

# CLIENT-SIDE CACHING FOR CLIENT-SERVER OLAP SYSTEMS

by

Elaheh Kamaliha

B. Sc., Sharif University of Technology, 2009

A 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

© Elaheh Kamaliha 2013  
SIMON FRASER UNIVERSITY  
Spring 2013

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:** Elaheh Kamaliha  
**Degree:** Master of Science  
**Title of Thesis:** Client-Side Caching for Client-Server OLAP Systems

**Examining Committee:** Dr. Uwe Glasser  
Chair

---

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

---

Dr. Oliver Schulte, Professor, Computing Science  
Simon Fraser University  
Supervisor

---

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

**Date Approved:** 8 April 2013

---

## Partial Copyright Licence



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

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

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

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

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

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

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

Simon Fraser University Library  
Burnaby, British Columbia, Canada

revised Fall 2011

# Abstract

Recently, client-side data caching has become popular for building highly interactive web applications. This research is about client-side data caching in a client-server OLAP system. Contrary to the traditional OLAP systems, the system we study here is client-centric which contains a light-weight OLAP engine and a data cache, such that queries posed by the user may be processed on the client without a round-trip to the server. It has been shown that this system works better than the traditional ones in some data visualization scenarios. In this study, we focus on the client-side data caching in this system. We implemented a new data structure for storing multidimensional data on the client-side, which occupies a small memory footprint and decreases the load time significantly with a minimal client-side processing overhead. Also, we investigate the efficiency of client-side caching when the processing of the query needs extra data that are not already client-resident. Our results show that in some cases it could be cost-effective to download more than %80 of the required data from the server, rather than to download the entire sub-cube from the server.

**Keywords:** OLAP; client-server system; client-side aggregation; client-side caching;

*To maman and baba  
for their endless love,  
and everlasting support*

*“In theory, there is no difference between theory and practice; In practice, there is.”*

*— Chuck Reid*

# Acknowledgments

I would like to thank Dr. Uwe Glasser for taking time out of his schedule to serve as the chair of the committee. I am also thankful from Dr. Oliver Schulte for accepting to be my supervisor and dedicating his valuable time for reviewing my thesis.

My special gratitude goes to my senior supervisor, Dr. Wo-Shun Luk, who has supported me through out my thesis from the beginning to the end with his knowledge and patience. Anytime I needed help, he made his time available and helped me with his valuable guidance and advice. It was not possible for me to finish this journey without his encouragement and useful points.

I would also like to acknowledge SAP-BusinessObjects for partially funding this project.

# 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 Server-side vs Client-side data caching . . . . .	1
1.2 Client-centric OLAP . . . . .	1
1.3 Objective and Motivations . . . . .	2
1.4 Research Problems . . . . .	3
1.4.1 Research Approach . . . . .	4
1.4.2 Research Findings . . . . .	4
1.5 Thesis Organization . . . . .	5



<b>2</b>	<b>Overview of the Client-Server System and Data Flow</b>	<b>6</b>
2.1	Server-side Components . . . . .	8
2.1.1	OLAP Server . . . . .	8
2.1.2	XML for Analysis and ADOMD . . . . .	9
2.1.3	Web Services . . . . .	11
2.2	JavaScript OLAP Client . . . . .	11
2.2.1	UI . . . . .	12
2.2.2	Query Engine . . . . .	13
2.3	Client-server Data Flow . . . . .	16
<b>3</b>	<b>Client-Side Caching for Web-Based OLAP</b>	<b>22</b>
3.1	Design and Implementation of Client-side Caching . . . . .	23
3.1.1	Using cell Table for Storing Cached Sub-Cube . . . . .	23
3.1.2	Searching Cached Data . . . . .	24
3.2	Performance Comparison with JSOC . . . . .	26
3.3	Using Cache for Answering Further Sub-Cube Requests . . . . .	29
3.3.1	Server-side Caching Implemented in JSOC . . . . .	30
3.3.2	Using Client-Side Caching for Answering Further Sub-Cube Requests	31
3.3.3	Proposed Algorithm for Using the Client-Side Cached Sub-Cube . . .	34
3.4	Evaluation for Using Client-Side Cached Sub-Cubes . . . . .	39
<b>4</b>	<b>User Workflow</b>	<b>43</b>
4.1	Workflow Scenario . . . . .	44
4.1.1	Trade-off between Downloading Answer Sub-cube and Inflated Version Sub-cube . . . . .	45
4.1.2	Data Aggregation for Drill-down Request . . . . .	46
4.1.3	Proposed Algorithm for Drill-down Operation . . . . .	49
4.2	Experiment Results for the Workflow . . . . .	50
4.2.1	Comparing the Proposed Aggregation Algorithm vs. the Original Al- gorithm . . . . .	51
4.2.2	Answer sub-cube vs. Inflated Sub-cube Comparison . . . . .	52
<b>5</b>	<b>Conclusion and Future Work</b>	<b>55</b>
5.1	Conclusion . . . . .	55

5.2 Future Work . . . . .	56
<b>Appendix A Partial XMLA Response</b>	<b>57</b>
<b>Appendix B Tables for Caching Experiment</b>	<b>59</b>
<b>Appendix C Tables for Workflow Experiment</b>	<b>63</b>
<b>Bibliography</b>	<b>67</b>

# List of Tables

3.1	Test environment . . . . .	27
3.2	Sub-cube download size . . . . .	28
3.3	Sub-cube download information . . . . .	29
3.4	A sample test case for client-side caching . . . . .	39
3.5	download information for the sample test case in table 3.4 . . . . .	40
3.6	Sub-cube download with/without using cache . . . . .	41
4.1	One sample of the workflow sequence . . . . .	45
4.2	workflow summary . . . . .	53
B.1	Evaluating Client-side caching for 2-dimensional . . . . .	60
B.2	Evaluating Client-side caching for 3-dimensional . . . . .	61
B.3	Continued from table B.2 . . . . .	62
C.1	User workflow . . . . .	63

# List of Figures

1.1	Drill-down/Roll-Up Operation . . . . .	3
2.1	Server-Centric OLAP architecture . . . . .	6
2.2	Our adopted Client-Centric OLAP architecture . . . . .	7
2.3	Multidimensional data [4] . . . . .	10
2.4	An screenshot of JavaScript OLAP Client user interface . . . . .	12
2.5	TreeView a)Before loading hierarchy content . . . . .	13
2.6	The <i>answer</i> object containing the aggregated measures. . . . .	15
2.7	(top)A sample cross-tab. (button)Drill-down on a cell . . . . .	16
2.8	Example of an MDX query . . . . .	18
2.9	A cell in the XMLA result . . . . .	19
2.10	Transforming Multidimensional result set to a fact table . . . . .	21
3.1	left: A complete fact table with 3 dimensions . . . . .	24
3.2	Determining the complete tuple for each CellOrdinal . . . . .	26
3.3	Cached sub-cube is a superset of the client's request . . . . .	32
3.4	2 MDX queries is needed to fetch the rest of the client's request. . . . .	33
3.5	1 MDX query is needed to fetch the rest of the client's request. . . . .	35
3.6	Cached sub-cube, new client's request . . . . .	36
4.1	An screen shot of the JavaScript OLAP Client. . . . .	47
4.2	The algorithm for drill-down process . . . . .	48
4.3	Comparing processing time for the new drill-down . . . . .	52

# Chapter 1

## Introduction

### 1.1 Server-side vs Client-side data caching

Data caching is one of the most popular and effective techniques to boost the performance of a client-server system. By caching the data accessed by users on the server side, e.g., DBMS and Web servers, data items that are frequently accessed will be available immediately without resorting to the database on the disk.

Recently, client-side data caching has emerged as a popular technique for building highly interactive web applications. These applications are web-based, i.e., they are written in Javascript that run on any generic browser. As such, there is nothing to maintain and runs anywhere internet access is accessible. Client-caching and pre-fetching of data will eliminate not only the round trip of retrieving data from the server, but also the unpredictable network delays that are beyond the users control. As a result of client-side processing, the server will be given additional resources either to serve more clients and/or to become more responsive to current clients. Development tools for client-side data caching are provided on major development platforms [6], [5].

### 1.2 Client-centric OLAP

In this thesis, we study client-side caching for a specific type of application, i.e., OLAP (OnLine Analytic Processing) applications that run in a client-server environment. In particular, the client targeted by this study is typically a resource-constrained device, i.e., a mobile device.

To the best of our knowledge, the first study of Web-based OLAP was published in [13]. Compared to many Web-based client products on the market, the Web-based OLAP described in [13] features light-weight OLAP engine, which is capable of performing OLAP operations on the data cached on the client side. This engine, written in Javascript, is embedded in the web page downloaded from the server side. To distinguish it from the traditional client-server OLAP systems, we call it client-centric OLAP.

Data prefetching is a common technique in reduction of I/O latency for server-side data caching, when the data in the file are processed sequentially, e.g., in a file scan operation. Data prefetching on the client-side works very well for some data visualization scenarios [13]. As an example, consider the scenario where an executive of a company would like to explore sales amounts in the top 10 regions and their contributions to the total amount, in the past 2 weeks. To this end, the application features a slider in its user interface that is bound to the time dimension. Every movement of the slider triggers a new query for different day, the answer of which is shown in a pie-chart. A server-centric client has to take a round trip to the server to find out the answer. In the case of Web-based OLAP, a data cube containing the answers for every day in the past 2 weeks, is first downloaded from the server. With the data in the cache on the client side, the answer will be produced by the local OLAP engine. Another data visualization scenario involves an  $x - y$  scatter plot, where  $x$  and  $y$  are measures of a data cube. By downloading the scatter plot plus the associated dataset, the user may zoom in a specific rectangle of the plot by a mouse move-over.

### 1.3 Objective and Motivations

This thesis is to study the issues related to client-side data caching in an OLAP system against a workflow, i.e., a sequence of queries. In comparison, each data visualization scenario as described in [13] is treated as a workflow, and pre-fetching of data is confined to the scope of the scenario. For a general workflow, the data downloaded from the server are stored in the data cache, which may, or may not, be utilized for processing subsequent queries.

Despite its speculative nature, we believe client-side data caching is still an effective strategy, because the subsequent queries to be posed by an OLAP user are often related to the answer of the current query. For instance, the user may want to compare the answer of the current query with the answer of a previous query. For another instance, the user

may create new queries based on the answer of an existing query. Consider drill-down, one of the OLAP operations, which allows the user to derive the aggregates by navigating from the current level of a hierarchy (-ies) to the more detailed level. Consider the 2-dimensional “cube” shown in figure 1.1 , with Geography and CalendarYear as the dimensions, has one cell storing the measure Sales. One may drill-down into the dimension CalendarYear, so that the sales for the year may be split by  $H1$  and  $H2$ .

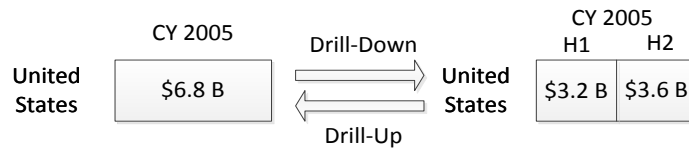


Figure 1.1: Drill-down/Roll-Up Operation

When a query is submitted by the user on the client side, the client may seek from the OLAP server, not only the answer sought by the user, but also the data associated with the more detailed levels of dimension hierarchies included in the query. For example, if the user requests the sales of US in the calendar year 2005, the client, in anticipation of the drill-down, may request the sales for first-half and second-half years instead. Pre-fetching of additional data is even more speculative, but it could work very well if the drill-down session is long and involved, since the user can develop various roll-up operations supported by the local OLAP engine. Similarly, a roll-up operation involves aggregation at a level higher than the current level of hierarchy. In this case, the data cache may not contain all necessary data required by the aggregation. The missing part could be downloaded from the server.

## 1.4 Research Problems

When a query is submitted without any indication of subsequent drill-down operations, the client will be faced with choosing one of the following options:

- (a) Send the query to the server, requesting only the answer to the query. We shall call it the answer sub-cube. The 2-D cube on the left side of figure 1.1 is an example.
- (b) Same as (a) , except that the sub-cube returned is an inflated version of the answer sub-cube, that is, each dimension of the answer sub-cube may include extra nodes, which are child nodes of nodes in the answer sub-cube. The 2-D cube on the right side of Fig. 1 is an example of inflated version of the answer sub-cube on the other side.
- (c) Answer the query with the data in the local cache.
- (d) Same as (c), except that a modified query is sent also to the server.

It is a fairly simple task for deciding to go for the option c), i.e., on the condition that the data in the local cache is sufficient for deriving answer to the query. Otherwise, one of the remaining three options will be chosen. In fact, the first step will be the choice between d) and a). If d) is not the choice, then the final choice is between a) and b). The objective of research is to develop a cost function with which to make these decisions.

#### 1.4.1 Research Approach

To be sure, the cost function depends on factors which are difficult to quantify, or are simply non-technical in nature. For example, the client needs to know to what extent the user will drill down into the inflated answer sub-cube. There is also a factor related to the preference of the user with working data off-line over on-line with the OLAP server. Our focus in this research is to consider only the cost in terms of the following components: (i) processing on the server side, (ii) the data download from the server to the client, and (iii) the processing at the client side. The first task to proceed is perhaps the most important one: to design and implement a client-side data cache that will minimize the cost in each of the three components. Next, we conduct experiments on sample data to determine the cost associated with each option. The experimental results hopefully will shed light on whether client-side data caching is effective.

#### 1.4.2 Research Findings

- We have developed a new data structure for storing data inside a cube on the client side, which takes a small memory footprint, and requires minimal overhead for transformation of the source data from the OLAP server into the cached data.



- Client-side data cache helps processing a new query, even though the processing of the query may require extra data that are not already client-resident. For some queries, it is faster to download over 80% of data required from the server, than to download the entire inflated answer sub-cube from the server. In other words, the option (d) is sometimes cost-justified over (b).
- Pre-fetching in anticipation of drill-down works. Our experiments show that for some queries, especially those with relatively smaller answer sub-cube, downloading an inflated answer sub-cube can be cost-justified, when they are followed by quite a few drill-down queries on the inflated answer sub-cube. In other words, the option (b) is sometimes cost-justified over (a).

## 1.5 Thesis Organization

The organization of the rest of this thesis is as follows : in chapter 2, we present an overview of the client-server OLAP system with the emphasis on the client-centric OLAP system. In this chapter, we explain each of the components of the client-based OLAP architecture. Also, we describe the data flow in the system for downloading and presenting a sub-cube to the user.

In chapter 3, we explain in details the client-side caching we have developed. We explain the method we have designed and implemented for storing multidimensional data and we explain how the client-side data engine is able to use this data structure for aggregation. Also, we propose an algorithm for using option (d) where for the new user's requests, we first check if it has any overlap with a cached sub-cube and we send modified MDX queries to the server to download the required data.

In chapter 4, we compare inflated sub-cube with the *answer sub-cube* in a workflow scenario. In this chapter, we present our algorithm for drill-down operation which is faster than the original algorithm. Then we compare the results with the *answer sub-cube* approach. In fact, we compare option (a) and (b).

In the last chapter, we discuss the conclusion of this work. We also present the future work for this research.

## Chapter 2

# Overview of the Client-Server System and Data Flow

In OLAP research, there is quite a lot of research done on transferring server-side aggregation to the client-side. Typically, the “thin” client is used to submit a query to the server and display whatever it receives from the server. All the communications with the OLAP server and the aggregation is done at the server side. Figure 2.1 shows an overview of the traditional server-centric architecture.

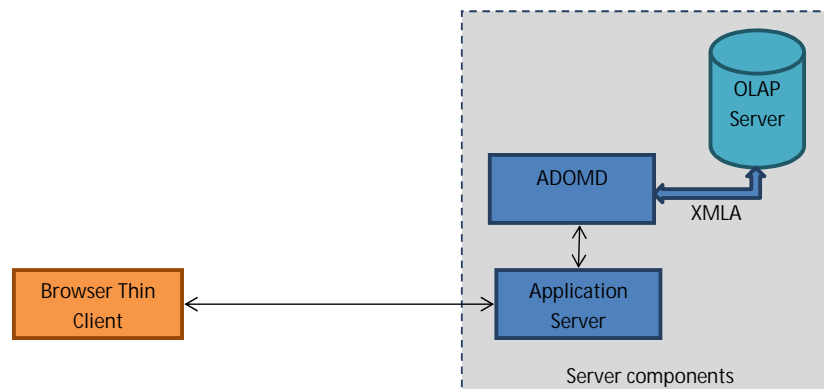


Figure 2.1: Server-Centric OLAP architecture

Hsiao explores in his thesis, the potential of a client-side OLAP system. He has proposed and implemented a client-centric OLAP system which is able to perform OLAP operation locally. In his design, the client application is not anymore a “thin” client. He has implemented a client-side query engine which connects to a web service to download metadata and multidimensional data and performs data aggregation locally. The engine code is in JavaScript and the engine runs in a browser platform.

Figure 2.2 is an adoption of the client-centric architecture proposed by Hsiao. The server-side components are: OLAP server, ADOMD,XMLA and web service. The server-side components are OLAP server, ADOMD, XMLA and the web service. OLAP server is the core of each OLAP system. It is the data provider that contains the databases in the form of data cubes. *XML for Analysis* (XMLA) is an Application Programming Interface for data access in OLAP server. ADOMD is a Microsoft .Net framework that implements the XMLA protocol. And the last component is the web service which acts as a middleware between the client-side and server-side applications. The web service receives the queries from the client and submits it to the server through the ADOMD layer. Also the data returned from the server is retrieved by the web service and the data is prepared for sending to the client in this component.

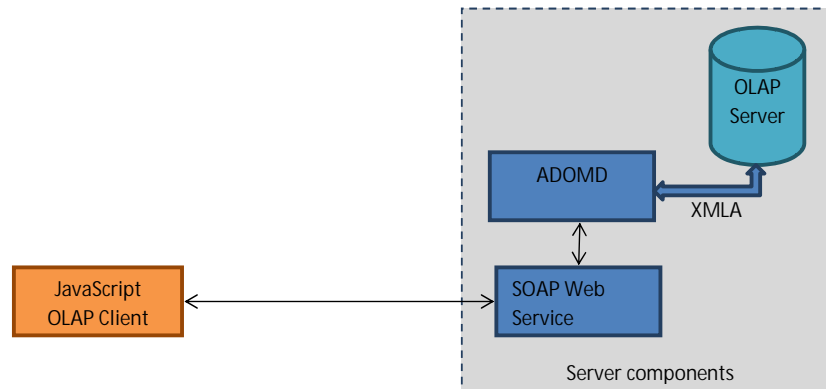


Figure 2.2: Our adopted Client-Centric OLAP architecture

The JavaScript OLAP Client (JSOC) shown in figure 2.2 consists of a User Interface

(UI) which is a web page displayed to the client, and a data engine which is written in JavaScript. The UI is designed to provide easy interaction with the user. User can select to download metadata from the drop-down menus. Also, a *TreeView* of the data is shown to the user so that she can select the dimensions to build a sub-cube and submit an MDX query.

When the client decides to submit a query, the local data engine accepts user's request from the UI. Then an MDX query is sent to the OLAP server via the web service. The OLAP server returns the answer in XMLA format. The web service will process the XMLA message and return the dataset to the client.

When the JavaScript OLAP Client receives the dataset, it is able to process user's requests regarding this data. JSOC implemented by Hsiao is mainly focused on Data Visualization. The local data engine is able to answer client's request for cross-tab generation, m-Day Moving Average, Scatterplot, and Top-k Contribution locally from the downloaded dataset. However, in our work we are focused on the user experience with the workflow when a sequence of MDX queries are requested.

The rest of this chapter is about the components of the client-centric system we have adopted and the data flow between the components. In section 2.1 we describe the server-side components and also we give general information about multidimensional data. In section 2.2, the JavaScript OLAP Client is introduced and the common OLAP operations handled by JSOC are explained. In the last section, we explain the data flow in the system in details.

## 2.1 Server-side Components

There are four different components on the server-side. In this section, we explain each of these components.

### 2.1.1 OLAP Server

There is plenty of work and research done on different aspect of OLAP such as aggregation and cubing, modeling multidimensional data and join optimization [1, 2, 3, 11]. Over the past decades, a new market has emerged for systems that provide the ability to analyze very large amount of data which is called *online analytical processing*(OLAP). Because OLAP systems are designed specifically for analyzing, they typically do not need to do both read

and write [14]. All is needed is reading data. Therefore, OLAP systems have the advantage of high speed data access. Another important factor that distinguishes OLAP from relational databases is multidimensional data structure. We explained about the conceptual model of multidimensional data earlier in this chapter. In general, the core of each OLAP server is an OLAP cube. This cube contains numeric values called *measures* which are the object of the analysis. An example of such data cube is shown in figure 2.3.

### **Multidimensional Data and Metadata**

Typically, each database contains one or more fact tables which are related to a number of dimension tables. Each fact table contains a number of records with one or more measures. Example of measures could be quantity or sale amount. Each record consists of attributes associated with it. Example of these attributes could be customer number, store number, product number.

Each dimension consists of a set of attributes. Examples of this attributes for date dimension are week, month and quarter. There may be a hierarchical relation between the attributes of a dimension. Figure 2.3 shows hierarchy levels for the three dimensions: Product, City and Date. Primary members are at the lowest level and are the values for the attributes in each record in the fact table. OLAP server builds a multidimensional cube for each fact table that pre-aggregates measures in the fact table in different level of hierarchy of each dimension.

Figure 2.3 shows a 3-dimensional cube. The cube is an aggregation of sales quantity measure for product, city and date dimension. Each combination of members from each dimension makes a tuple. That is, for an  $n$ -dimensional cube, a tuple consists of  $n$  members that specify one cell. If a tuple does not specify any member for one dimensions of the cube, every member in that dimension is considered.

OLAP vendors have standardized an interface that allows an application execute MDX query. This interface is called *XML for Analysis* or XMLA. Any application can access any multidimensional database that supports XMLA.

#### **2.1.2 XML for Analysis and ADOMD**

XMLA is a protocol developed for communication between applications and analytic data provider. It is a standard XML format that defines the messages and representation of

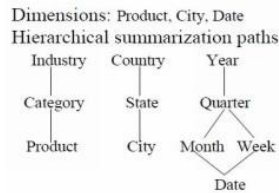


Figure 2. Example of hierarchies [2]

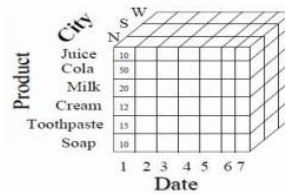


Figure 3. A 3-Dimensional cube [2]

Figure 2.3: Multidimensional data [4]

multidimensional data types. The back and forth communication of messages is done using web standards: HTTP, SOAP and XML. Essentially, XMLA is a SOAP envelope with XML content sent over the HTTP layer. The query language used is MDX which is explained later in this chapter.

Technically speaking, XMLA is a specification for a set of XML message interfaces that uses *Simple Object Access Protocol* (SOAP) to define data access interaction between a client application and a data provider working over the Internet. Using a standard API, XMLA provides open access to multi-dimensional data from varied data sources – any client platform to any server platform – through web services that are supported by multiple vendors.

The reason that XMLA has become a standard is that XMLA is adopted by all major OLAP service providers such as IBM Infosphere Warehouse Cubing Services, Oracle/Hyperion Essbase, Microsoft Analysis Services and Mondrian on top of MySQL.

XMLA has two methods: *Execute* and *Discover* [8].

*Discover* message will return the *metadata* to the application. Metadata contains information about dimension, hierarchies and measures.

*Execute* message will execute an MDX query on the OLAP server. The result of the XMLA execute operation for an MDX query shown is included in appendix A.

Since the result of the XMLA execute message contains a lot of details, it will produce a lot of overhead for the client. The results include axes information, but this information is already available at the client when *metadata* is downloaded. Removing this overhead from the XMLA response will decrease the amount of the data sent over the network.

ADOMD.NET is a Microsoft.NET Framework that uses XMLA protocol to communicate with the analytical data sources. ADOMD is the so-called middleware, which makes it easy

for the web service to communicate with the OLAP server. A call from the web service to ADOMD, will be translated into a method of XMLA to submit to the OLAP server. The server returns the answer within an XMLA message to ADOMD. The ADOMD has functions to parse the results and produce convenient objects such as dimension, hierarchy and fact table for the web service.

### 2.1.3 Web Services

The web service layer is a middleware between the JavaScript OLAP Client and ADOMD layer. A web service consists of a set of methods that can be called from the remote distance client. Web service receives client's requests and depending on the type of the request, appropriate ADOMD function is used to retrieve the data from the OLAP server. Examples of client requests could be the list of dimensions, the contents for a hierarchy, or an MDX query for retrieving a sub-cube. Since XMLA is needed to communicate with the OLAP server, the web service uses ADOMD to access the OLAP server.

Once the data is retrieved from the server, the web service processes the results to clean the data and put it in the structure desired by JSOC.

In general, the web service is designed to provide server side support for the JavaScript OLAP Client on two aspects transmitting: (i) metadata and (ii) multidimensional data. Together with the JavaScript OLAP Client, they are the main parts in this research.

## 2.2 JavaScript OLAP Client

JavaScript OLAP Client (JSOC) is developed on a browser platform. The benefit of this approach is that any device with an internet connection and a browser can be an OLAP client. No application is installed on the client machine and therefore the client machine is maintenance-free: there is no cost for updating the application on every user device and the user is always running the latest version of the code.

The JavaScript OLAP Client consists of a graphical User Interface which is displayed to the user, and a data engine which manages the data flow on the client-side. The UI is designed to interact with the user: user can select to download the list of dimensions and download a sub-cube with desired dimensions. The data engine collects the information from the UI and manages the communication with the server and displays the result to the client.

In the following, we first explain the graphical User Interface(UI) used to interact with the user client. Then we describe the tree view used for metadata representation. We also discuss the query engine and the way it performs OLAP operations.

### 2.2.1 UI

Figure 2.4 shows a screen-shot of the interface displayed to the user. After user has selected the server and cube she wants to connect to, a list of dimensions and categories is downloaded and represented in a *tree* structure. The box for the dimensions tree is specified in the figure by number 1. The open source *TreeView* from *Yahoo! UI 2*[12] is adopted to give the user the option to select axes from the dimensions and hierarchies. The components provides many features for developing the JavaScript client such as: the ability to instantiate the tree nodes as objects, methods and properties to access node level, children nodes, dynamic loading of sub-tree node on the node expansion.

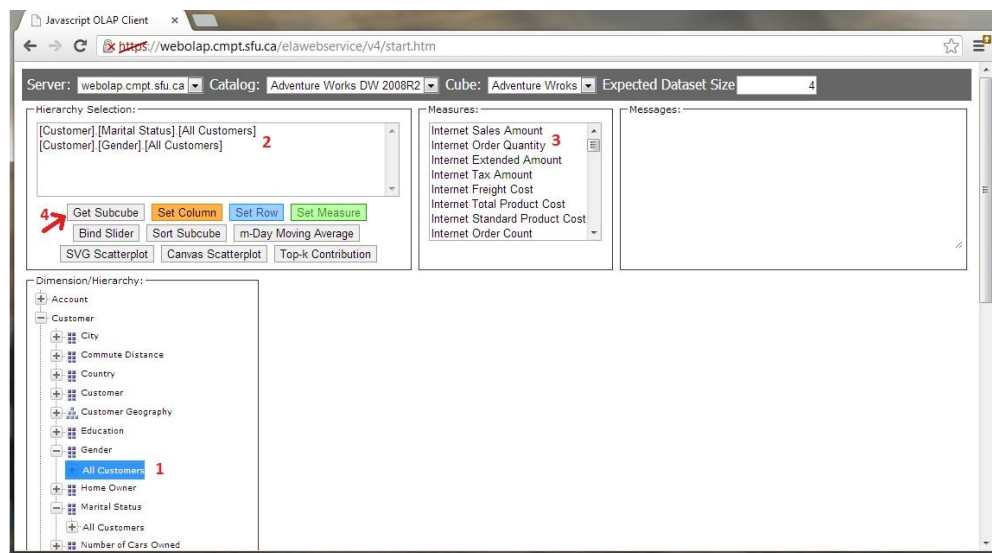


Figure 2.4: An screenshot of JavaScript OLAP Client user interface

Figure 2.5 shows an example of dimensions on the first level of the TreeView. Note that at this stage, the JavaScript OLAP Client does not load the whole content of every hierarchy. This is due to the fact that the complete hierarchy data are too large to be downloaded at once. Instead, the TreeView components provides the ability to define a loading function when the user expands a node. When user selects to expand a hierarchy,



JSOC downloads the contents of the sub-tree from the web service layer. The [Geography] dimension contains five hierarchies: [city], [country],[Geography], [postal code] and [state-province]. The [Geography] hierarchy contains a multi-level sub-tree.

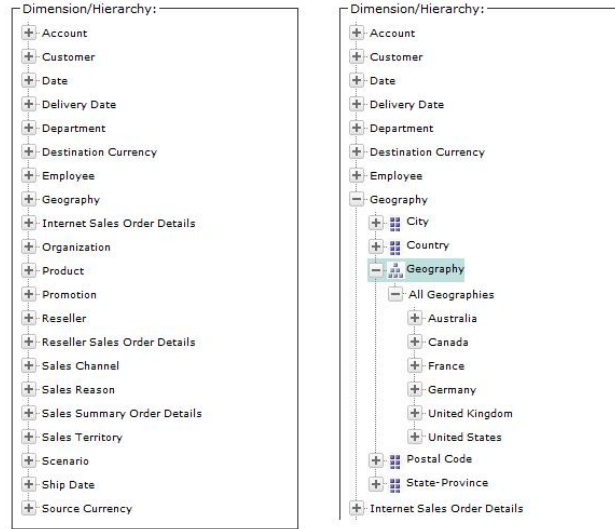


Figure 2.5: TreeView a)Before loading hierarchy content, b) After loading Hierarchy content

### 2.2.2 Query Engine

User can select one or more hierarchies from the TreeView to build a sub-cube. For example, user selects to download a 2-dimensional sub-cube: [Geography].[Geography].[Canada] and [Product].[product Categories].[All Products]. These selected nodes are added to the *Hierarchy Selection* textbox in the UI which is just above the tree view. The list of hierarchies shown in this box will make the dimensions of the sub-cube requested from the web service.

The TreeView component provides the option of retrieving the tree nodes for the hierarchy selection. Therefore the data engine have all the information about the sub-cube such as size and hierarchy levels.

When the user selects to download a sub-cube, the *fact table* for that sub-cube is loaded to the browser memory. Fact table is a 2D array with  $n$  rows and  $(totalDim+totalMeasure)$  columns. Each row in the fact table represent a tuple in the dataset. the first  $totalDim$  columns contain primary member values for each dimension of the tuple. The  $totalMeasure$  columns contain the numeric measure values.

The JavaScript OLAP Client provides cross-tab, drill-down and rull-up operations locally. In the following we explain each of these operations and how they are implemented in JSOC.

### Cross-tab

One of the operations that OLAP engine process is cross-tab (or pivot table). Suppose that the user requests a cross-tab query based on  $[rowAxis].[rowNode]$  and  $[columnAxis].[colNode]$ . The OLAP engine performs a linear scan on the fact table. A tuple (a row in the fact table) is determined to be part of the cross-tab if the tuple members for the row and column axes are descendants of the  $rowNode$  and  $colNode$ . Then the tuple is aggregated to a cell in the answer matrix. The following algorithm shows the general idea for cross-tab designed by Hsiao:

---

#### Algorithm 1 getMatrixAnswer

---

```

1: var answer=new object
   {initialize the answer object by adding rowGroup member names as object properties}
2: for i=0 → rowGroup.length do
3:   {initialize each property to an object of colGroup members}
4:   answer[rowGroup[i]]=new object
5:   for j= → colGroup.length do
6:     answer[rowGroup[i]][colGroup[j]]=0
7:   end for
8: end for
9: {Linear scan of the fact table}
10: for i=1 → factTable.Length do
11:   currentTuple=factTable[i]
12:   rowMember=current[rowAxis]
13:   colMember=curren[colAxis]
   {lookup the actual node in the treeview}
14:   rowMemberNode=keyToNodeMap(rowMember)
15:   colMemberNode=keyToNodeMap(colMember)
   {the aggregation point is the ancestor node at the level of the rowAxisNode+1}
16:   row=rowMemberNode.getAncestor(rowAxisNode.depth+1)
17:   col=comMemberNode.getAncestor(colAxisNode.depth+1)
18:   if row is a property of answer AND col is a property of answer[row] then
19:     answer[row][col]+ =currentTuple[measureIndex]
20:   end if
21: end for

```

---

The variable *answer* is an object with  $|rowGroup|$  properties. *rowGroup* is an ordered list of members that are the children of *rowNode*. Each *rowGroup* is itself an object with  $|colGroup|$  properties whose values are the aggregated measures. Similarly, *colGroup* is an ordered list of members that are the children of *colNode*. Figure 2.6 shows the answer object returned by this algorithm for the cross-tab query on [Geography].[Geography].[Canada] and [Product].[product Categories].[All Products]. The *answer* object contains six properties for six states of Canada. For example, the first property is AB(Alberta). Then each property contains four properties for the product categories 1 to 4. The fact table is scanned row by row. If current tuple in the fact table is  $[PostalCode.T2P2GB : Calgary : AB, Product : 483]$ , then *row* and *col* nodes will be set to “State-Province.AB:CA” and “Category.4” and since both values for *row* and *col* nodes exists in the answer object, the measure value will be added to the corresponding cell in the answer matrix.

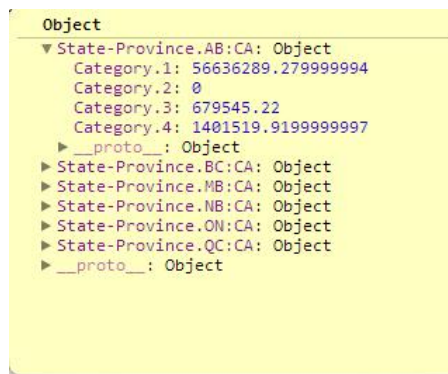


Figure 2.6: The *answer* object containing the aggregated measures.

This algorithm uses a linear table scan method. Therefore the algorithm runs in  $O(n.H(t))$  time where  $H(t) = \max(H(rowAxis), H(columnAxis))$

### Drill-down

Drill-down means viewing data at a level of increased detail. When a cross-tab is performed on [rowDim]. [rowHier]. [rowMember].Children and [columnDim]. [columnHier]. [columnMember].Children, a drill-down operation on a cell can occur on any row member  $r \in rowMember.Children$  and any column member  $c \in colMember.Children$ . As shown in figure 2.7, the operation will generate an embedded cross-tab within the intended cell. User

can also perform drill-down operation on row  $r$  or on column  $c$ . In this case, a new cross-tab table will be generated which is not embedded. In each case, the *getMatrixAnswer* algorithm will be supplied with sufficient *rowMember* and *columnMember*.

		All Products			
		+ Accessories	+ Bikes	+ Clothing	+ Components
Canada	Alberta	1401519.92	56636289.28	679545.22	0.00
	British Columbia	20322038.84	821226194.56	9853405.69	0.00
	Brunswick	700759.96	28318144.64	339772.61	0.00
	Manitoba	700759.96	28318144.64	339772.61	0.00
	Ontario	18219758.96	736271760.64	8834087.86	0.00
	Quebec	9810639.44	396454024.96	4756816.54	0.00

		All Products						
		+ Accessories	+ Bikes			+ Clothing	+ Components	
			+ Mountain Bikes	+ Road Bikes	+ Touring Bikes			
Canada	Alberta	1401519.92	9952759.56	14520584.03	3844801.05	679545.22	0.00	
			Calgary					
			Edmonton					
	British Columbia	20322038.84				821226194.56	9853405.69	0.00
	Brunswick	700759.96				28318144.64	339772.61	0.00
	Manitoba	700759.96				28318144.64	339772.61	0.00
Ontario	18219758.96				736271760.64	8834087.86	0.00	
Quebec	9810639.44				396454024.96	4756816.54	0.00	

Figure 2.7: (top)A sample cross-tab. (bottom)Drill-down on a cell

### Roll-up

Roll-Up operation refers to the process of decreasing the detail in data view. In JSOC, when a roll-up is requested, the data will be re-stored with the original cell value and embedded table will be removed which is done at no additional cost. In the case that drill-down is led to generating a new grid, data can be restored with re-generating the original cross-tab by choosing the row and column member.

## 2.3 Client-server Data Flow

In this part, we go through the data flow between the client and server application. We explain how the user initiates a requests via the User Interface and the process of back and forth communications between the client and server applications until the result is displayed to the user.

- (a) **Accept user’s request from UI:** The UI shown in figure 2.4 provides the user with the options to download metadata and subcubes. At the beginning, the user needs to view the list of dimensions for the selected cube. So in each session, the first thing that

the user should do is select a cube to view the list of dimensions. Then the user is able to formulate an MDX query from the available dimensions. In the following we explain the data flow for each of the two cases: Metadata download and MDX query.

- (I) **Metadata query:** A request for metadata is submitted when the user clicks on a node in the tree view. In figure 2.4, user has selected two hierarchies: [Customer]. [Marital Status] and [Customer]. [Gender]. The data engine adds the selected nodes to the “Hierarchy Selection” textbox which is specified with number 2 in this figure. This textbox keeps the hierarchy selection for the sub-cube that is going to be downloaded later. Then the data engine establishes a communication with the server to requests the detailed information about the content of the selected hierarchies.

The web service receives the request for the metadata, first it specifies the object names such as the cube name, dimension name, and hierarchy name and then invokes ADOMD method to send an XMLA *discovery* message to the OLAP server. The ADOMD will return the metadata to the web service where the data is processed and transmitted to the client.

Hsiao explains in detail that how he transfers the tree structure of dimensions and hierarchies to the client. Note that the content of the dimensions are downloaded only for the first level, i.e. the first order children of the dimensions. User can explore each dimension by choosing to open a hierarchy node which leads to downloading the content of that hierarchy. This is to avoid the cost of downloading the whole cube at once.

When the data engine receives the metadata, it will store the data in the tree view structure. The *TreeView* component provides this ability to load each node in the tree view with the corresponding content and display it to the user.

- (II) **Formulation of MDX query:** When the user selects the hierarchies for each dimension of the sub-cube, she will select one or more measures from the list of measures specified by number 3 in the figure 2.4. Then she will click on “Get sub-cube” button which is numbered as 4 in the figure.

The data engine collects the list of hierarchy selections and the measures from the UI. Then it will formulate an MDX query. Multidimensional Expression(MDX) is a query language for OLAP databases. MDX is analogous to SQL which is the

query language for relational databases. Figure 2.8 shows an example of MDX query which has 2 dimensions: [Customer].[Education] and [Geography].[Geography]. [All Geography] and 1 measure: Internet Sales Amount.

```
SELECT {[Measures].[Internet Sales Amount]} on axis(0),
DESCENDANTS([Customer].[Education].[All Customers] , ,LEAVES) on
axis(1),
DESCENDANTS([Geography].[Geography].[All Geographies] , ,LEAVES)
on axis(2)
FROM [Adventure Works]
```

Figure 2.8: Example of an MDX query

The data engine will send this query to the server-side and wait for the respond.

- (b) **Send MDX query to OLAP server via web service:** The MDX query submitted to the server will be received at the web service. The web service includes the MDX query inside an XMLA *execute* message and submits it to the OLAP server. The data provider executes the query and returns the multidimensional data set in the form of an XMLA message to the web service.

The result of performing *execute* message is highly detailed and needs to be cleaned and converted to a data structure that the client can consume. In the web service layer, the data received in the XMLA format is parsed and converted to the desirable format for JSOC.

- (c) **Processing of XMLA messages in web service layer:** One of the main parts that our system differs from the Hsiao's is in the approach used for multidimensional data transformation. In Hsiao's system, the fact table is built from the dataset prior to sending it to the client. However, in our design we use a different way to present the dataset. In the following, we explain the approach that Hsiao has used for this purpose. We discuss our own design which leads to reducing the data sent over the network in chapter 3.

In the XMLA message returned from the OLAP server to the web service (appendix A) the cells are addressed in a special format: each cell is uniquely identified by a numeric

value named *CellOrdinal*. This number specifies the exact position of a cell in a sub-cube as if the multidimensional data is a  $d$ -dimensional array and the array is traversed in a row-major. Each cell in the XMLA message is equivalent to a row in the fact table. The web service transforms the XMLA result into a fact table by parsing the XMLA message: for each cell, it takes out the *CellOrdinal* and produces the corresponding row in the fact table. In the following, we explain how *CellOrdinal* of a cell is calculated from the axes of a sub-cube and similarly, how one can extract the tuple ordinal of a cell from its *CellOrdinal*.

- (I) **Converting CellOrdinal into a tuple of fact table:** The multidimensional result is a sub-cube in which the cells are addressed as **CellOrdinals**. A CellOrdinal is a number assigned to a cell as if the dataset is a  $D$ -dimensional array where  $D$  is the number of dimensions in the result set[7]. For a cell with tuple ordinal  $(s_0, s_1, \dots, s_{D-1})$ , cellordinal is:

$$\sum_0^{D-1} s_i.e_i \text{ where } e_0 = 1 \text{ and } e_i = \prod_0^{i-1} U_k \quad (2.1)$$

Where  $U_k$  is the number of members in axis  $k$ . Cells are numbered from 0 to  $n$  in a row-major format for a sub-cube of size  $n$ . For example, the block of code shown in figure 2.9, is part of the result returned by XMLA which is related to a cell in a sub-cube.

```
<CellData>
  ...
  <Cell CellOrdinal="5">
    <Value xsi:type="xsd:decimal">9900142.75</Value>
    <FmtValue>$9,900,142.76</FmtValue>
  </Cell>
  ...
</CellData>
```

Figure 2.9: A cell in the XMLA result

For each cell in the multidimensional dataset, we can calculate the corresponding CellOrdinal using the formula in 2.1. Conversely, if we have the CellOrdinal number of a cell, number of axes and the number of members in each axis, we can uniquely specify the related tuple ordinal  $(measure_1, \dots, measure_m, s_1, \dots, s_n)$  where  $s_i$  is the value for dimension  $i$  for this tuple. For example, in the XMLA code in figure 2.9, CellOrdinal= 5 and cell Value is 9,900,142.76. We know that the sub-cube has 2 dimensions with cardinalities of  $U_0 = 5$  and  $U_1 = 655$ . by substituting these values in 2.1 we will find  $s_0 = 0$  and  $s_1 = 1$ . With the axis members in hand, the tuple for this particular cell is [Bachelor, Postal Code.2450:Coffs Harbour:NSW, 9900142.76].

- (II) **Fact table representation:** As we explained in subsection 2.2.2, the JavaScript OLAP Client implemented by Hsiao uses the in-memory dataset as an  $n * (|D| + |M|)$  array which corresponds to a fact table in the data warehouse used to build a cube. The web service is responsible to provide such array for the client.

The web service implemented by Hsiao performs the *execute* message on the OLAP server and collects the multidimensional result set such as the one shown in appendix A. The transformation of this result set to a fact table is done at the web service layer.

Above, we explained the process of converting each *CellOrdinal* into the stripped down dimensions members. The measure value is moved to the end of the  $D$  dimensions, then the row is inserted as a tuple into the fact table. The headings of the fact table are dimensions and hierarchy names. Table 2.10 shows a portion of a fact table created by transforming the multidimensional result set.

It is worth mentioning the potential size of the sub-cube fact table. In theory, if the data cube is dense, the maximum size of the sub-cube fact table is the size of the Cartesian product of all dimensions which is equal to:

$$\prod_0^{D-1} U_k \quad (2.2)$$

where  $U_k$  is the the number of members in axis  $k$  and  $D$  is the number of dimensions

- (d) **JSOC processes user's request:** Once the dataset is downloaded to the browser's memory, JSOC is able to process further queries regarding this dataset locally. In



	[Geography],[Geog	[Geography],[Geog	[Geography],[Geog	[Geography],[Geog Code],[MEMBER_C	[Customer],[Educat	[Measures],[Interne Sales Amount]
Australia	New South Wales	Alexandria	2015	Bachelors	9900142.7571	
Australia	New South Wales	Alexandria	2015	Graduate Degree	5460560.2513	
Australia	New South Wales	Alexandria	2015	High School	4638026.0686	
Australia	New South Wales	Alexandria	2015	Partial College	7723542.8848	
Australia	New South Wales	Alexandria	2015	Partial High School	1636405.2589	
Australia	New South Wales	Coffs Harbour	2450	Bachelors	9900142.7571	
Australia	New South Wales	Coffs Harbour	2450	Graduate Degree	5460560.2513	

Figure 2.10: Transforming Multidimensional result set to a fact table

Hsiao’s work, the focus is on data visualization: queries for cross-tab, drill-down, m-Day Moving Average, Scatterplot, and Top-k Contribution for the loaded sub-cube is processed in the data engine and displayed through the UI to the user. Hsiao does not study the cost of sub-cube download in details.

## Chapter 3

# Client-Side Caching for Web-Based OLAP

Depending on the client's requests, the server will return the data sub-cube in two different formats. In chapter 1 we defined two terms for identifying the formats of the data returned from the server to the client. The client may send a query to request the data sub-cube to answer only the current query. We called this sub-cube "answer sub-cube". Otherwise, the client requests a sub-cube that not only answers the current query, but also contains the answer for any query on the child nodes of the nodes in the answer sub-cube. We called this data sub-cube "inflated version of the answer sub-cube".

In this chapter, we assume that the user always chooses to download the inflated version of the answer sub-cube. This data is stored in the local machine. As long as the user submits queries within the current sub-cube, there is no communication cost and the client application will answer the queries with the local data engine. In the rest of this chapter, for the sake of conciseness we use "sub-cube" instead of "inflated version of the answer sub-cube".

In chapter 2, we explained that the web service designed by Hsiao returns the dataset to the client in the form of a fact table. In this chapter, we propose a new approach to represent multidimensional dataset: instead of fact table, our web service will send a *cell table* to the client which consists of only *CellOrdinals* and measure values extracted from the XMLA message. This way, we will decrease the volume of the data sent over the network and thus the network transmission time.

Also, since the client has already have paid the cost of downloading the sub-cube to the client, we intend to perform client-side caching. When the client decides to download a sub-cube, the local data engine first checks to see if this cube has already been cached. If the sub-cube has any intersection with the cached-cube, the data engine manages to use the part that exists in the cache and generate appropriate query only for the parts that are not already cached.

In the following sections, first we define *cell table*, then we explain how the JavaScript OLAP Client processes the cell table for data aggregation. Next, we explain the design and implementation of client-side caching. The evaluation results are discussed in each related part.


### 3.1 Design and Implementation of Client-side Caching

One of the goals of this thesis is reduce data sent over the network to decrease client's waiting time. To achieve this purpose, we want to avoid building the fact table on the server. This will save the time needed by the web service to respond to the client's request. Also, we can send the data in a more efficient and compact format.

#### 3.1.1 Using cell Table for Storing Cached Sub-Cube

As we explained in the previous chapter, each cell in a sub-cube is identified by an integer called *CellOrdinal*. To minimize the data transmitted over the network, we do not send the fact table. Instead, we send corresponding cellordinal and measure value for each cell. Figure 3.1 shows how fact table is reduced to a smaller table. We call this table *cell table*.

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 3.1: left: A complete fact table with 3 dimensions and a measure. right: Corresponding cell table for this fact table

By using cell table, we avoid building and transmitting the massive fact tables over the network. The size of a cell table is in average equal to 30% of the size of the fact table for the same sub-cube. As we will see later in the evaluation results, this reduction in the size of the data leads to reduction of dataset download time which is strongly desired.

When the cell table is loaded to the browser's memory, the client-side data engine should be able to process the data and perform aggregation on it. Since JSOC designed by Hsiao only works on the fact table, we need to modify his cross-tab algorithm so that the data can be extracted from the cell table. For this purpose, the *CellOrdinals* should be used to generate the stripped down tuple ordinals.

### 3.1.2 Searching Cached Data

In Hsiao's design, the web service is responsible for building the rows of the fact table from the *CellOrdinals*. In our design, the web service sends the cell table that includes the *CellOrdinals* to the client. Here, we perform the same process to extract a tuple ordinal from the *CellOrdinal*, but we don't store the row in the fact table. In fact, we do not build any fact table. The reason is that the browser's memory is limited and we want to use it as efficient as possible.

Remember that the user builds the MDX query by selecting nodes from the TreeView. The client keeps the records of these selected dimensions and hierarchies. So the contents of

each hierarchy is at hand by using the TreeView, i.e the parents and the list of leaf nodes.

The algorithm for decoding the *CellOrdinals* is shown below. In this algorithm, `memberList[i]` contains all the members(leaves) for dimension  $i$ . Then through a *ForLoop*, for each dimension the index of the corresponding member is calculated and the member is retrieved from the current dimension array. In *getMatrixAnswer* algorithm explained in section 2.2, row number `< 11 >` is replaced with a call to the *Decode\_cellordinal* algorithm :

```
11: currentTuple=decode_cellordinal(cellTable[i])
```

---

**Algorithm 2** Decode CellOrdinal

---

**Input:** Current Row = [cellOrdinal, measureValue] in the sub-cube data and membersList that contains member values for all dimensions in the query.

**Output:** Stripped down tuple with members value for current row.

{Parsing CurrentRow to get cellOrdinal and measureValue.}

```
1: cellOrdinal ← currentRow[0]
2: measureValue ← currentRow[1]
3: for  $j = 0 \rightarrow totalDimension$  do
4:   currentDimension ← membersList[j]
5:    $S[j] \leftarrow temp \bmod (currentDimension.Length)$ 
6:    $tuple[j] \leftarrow currentDimension[S[j]]$ 
7:    $temp \leftarrow (temp - S[j]) / currentDimension.Length$ 
8: end for
```

---

In figure 3.2, the process of decoding a CellOrdinal into a stripped down member names is shown: The fact table is scanned for aggregation and for each row in the cell table, the *CellOrdinal* is read and passed to the *Decode CellOrdinal* algorithm. The algorithm computes the tuple and returns it to the function. The rest of the function is the same as *getMatrixAnswer* algorithm explained in chapter 2.

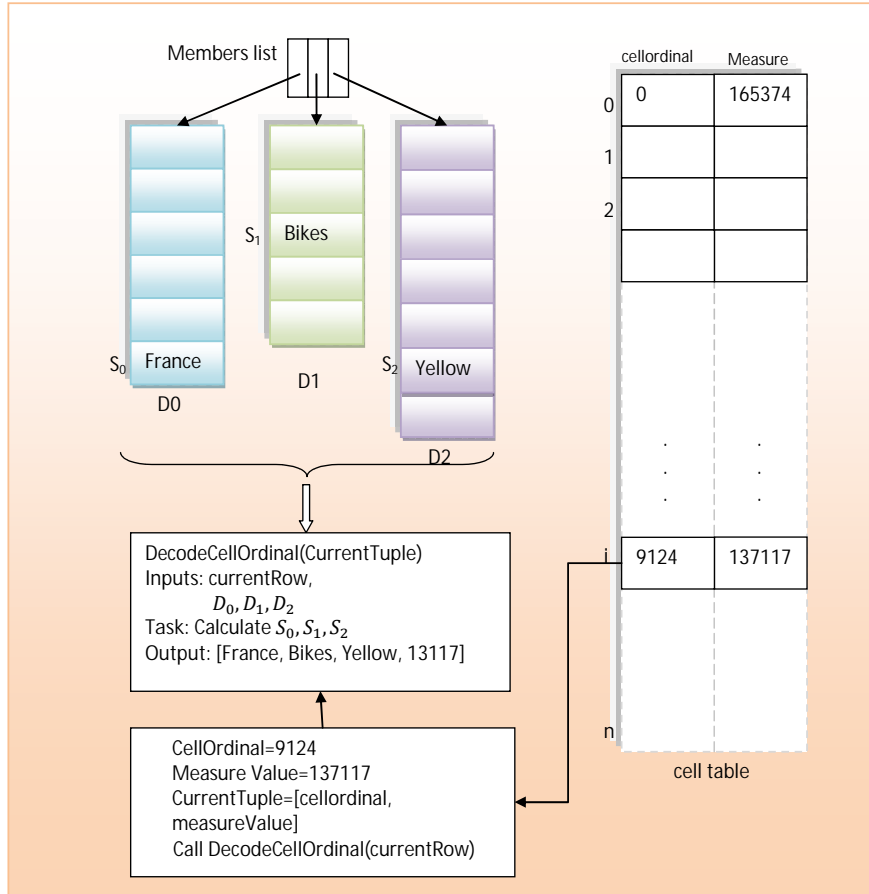


Figure 3.2: Determining the complete tuple for each CellOrdinal.

### 3.2 Performance Comparison with JSOC

Table 3.1 shows the test environment and the client-server specifications we have used for evaluation throughout this work.

	Test Environment
Front-End Machine	Core i5 @1.7GHz 4GB RAM Windows 7 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	Firefox 17.0.1
Dataset	Adventure Works DW

Table 3.1: Test environment

The time that client is waiting for the server's response consists of two parts: server execution time and data transmission time over the network. In section 3.1, we explained how we reduced the size of data transmitted over the network by sending the cell table instead of the fact table. Table 3.2 compares the sizes of fact table and cell table of the same sub-cube. As one can see, the size of cell table is in average %30 of the size of the fact table for a sub-cube.

sub-cube number	Sub-cube Size (rows)	Download Size		Size Ratio
		(Kilo Bytes)		
		fact table	cell table	
1	80	4.1	1.4	%34
2	130	6.5	2.3	%35
3	548	23	2.7	%33
4	1880	117.4	32.8	%27
5	3275	205.2	57.9	%28
6	3760	248.8	65.4	%26
7	12629	800.5	196.7	%24
8	16059	1229.6	250.1	%20
9	22578	1470.2	363	%24
10	38480	2491.9	758.5	%30
11	68056	4049.9	1132.7	%27
12	93624	4714.7	1861.4	%39
13	163095	8239.5	3313.9	%40
14	234248	17543.3	3920.6	%22

Table 3.2: Size of Sub-cube download:(I) fact table is downloaded and when (II) cell table is downloaded

In table 3.3, for each of the sub-cubes in table 3.2, we show the load time, web service time, network transmission time and client processing time. Load time is the total time that client is waiting before receiving a response which is:

$$\text{load time} = \text{server time} + \text{network transmission time}.$$

Server time is the time that web service spends for generating the results. Network transmission time is the total time that data is on the network until it is fully received by the client.

As table 3.3 shows, the cell table approach outperforms the fact table download in both server time and network transmission time. However, as it is expected, the client has to do more processing when scanning the cell table. In table 3.3, the processing time for each approach is shown. Although the processing time for cell table approach is larger than the fact table, the difference is still much smaller compared to the difference of the load time.

For the sub-cubes shown in the table, the average difference between the load time of fact tables and cell tables is +2084 milliseconds. However, the average difference between



the processing time of the cell tables and the fact tables is +83 millisecond. So we can conclude that downloading the sub-cube in the form of a cell table will increase the client-side processing time to some extent, but the time saved in server-side processing and network transmission in this approach is quite large so that we can say the cell table approach is faster. In addition, our approach is more efficient in the terms of client's memory since we store cell table instead of fact table.

sub-cube number	Load Time (ms)		Server Time (ms)		Network Transmission Time (ms)		Processing Time (ms)	
	fact table	cell table	fact table	cell table	fact table	cell table	fact table	cell table
1	214	81	140	51	74	30	5	5
2	220	93	130	56	90	37	2	4
3	1253	150	1126	81	127	69	3	5
4	2485	343	2213	307	272	36	4	6
5	4202	251	3442	156	760	95	6	17
6	5769	524	5372	453	397	71	7	19
7	15548	691	14986	499	562	192	12	43
8	28751	1779	26699	834	2052	945	20	51
9	35723	1655	28793	802	6930	853	25	67
10	50344	2887	44511	2107	5833	780	65	138
11	60704	2710	51525	2181	9179	529	112	138
12	46401	3820	40737	2962	5664	858	120	251
13	88833	12461	69908	4977	18925	7484	202	447
14	527124	13494	512674	10377	14450	3117	307	823

Table 3.3: Sub-cube download information when:(I) fact table is downloaded and when (II) cell table is downloaded

### 3.3 Using Cache for Answering Further Sub-Cube Requests

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. In other words, the sub-cube user is waiting for and the already cached sub-cube have an overlap. The client data engine can manage to use this part locally, and submit necessary queries to retrieve the rest of the data which are not already client-resident.

In the most basic scenario, when the user requests a repeated sub-cube which is retrieved once from the OLAP server, we can escape the phase of executing the query on the OLAP

server again if we have already stored the sub-cubes in the memory. Then the engine will first look into the cached sub-cube for the answer. If it is not one of the previously cached sub-cubes, then a query is sent to the web service retrieve it from the OLAP server.

Note that cached sub-cube can be stored at either the server-side or the client-side. In JSOC, server-side caching is implemented while we investigate the possibilities and advantages of client-side caching.

In the following sections, first we briefly explain the server-side caching implemented by Hsiao and the changes in the data flow described in section 2.3. Then we present our client-side caching approach and the corresponding data flow.

### 3.3.1 Server-side Caching Implemented in JSOC

Hsiao has implemented sub-cube caching in the web-service layer. Remember the data flow we explained in section 2.3: we explained that the web service submits the MDX query to the OLAP server and transmits the results to the client. For implementing server-side caching, this part of the data flow will change. When the JavaScript OLAP Client submits a request to the web service to fetch a sub-cube, the web service generates a unique tag for the sub-cube based on a few properties : the number of the axes in the dataset, the hierarchies specified for each dimension, the depth of the hierarchy and the selected measure. All these properties can be derived from the MDX query. When the web service has the MDX query, it will use this information to generate a file name that uniquely determines this cube. Then the web service searches through the cached files to see if this sub-cube is already cached. If the sub-cube has been previously cached, it is returned to the client. Otherwise, the web service executes the query on the OLAP server to get a new sub-cube. When the OLAP server returns the result set, web service sends the transformed sub-cube fact table to the client and also saves a copy of the fact table on the disk. The name of the file is generated with a hash function using the properties mentioned above.

Note that in Hsiao's approach, the cached sub-cube fact table has to be an exact match of the client request. Otherwise the web service does not use the cache file. Hsiao mentions that he had decided to implement server-side caching over client-side caching due to the fact that client memory is limited. The cached cube should be kept in the browser's memory which has a limited capacity of three to ten megabytes per domain [10].

### 3.3.2 Using Client-Side Caching for Answering Further Sub-Cube Requests

In our system, we have decided to implement client-side sub-cube caching. In fact, when the client receives a sub-cube and stores it in local memory, there is no reason for not using this dataset if it can answer further requests. Not only we are able to answer the query if the same sub-cube is requested, but also any sub-cube that is a subset of the cached sub-cube can be retrieved locally without communicating with the server.

For developing client-side caching, the data flow (section 2.3) needs some modifications. When the user submits the MDX query from the UI, the client data engine will first consider the cached sub-cubes in the browser's memory. When comparing the two sub-cubes (cached sub-cube and the requested sub-cube) three cases may occur:

- The cached sub-cube and the client's request are completely separate. In this case, the data engine will submit an MDX query to the server to retrieve the client's requested sub-cube.
- The cached sub-cube is a superset of the client's request. This means that the sub-cube requested by the user is totally inside the cached sub-cube. Therefore, the data engine can perform aggregation from the cached sub-cube and no query is sent to the server.

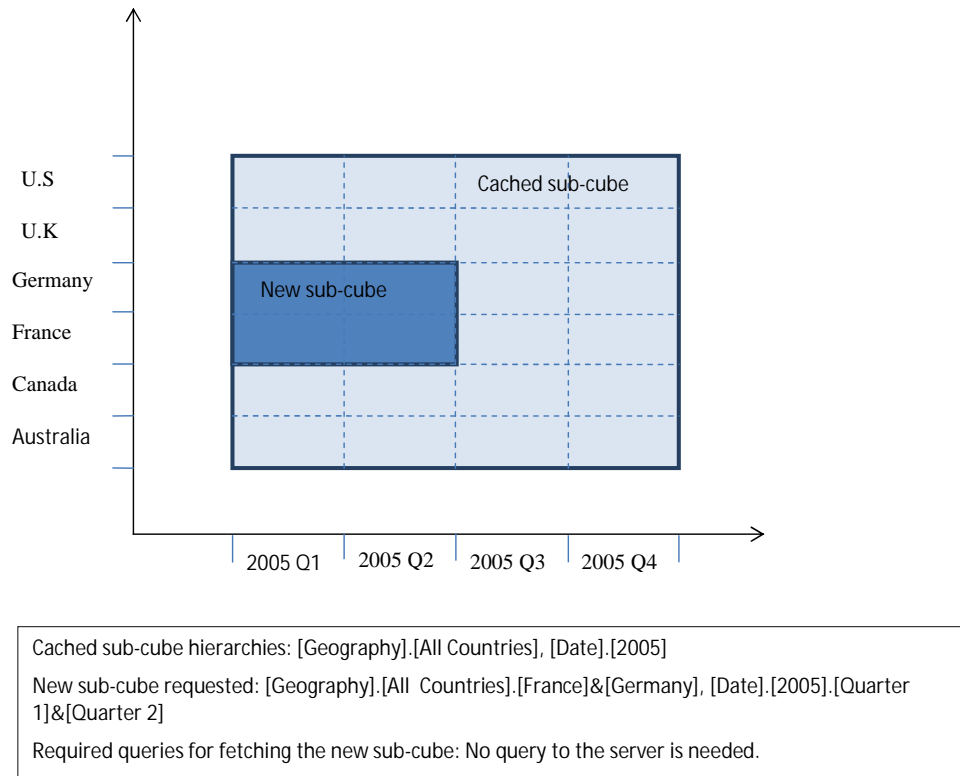


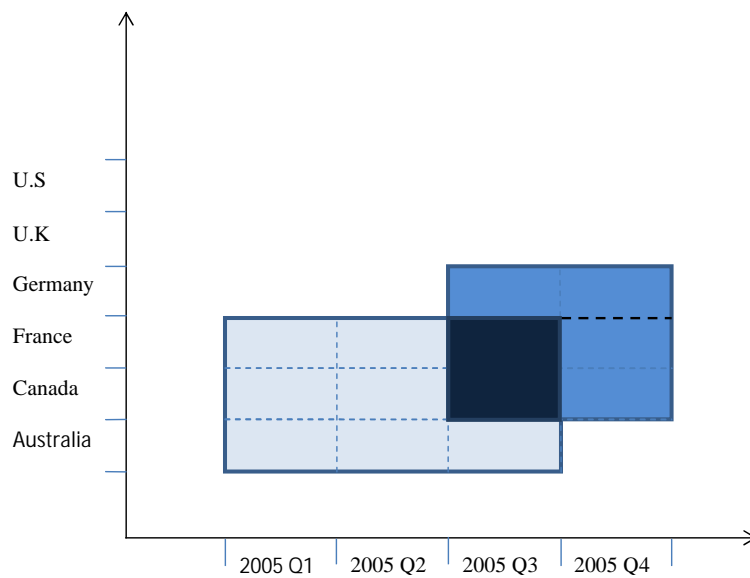
Figure 3.3: Cached sub-cube is a superset of the client's request

For example, suppose that the client has downloaded a sub-cube with the *sale amount* measure for two hierarchies: [All Countries] and [Date 2005] and stored the result set in the local machine. If later, the client has a query on *sale amount* of [All Countries].[France]&[Germany] and [Date 2005].[Quarter 1]&[Quarter 2], the answer exists in the client's memory.

- The third case is when the requested sub-cube has an overlap with the cached sub-cube. In this case, the data engine will manage to use the part of the sub-cube that is already cached and send proper queries to retrieve the rest of the sub-cube.

For example, the data engine has cached a 2-dimensional sub-cube: one dimension consists of [All Countries].[Australia]&[Canada]&[France], the other dimension is [Date].[2005].[Quarter 1]&[Quarter 2]&[Quarter 3] and the measure is [Internet Sale Amount]. Suppose that later the client request a query on the same measure for

[All Countries].[Canda]&[France]&[Germany] and [Date].[2005].[Quarter 3]&[Quarter 4]. These two sub-cube are not the same. Also, the cached sub-cube is not a superset of the requested sub-cube. However, part of the answer exists in the cached sub-cube. I.e., the intersection of the two sub-cubes can be retrieved from the local cached sub-cube and the rest should be retrieved from the server. In this example, a sub-cube consisting of [All Countries].[France] and [Date].[2005].[Quarter 3] can be retrieved from cache. The rest of the data can be retrieved with two separate queries: one query for [All Countries].[France], [Date].[2005].[Quarter 4] and another query for [All Countries].[Germany], [Date].[2005].[Quarter 3]&[Quarter 4].



Cached sub-cube: [Geography].[All Countries].[Australia]&[Canada]&[France],  
 [Date].[2005].[Q1]&[Q2]&[Q3]  
 New sub-cube requested: [Geography].[All Countries].[Canada]&[France]&[Germany],  
 [Date].[2005].[Q3]&[Q4]  
 Required queries for fetching the new sub-cube:  
 1. [Geography].[All Countries].[Canada]&[France], [Date].[2005].[Q4]  
 2. [Geography].[All Countries].[Germany], [Date].[2005].[Q3]&[Q4]

Figure 3.4: 2 MDX queries is needed to fetch the rest of the client's request.

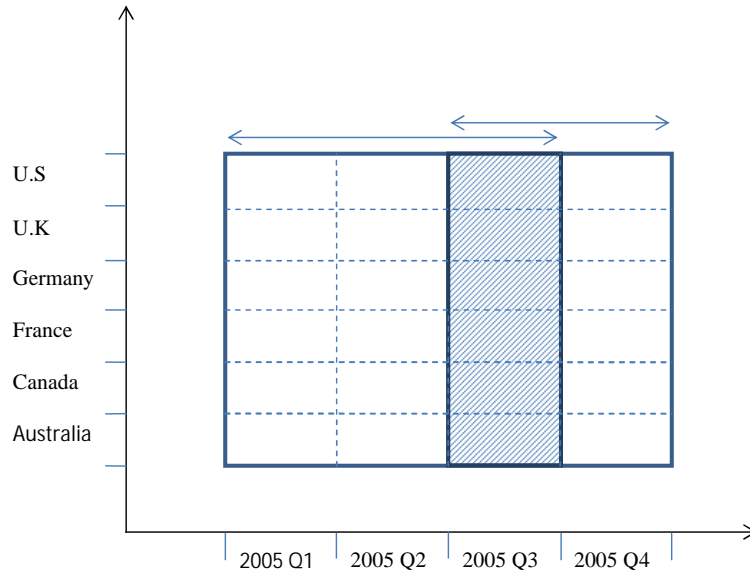
Since case one and two are straight forward, our attention in this section is mainly in the last case. Remember that user builds the MDX query by selecting dimensions and hierarchy nodes from the TreeView. The data engine has every information about the sub-cube except the values for the measure. So with comparing the selected hierarchies with the hierarchies of the cached sub-cube, the data engine can understand whether the two sub-cubes have an intersection. If yes, the data engine needs to figure out which hierarchies does not exist in the cache and generate appropriate queries to retrieve them from the server.

Although it is possible to cache multiple sub-cubes in the browser's memory, in this work we focus on methods that a cached sub-cube could be used. In other words, we only consider the situation that we have a cached sub-cube that has an intersection with the client's request. However, the work is extendable to the case of multiple sub-cubes.

### 3.3.3 Proposed Algorithm for Using the Client-Side Cached Sub-Cube

Depending on the shape of the intersection of the two sub-cubes, the query engine will have to generate different number of MDX queries to build the requested sub-cube. Figures 3.3, 3.4 and 3.5 show three different situations that may occur between two 2-dimensional sub-cubes. In each figure, the hierarchies of the MDX queries are written in the box below the graph. As you can see, for 2-dimensional sub-cubes we can imagine three kinds of intersection which may lead to 0, 1, or 2 MDX queries:

- No query is needed: The cached sub-cube may fully contain the answer for the whole new sub-cube. In this situation no query is generated and the data engine will use the data from the cached sub-cube.
- one query is needed: The cached sub-cube may have an intersection with the new sub-cube. The rest of the data may be retrievable with only one query. The example of this case is shown in figure 3.5.
- two queries are needed: The data engine may have to retrieve the rest of the data from the server by submitting two separate queries. The example for this case is shown in figure 3.4.



<p>                 Cached sub-cube: [Geography].[All Countries], [Date].[2005].[Q1]&amp;[Q2]&amp;[Q3]                  New sub-cube requested: [Geography].[All Countries], [Date].[2005].[Q3]&amp;[Q4]                  Requires queries for fetching the sub-cube:                  [Geography].[All Countries], [Date].[2005].[Q4]             </p>
---

Figure 3.5: 1 MDX query is needed to fetch the rest of the client’s request.

In general, for a  $d$ -dimensional sub-cube, the data engine may have to submit 0 to  $d$  sub-cubes to fetch the dataset. In the following we will explain how the data engine determines the number of queries and the hierarchies for each of them.

Suppose we have cached a  $d$ -dimensional sub-cube in the client’s memory. By caching the sub-cube, we mean caching the list of its hierarchies and the corresponding cell table. Later, client submits a new request that is within the same  $d$  dimensions. Before sending the new request to the server, we compare the hierarchy members of the client’s request with the cached sub-cube. For each dimension  $i$ , we separate hierarchies in 2 parts:  $a_i$  is the part of the dimension that is in common with the cached sub-cube,  $b_i$  is the rest of the hierarchy members. Figure 3.6 illustrates these definitions for a 2-dimensional sub-cube. The intersection is a sub-cube itself consisting of dimensions  $a_0$  and  $a_1$ . In general, the intersection of two  $d$ -dimensional sub-cube is a sub-cube consisting of dimensions  $a_0, a_1, \dots, a_{d-1}$  where  $d$

is the number of dimensions.

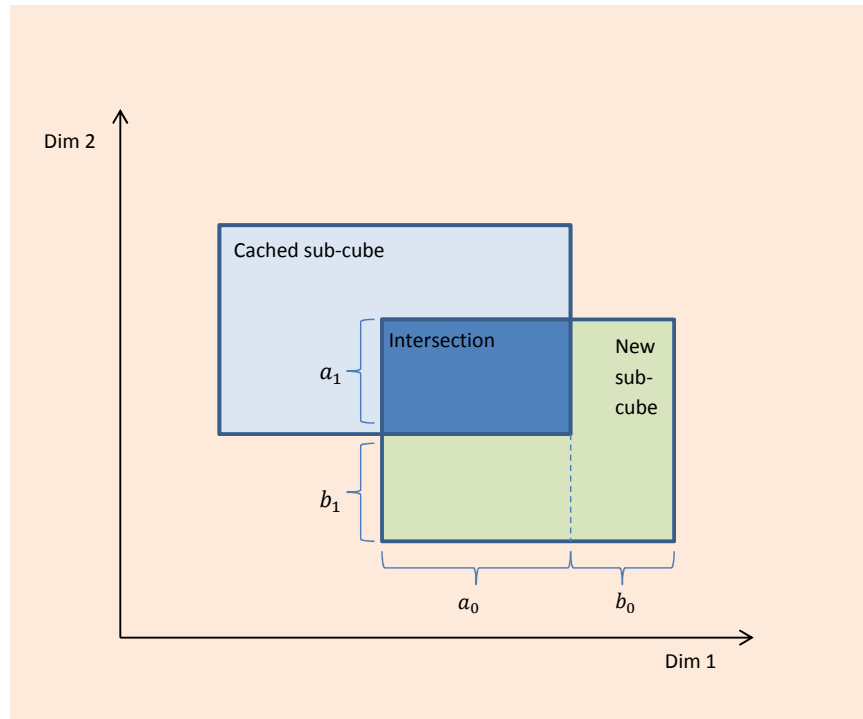


Figure 3.6: Cached sub-cube, new client’s request and their intersection. The intersection is an  $n$ -dimensional sub-cube named  $a_0 \times a_1 \times \dots$ .

Our goal is to use the part  $a_0 \times a_1$  that is already cached and to submit a request to download the rest of the new sub-cube. *Retrieve From Cache* algorithm specifies the queries for this purpose which runs in  $O(n^2)$  where  $n$  is the number of total dimensions. Notice that when we omit the intersection part from the query, we may need more than one query to retrieve the rest of the data. The number of queries is determined as follows: Each dimension  $i$  is separated into two parts  $a_i$  and  $b_i$  as defined earlier. There are  $2^d$  sub-cubes such as  $c_0 \times c_1 \times \dots \times c_{d-1}$  where  $c_i = a_i$  or  $b_i$ . We need to retrieve all these sub-cubes except one with  $c_i = a_i \mid i = 0, \dots, d - 1$ . This sub-cube is already cached. However, we can reduce the number of queries with some simplifications. For instance, instead of generating two queries for  $b_0 \times a_1$  and  $b_0 \times b_1$  in figure 3.6, we build one query such as  $b_0 \times (a_1 + b_1)$ .



Therefore, we will have maximum number of  $d$  queries where  $d$  is the number of dimensions. The format of hierarchies for  $j$ -th query, where  $0 \leq j \leq d - 1$ , would be:

$$subcube_j = c_0 \times c_1 \times \dots \times c_i \times \dots \times c_{d-1} \text{ where } 0 \leq i \leq d-1 \text{ and } c_i = \begin{cases} a_i & i < j \\ b_i & i = j \\ (a_i + b_i) & i > j \end{cases} \quad (3.1)$$

As an example, consider a 3-dimensional sub-cube which has an overlap in all three dimensions with a cached sub-cube. This means that  $b_i$  is a non-empty set for  $i = 0, 1, 2$ . The result of applying the algorithm on the sub-cube will generate three lists as follows:

- $[b_0, \quad a_1 + b_1, \quad a_2 + b_2]$
- $[a_0, \quad b_1, \quad a_2 + b_2]$
- $[a_0, \quad a_1, \quad b_2 \quad ]$

Each list, has hierarchies for a 3-dimensional MDX query. For example, the first list means that the MDX query will have hierarchies in  $b_0$  as the first dimension, hierarchies in  $a_1 + b_1$  as second dimension and hierarchies in  $a_2 + b_2$  as the third dimensions. Another function will generate the appropriate structure for the MDX query from this list.

---

**Algorithm 3** Retrieve From Cache

---

**Input:** An array containing cached cube dimensions and an array containing client's request dimensions

**Output:** queries: list of queries to be sent to the web service.  $\{a[i]$  is the intersection of  $\text{cachedCubeDimension}[i]$  and  $\text{newCubeDimension}[i]\}$   $\{b[i]$  is  $\text{newCubeDimension}[i] - a[i]\}$

```

1: if  $a[i] == 0$  for all  $0 \leq i < \text{totalDimension}$  then
2:   RETURN
3: end if
4: for  $i=0 \rightarrow \text{totalDimension}$  do
5:   list=[]
6:   for  $j=0 \rightarrow \text{totalDimension}$  do
7:     if  $j < i$  then
8:       list.push(a[j])
9:     else
10:      if  $j == i$  then
11:        if  $b[j].\text{length} > 0$  then
12:          list.push(b[j])
13:        end if
14:      else
15:        list.push(concat(a[j], b[j]))
16:      end if
17:    end if
18:  end for
19:  if  $\text{list.length} == \text{totalDimension}$  then
20:    queries.push(list)
21:  end if
22: end for

```

---

The client query engine will send a request containing all MDX queries to the server instead of sending each query separately. The web service will create a connection to the server and execute the queries one by one. Then the web service concatenates the response in an array and returns it to the client.

### 3.4 Evaluation for Using Client-Side Cached Sub-Cubes

Re-using the cached sub-cube at the client-side will obviously decrease the communication with the server and the client has to download less data if any part of the data is already cached. In this section, we want to compare the download time and size when the client-side caching is used with the case that caching is not used.

The complete results of caching experiments for 2-dimensional and 3-dimensional sub-cubes are shown in appendix B. Each group of three rows which are separated with a double solid line are related to one case of testing. A sample of the test case is shown in table 3.4 which is one of the test cases in appendix B. Table 3.4 shows the tree dimensions for the cached sub-cube and for the requested sub-cube. As you can see, the first dimension is the same, and the second and the third dimensions have intersections. %34.6 of the requested sub-cube exists in the cached sub-cube.

sub-cube number	Dimension 1	Dimension 2	Dimension 3	percentage cached
cached sub-cube				
1	Customer.Education: Grad, High sch, partial C, partial H	Date: 2007.H1	Countries:Ger, U.K, U.S	
requested sub-cube				
2	Customer.Education: Grad, High sch, partial C, partial H	Date: 2007.H1, H2	Countries:Aust,ca,Fr, Ger, U.K, U.S	%34.6

Table 3.4: A sample test case for client-side caching

The size of the sub-cubes and timing for this sample test case is shown in table 3.5. When the data engine uses the cached part, it needs to send two MDX queries to the server to retrieve the rest of the data. However, when the cached part is not used, the whole data is returned with one MDX query. In this specific example the server time, network transmission time and aggregation time are all lower when cache is used.

sub-cube number	# of queries	# of rows	size( kilo byte)	server time	network transmission time	Total Load Time	aggregation time
cached sub-cube							
1	1	278588	4582.9	12395	6887	19282	983
requested sub-cube (using the cached part)							
2	2	525752	8715.3	25446	17276	42722	1755
requested sub-cube(cache is not used)							
2	1	804340	13457.5	34666	21749	56415	2892

Table 3.5: download information for the sample test case in table 3.4

In table 3.6 we summarized the tables in appendix B for easier comparison. In all the experiments, three parameters are measured: server time, network transmission time and aggregation time (“download time” in the summary table is the sum of server time and network transmission time). It is important to compare the two scenarios (client-side caching and no cache) for these three parameters. Essentially, using the partial cached sub-cube leads to multiple sub-cube download. We are interested to figure out how this fact will affect each of the three timing parameters. We can summarize the results for each parameter as follows:

- **Server time:** which is the total time spent on the server-side for generating and preparing the dataset for the client. The results show that when one query is used to retrieve all the not-cached-part of the client’s request, server time is less than the no-cache scenario. However, when multiple queries are submitted to the server, the server performs better or equal to the no-cache scenario if the percentage of the data cached is greater than 10%.

This result seems rational. When a large percentage of the requested sub-cube is cached, less data is needed from the server. Therefore server spends less time on collecting the results from the database. But when this percentage is low, the two scenarios need to retrieve approximately the same amount of data from the server, plus that in the case of multiple queries, there is an overhead for running the queries on the server. Therefore, when a low percentage of the sub-cube is cached ( $< 10\%$ ), and multiple queries are needed to retrieve the data, the server time may be greater than downloading the whole sub-cube without using cache.

	# of rows	Download Time (ms)		Processing Time (ms)		Percentage Cached
		with cache	without cache	with cache	without cache	
2-dimensional sub-cube						
1	159165	10352	10041	450	752	%0.5
2	151143	6752	8861	1260	1269	%6.8
3	33630	3567	3746	180	328	%10.1
4	98644	4532	7829	224	380	%34.7
5	23797	6117	9827	67	140	%53.9
6	163095	880	11169	54	612	%93
3-dimensional sub-cube						
7	846390	52040	60673	2850	2815	%0.62
8	859070	60857	55615	3053	2990	%0.67
9	297370	17566	18546	917	1059	%10.63
10	804340	42722	56415	1755	2892	%34.63
11	575604	14944	25944	1813	1848	%46.85
12	476840	7328	28078	1616	1707	%72.82
13	450640	3113	28615	114	1636	%93.89
4-dimensional sub-cube						
14	362068	29066	28316	1356	1390	%1.2
15	259064	16241	15931	924	986	%1.8
16	33600	970	2257	119	109	%47.5

Table 3.6: Sub-cube download with/without using cache

- **Processing time:** is the time spent by the client-side data engine to aggregate the result from the dataset. The results for processing time shown in table 3.6. If we compare the two columns under the “processing time” header (with cache and without cache), we see that for the cases that cached part is greater than 10 percent, the caching outperforms the no-cache approach. Again, we can argue that for the lower percentages of cache, the overhead of dealing with multiple sub-cubes may be larger than the advantage of using cache.
- **Network transmission time:** In our results, this parameter is always better when cache is used expect for one or two of the test cases. It is clear that when less data is transmitted, less time is needed. For the cases that using cache is not better than

the no-cache method, the percentage of cache is low. Therefore, the two methods transmit “about” the same amount of data and either approaches may become the winner. However, the difference is not very large in these test cases.

To conclude, we can say that for the three parameters discussed above, the benefit of using client-side cache outperforms its overhead specially when a large portion of the sub-cube exists in the cache. Therefore, when the user requests a new sub-cube which is partly cached, it is beneficial to use the cached part from the local memory and send modified queries to retrieve the additional data to complete the answer for the requested sub-cube. In our experiment, the benefit is recognized when the cached percentage of the sub-cube is as low as 10%.

## Chapter 4

# User Workflow

In the previous chapter, we explained in details the process of downloading the multidimensional data and client-side caching. As we stated earlier, we download the sub-cube when the user knows in advance that the next couple of queries are from this sub-cube. The user is going to explore the sub-cube by requesting different cross-tab and drill-down queries. Therefore, it worth to download the inflated answer sub-cube rather than generating the exact response for the client's request.

In this chapter, we focus on the workflow, i.e., a sequence of queries for cross-tab, drill-down and roll-up for two cases :

- (I) when for each query, only the *answer sub-cube* is downloaded from the server and,
- (II) when the inflated version of the *answer sub-cube* is downloaded.

We described in chapter 1 these two terminologies. As a reminder, the *answer sub-cube* is the data sent to the client when the client requests only the exact answer to the query. And an inflated version of the *answer sub-cube* is a sub-cube where each dimension of the answer sub-cube includes extra nodes, which are child nodes of nodes in the answer sub-cube. The challenge in this chapter is to compare cases (I) and (II) in a workflow scenario.

Downloading the inflated answer sub-cube decreases the communication time and enables the client to work with the local dataset. However, the processing time is affected since the local engine should process the large sub-cube. In this chapter, we present our algorithm for searching the sub-cube. The new algorithm for data aggregation is much better than the *getMatrixAnswer* algorithm discussed in chapter 3.

In the following, first we explain the workflow scenario through an example. Then we discuss the trade-off between downloading the *answer sub-cube* and the inflated sub-cube. In section 4.1.3, we present the algorithm we used for drill-down operation which leads to a lower processing time. And finally, we evaluate the workflow for the two approaches (answer sub-cube and inflated answer sub-cube) and we present the results for our workflow experiment.

## 4.1 Workflow Scenario

A workflow is a sequence of queries requested by the user. In each case, the client-side data engine has two options. It can download only the aggregated data that answers the current query or download a sub-cube containing the measure value for the lowest level of aggregation. Table 4.1 shows a sample sub-cube from the workflow.

In this table, the workflow sequence and the time for each version is shown. For inflated version of answer sub-cube (which we we call version *II*), there is a one-time cost for downloading the sub-cube. The time for downloading the sub-cube is in millisecond and the size of the sub-cube  $s$  in kilobytes. For the answer sub-cube (which we will call version *I*), we do not have such a cost since we only download the data needed for each query when the query is submitted.

In this specific example, the client submits four queries one by one: a cross-tab and three drill-downs. Because of the design of the UI, a cross-tab on the sub-cube is necessary before one can perform any drill-down. In the answer sub-cube approach, for each of these queries, the client data engine will submit a query to the server. Server will generate the result and return it to the client. In this approach, since the amount of data returned for each query is low, the processing time is negligible. Therefore we only show the load time for each query in the table.

For the other approach, downloading the inflated version sub-cube with all leaf nodes, no data is downloaded from the server for further queries and all the queries are answered locally from the sub-cube downloaded at the beginning. However, since this sub-cube is large, the processing time becomes a matter. Therefore, for this version we do not have any load time for cross-tab and drill-down queries, but we have processing time for them.



workflow	Inflated sub-cube	answer sub-cube
	script time	load time (size)
1. Download [Customer].[Education], [Date].[ 2008]	Load (size)	
	288 (15)	
2. Generate cross-tab on [Customer].[Education], [Date].[ 2008]	8	102
3. Drill down on [Date].[2008].[H2]	2	157
4. Drill down on [Date].[2008].[H2].[Q3]	2	117
5. Drill down on [Date].[ 2008].[H2].[ Q3].[July]	3	144

Table 4.1: One sample of the workflow sequence

#### 4.1.1 Trade-off between Downloading Answer Sub-cube and Inflated Version Sub-cube

We already mentioned that it is difficult to choose between downloading the answer sub-cube and the inflated version of it, because it depends on non-technical and non-quantitative factors. For example, one needs to know to what extent the user will perform drill-down on the inflated answer sub-cube. In the example shown in table 4.1, it seems clear that even with considering the download time of the inflated sub-cube, it is better to download the inflated version of the sub-cube because the average response time for the inflated sub-cube processing is much lower than the *answer sub-cube* load time.

In general we can see that there is a trade-off between the two approaches: When the inflated sub-cube is downloaded, we avoid the cost of network transmission time and server time, instead we have to pay the one time cost for downloading the large sub-cube. If the user is willing to apply several drill-downs on the local sub-cube, the cost of sub-cube downloading is compensated. In addition, the server time is saved to serve another user or even the same user for other queries.

In the other hand, if the inflated sub-cube is not really used for further drill-downs, then there is no reason to download the large dataset since we could download only the answer to the current query. So the more the sub-cube is used for further queries, it is more beneficial to download the inflated sub-cube.

Note that the processing time for drill-downs increases as the size of the inflated sub-cube goes up. To decrease the processing time (scripting time), we came up with a new algorithm for data aggregation. In the next part, we explain briefly how the data for cross-tab and drill-downs are generated in the current algorithm, and we present the new algorithm for

this purpose.

### 4.1.2 Data Aggregation for Drill-down Request

The basic operations in an OLAP system are cross-tab and drill-down/roll-up. Here we briefly describe how these operations are addressed in the *getMatrixAnswer* algorithm. Then we describe our new implementation.

#### Full Sub-cube Scan for a Drill-down Operation

In section 2.2 we explained the *getMatrixAnswer* algorithm used in JSOC for performing cross-tab and drill-down. Then in section 3.1 we explained how we adopted this algorithm to process a cell table instead of a fact table. The key point about using this algorithm for both cross-tab and drill-down is that, for each of these queries, the algorithm scans the whole sub-cube and for each row of the sub-cube, it will determine if this row is part of the answer or not.

For example, suppose that the user chooses to download a 2-dimensional sub-cube: [Date]. [Calendar].[All Periods].[CY 2005] and [Geography]. [All Geography].[Canada] which has 13213 rows. The user then requests a cross-tab on these two dimensions: the row node is [All Geography]. [Canada] and the column node is [All Periods].[CY 2005]. A full sub-cube scan is performed to generate the matrix for cross-tab table. The result tables are shown in figure 4.1.

Suppose that later, the user chooses to perform a drill-down on [All Geography].[Alberta]. The query engine, handles the drill-down requests as a subquery. The *getMatrixAnswer* function is called with [All Geography].[Canada].[Alberta] as the row node and [Date]. [Calendar]. [All Periods]. [CY 2005] as the column node. However, the full sub-cube is scanned again: for each row, if the row member is a descendant of the row node ([All Geography]. [Canada]. [Alberta]) and column member is a descendant of the column node ([Date]. [Calendar]. [All Periods]. [CY 2005]), the row will be computed as a part of the drill down table. Therefore, for each cross-tab or drill-down, the algorithms iterates through the loop on the whole dataset downloaded from the web service.

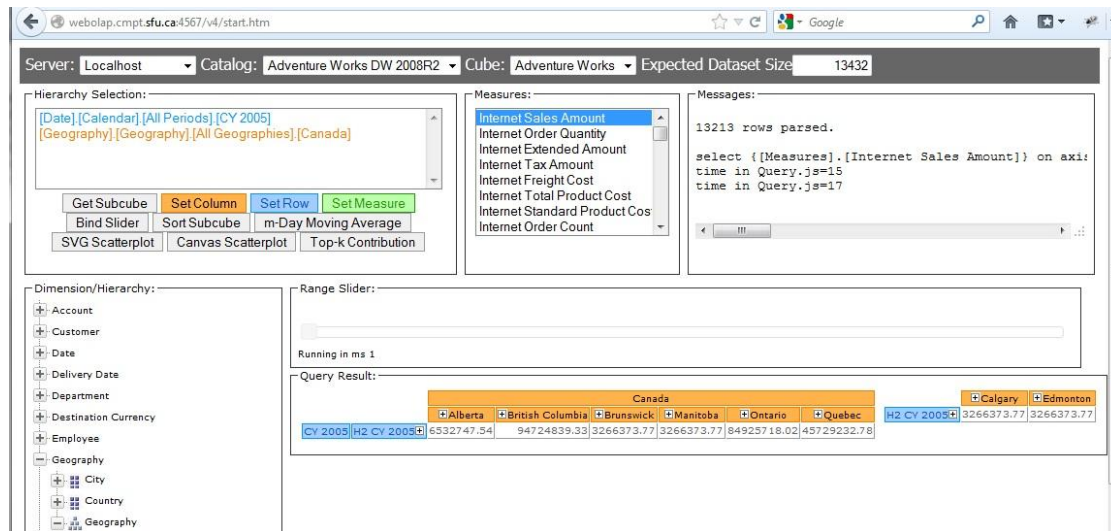


Figure 4.1: An screen shot of the JavaScript OLAP Client.

### Selective Part Sub-cube Scan for a Drill-down Operation

A better solution for performing the drill-down operation is scanning the part of the dataset that is intended for a drill-down instead of scanning the whole dataset.

In the new algorithm, instead of scanning the cell table for a drill-down, we start from the *Members List* containing the child nodes for the hierarchies. The process of the work is shown in figure 4.2. The idea is for each child node in the *Members List*, check to see if it is part of the requested drill-down and then compute the cell ordinal only if both row child and column child are part of the drill-down. Then the algorithm looks up for the generated cell ordinal in the cell table and returns the measure value for aggregation. With this approach, we avoid the cost of calculating cell ordinal for unused cells.

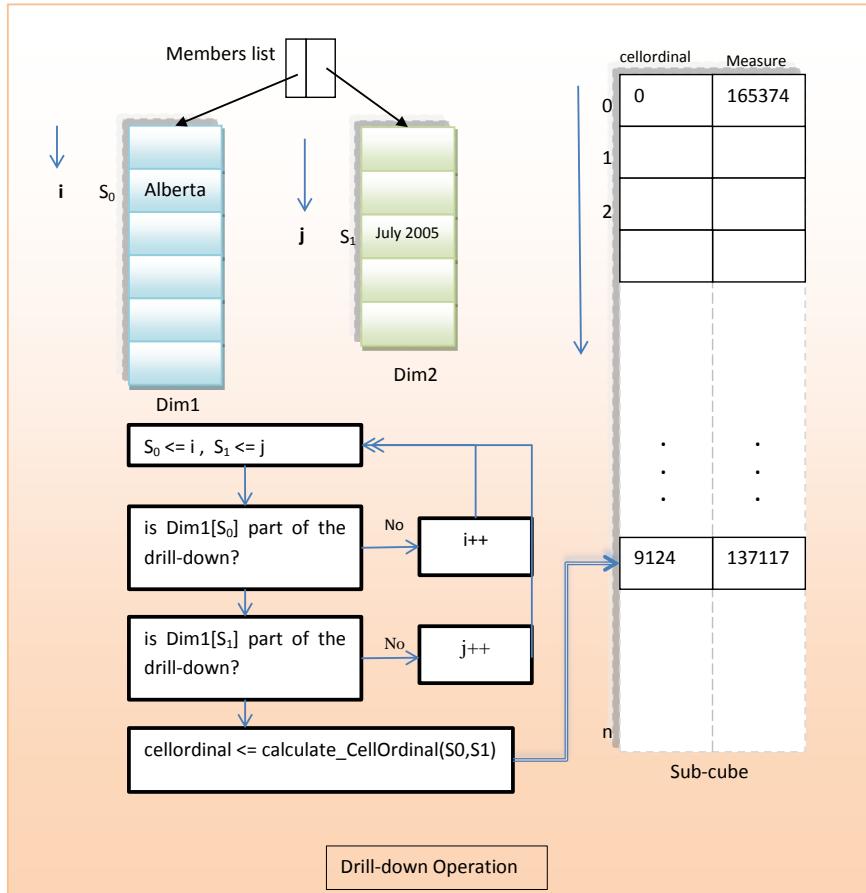


Figure 4.2: The algorithm for drill-down process

### 4.1.3 Proposed Algorithm for Drill-down Operation

The algorithm needed for this purpose is shown below. `columnNode` and `rowNode` are the nodes selected as dimensions when the sub-subcube was downloaded. `columnMemberNode` and `rowMemberNode` are the nodes selected for drill-down. When a drill-down is performed, first for each leaf node in `columnAxis`, the algorithm will check whether this node is part of the drill-down or not. If yes, then the loop will continue and check whether the row leaf is a member of drill-down too. Then the cellordinal is calculated for these two tuple ordinal.

The function “`isAncestor`” takes two tree nodes as input and determines if the first one is an ancestor of the second node. It will take  $O(h)$  to determine the ancestor where  $h$  is the height of the latter node.

---

**Algorithm 4** Modified-GetMatrixAnswer

---

**Input:** `columnNode`, `rowNode`

---

**Output:** `cellordinal`

```

1: for  $i = 0 \rightarrow \text{columnAxis.Length}$  do
2:   current=[]
3:   S=[]
4:   if isAncestor(columnNode,ColumnMemberNode) then
5:     current.add(columnAxis[i])
6:     S.add(i)
7:     for  $j=0 \rightarrow \text{rowAxis.Length}$  do
8:       if isAncestor(rowNode, rowMemberNode) then
9:         current.add(rowAxis[j])
10:        S.add(j)
11:        cellordinal=calculateCellOrdinal(current, S)
12:        fillMatrixAnswer(cellordinal)
13:      end if
14:    end for
15:  end if
16: end for

```

---

The `cellordinal` calculated in algorithm 4 will be passed to `fillMatrixAnswer` shown in algorithm 5 for calculating the aggregation. The `fillMatrixAnswer` algorithm, receives a

cellordinal as an input and searches the cell table to find this cellordinal. If it is found, it will add the corresponding measure value to the answer. *searchIndex* is the index of the last cellOrdinal found in the cell table. Since the cell table is sorted by the cellordinal, and also the cellrodinals generated in *Modified-getMatrixAnswer* are incremental, finding a cellordinal in the cell table will take  $O(1)$  time.

---

**Algorithm 5** fillMatrixAnswer
 

---

**Input:** cellordinal

**Output:** matrixAnswer

```

1: if cellTable[searchIndex][0]==cellordinal then
2:   measureValue  $\leftarrow$  cellTable[searchIndex][1]
3:   SearchIndex++
4: end if
5: if cellTable[searchIndex][0]> cellordinal then
6:   # The cellordinal does not exists in the cell table
7:   return
8: end if
9: row=rowMemberNode.getAncestor(rowMemberNode.depth+1)
10: col=colMemberNode.getAncestor(colMemberNode.depth+1)
11: if row is a property of answer AND col is a property of answer[row] then
12:   answer[row][col]+ = measureValue
13: end if

```

---

## 4.2 Experiment Results for the Workflow

Table C shows the workflow experiment results. Our goal is to compare the *answer sub-cube* and the inflated version of the sub-cube for the same workflow.

We discuss the results in two part. In part one, we compare the proposed drill-down algorithm with the previous version of the getMatrixAnswer algorithm explained in chapter ???. In the second part, we compare downloading the *answer sub-cube* with downloading the inflated version of it.

### 4.2.1 Comparing the Proposed Aggregation Algorithm vs. the Original Algorithm

We expect that our algorithm performs better than the original *getMatrixAnswer* algorithm in drill-downs since less data is processed. The results meet our expectation. For all drill-downs, our algorithm beats the original algorithm. This is reasonable since less data is processed in the new algorithm.

The graph in figure 4.3 shows a comparison between the new algorithm and the previous version for a sequence of random drill-downs on different sub-cubes. As the graph shows, the new algorithm always performs better than the old one. The drill-down sequence is sorted by the size of the sub-cubes. The graph shows that we have a huge difference between the two methods when the sub-cube is large i.e. the right side of the graph. This is rational since when the sub-cube is large, the old method that scans the whole sub-cube will take a long time to perform. But the new method that only scans the part needed for the drill-down has a very lower peak.

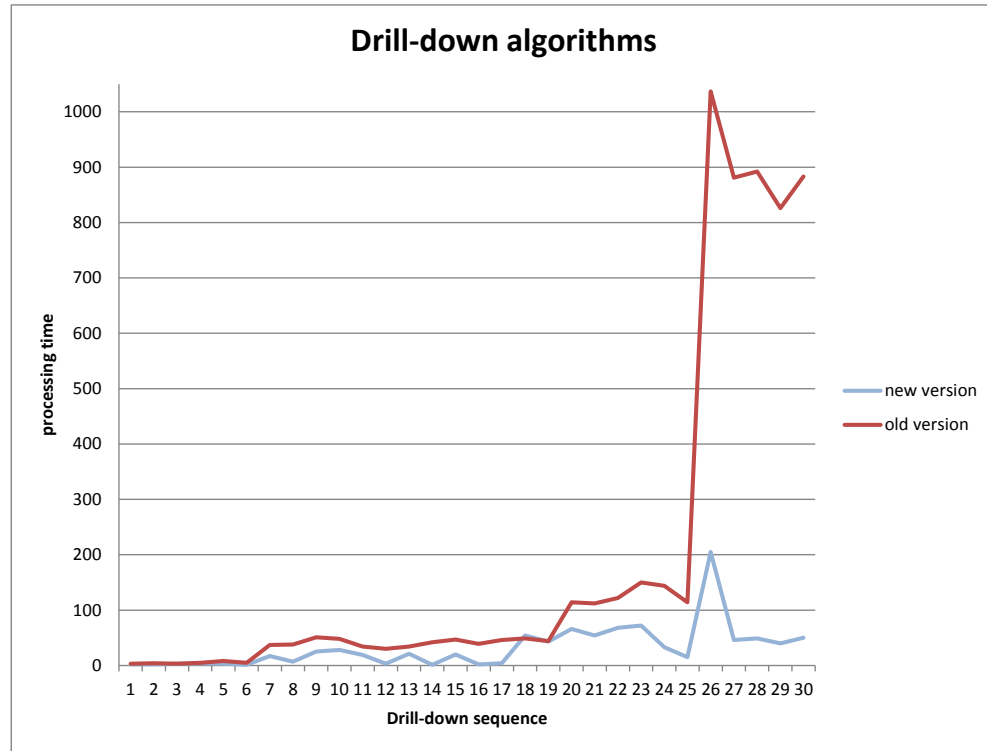


Figure 4.3: Comparing processing time for the new drill-down algorithm with the previous version. The lower is the better.

#### 4.2.2 Answer sub-cube vs. Inflated Sub-cube Comparison

As we addressed the problem in the Introduction chapter, when the user wants to launch a drill-down, the client engine has two options : (I) requesting only the answer sub-cube which usually has a small size, but still there is a communication with the server or (II) the answer is derived by the local engine working entirely on the cache data. This cached data is the inflated sub-cube with the lowest level of aggregation i.e the leaf nodes. This sub-cube has a larger size and therefore it takes longer to download, but instead the user has the benefit of working locally with the data and only pays the processing time.



	# of drill-downs	Inflated answer sub-cube		answer sub-cube
		Load Time (size)	sum of script time	sum of load time
1	4	288 (15)	15	520
2	5	320 (25.9)	17	933
3	6	389 (66.5)	130	528
4	5	519 (162)	82	433
5	6	1401 (274.5)	98	892
6	4	1669 (326.5)	99	597
7	8	1601(421)	105	1067
8	7	3006 (431)	319	1470
9	13	2717 (679)	593	1504

Table 4.2: workflow summary

Table 4.2 summarizes the results of the workflow experiment. The expanded table is attached in appendix C which contains more details about the queries. In table 4.2, for each sub-cube requests, we show the number of further drill-downs on that sub-cube. In the case of inflated answer sub-cube, we show the load time of the sub-cube and the sum of processing time for the consequent queries. However, in the case of answer sub-cube download, we only show the sum of load times since the processing time is very small and negligible. Two factors are important in the workflow scenario : number of up-coming drill-downs and size of the inflated sub-cube.

1. *Number of further drill-downs* : When several number of drill-downs are launched, it is more likely that downloading inflated sub-cube is worthy. For example, consider query number 3 in table 4.2. It takes 389 millisecond to download the inflated sub-cube. For this sub-cube, a couple of further requests are submitted. As you can see, the response time of the inflated sub-cube approach is better than the answer-sub-cube approach. Therefore, the client may prefer to download the inflated answer sub-cube and work locally instead of communicating back and forth with the server.

But for some cases, such as row number 6 in the table, still processing the large cell table is faster than downloading the answer sub-cube from the server. However, there are a few requests for drill-down on this sub-cube. Therefore, it may not worth for the client to pay the download time cost for the inflated sub-cube.

2. *Size of inflated sub-cube* : When the inflated sub-cube is large, the load time may

become too large that is not tolerable for the user. In addition, the processing time increases with the sub-cube size. For example, consider the last sub-cube in the workflow table. Even without considering the load time for the inflated sub-cube, the processing time for a drill-down in this sub-cube is equal or larger than downloading the *answer sub-cube* for it. So it is not beneficial to download the inflated sub-cube for such a large sub-cube.

In conclusion, we can say that for large sub-cube we should be cautious about downloading the inflated sub-cube. In our workflow experiment, for larger sub-cubes, we should hesitate to download the inflated sub-cube since the cost of the load time may not be overcome with even several drill-down requests. However, for smaller sub-cubes the load time of the large sub-cube may be tolerable since it is less than 4 seconds. Instead, the data processed locally with a lower cost, plus that the facts that the task load of the server is reduced and the client is able to work offline.

## Chapter 5

# Conclusion and Future Work

### 5.1 Conclusion

To the best of our knowledge, Hsiao did the first study of client-centric OLAP system [10]. The study is concerned with data pre-fetch as well, but in the context of data visualization. Once the data visualization is over, the data cube is discarded. There is no client-side data cache. We improved the client-centric OLAP system by reducing the size of the data cube downloaded to the client. Also, in our design, the client-side data engine will re-use the cached data cube for answering further queries.

The focus of this study is client-side caching for a client-centric OLAP system. As the first step, we designed a new data structure for transmitting the multidimensional data from the server to client, and storing it to the client's memory. The new data structure is specially beneficial when inflated answer sub-cube is downloaded. Our results show that we save about 70% of the memory compared to JSOC. The new data structure increases the client-side processing time, but the overhead is tolerable since the network transmission time is decreased significantly.

When we download the inflated answer sub-cube, we can use it for processing a new query if the new query has any intersection with the cached sub-cube. We made the data engine capable of comparing the new query with a cached sub-cube. If the two sub-cubes have an overlap, that part of the data is used and the data query engine will generate the MDX queries to retrieve the rest of the data from the server. Our results show that this approach is useful even when more than 80% of the data still needs to be retrieved from the server.

Also in this work we studied the efficiency of downloading the inflated answer sub-cube in expectation of further drill-downs. We discussed that when the client submits a query, we can either download a sub-cube that only answers the current query (which we named *answer sub-cube*, or we can download the inflated version of the answer sub-cube which contains all the measure values for the child nodes of the selected hierarchies in the query. If we choose the latter one, then further drill-downs on this sub-cube is performed locally. The workflow shows that downloading the inflated sub-cube is specially cost-effective for smaller sub-cubes.

## 5.2 Future Work

There are different directions to continue and expand this research. First, when we use client-side caching, like any other caching application, we should think of a system for managing multiple sub-cubes in the client's memory. For example, when the client's memory reaches its limit, there should be a pre-defined algorithm that chooses a sub-cube and excludes it from the memory and includes a new sub-cube. Different strategies could be first-in/first-out, excluding the largest sub-cube, excluding least wanted, or random selection.

Also, it is a good idea to repeat the workflow experiment several times to obtain a better idea of the fact that to what extent the user will explore the current sub-cube.

Another idea is to work on size of the inflated answer sub-cube. In this work, we build the inflated version with the lowest level of the aggregation for each hierarchy in the sub-cube, i.e. the leaf members. However, we may not always need to drill down the sub-cube all the way down to the leaf nodes. Instead, the client data engine can decide on the level of the hierarchies to be fetched. Since we have enough information from the metadata to calculate the size of the sub-cube, we can decide how deep we want to build the inflated answer sub-cube specially for large sub-cube. This way, we can reduce the processing time and still, drill-downs are possible for a number of levels.

## 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/analysiservices/2003/
engine">2011-03-25T21:16:58.49</LastDataUpdate>
      <LastSchemaUpdate xmlns="http://schemas.microsoft.com/analysiservices
/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" />
      </HierarchyInfo>
    </AxisInfo>
  </AxesInfo>
</OlapInfo>
</return>
```

```

        <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 Caching Experiment

dimension 1	dimension 2	# of queries	# of rows size(byte)	server time	network transmission time	Total Load Time	aggregation time	% cached
1 Employee: All Employees	Countries:Ca, Fr, Ger, U.K, U.S	1	3104831	5764	4277	10041	519	
2 Employee: All Employees	Countries:Aust, Ca, Fr, Ger, U.K, U.S	1	188235	393	487	880	54	93.9
3 Employee: All Employees	Countries:Aust, Ca, Fr, Ger, U.K, U.S	1	3313991	5758	5411	11169	612	
1 Employee: All Employees	Countries:Aust	1	188235	461	497	958	80	
2 Employee: All Employees	Countries:Aust, Ca, Ger, U.K, U.S	1	2853839	5397	1355	6752	1269	6.8
3 Employee: All Employees	Countries:Aust, Ca, Ger, U.K, U.S	1	3062999	6571	2290	8861	1260	
1 Employee Department: 1-1,2,3,4	Countries:Aust, Ca, Fr	1	63234	265	1132	1397	42	
2 Employee Department: 1-8	Countries:Aust, Ca, Fr, U.K, U.S	2	583874	1093	2474	3567	180	10.1
3 Employee Department: 1-8	Countries:Aust, Ca, Fr, U.K, U.S	1	661596	1308	2438	3746	328	
1 Employee Department: 5	Countries:Canada	1	13134	84	151	235	11	
2 Employee Department: 1-10	Countries:Aust, Ca, Fr, Ger, U.K, U.S	2	3203398	5527	4825	10352	450	0.5
3 Employee Department: 1-10	Countries:Aust, Ca, Fr, Ger, U.K, U.S	1	3231461	5176	4865	10041	752	
1 Products:Accessories, Bikes, Comp	Date:2005,2006,2007	1	194361	3861	557	4418	61	
2 Product:Accessories, Bikes, Comp, Clothing	Date:2005, 2006, 2007, 2008, 2010	2	155336	5717	400	6117	67	53.9
3 Product:Accessories, Bikes, Comp, Clothing	Date:2005, 2006, 2007, 2008, 2010	1	360174	8910	917	9827	140	
1 Countries:U.S	Date:2008.Q1	1	34216	1755	1000	2755	143	
2 Countries:Fr, Ger, U.K, U.S	Date:2008.Q1, Q2	2	1063975	2410	2122	4532	224	34.7
3 Countries:Fr, Ger, U.K, U.S	Date:2008.Q1,Q2	1	1651852	4536	3293	7829	380	

Table B.1: Evaluating Client-side caching for 2-dimensional sub-cubes. In each group of three rows separated by double line: row 1 shows the cached sub-cube, row 2 shows client's sub-cube requested which has an intersection with the cached sub-cube, row 3 repeats downloading client's sub-cube without using cache.



	Dimension1	dimension2	dimension3	# of queries	# of rows size(byte)	server time	network transmission time	total load time	Aggregation time	Aggregation % cached
1	education	Date: 2007.H1	Countries:ca, Fr, Ger, U.K, U.S	1	423120	19503	11335	30838	1499	
2	Customer.All Education	Date: 2007.H1	Countries:Aust, Ca, Fr, Ger, U.K, U.S	1	27520	1661	1452	3113	114	93.8
3	Customer.All Education	Date: 2007.H1	Countries:Aust, Ca, Fr, Ger, U.K, U.S	1	450640	18200	10415	28615	1636	
1	Customer.Gender	2007	Fr, U.K, U.S	1	347256	14370	2597	16967	1228	
2	Customer.Gender	2007	Aust, Ca, Fr, Ger, U.K, U.S	1	129584	5735	1593	7328	1616	72.8
3	Customer.Gender	2007	Aust, Ca, Fr, Ger, U.K, U.S	1	476840	17760	10318	28078	1707	
1	Customer.Education: Grad, High sch, partial C, partial H	Date: 2007.H1	Countries:Ger, U.K, U.S	1	278588	12395	6887	19282	983	
2	Customer.Education: Grad, High sch, partial C, partial H	Date: 2007.H1, H2	Countries:Aust,ca,Fr, Ger, U.K, U.S	2	525752	25446	17276	42722	1755	34.6
3	Customer.Education: Grad, High sch, partial C, partial H	Date: 2007.H1, H2	Countries:Aust, Ca, Fr, Ger, U.K, U.S	1	804340	34666	21749	56415	2892	

Table B.2: Evaluating Client-side caching for 3-dimensional sub-cubes. In each group of three rows separated by double line: row 1 shows the cached sub-cube, row 2 shows client's sub-cube requested which has an intersection with the cached sub-cube, row 3 repeats downloading client's sub-cube without using cache.

	Dimension1	dimension2	dimension3	# of queries	# of rows size(byte)	server time	network trans- mission time	total load time	Aggregation time	% cached
1	Customer.Education: Grad, high sch	Date: 2007.H1	Countries: Aust, Ca	1	488492	1626	1194	2820	145	
2	Customer.Education: Bachelor, Grad, High sch	Date: 2007.H1	Countries: Aust, Ca, Fr, Ger, U.K, U.S	2	4281656	11613	5953	17566	917	10.6
3	Customer.Education: Bachelor, Grad, High sch	Date: 2007.H1	Countries: Aust, Ca, Fr, Ger, U.K, U.S	1	4910105	11515	7031	18546	1059	
1	Bach, Grad, High sch	h1 2008	Ger, U.K, U.S	1	4589406	9839	3187	13026	826	
2	Bach, Grad, High sch, partial c, partial h	h1, h2 2008	Fr, Ger, U.K, U.S	3	4939869	11909	3035	14944	1813	46.8
3	Bach, Grad, High sch, partial c, partial h	h1, h2 2008	Fr, Ger, U.K, U.S	1	9732351	21771	4173	25944	1848	
1	Customer.Education: High school	Date: 2007.H1	Countries:Australia	1	109115	391	316	707	36	
2	Customer.Education: Bachelor, Grad, High sch, partial c, partial h	Date: 2007.H1, ,H2	Countries:Aust, Ger, U.K, U.S	3	13969254	31958	20082	52040	2850	0.6
3	Customer.Education: Bachelor, Grad, High sch, partial c, partial h	Date: 2007.H1, ,H2	Countries:Aust, Ger, U.K, U.S	1	14239094	40038	20635	60673	2815	
1	Customer.Education: Partial h	Date: 2007.H1	Countries:Canada	1	86223	542	974	1516	32	
2	Customer.Education: Bachelor, Grad, High sch, partial c, partial h	Date: 2007.H1, ,H2	Countries:Aust, Ca, U.K, U.S	3	14208029	35711	25146	60857	3053	0.6
3	Customer.Education: Bachelor, Grad, High sch, partial c, partial h	Date: 2007.H1, ,H2	Countries:Aust, Ca, U.K, U.S	1	14453854	30609	25006	55615	2990	

Table B.3: Continued from table B.2

## Appendix C

# Tables for Workflow Experiment

Table C.1: User workflow

	Inflated version of sub-cube	Answer sub-cube
<b>Workflow</b>	<b>Script time</b>	<b>Load time</b>
1. Download [Customer].[Education], [Date].[2008]	Load time (size) 288 (15)	
2. Generate cross-tab on [Customer].[Education], [Date].[2008]	8	102
3. Drill down on [Date].[2008].[H2]	2	157
4. Drill down on [Date].[2008].[H2].[Q3]	2	117
5. Drill down on [Date].[2008].[H2].[Q3].[July]	3	144
6. Download [Geography].[Australia].[New South Wales], [Date].[2006].[Q1]	Load time (size) 420 (25.9)	
7. Generate cross-tab on [Geography].[Australia].[New South Wales], [Date].[2006].[Q1]	4	144
8. Drill Down on [Geography].[Australia].[New South Wales].[Alexandria]	4	151
9. Roll up on [Geography].[Australia].[New South Wales]	3	151
10. Drill down on [Date].[2006].[Q1].[January]	1	276

*Continued on next page*

Table C.1 – Continued from previous page

Workflow	Script time	Load time
11. Drill down on [Date].[2006].[Q1].[Feb]	5	211
12. Download [Customer].[Education], [Date].[All Periods]	Load time (size) 389 (66.5)	
13. Generate cross tab on [Customer]. [Education], [Date]. [All Periods]	37	133
14. Drill down on [Date]. [2006]	17	65
15. Drill down on [Date]. [2006]. [H2]	7	71
16. Roll up on [Date].[2006]	39	114
17. Drill down on [Date]. [2008]	18	76
18. Drill down on [Date]. [2008]. [H1]	12	69
19. Download [Geography]. [Canada]. [BC], [Employee]. [All Employee Departments]	Load time (size) 519 (162)	
20. Generate cross-tab on [Geography]. [Canada]. [BC], [Employee]. [All Employee Departments]	41	127
21. Drill down on [Geography]. [Canada]. [BC]. [Vancouver]	14	91
22. Drill down on [Employee]. [All Employee Departments]. [Engineering]	4	65
23. Roll up on [Employee]. [All Employee Departments]	7	83
24. Drill down on [Geography]. [Canada]. [BC]. [burnaby]	16	67
25. Download [Geography]. [United States].[California], [Date].[2007].[H1]	Load time (size) 1401 (274.5)	
26. Generate cross tab on [Geography].[United States].[California], [Date].[2007].[H1]	37	173
27. Drill down on [Geography].[United States].[California].[San Francisco]	17	137
28. Drill down on [Date].[2007].[H1].[Q1]	7	136

Continued on next page

Table C.1 – Continued from previous page

Workflow	Script time	Load time
15. Download sub-cube [Customer]. [Education].[Bachelors], [Date]. [2007].[H1], [Geography]. [Canada]&[France]	Load time (size) 1669 (326.5)	
29. Generate cross tab on [Customer]. [Education].[Bachelors], [Geography]. [Canada]&[France]	45	146
30. Drill Down on [Geography].[France]	25	167
31. Drill Down on [Geography].[France].[Nord]	28	146
32. Roll up on [Geography].[France]	1	138
33. Download sub-cube [Geography].[Canada], [Employee].[all employee departments]	Load time (size) 1601 (421)	
34. Generate cross-tab on [Geography].[Canada], [Employee].[all employee departments]	35	108
35. Dill down on [Geography].[Canada].[BC]	19	110
36. Drill down on [Employee].[All Employee Department].[marketing]	3	101
37. Roll-up on [Employee].[All Employee Department]	21	117
38. Drill down on [Employee].[All Employee Department].[executives]	1	203
39. Roll-up on [Employee].[All Employee Department]	20	120
40. Drill Down on [Employee].[All Employee Department]].[engineering]	2	216
41. Drill down on [Employee].[All Employee Department].[senior designer]	4	92
42. Download Subcube [Product].[All products], [All Geography].[Australia]&[Canada]&[France]	Load time (size) 3006 (431)	

Continued on next page

Table C.1 – Continued from previous page

Workflow	Script time	Load time
43. Generate cross tab on [Product].[All products], [All Geography].[Australia]&[Canada]&[France]	63	150
44. Drill down on [Product].[All products].[bikes]	54	147
45. Drill down on [Product].[All products].[mountain bike]	43	366
46. Roll up on [Product].[All products]	2	189
47. Drill down on [Geography].[Australia]	46	205
48. Drill down on [Geography].[Canada]	60	192
49. Drill down on [Geography].[France]	51	221
50. Download [Geography].[Germany]&[France], [Date].[2006]	Load time (size) 2717 (679)	
51. Generate cross tab on [Geography].[Germany]&[France], [Date].[2006]	88	170
52. Drill down on [Geography].[Germany].[Hessen]	39	141
53. Drill Down on [Geography].[Germany].[Hessel].[Berlin]	37	141
54. Roll up on [Geography].[Germany]	1	141
55. Drill Down on [Date].[2006].[H1]	30	173
56. Roll up on [Date].[2006]	1	141

# Bibliography

- [1] Sameet Agarwal, Rakesh Agrawal, Prasad M. Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES*, pages 506–521, 1996.
- [2] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *Proceedings of International Conference on Data Engineering (ICDE97)*, pages 232–243, 1997.
- [3] Luca Cabibbo and Riccardo Torlone. A logical approach to multidimensional databases. In *Advances in DB Technology*, pages 183–197. Springer, 1998.
- [4] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26:65–74, 1997.
- [5] IBM Corporation. Client-side data caching using faces client components. <http://publib.boulder.ibm.com/infocenter/iadthelp/v6r0/index.jsp?topic=/com.ibm.odc.doc/topics/cfcfintro.html>.
- [6] Microsoft Corporation. High-performance asp.net caching. <http://visualstudiomagazine.com/Articles/2011/12/01/High-Performance-ASPNET-Caching.aspx>.
- [7] Microsoft Corporation. Cellordinal attribute, 2012. [http://msdn.microsoft.com/en-us/library/ee320764\(v=SQL.105\).aspx](http://msdn.microsoft.com/en-us/library/ee320764(v=SQL.105).aspx).
- [8] Microsoft Corporation and Hyperion Solutions Corporation. Xml for analysis specification, 2012. <http://msdn.microsoft.com/en-us/library/ms977626.aspx>.
- [9] XML for Analysis Council. Xml for analysis, 2012. <http://www.xmla.org>.
- [10] Tim Hsiao. Web-based olap, 2010.
- [11] Torben Bach Pedersen and Christian S. Jensen. Multidimensional data modeling for complex data, 1999.
- [12] YUI Team. Yui 2: Treeview, 2012. <http://developer.yahoo.com/yui/treeview/>.

- [13] 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.
- [14] Microsoft SQL Server 2008 Analysis Services Unleashed. *Irina Gorbach and Alexander Berger and Edward Melomed*. Sams, 2008.
- [15] Zheng Xu. Client-centric mobile olap, 2012.