WEB-BASED OLAP

By

Tim I-Hsuan Hsiao
Bachelor of Science, Simon Fraser University, 2000

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

MASTER OF SCIENCE

In the
School
of
Computing Science

© Tim I-Hsuan Hsiao 2010

SIMON FRASER UNIVERSITY

Term Fall 2010

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:	**Tim I-Hsuan Hsiao**
Degree:	**Master of Science**
Title of Thesis:	**Web-based OLAP**

Examining Committee:

Chair: **Dr. Ke Wang**
Professor

Dr. Wo-Shun Luk
Senior Supervisor
Professor

Dr. Jian Pei
Supervisor
Associate Professor

Stephen Petschulat
External Examiner
Senior Architect
SAP – BusinessObjects

Date Defended/Approved: December 3 2010

SIMON FRASER UNIVERSITY
LIBRARY

Declaration of Partial Copyright Licence

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

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

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

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

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

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

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

Simon Fraser University Library
Burnaby, BC, Canada

Abstract

Rich internet applications (RIA) are the next-generation of web applications that share many characteristics of desktop applications. In-memory OLAP brings to the user the ability to explore the data warehouse in real-time on a powerful desktop. Combining these two areas together, we investigate challenges concerning development of a RIA for OLAP users. A functional prototype of a Web-based OLAP was built and run in a browser connected to an OLAP server inside a SAP – BusinessObjects testing lab. The performance of this client-centric, JavaScript-based system is compared to a version of the SAP's traditional server-centric system. We find that the choice of browser can significantly affect the performance of the Web-based OLAP. With the new graphic capabilities of HTML5, our Web-based OLAP attains a level of data visualization beyond the capabilities of a traditional client-server system.

Keywords: OLAP; Client-side aggregation; Partial pre-aggregation; in-memory OLAP; data visualization; HTML 5.

Dedication

To my sons Aiden & Isaac, for you are my motivation in this journey.

To my wife Tina, for your love and patience.

To my parents and brother, for your on-going support.

Acknowledgements

I would like to thank Dr. Ke Wang for taking time out of his busy schedule to serve as the Chair of the Committee. I am also grateful for Dr. Jian Pei's assistance in my admission to the graduate school at SFU, his guidance at the beginning of the program, and his time dedication to review my thesis and to serve the Committee as my supervisor. I was not familiar at all with my current research area initially, but the classes I took from them really broadened my knowledge in the advanced database topics. For this, I cannot thank them enough.

My deepest gratitude goes out to my senior supervisor, Dr. Wo-Shun Luk. I appreciate his patience for taking on a student who has little to no research skills. For someone like me who completed his undergraduate almost ten years ago, there never seems to be light at the end of the tunnel during this journey. However, through his encouragement and pointers in the right directions, I was able to make the journey count. He saw the potential and recommended me to a thesis research from a project option. Dr. Luk's research proposal is also the main reason why I won the ARC Fellowship from SAP – BusinessObjects. The award really accents my academic profile, and establishes good liaison with a leader in the OLAP industry. For this, I am forever in his debt.

I would like to acknowledge SAP – BusinessObjects for funding this project and providing a lab environment for the thesis development and experiments. From my point of view, a leader in the industry showing a strong interest in the research topic already says a lot about recognition. I had the pleasure to work with many experts from the company; in particular, I would like to thank Dr. Michael McAllister, Stephen Petschulat, Gavin Olle, Deejay Johal, and Parviz from the testing team, for their expertise and assistance that I cannot put a monetary value on. I especially want to express my gratitude to Steve, for not only his countless advices, ideas and positive feedbacks, but also showing great enthusiasm during our discussions, and always being flexible and supportive for my requests.

Table of Contents

Approval ... ii

Abstract ... iii

Dedication ... iv

Acknowledgements ... v

Table of Contents ... vi

List of Figures ... viii

List of Tables ... ix

1 Introduction ... 1

1.1 An Overview of Data Warehouse and OLAP ... 1
1.2 Web-based OLAP Client ... 2
1.3 Problem with a Server-centric OLAP Model ... 3
1.4 Related Work ... 4
1.5 Thesis Organization ... 5

2 JavaScript OLAP Client ... 6

2.1 Multidimensional Data and Metadata ... 6
2.2 Web Technologies and JavaScript ... 9
2.3 Standalone In-Memory JavaScript Engine ... 11
2.4 Evaluating the JavaScript OLAP Engine ... 15
2.5 Revised JavaScript OLAP Client ... 19
2.5.1 Metadata Representation ... 19
2.5.2 Modified *getMatrixAnswer* ... 22
2.5.3 OLAP Operators ... 24

3 Web-based OLAP Architecture ... 26

3.1 Traditional Web-based OLAP Architecture ... 26
3.2 Overview of the Proposed Architecture ... 27
3.3 OLAP Server ... 27
3.4 XML for Analysis Layer ... 28
3.5 OLAP Data Access Layer ... 30
3.6 RESTful Web Service Layer ... 30
3.6.1 Server Exploration ... 30
3.6.2 Multidimensional Data Transformation ... 33
3.6.3 Data Caching ... 36
3.6.4 Dataset Compression ... 37

3.7 Lightweight OLAP User Workflow 39
3.8 Performance Degradation with Respect to Sub-cube Size 44

4 Calculations and Data Visualization 48

4.1 Data Visualization in Traditional Web-based OLAP 48
4.2 Top-*k* Percentage Contributions Over *m*-time Span 49
4.3 *m*-day Moving Average 51
4.4 Scatter Plot Aggregation on Mouse Selection 52

5 Conclusion and Future Work 56

5.1 Conclusion 56
5.2 Future Directions 56

Appendices 58

Appendix A: Weather Dataset Specification 58
Appendix B: Partial XMLA Response 59
Appendix C: REAL Warehouse Sample V6 MT Dataset Specification 60
Appendix D: Workflow Sequence 65

References 69

List of Figures

Figure 1. A Star schema [2] ... 7
Figure 2. Example of hierarchies [2] ... 7
Figure 3. A 3-Dimensional cube [2] ... 7
Figure 4. An example of a cross-tab ... 8
Figure 5. JavaScript OLAP Engine ... 12
Figure 6. Algorithm *getMatrixAnswer* pseudo code ... 14
Figure 7. *isAncestor* pseudo code ... 14
Figure 8. *ancestorAtLevel* pseudo code ... 15
Figure 9. JavaScript OLAP engine speed on a query size of 80 cells ... 17
Figure 10. Query size of 680 cells ... 17
Figure 11. Query size of 1010 cells ... 18
Figure 12. Query size of 3030 cells ... 18
Figure 13 a) Before loading hierarchy content, b) [Accounts] sub-tree loaded. ... 21
Figure 14. Modified *getMatrixAnswer* pseudo code based on *TreeView* ... 23
Figure 15. Server-centric OLAP Architecture ... 26
Figure 16. Client-centric OLAP Architecture ... 27
Figure 17. Reduced fact table through partial pre-aggregation ... 28
Figure 18. Basic MDX query to obtain leaf level data ... 28
Figure 19. Member node selection with *descendants*(*root*, *3*, *leaves*) ... 34
Figure 20. CellOrdinal calculation ... 34
Figure 21. Compression at HTTP level [31] ... 38
Figure 22. Original versus deflated sub-cube size ... 38
Figure 23. Traffic shaping with *Dummynet* [34] ... 40
Figure 24. Performance time comparison on workflow sequence ... 42
Figure 25. Performance degradation ... 46
Figure 26. Top-*k* selection ... 49
Figure 27. Custom array insertion method ... 50

List of Tables

Table 1. Browser script and render time..16
Table 2. Multidimensional result set transformation..35
Table 3. Testing environment..39
Table 4. Average performance time on model comparison..40
Table 5. Total data size transferred..44
Table 6. Performance degradation with respect to sub-cube size..45
Table 7. Sub-cube size explosion..45
Table 8. Increased member set size..46
Table 9. Top-*k* percentage contribution performance..51
Table 10. *m*-day moving average performance..52
Table 11. SVG versus Canvas scatter plot aggregation..55

1 Introduction

1.1 An Overview of Data Warehouse and OLAP

OLAP (On-Line Analytic Processing) refers to the analysis of business data by casting the data in a multidimensional model [1]. An excellent end-to-end overview of data warehouse and OLAP is described by Chaudhuri and Dayal [2]. Though not limited to the retail industry, consider a typical scenario for OLAP application of a business with several retail locations spread across a country, or even around the globe. Each location maintains a transactional database that records details of daily business operations. Periodically, transactional data is extracted, transformed, and loaded into a centralized data warehouse or several data marts if extensive business modelling is required to perform a global integration. Together with data mining, this is better known in the information industry as BI, or Business Intelligence.

An OLAP tool is an application that allows the user to explore the data warehouse at ease. One of the first OLAP tools was a spreadsheet system [3], such as Microsoft Excel, which did not scale for large data sets. For businesses that have been established for years, the database sizes, in terms of number of records, can easily be in the order of tens of millions. As data volumes increased, the debate in the industry focused on whether to house the data in a relational database system or a separate, specialized OLAP server. Over the years, separation of largely read-only from data warehouse into OLAP server has become the norm.

Recently, 64-bit computing, rapidly declining memory prices and multi-core CPUs have conspired to instigate a software revolution in database technology. For the first time in the long history of main-memory Database Management System research, in-memory OLAP technology has become the mainstream [4, 5]. The *PowerPivot* extension for Excel 2010 from Microsoft is the latest example that displays the capability of a desktop application for analyzing data from a server in real-time. These products allow the user to explore the OLAP data cube on a powerful desktop, and pose “what-if” questions without re-doing the entire cube.

1.2 Web-based OLAP Client

On the other hand, OLAP clients that run in a web browser have limited flexibility as they were introduced before various browsers competed for JavaScript execution speed. Up to now, the primary function of a browser-based OLAP client is to present results calculated by an OLAP server. In contrast to OLAP-specific applications, web applications in general have been advancing at a hectic pace. The specific technological advancement relevant to this research is Rich Internet Application. A Flex, Silverlight, or Java FX Rich Internet Application, which is downloadable from a web server, runs as a browser plug-in but with features of a desktop application (such as access to local file system, with elevated trust level, and a plethora of graphical functions for data visualization). However, these plug-ins are proprietary products. With Adobe, Sun and Microsoft being the major vendors of these plug-in, they rely on the collaboration between vendors to run on certain operating system and browser platforms, and the termination of such collaboration can lead a development project to a dead end [6].

Most web developers, however, prefer to avoid such dependencies by running their web applications on the native browser. JavaScript has been the predominant programming language for browsers since the nineties and naturally became the undisputed client-side language for web browsers after the failure of the Java applet. Consequently, browser vendors have been continually improving their JavaScript engine speeds as W3C updates the HTML specification.

Although HTML 5 is still in the draft stage, Google, Apple, and Mozilla have already implemented many features from the specifications in the latest versions of the browsers. First, on the graphics library side, both SVG and the Canvas element allow dynamic rendering of 2D shapes directly with JavaScript in Chrome, Safari, Firefox, and Opera. Second, every major browser now supports session and persistent storage, albeit limited in disk quota. Third, Chrome, Safari, and Firefox have implemented database storage, which essentially embeds a persistent relational database engine for a web site. Last and not least, web workers currently implemented by Chrome and Firefox allow long running, computationally intensive scripts to execute in parallel with the main thread. Being the browser with the largest user base, Internet Explorer 8 lacks most of these functionalities; however, that will change with the up and coming IE 9 [7]. Overall, these proposals promote development of complex web applications on the web platform. In our opinion, JavaScript will remain as a strong contender in future generations of RIA, making it a suitable client-component in OLAP architectures.

1.3 Problem with a Server-centric OLAP Model

The majority of OLAP systems nowadays are server-centric. This means that the processing of data occurs entirely on the OLAP server. In contrast to the technological advances to OLAP technology on the server side, the progress of client-centric OLAP technology has been quiet. Consider the following realistic scenario for an electronics store chain: suppose that the OLAP server is physically located at the corporate headquarters in Toronto. A retail store manager in Vancouver uses an OLAP client application to connect to the OLAP server. First, assume that the one-way network latency, which is defined as the time from the start of packet transmission to the start of packet reception, is approximately 60 milliseconds. As a simplifying assumption, let us also assume that the OLAP server is powerful enough to process any type query in negligible time and that the query and response messages are small enough for us to take network bandwidth out of the equation. A typical OLAP client gives users the ability to submit a query (for example, a cross-tab of total sales amount by product category and store location to the server) receives the response in no less than 120 milliseconds, and present the results in a data grid.

At first glance, there does not seem to be any problem with this example. However, notice there is practically zero resource utilization on the client-side, outside the presentation layer. Naturally this brings up a point that is not necessarily a problem in a server-centric model but rather, an advantage of a client-centric model: scalability. By allocating some calculations to the client-side, the server load is eased so it can accept more requests. This may not seem problematic when the number of simultaneous users is small; however, in the following scenario the impact is non-trivial.

A “thin” browser may be suitable when users do not need real-time feedback of the queries, or the number of queries is small across time. However, if the manager wishes to perform further operation on the data (such as swapping row and column, viewing the total sales of a particular store with respect to each product category, or applying a date range filter to the data) the OLAP client must reconstruct the query and inquire the server. A rapid succession of such operations, which is common when a user knows the interface inside out, would make the existence of the 120 milliseconds lag conspicuous. The lack of smoothness is most apparent when an analyst binds a slider, a highly interactive and desirable control in data visualization, as a sliding window filter in a query. For example, the query can be the top-k products sold in the past m days or the m-day moving averages of store sales, in which the slider bar is used to adjust the value of m and each movement can potentially affect the outcome drastically. In this sort of case,

it is extremely important for an analyst to investigate the reasons behind such trend deviations. In this application, each tick in the slider movement pounds the server, which is amplified by the number of concurrent users. Even if the server is capable of serving all these simultaneous requests through software or hardware enhancements, the existence of consistent network latency, or even worse, high packet delay variation over the Internet is beyond any organization's control. This variable alone contributes to the increased query response time, and limits the quality of service of smooth data visualization. This phenomenon motivates us to utilize client-side resources to bridge such a gap.

Our contribution to the OLAP research community is three-fold. First, to the best of our knowledge, we built a direct side-by-side comparison of a traditional approach and a new browser-based in-memory approach. Second, we show by a proof-of-concept that the client-centric aggregation scheme can perform better than a traditional server-centric OLAP system in the presence of network latency. Third, utilizing the new HTML5 features, we develop a novel aggregation technique with the traditional scatter plot that is achievable only in a client-centric model and the new HTML5 Canvas element. Fourth, we compare data visualization between the traditional server-client OLAP system and our architecture, and show that by delegating data processing to the client-side, we eliminate network latency from the equation, facilitating smooth data visualization.

1.4 Related Work

Researchers have extensively studied various aspects of Online Analytical Processing for many years and have published numerous research papers on this topic [2, 8-15]. Although they do not explicitly state that these group-by algorithms, index strategy, and cubing methods are for an OLAP server, it can be assumed, due to the nature and size of datasets that would require analytical processing. On the web technology side, applications such as SAP's server-based browser client, Strategy Companions Analyzer, Kyubit AnalysisPortal and Ranet.UILibrary.OLAP have been around as OLAP clients; however, these applications rely entirely on server computation for all OLAP operations. On the other hand, QlikView and BusinessObjects Desktop Intelligence best exemplify client-side in-memory OLAP applications that run in a desktop environment.

To the best of our knowledge, there has not been any published research work based on client-side OLAP that runs in a restrictive web-based environment. The closest work that comes to mind is an Adobe Flex 3 *AdvancedDataGrid* control that mimics the functionality of client-

side aggregation on relational data. However, as stated by the Flex development team at the time of this writing, the current state of *AdvancedDataGrid* cannot handle more than a few thousand records even thought it runs as a compiled binary and users must explicitly define relationships between the data sources [16].

1.5 Thesis Organization

The remainder of this thesis is organized as the following: Chapter 2 presents an overview of key concepts in multidimensional data, details JavaScript as the runtime environment for an OLAP application, and describes the first stage of this research where we benchmark client-side aggregation using JavaScript against various parameters. In the later sections of this chapter, we describe several enhancements to the baseline JavaScript OLAP Client which, in conjunction with the architecture in Chapter 3, overcomes many of the challenges we observed. In Chapter 3, we propose a hybrid OLAP architecture that is different from traditional client-server OLAP system and describe each individual component in detail. A set of common OLAP operators are introduced and a set of workflow sequences is devised to benchmark the performance of our proposed architecture against a real-world OLAP system and real-life dataset from SAP - BusinessObjects. In Chapter 4, we take the OLAP operators a step further to demonstrate the capability of the client-centric model by implementing several calculations commonly used by power users. We compare our results to a simulated server-centric model not only on the aggregation performance aspect, but also on the client's ability to present graphical visualizations. Finally, we summarize our contributions in Chapter 5 and discuss potential future works regarding this research topic.

2 JavaScript OLAP Client

There has been very little study on offloading server-side aggregation to client-side in both the academic and industrial research areas. Security and privacy issues aside, we suspect there are several reasons for this. First, the amount of data stored on the server is too massive for a desktop client to withhold in memory or even on disk. Second, delivering such data over the network to each OLAP user is neither efficient nor cost-effective. Third, without pre-aggregation and heavy indexing, the client is not powerful enough to process multidimensional queries. Due to these reasons, most OLAP systems rely on the server to process client requests, be it an OLAP server or an application server.

In our initial stage of research, we explore the potential of a standalone OLAP system. In contrast to a main-memory server-based OLAP system, this engine is written in JavaScript and runs in a browser. Our engine downloads dataset and metadata from a general-purpose web server and performs aggregation locally.

In this chapter, we start with a preliminary overview of multidimensional databases, and describe key technologies in modern web applications that are particularly critical in this research. We then describe the initial data structures, general architecture, and aggregation algorithm used by the in-memory OLAP engine. An OLAP cross-tab test is conducted with this initial prototype using the Weather Dataset by comparing its performance in different browsers, using datasets and queries of various sizes. With results from the initial prototype, we enhance several aspects of the JavaScript OLAP Client. Together with a server component introduced in the next chapter, they form a novel architecture to deliver a client-centric OLAP system.

2.1 Multidimensional Data and Metadata

We first provide an overview of the multidimensional data associated with OLAP applications. A database system, be it a small business relational database or a giant corporate data warehouse, typically contains one or more fact tables that are related to several dimension tables. Figure 1 shows an example of a data set in a relation known as a star schema.

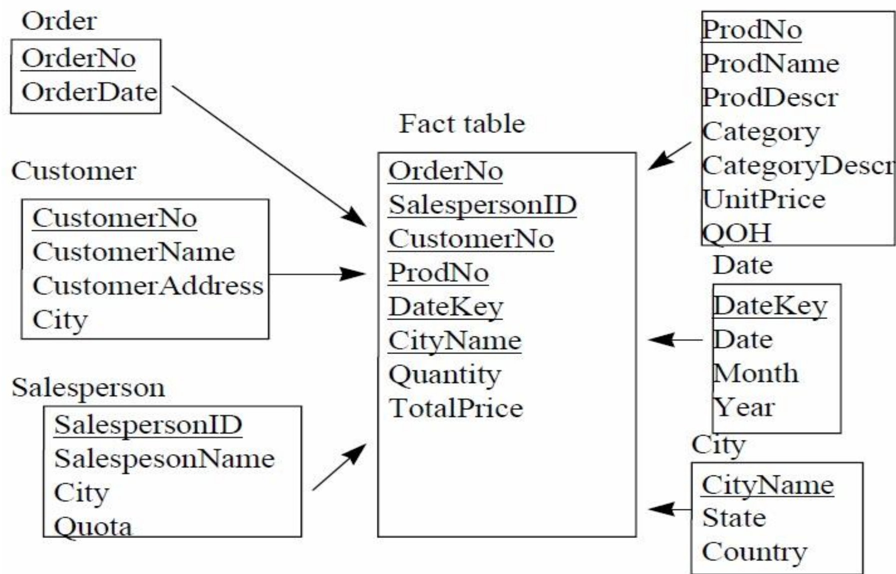


Figure 1. A Star schema [2]

Each fact table essentially contains records with one or more measures such as sales quantity and sales amount. Each record also has attributes associated with it, such as customer number, store number, and product number.

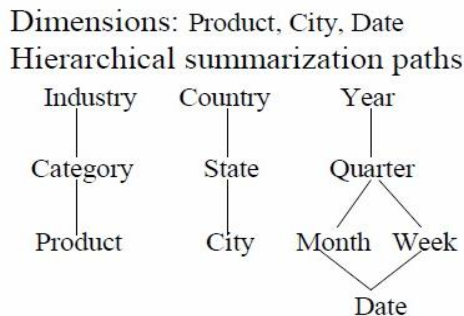


Figure 2. Example of hierarchies [2]

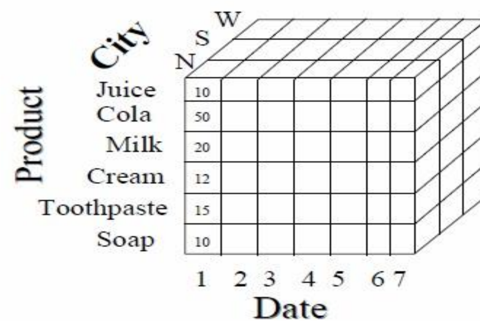


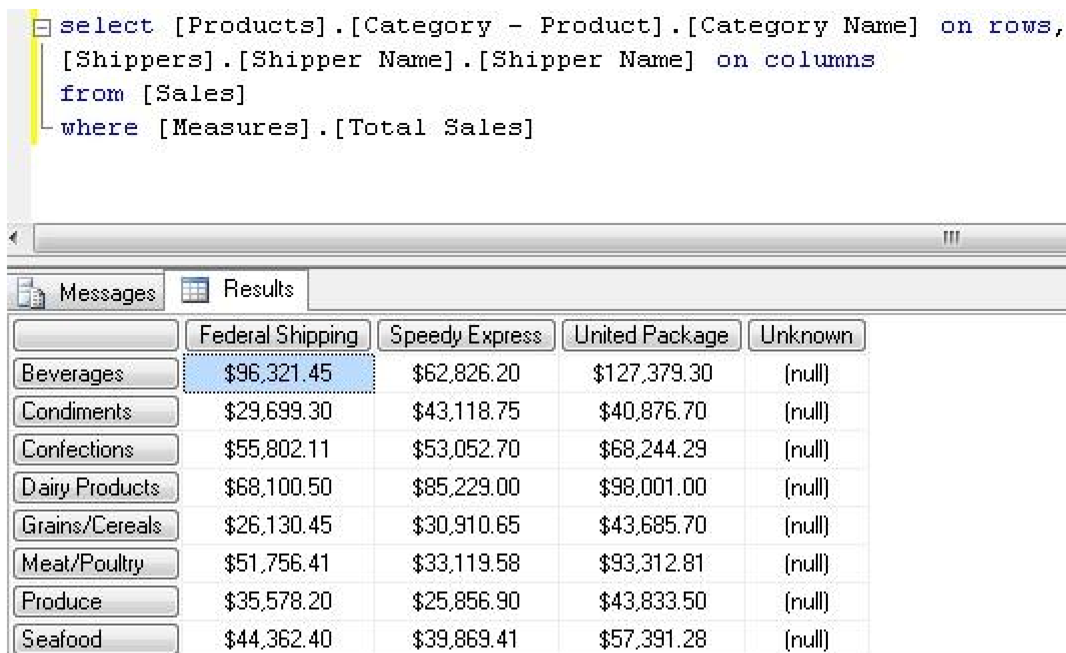
Figure 3. A 3-Dimensional cube [2]

These attributes are known as dimensions; each dimension may contain several concept hierarchies. For instance, the product number dimension may contain a hierarchy based on category and another hierarchy based on price range. Each hierarchy consists of hierarchy members. Figure 2 depicts hierarchy levels for the Product, City and Date dimension and each node in the hierarchy represents a member. Primary members are those at the lowest granularity

and are the attribute values recorded in each row of the fact table. The purpose of an OLAP server is to build, for each fact table, a multidimensional cube or cuboid that pre-aggregates measures in the fact table with respect to various levels of the hierarchy trees of various dimensions.

Figure 3 shows a visualization of a 3-dimensional cube: it is an aggregation of sales quantity for every combination of product categories, city regions, and days of a week. A combination of members from each dimension forms a tuple. Given an *n*-dimensional cube, a tuple can consist of *n*-members that pinpoint a specific cell. In the case where a tuple does not specify a member for a particular dimension, every member in that dimension is implicitly included.

As the number of dimensions increase in a multidimensional dataset, the number of possible cuboids grows exponentially because a dimension may contain several hierarchies and each hierarchy may contain different levels of granularity. Therefore it is costly for OLAP servers to generate excessive cubes. However, such pre-computation mechanisms provide responsive answers to a user query based on specific dimensions and granularity as shown in Figure 4.



	Federal Shipping	Speedy Express	United Package	Unknown
Beverages	$96,321.45	$62,826.20	$127,379.30	(null)
Condiments	$29,699.30	$43,118.75	$40,876.70	(null)
Confections	$55,802.11	$53,052.70	$68,244.29	(null)
Dairy Products	$68,100.50	$85,229.00	$98,001.00	(null)
Grains/Cereals	$26,130.45	$30,910.65	$43,685.70	(null)
Meat/Poultry	$51,756.41	$33,119.58	$93,312.81	(null)
Produce	$35,578.20	$25,856.90	$43,833.50	(null)
Seafood	$44,362.40	$39,869.41	$57,391.28	(null)

Figure 4. An example of a cross-tab

2.2 Web Technologies and JavaScript

In our initial design, the main-memory OLAP engine works similarly to an OLAP server. The critical difference is that it runs in a restrictive browser environment and there is no pre-aggregation at any level. Given the same fact table as the OLAP server, the engine takes a linear scan approach and aggregates a cross-tab query on the fly.

As mentioned in the previous chapter, JavaScript has been under the spotlight in the past decade, getting special treatments to support a graphically-rich, highly interactive user interface environment that is ideal for data visualization. However, its ability to perform OLAP operations is not evident. What is more daunting is the restrictive sandbox environment associated with a web-browser in comparison to a desktop application. A sandbox environment restricts the application from fully accessing local file systems, which rules out caching of data on the local disk without resorting to an external plug-in. The browser host carries out memory management; a JavaScript application cannot dynamically allocate more memory even if there is an abundance of free memory available. While a desktop application can maximize its performance through concurrency, the current state of parallel processing in JavaScript is still limited. The primary function of Web Workers is to compute long running, CPU intensive tasks such as calculating prime numbers in the background while preventing the user interface from freezing. Since our goal of an OLAP client is to respond to a query in sub-second, Web Workers would offer minimal benefit at this stage, if any at all.

With the aforementioned drawbacks, there are benefits for developing our prototype on a browser platform. Like any other web application, any device with an internet connection and a browser can become an OLAP client in our proposed system. Other than installing a browser, preferably one that has a fast JavaScript engine, the client machine is literally maintenance-free. This implies there is no cost associated with deploying application patches onto every user device: when users access OLAP data, he/she is running the latest code. Another compelling reason is the research collaboration in the area of web-based OLAP with SAP – BusinessObjects, one of the leaders in the business intelligence industry. SAP has developed a web-based BI product suite. Their tool offers a “thin” browser client that presents multidimensional data from the OLAP server using dynamic HTML. The current generation of web-based OLAP does not perform any local processing of data. Therefore, it is the common interest of both parties to validate our test results against real-life queries and real-life datasets made available by SAP.

Since JavaScript is the language used to write the engine code, we describe three key web terminologies relevant to a JavaScript OLAP client: *Document Object Model*,

XmlHttpRequest, and *JSON*. The *HyperText Markup Language* (HTML) is the predominant markup language of the Internet. An HTML document contains tags known as HTML elements. Web browsers display document contents based on the ordering of HTML elements, which usually nest within one another. This hierarchical element representation is known as the *Document Object Model* (DOM). The DOM exposes a set of functions to JavaScript to manipulate HTML elements. For instance, the *getElementById* function allows JavaScript to retrieve an element object at runtime using the ID associated with the HTML element, a function known as DOM Traversal. As we shall see in the remainder of this thesis, DOM traversal plays a critical role that contributes to the performance loss of a web application in which data visualization is involved.

Before the era known as Web 2.0, web applications were limited in the sense that when the client browser needed to send data or required data from the web server, form submission was required. When the user submitted a form, the front-end browser waited for server response before proceeding. This type of user interaction is no longer the case with the introduction of *XmlHttpRequest* (XHR), or more commonly known as AJAX. Essentially, XHR allows JavaScript to communicate with a remote entity, fetch resources, or consume a web service over the HTTP layer, asynchronously. Being asynchronous means the interface does not lockup while the request is in progress. Rather, a custom event handler is bound to the XHR object so that when the remote entity responds, JavaScript passes control to the event handler to process the response.

JavaScript Object Notation (JSON), as the name implies, is an open standard object declaration convention designed for data interchange. Originating from JavaScript's simple data structure and object representation, it has become the standard format with parsers for almost every programming language. JSON provides two main methods for data interchange: *stringify* and *parse*. Consider the following JavaScript object in the heap:

```
var student = {
    studentNumber: 12345,
    firstName: "Jason",
    lastName: "Smith"
}
```

JSON.stringify(student) effectively serializes the object into a string. The benefit of JSON is not apparent at this point. However, this is a powerful and efficient feature if used in conjunction

with the *XmlHttpRequest* object: the server can respond to a client request by creating the object on the server-side, serialize it with stringify, and send it to the client. When the client receives the XHR response, it can invoke JSON.parse and immediately instantiate the object on the client-side, at runtime. This eliminates manual parsing of the response content.

In addition to the above, we believe the most powerful feature of JavaScript is its dynamic, weakly typed nature. JavaScript objects and data structures can be easily manipulated, and object properties/methods can be searched on, at runtime. The downside of JavaScript is the limited selection of primitive types. In contrast to other languages that differentiate small and big integers, single and double floating points, JavaScript treats all numbers as 64-bit floating points. From a memory management perspective, large JavaScript data structures would leave a massive footprint.

2.3 Standalone In-Memory JavaScript Engine

We begin by implementing a prototype to show the concept of client-side aggregation by moving the entire fact table and metadata to the client and letting the client compute the aggregation on the fly. In contrast to disk-based and index-heavy OLAP query processing methods [11, 13], we start with an experimental in-memory OLAP engine written in C++, which is meant for a desktop platform [17]. Like most in-memory OLAP systems, it does not do much pre-aggregation, nor does it keep an elaborate data structure. Queries are processed by a linear table scan of the dataset and the ancestor-descendant relationships are determined on the fly to answer a pivoting of two dimensions [18, 19]. Preliminary tests show that the response time is in the order of microseconds on a laptop computer. Given the low-cost of memory nowadays, we expect even a faster performance if the dataset resides in memory.

This indicates that client-side OLAP is not a far-fetched idea as a desktop application; however, as our focus is a web-based platform, we port the engine into JavaScript with the components as shown in Figure 5.

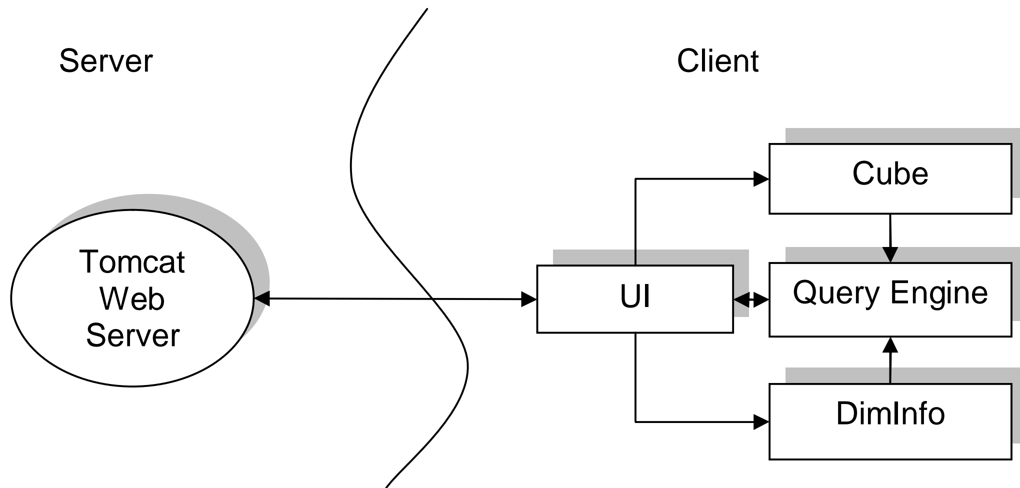


Figure 5. JavaScript OLAP Engine

To simulate dataset download from a data warehouse, we encode the dataset and metadata in JSON format and store them on a Tomcat web server. The web server is merely a repository for the JavaScript code, dataset, and metadata. At the start of the web application, the browser downloads the dataset and metadata using *XmlHttpRequest*, and the Tomcat web server will not be called again.

Cube

We use JSON.parse to convert the XHR response text directly into a JavaScript object "Cube", an in-memory 2-dimensional Array Object that resembles the fact table.

DimInfo

The metadata is instantiated into a JavaScript object "DimInfo" that specifies:

- the number of dimensions,
- array of dimension names,
- array of total members, total primary members, member ids and member names for each dimension,
- number of measures, array of measure names,
- array of parent-child member relationships in each dimension,

- and pre-compute ancestor members at every level of the hierarchy for each primary member. This pre-computation facilitates a constant time factor to determine aggregation point during the query answering stage.

UI

The UI object provides a basic HTML interface allowing the user to select which dataset and metadata to fetch from the server, displaying dimensions and members in drop down boxes, and for rendering the output pivot table. We integrated a slider bar from *jQuery UI* [20] to our application to act as a range filter. When the user slides the slider thumb, the query engine re-aggregates the data and UI object redraws the pivot table.

Query Engine

The reason we called this version a prototype is because its functionality as an OLAP client is rather limited. At this point, the OLAP engine can process the most basic form of an OLAP operation, a cross-tab (or pivot table). Given a cross-tab query based on [*rowAxis*].[*rowNode*], [*columnAxis*].[*columnNode*] and [*measureIndex*] similar to that of Figure 4 and a slider bound to the [*SliderAxis*] dimension. We define the following:

TotalDim, total number of dimension in the dataset.

TotalMeasure, total number of measures in the dataset.

FactTable, an 2D array with *n* rows and (*TotalDim* + *TotalMeasure*) columns. Each row represents a tuple in the dataset. The first *TotalDim* columns contain primary member ids of each dimension of the tuple. The last *TotalMeasure* columns contain the numeric measure.

rowGroup, an ordered list of members that are the children of *rowNode*, where |*rowGroup*| = *r*.

colGroup, an ordered list of members that are the children of *colNode*, where |*colGroup*| = *c*.

As previously mentioned, the list of ancestors of a primary member is pre-determined in *DimInfo*. Thus, the function *isAncestor*(m_1,m_2) runs in $O(H(d))$ where $H(d)$ is the height of the hierarchy tree d in which m_1 and m_2 reside. We compute an r x c cross-tab using the following algorithm:

```
Algorithm getMatrixAnswer (Array[][] answer)
--------------------------------------------
FactTableLoop:
  //Linear Table Scan
  For i = 1 to FactTable.length
    currentTuple = FactTable[i]
    rowMember = currentTuple[rowAxis]
    colMember = currentTuple[colAxis]
    sliderMember = currentTuple[sliderAxis]
   // skip currentTuple if it does not satisfy the three conditions below
    if !isAncestor(rowNode, rowMember) Next i
    if !isAncestor(colNode, colMember) Next i
    if (sliderMember is not in slider range) Next i
   // compute the position of the ancestor of rowMember in rowGroup
    row = ancestorAtLevel(rowMember, rowNode.level) – rowNode.index
    // compute the position of the ancestor of colMember in colGroup
    col = ancestorAtLevel(colMember, colNode.level) - colNode.index
    //aggregate the tuple measure to the cell in the answer matrix
    answer[row] [col] += currentTuple[measureIndex]
 End For
```

Figure 6. Algorithm *getMatrixAnswer* pseudo code

```
Function isAncestor (member m1, member m2)
------------------------------------------
    Return true if m1 is an ancestor of m2, false otherwise
```

Figure 7. *isAncestor* pseudo code

Function ancestorAtLevel (member m, integer l)
Return the ancestor member of *m* at level *l*

Figure 8. *ancestorAtLevel* pseudo code

In general, a tuple is determined to be part of the cross-tab if the tuple members associated with the row and column axes are descendants of the *rowNode* and *colNode* respectively, and the tuple member associated with the slider axis is within filter range. The tuple is aggregated to a cell in the answer matrix. The cell address is determined by subtracting the index of the associated row/column ancestors in the *rowGroup*/*colGroup* from the index of the first member in the *rowGroup*/*colGroup*. The indices of the members in this dataset are in sequential descending order, allowing us to compute the offset relative to the hierarchy level easily. This algorithm uses a linear table scan query processing method. Suppose $H(t) = \max(H(rowAxis), H(columnAxis))$ then the algorithm runs in $O(n \cdot H(t))$ time.

2.4 Evaluating the JavaScript OLAP Engine

Back in the days before the browser vendors were serious about JavaScript execution speed, the consensus was that JavaScript lacks the speed to delivery anything valuable as an OLAP client. Internet Explorer, in particular, has the slowest JavaScript environment and yet it has the largest user base. With that in mind, the current generation of web-based OLAP clients are built around presenting results rather than generating results, which yields the fact that web-based OLAP aggregation is such an unexploited research area. The experiments we perform at this stage allow us to quantify exactly what we are dealing with, and whether client-side aggregation on a browser can offer reasonable improvement to user interaction with the data. We performed testing on a desktop computer equipped with a Core 2 Duo 2.66 GHz processor, 3.25 GB memory, running 32-bit Windows XP SP2. The Weather Dataset specified in Appendix A was used to conduct this experiment.

First, we define three parameters used in this experiment: **Dataset size** refers to the number of tuples in the fact table. To create dataset sizes of 314,922, 508,159, 720,724, and 1,015,367 rows, we randomly sampled records from the original pool of ten million records to prevent sparseness in the smaller datasets. There are many browser benchmark results out there, indicating that certain browsers dominate in applications with lots of recursive function calls, while others succeed at computationally intensive tasks. Therefore, **browser platform** is another

interesting variable to compare the differences between browser engines from an aggregation speed perspective. We tested the application on the latest versions of the five most popular browsers at the time of this writing. **Query size** refers to the number of cells in the output data grid. We selected dimension members for the row and column axes to obtain query sizes of 80, 680, 1010, and 3030 cells. **Output rendering** is a binary variable that dictates whether the browser displays the data grid or not after the cross-tab aggregation is complete. As we shall see shortly, it has a profound impact on the performance given the number of DOM traversals required to render a large query size.

In addition to these four parameters in the experiment, there are three important metrics in measuring the performance of a web-application: load time, script time, and render time. Load time refers to the time it takes to download the html page, style sheets, images, and JavaScript files, as well as any time spend on AJAX calls. Script time refers to the time spent on JavaScript execution. Render time refers to the time for the browser to display web content in the window. We divide the test results in two parts, first by taking output rendering into consideration, and then by pure aggregation time. Load time is another serious issue that we will address in a later chapter. All time units are measured in milliseconds.

A web-based OLAP client incurs rendering time as a cost, be it part of a client-centric or a server-centric model. Table 1 shows the time it takes to execute the query in Figure 12 with and without rendering the data grid in the three fastest browsers.

	B1		B2		B3	
Dataset Size	Script	Script + Render	Script	Script + Render	Script	Script + Render
314,922	45	377	99	335	373	525
508,159	79	393	160	390	580	752
720,724	111	460	193	462	603	990
1,015,367	171	509	299	585	1,112	1,390

Table 1. Browser script and render time

As seen in the table, we observe the effect of dynamic content update in a browser through DOM traversals as the time it takes to draw 3030 table cells takes several times longer than that of pure aggregation.

The following four figures show the results of the experiment without considering rendering time. That is, the raw speed of the standalone OLAP engine to computes the cross-tab aggregation with an in-memory fact table.

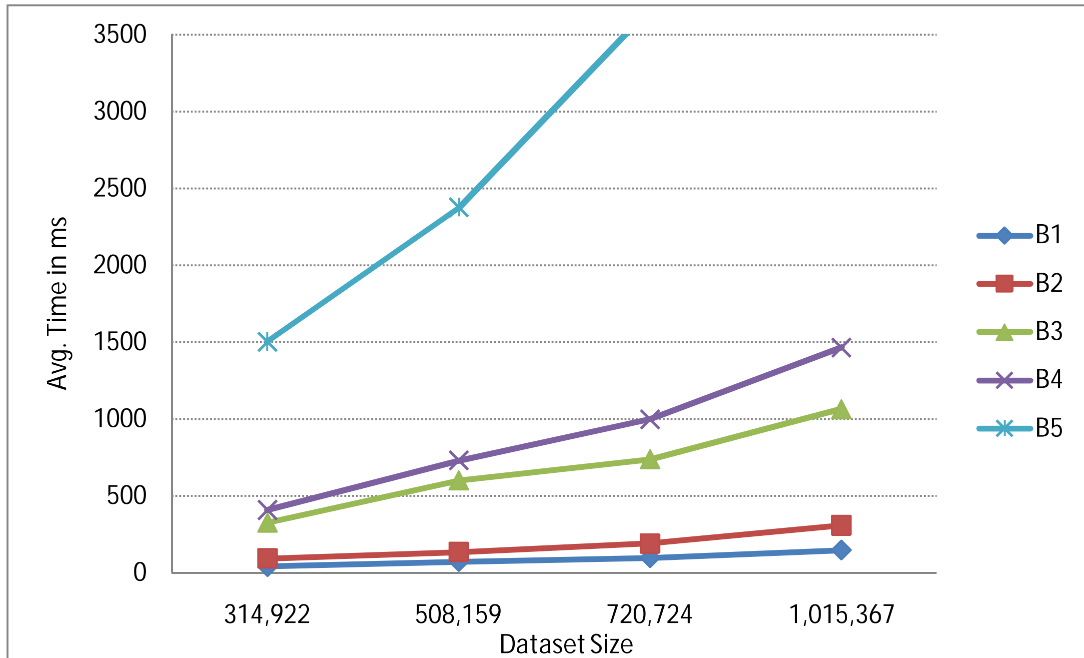


Figure 9. JavaScript OLAP engine speed on a query size of 80 cells

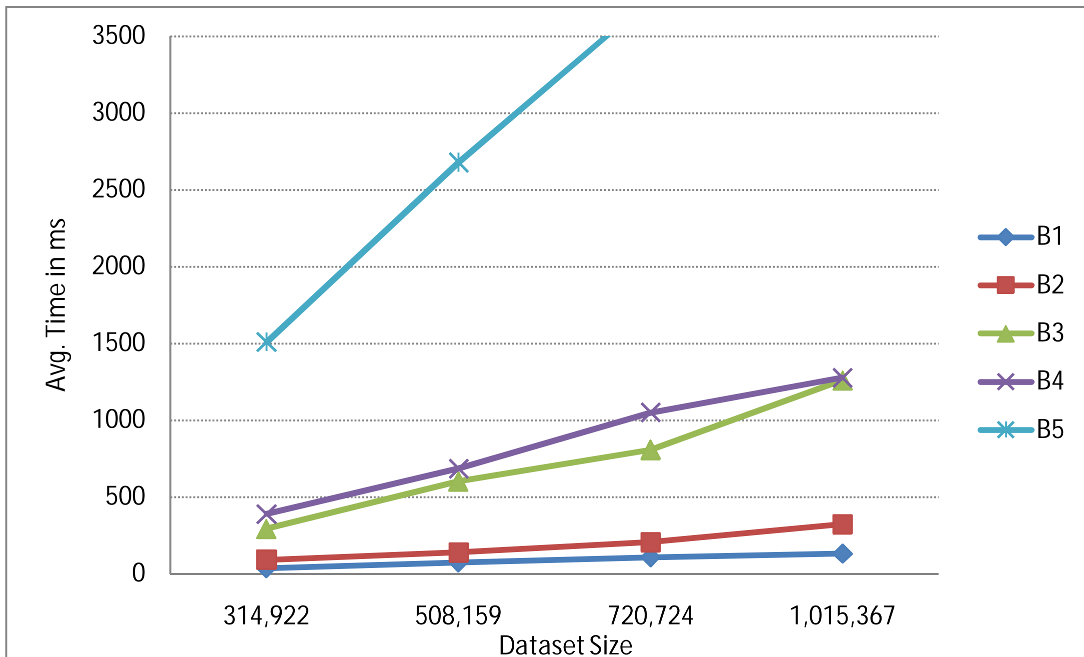


Figure 10. Query size of 680 cells

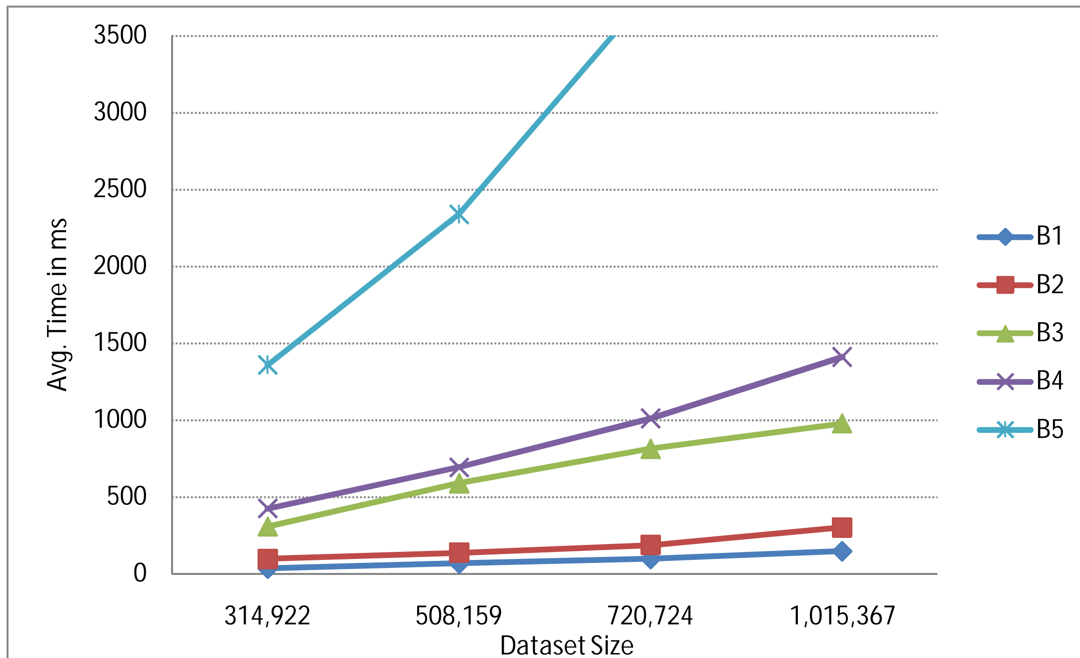


Figure 11. Query size of 1010 cells

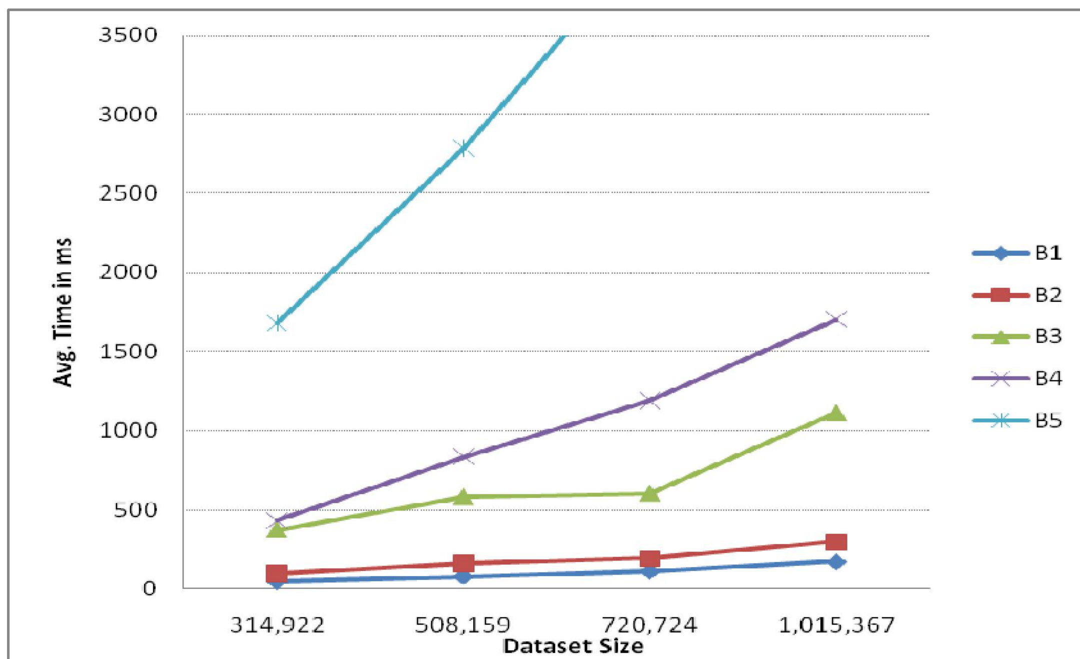


Figure 12. Query size of 3030 cells

To obtain different query sizes, the row axes and column axes of these queries are selected from the dimensions of the Weather Dataset to form queries of different sizes. The number of primary members in these hierarchies in Appendix A reflects the sizes of the hierarchies.

From the prototyping experience and experimental results of the prototype, we draw a strong conclusion that the performance the main-memory OLAP engine is heavily dependent on the browser or, more accurately, the JavaScript Engine version. In addition, we identify a number of issues pertaining to the JavaScript in-memory OLAP engine:

- Current algorithm quickly suffers performance degradation as the size of the dataset increases. Although query size is the second major contributor to pure aggregation time, its impact is primarily on rendering time. With the current approach, the size of the data that JavaScript can process in a reasonable amount of time is rather limited. By reasonable amount of time, we mean the time span it takes to send the query and receive the result from an OLAP server over network latency. The actual amount of time differs among actual OLAP environments.
- Most real-life databases have dimensions that contain multiple hierarchies of various depth and some hierarchies may contain tens of thousands of primary members.

2.5 Revised JavaScript OLAP Client

Other than performance bottleneck, the JavaScript OLAP prototype described in section 2.3 is not nearly a functional OLAP system. It does not have efficient and intuitive way to handle metadata. The user interface is good for only cross-tab operations; it does not do common OLAP operations such as roll-up and drill-down. Architecturally, it lacks a visualization layer for these UI operations. We address each issue and describe how to overcome these drawbacks.

2.5.1 Metadata Representation

We developed the original prototype with the Weather Dataset by using array structures to store metadata. Part of the data structure choice has to do with properties of the Weather Dataset, properties that are unrealistic to presume in any other dataset. First, a dimension has only one hierarchy. This simplifies the aggregation process, as there is no need to determine the actual hierarchy when a user selects a member from a dimension. Second, members in each hierarchy have an unsigned integer as a unique identifier in the hierarchy namespace. This ordered and sequential integer appears as if the primary key field is extracted from a single dimension table with a self-referencing parent member column. With the identifiers being sequential, it makes it possible to compute offsets to pinpoint the row and column a measure should aggregate to in the *answer* array. Numeric identifiers provide superior advantages: determining member equality during the aggregation process is also faster than comparing

alphanumeric identifiers; fact tables stay compact as each tuple stores numbers for each dimension, rather than strings. These conditional properties of the Weather Dataset allow us to use nested arrays to store hierarchies and use array element indexes directly as member identifiers because they facilitate efficient member lookups, reduced aggregation time, and minimize fact table size for transmission.

These conditions break easily in real-life datasets. In reality, one hierarchy per dimension is possible in a dimension with a small set of members; however, this is rarely the case for every dimension in a database. In contrast to self-referencing dimension tables, a hierarchy level structure can span across multiple tables such as [*Product*], [*Product Subcategory*], and [*Product Category*]. Furthermore, sequential numeric primary keys are not always accessible in the context of a multidimensional dataset without explicit manual definition. In a typical OLAP server, members are unique in the context of a hierarchy namespace. For instance, in the [*Product*].[*Product Categories*] hierarchy the unique alphanumeric identifiers would be [*Product*].[*Product Categories*].&[*LCD Monitor*]&[*166*]; however, the same product in a different hierarchy such as [*Product*].[*By Model*] has an different identifier such as [*Product*].[*By Model*].&[*C2331*], even though both hierarchies belong to the same [*Product*] dimension. Thus, to generalize the metadata, we need to assume member identifiers change in different hierarchies.

In the original prototype, the user interface component to let the user create an ad hoc query contains no more than a handful of drop down boxes for axes and measure selections. Drop down boxes are easily bound to arrays that contain the metadata; however, they are inconvenient as controls for exploring members nesting on each other.

We adapt a different data structure to replace the *DimInfo* object, which uses a set of arrays to maintain the metadata. The open source *TreeView* component from *Yahoo! UI 2* [21] provides an intuitive way to select axes from the dimensions and hierarchies. The component complements the JavaScript OLAP Client by providing the following: a set of API methods to instantiate a tree data structure and tree nodes as objects, methods to access node level, children nodes, an ancestor of a node at a particular level, ability to add custom properties to a node, and dynamic loading of sub-tree on node expansion. We modify the component with additional recursive functions: retrieving leaf nodes of a given sub-tree; retrieving descendant nodes at a specific depth of a specific node; obtaining the size of a sub-tree; and a hash map of member keys to actual node objects. However, the use of *TreeView* comes with a cost - instantiating members as document objects rather than array elements incurs overhead associated with DOM traversal and additional

properties maintained by the *TreeView* component. We will elaborate the impact of these pros and cons throughout this section as we discuss other changes to the JavaScript application.

The issue of single hierarchy per dimension in the Weather Dataset is non-existent in the new JavaScript OLAP Client. Upon selecting a cube, the JavaScript OLAP Client fetches a list of dimensions and their hierarchies from the web service layer described in section 3.6, instantiates a tree, a set of dimension nodes, and hierarchy nodes under each dimension node.

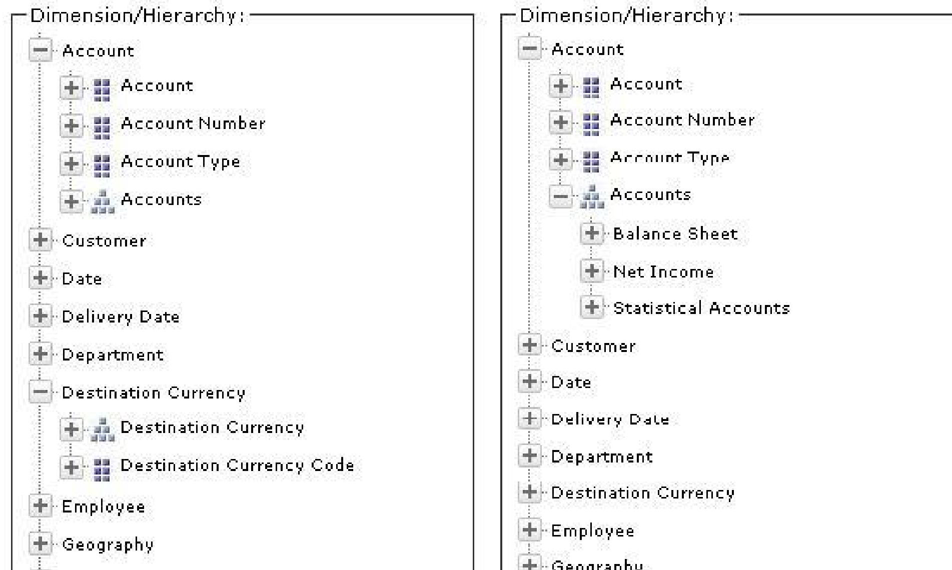


Figure 13 a) Before loading hierarchy content, b) [Accounts] sub-tree loaded.

Figure 13 shows an example of dimension nodes on the first level of the *TreeView*. The [*Account*] dimension contains four hierarchies, with the first three being flat member lists, and the fourth hierarchy [*Accounts*] containing a multi-level sub-tree. At this point, the JavaScript OLAP Client has not loaded actual content of each hierarchy from the server. If the client does that pre-emptively, it will take minutes to complete depending on the size of the entire metadata. Instead, the *TreeView* component allows us to supply a self-defined loading function dynamically when the user expands a node. This alleviates two potentially unforeseeable problems to the JavaScript application and the user. The first of these is described in section 3.6.1, where a hierarchy containing tens of thousands of leaf members can bog down the browser due to the number of DOM traversals as each member needs to be instantiated into a node object. When node expansion triggers the loading function, the JavaScript OLAP Client requests a hierarchy of acceptable height from the web service layer. The second problem is more of a user experience issue where the application is loading hierarchies which the user is not interested in at all. By

giving the user freedom of choice, both the client and server are refrained from wasting processing cycle on irrelevant dimensions.

2.5.2 Modified *getMatrixAnswer*

The modified *getMatrixAnswer* algorithm makes use of *TreeView* API and our custom functions to determine ancestor and descendant relationship between member nodes. Hence, the JavaScript OLAP Client no longer pre-computes the ancestors at every level of the hierarchy for every member. It still performs a linear table scan of the sub-cube data, but rather than returning a |*rowGroup*| x |*columnGroup*| array as the answer, the algorithm returns an *answer* object with |*rowGroup*| properties. Each property is itself an object of |*columnGroup*| properties whose values are the aggregated measures. This is because member identifiers are no longer sequential numbers and calculating offsets is not possible. Although we lose the efficiency of offset computation, we take advantage of the dynamic nature of the JavaScript language to augment objects at runtime. Therefore, by using member identifiers as object property names, quick assignment and retrieval of values is achievable without a linear search on object properties as seen in the following algorithm:

```
Algorithm getMatrixAnswer

var answer = new Object
// initialize the answer object by adding rowGroup member names as object properties
For i = 1 to rowGroup.length
   //initialize each property to an object of colGroup members
   answer[rowGroup[i]] = new Object
   For j = 0 to colGroup.length
     answer[rowGroup[i]][colGroup[j]] = 0;
   End For
End For

For i = 1 to FactTable.length
   current Tuple = FactTable[i]
   rowMember = currentTuple[rowAxis]
   colMember = currentTuple[colAxis]
   // lookup actual node in TreeView using member names
   rowMemberNode = KeyToNodeMap(rowMember)
   colMemberNode = KeyToNodeMap(colMember)
   // the aggregation point is the ancestor node at the level of the rowAxisNode + 1
   row = rowMemberNode.getAncestor(rowAxisNode.depth + 1)
   col = colMembernode.getAncestor(colAxisNode.depth + 1)
   if (row is a property of answer) AND (col is a property of answer[row])
      answer[row][col] += currentTuple[measureIndex]
End For
```

Figure 14. Modified *getMatrixAnswer* pseudo code based on *TreeView*

Chapter 3.6.2 introduces the notion of sub-cube fact table. For now, imagine a sub-cube fact table as a subset of the columns from the original fact table. Each tuple or row in the sub-cube fact table contains stripped down member names as unique keys in each hierarchy. Member names are irrelevant to the revised *getMatrixAnswer* algorithm: it must be able to determine the tuple with respect to a node in the hierarchy tree in order to perform aggregation. Although the

TreeView API provides several methods to search for a node on the tree, there is no question that performing such tasks for every tuple is a great performance hit. To improve the speed, we create a *KeyToNodeMap* index to obtain a quick reference to the associated node object given a dimension, hierarchy, and member key. Note that for the cross-tab axes nodes no lookup is required, as we obtain the actual references to the axis nodes when the user selects them from the *TreeView* interface.

2.5.3 OLAP Operators

Another major change to the JavaScript OLAP Client is to introduce additional functionalities of an OLAP tool. While the original prototype has the ability to compute a cross-tab, it does not implement other major operators, which are necessary to perform empirical experiments against a real-life web-based OLAP product. Numerous studies on the topic of OLAP and data warehouse have proposed a set of OLAP operators that exist in real-life OLAP products nowadays [9, 12, 14, 15, 22]. We do not implement all the operators in the way they work in real life systems because it is not our intention to build a fully-fledged OLAP product. With that said, we provide the following operators:

Cross-tab

It is common in OLAP tools to perform a cross-tab with rows and columns set to the entire hierarchy level. The original JavaScript prototype creates a cross-tab on the children of a *rowAxisNode* or *columnAxisNode*, which would only be a fraction of the members on a particular hierarchy level. To make a justified cross-tab comparison, we modified the *TreeView* component with functions to retrieve nodes at a specific level, and assign those nodes as the *rowGroup* and *columnGroup* supplied to the *getMatrixAnswer* algorithm.

Drill-down

In a cross-tab of [*rowDim*].[*rowHier*].[*rowMember*].*Children* and [*columnDim*].[*columnHier*].[*columnMember*].*Children*, a drill-down operation on a cell incident on a row member $r \in rowMember.Children$ and a column member $c \in columnMember.Children$ generates an embedded cross-tab where the row members are *r.Children* and the column members are *c.Children*. To implement this operation, we supply the *getMatrixAnswer* algorithm with [*rowDim*].[*rowHier*].[*r*] as *rowMember* and [*columnDim*].[*columnHier*].[*c*] as *columnMember*, receive an answer object, and bind it to a sub

grid inside the original data grid. Similarly, a drill-down operation on a row *r* or column *c*, we supply the *getMatrixAnswer* algorithm with *r.Children* and *c*, or *r* and *c.Children*, respectively.

Roll-up

There are two ways to perform roll-up in the JavaScript OLAP Client. The first is to roll-up after a cell drill-down. In the case of a server-centric model, the client must query the server again to obtain original cell value. In our case, we store the original cell value with the HTML element. When rolling-up a cell, we remove the sub-grid and restore that value at no additional cost. The other roll-up operation is to aggregate to a higher level in the hierarchy tree. By using the same axes but reselecting the row and/or column member levels, the *getMatrixAnswer* algorithm can return a roll-up aggregation result.

Slice and Dice

Slicing refers to the selection of a subset of members from a dimension and dicing refers to the custom selection of a subset of dimensions from the entire set. Note that these two operations can reduce the size of the sub-cube by several factors, and are highly desirable in a client-based aggregation scheme. In essence, we are performing slicing and dicing during the sub-cube retrieval process in our architecture: we let the user select a set of hierarchies from the dimensions and, optionally, a subset of members from those hierarchies that he/she is interested in, and retrieve a sub-cube to the local workspace.

Pivot

The application allows pivoting the cross-tab by swapping row and column axes with each other, or with any other hierarchy that has been part of the sub-cube retrieval. This operation is also supported in the sense that the user can dynamically swap current cross-tab axes with any hierarchies retrieved in the sub-cube.

3 Web-based OLAP Architecture

Realizing the shortcomings of our initial JavaScript OLAP prototype, we propose and implement an end-to-end architecture to support the revised JavaScript OLAP Client in section 2.5. This architecture leverages the power and resources of an existing OLAP server to alleviate, if not completely resolves the issues we identified with the initial prototype in section 2.4. We first give a high-level description of the traditional web-based client-server OLAP architecture in the first section. In the remainder of the chapter, we describe our proposed architecture and detail each component, and discuss several OLAP operations that are common in a lightweight user's workflow. Finally, we present experimental results of the workflow comparison between the two models, and the performance degradation of the JavaScript OLAP Client.

3.1 Traditional Web-based OLAP Architecture

Figure 15 depicts the current generation of web-based OLAP applications.

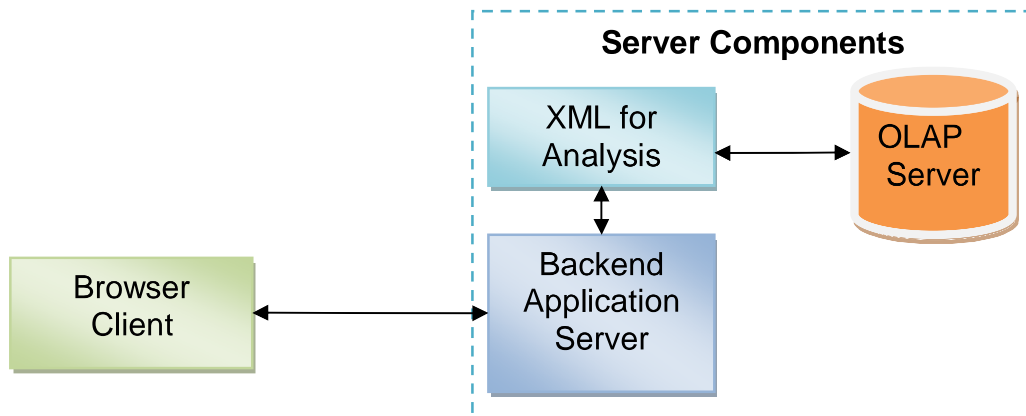


Figure 15. Server-centric OLAP Architecture

In this server-centric model, the "thin" browser client is merely an interface to display whatever the backend server passes on. In some implementations, such as xmla4js, the browser client communicates directly to the OLAP Server through the XMLA interface without a backend application server.

3.2 Overview of the Proposed Architecture

Figure 16 shows the component layout in the architecture we implemented.

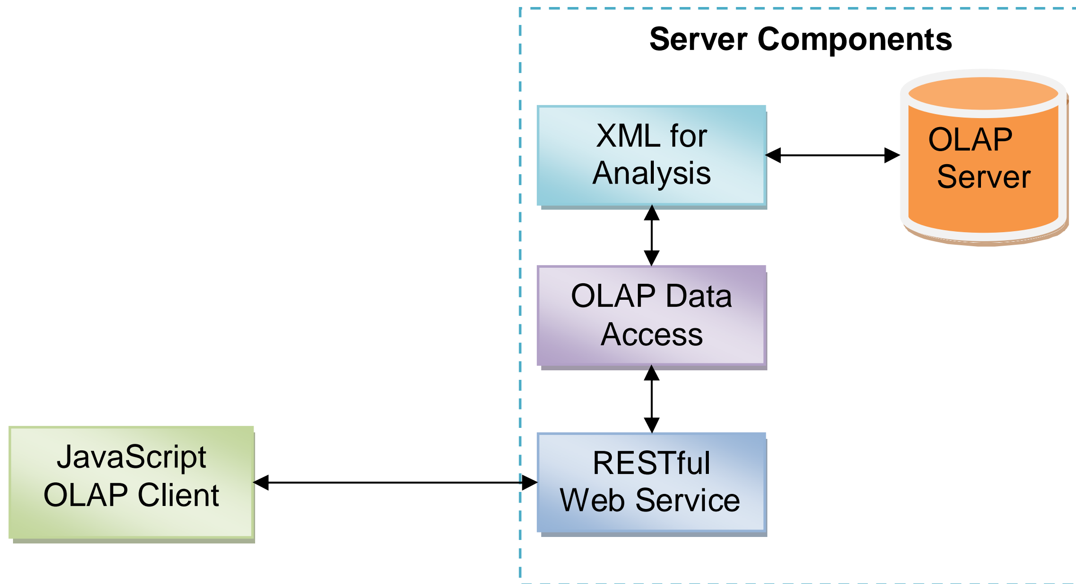


Figure 16. Client-centric OLAP Architecture

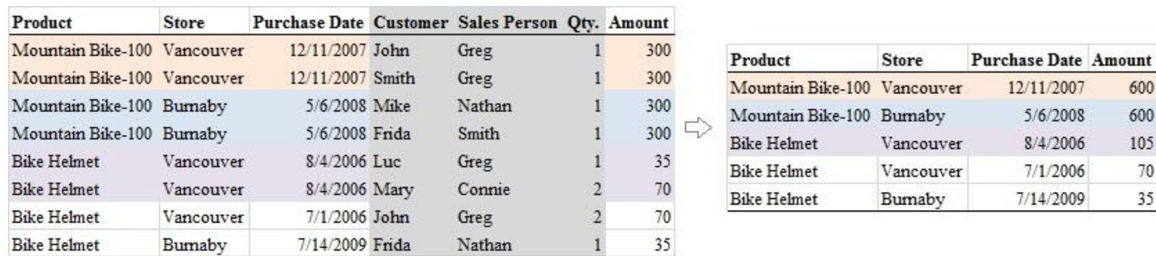
Message exchange between entities in the entire system is over the HTTP transport layer. Although not explored in this research, this architecture can benefit from HTTP features such as caching and security enforcement.

3.3 OLAP Server

Various aspects of OLAP, such as modelling multidimensional databases, aggregation and cubing, index selection, and join optimization, have been extensively studied by researchers for years [8-11, 13, 14]. In the business intelligence industry, there are plenty of vendors providing OLAP server products that offer myriad options for configuration and optimization tailored to the most frequently asked queries. Among these, partial pre-aggregation and leaf level optimization are two features we can leverage.

When we look back at the fact table transferred to the web-based client, we ask ourselves the question "is every row in the fact table required to compute an answer at any given granularity of the dimension hierarchy?" Consider the case where a user is only interested in exploring data on three dimensions and one measure from a fact table as shown in Figure 17. The reduced fact table through partial pre-aggregation is still capable of answering any query on these

three dimensions at every level of granularity. A take-only-what-you-need approach on the fact table can potentially reduce a huge portion of the dataset.



Product	Store	Purchase Date	Customer	Sales Person	Qty.	Amount
Mountain Bike-100	Vancouver	12/11/2007	John	Greg	1	300
Mountain Bike-100	Vancouver	12/11/2007	Smith	Greg	1	300
Mountain Bike-100	Burnaby	5/6/2008	Mike	Nathan	1	300
Mountain Bike-100	Burnaby	5/6/2008	Frida	Smith	1	300
Bike Helmet	Vancouver	8/4/2006	Luc	Greg	1	35
Bike Helmet	Vancouver	8/4/2006	Mary	Connie	2	70
Bike Helmet	Vancouver	7/1/2006	John	Greg	2	70
Bike Helmet	Burnaby	7/14/2009	Frida	Nathan	1	35

Product	Store	Purchase Date	Amount
Mountain Bike-100	Vancouver	12/11/2007	600
Mountain Bike-100	Burnaby	5/6/2008	600
Bike Helmet	Vancouver	8/4/2006	105
Bike Helmet	Vancouver	7/1/2006	70
Bike Helmet	Burnaby	7/14/2009	35

Figure 17. Reduced fact table through partial pre-aggregation

Multidimensional Expressions (MDX) is a query language for OLAP databases that is analogous to SQL, which is a query language for relational databases. Executing an MDX query on the OLAP server would yield such result set:

```
Select Descendants([Product].[All Products],, Leaves) on Axis(0),
Descendants([Store].[All Stores],,Leaves) on Axis(1),
Descendants([Purchase Date].[Date],, Leaves) on Axis(2)
From [Sales]
Where [Measures].[Amount]
```

Figure 18. Basic MDX query to obtain leaf level data

This query effectively creates a multidimensional result set that only contains the three pre-aggregated dimensions at the leaf member level. This is the first step we take to reduce the data required by the client, by retrieving a sub-cube from the OLAP server.

In order to provide the ability to execute MDX queries from another application, e.g. a browser, OLAP vendors have standardized an interface called *XML for Analysis*, or XMLA for short.

3.4 XML for Analysis Layer

Throughout this research, we used SQL Server Analysis Service as our OLAP server. The XMLA Application Programming Interface is not a proprietary format; it is an open standard initiated by Microsoft for connecting any application to a multidimensional data source [23]. It is widely adopted by all major OLAP service providers such as Oracle/Hyperion Essbase, IBM

Infosphere Warehouse Cubing Services, Microsoft Analysis Services, and Mondrian on top of MySQL.

The XMLA API allows application software to communicate with a data provider through web standards: HTTP, SOAP, and XML. XMLA is essentially a SOAP envelope with XML content sent over the HTTP Transport Layer. As SQL Server Analysis Service does not listen on HTTP ports, a driver named *msmdpump.dll* is set up in Internet Information Server to redirect any XMLA requests to the SQL Server Analysis Service instance [24]. In the XML content, we can specify two types of operations: *Discovery* and *Execute* [25].

Discovery messages allow an application to retrieve metadata on the OLAP Server. In our case, we are able to obtain dimension, hierarchy, member, and measure metadata directly from the OLAP server without prior knowledge of the relations or explicit manual definition.

As the name implies, *Execute* messages allow execution of any valid MDX query against the OLAP server. Typically, an OLAP client uses this method to obtain only the data required to answer a query. This brings us back to our original problem definition: if a user wishes to perform other operations such as drill-down or change filter values, subsequent queries must be sent per user action. In our architecture, we use it to obtain the lowest level of aggregation whenever possible.

The result of an XMLA *Execute* message is included in Appendix A. XMLA response is extremely detailed. In addition to the partially aggregated data we need, it includes detailed axes information associated with the result set. Such data is an overhead as we already have such information from *Discovery* messages. Removing this portion of the response before sending the dataset to the client minimizes transmission time and avoids the overhead for JavaScript to parse such data.

When XMLA receives an MDX query request such as the one in Figure 18, it returns a result set that is quite different from a traditional SQL query record set. It uses an ordinal value system to identify cells in a multidimension set. Typically, when an OLAP client uses MDX to query a cross-tab, it can use the cell ordinal to determine the row and column indices and display the result in a 2D grid. As the sub-cube that we retrieve has more dimensionality, and we require a data structure closely resembling the fact table used by the JavaScript prototype in section 2.1, a transformation process is required and explained in section 3.6.2.

3.5 OLAP Data Access Layer

The OLAP Data Access Layer is a server vendor agnostic layer for accessing OLAP data sources, developed by SAP – BusinessObjects. It is a Java library, implementing the XMLA protocol to provide methods for subsequent layers to consume XMLA without getting into the gritty details. The main functionality of the ODA library is to parse XMLA results and return convenient Java objects such as Cube, Hierarchy, Hierarchy Level, Tuple, MemberSet, and Member. Although there is a client-side JavaScript XMLA library, *xmla4js*, that provides similar functionality [26], we purposely move the XMLA processor to the server-side, to conserve resources on the JavaScript OLAP Client.

3.6 RESTful Web Service Layer

Representational State Transfer or REST, is a coordinated set of architectural constraints that focuses on minimizing network latency and communication and maximizing independent and scalable component implementations [27]. A web application is said to be RESTful if it confirms to being a: client-server, stateless, cacheable, uniformly interfaced, and layered system [28].

The RESTful Web Service (RWS) Layer is a middle tier between the JavaScript OLAP Client and the ODA Layer. This layer is created using jRuby, a Java implementation of Ruby. SAP – BusinessObjects originally created this application as an example of exposing the Java ODA library functions as web services. Enhancing its original functionality to explore OLAP server objects and execute MDX statements with ease, we seek to provide server side support for the JavaScript OLAP Client on the following aspects: (i) transformation of multidimensional data into fact tables; (ii) data caching; and (iii) dataset compression. Together with the JavaScript OLAP Client, they are the core of this research. The RESTful Web Service performs four major functions: server exploration, multidimensional data transformation, data caching, and data compression.

3.6.1 Server Exploration

In order to provide a basic interface to submit queries, the JavaScript OLAP Client must allow the user to explore the cubes, dimensions, hierarchies, and measures available on the OLAP server. The RWS Layer provides methods that are exposed to the JavaScript client in the form of URLs. Not limited to browsers, any client application that has the ability to send an HTTP request is able to take advantage of our RESTful Web Service. In the case of JavaScript, an

XmlHttpRequest object that is available on every browser is all that is required to consume the endpoint.

Ruby by itself is just another programming language with a different expressive syntax than conventional programming languages. However, by adding a Sinatra extension, it immediately provides a server-side web application framework for developing web services. Without the encumbering service declaration, definition and registration of web services written in other programming languages, Sinatra allows Ruby to define HTTP request blocks that look like the following:

```
GET '/ssas/:user\::pass@server/:catalog/:cube/:dimension' do
        Execution_block
End
```

When the RWS receives a request, it runs the *Execution_block* in which the request URL matches a defined directive. Once a directive is matched, it binds parameters such as *catalog*, *cube*, and *dimension* to actual names in the URL path. For example, *http://myServer/ssas/user:pass@myServer* will return a list of catalogues available on *myServer* and *http://myServer/ssas/user:pass@myServer/Adventure Works DW* will return a list of cubes in the *Adventure Works DW* catalogue. Manipulating the directive names in the JavaScript OLAP Client allows it to send requests to the RWS. The web service then specifies object names and invokes ODA methods to send XMLA *Discovery* messages.

Because the number of catalogues, cubes, dimensions and measures in an OLAP application is typically much less than one hundred, when the ODA Layer returns a response to the RWS, the web service serializes the response into a *JSON* string, readily available for JavaScript to parse and instantiate into Arrays. Hierarchy trees, such as the product hierarchy of a supermarket, may contain members in the order of tens of thousands. This raises two issues: (i) transferring the tree structures efficiently to the client and (ii) instantiating the products as JavaScript document objects will crash the browser.

Concerning the first issue, the RWS generates a list of hierarchy members by taking a pre-order traversal of the tree. For each hierarchy member, we include the *level* of the member, *MemberCaption*, and *UniqueMemberName* delimited by the vertical bar character:

```
0|All Products|All Products
1|Accessories|Category.4
```

2|Bike Racks|Subcategory.26

3|Hitch Rack - 4-Bike|Product.483

2|Bike Stands|Subcategory.27

3|All-Purpose Bike Stand|Product.486

2|Bottles and Cages|Subcategory.28

3|Water Bottle - 30 oz.|Product.477

3|Mountain Bottle Cage|Product.478

As member captions are for display purposes and are not unique in each hierarchy, they are unreliable for accurate aggregation. Therefore, we include *UniqueMember Name* in the metadata. This compact representation of the tree yields minimum transmission time to the client-side as no tree node appears twice in this list. When the JavaScript OLAP Client receives this response, it parses the result by maintaining a stack to keep track of the current list of ancestors, and rebuilds the tree in one pass.

For the second issue, when the JavaScript OLAP Client requests a hierarchy h from dimension d, the RWS obtains the hierarchy level breakdown from the ODA Layer. With knowledge of member count on every level of h, and a predetermined tolerance τ, the wrapper returns a sub-tree t of h from the root r to a level l where the member count in l in h is at most τ.

Although the choice of τ is not part of our research, it should be a function of several parameters: browser version, JavaScript engine, and device hardware (such as CPU and memory). There is a trade off with the value of τ: the higher the tolerance, the larger is the sub-cube provided by the RWS, which offers aggregation at low granularity. With low tolerance, the RESTful wrapper can only provide aggregation at higher levels of the hierarchy and shallow drill-downs, although it is capable of providing reasonable responses to devices with slower JavaScript engine and hardware specs.

Note that we are strictly discussing hierarchies with high branching factor, not fact tables with massive amount of transactions. In real life data warehouses, usually the product and customer dimensions are on such scale and, even in such cases, obsolete products and inactive customers can be pruned from the hierarchy trees.

3.6.2 Multidimensional Data Transformation

Recall in section 2.1, we described the in-memory dataset used by the JavaScript prototype as an n x ($|D|$ + $|M|$) array, which corresponds to a fact table in the data warehouse used to build a cube. We used the MDX query such as the one in Figure 18 embedded in an XMLA *Execute* message to fetch a sub-cube; however, there are still two potential problems with this approach. First, specifying the measure in the slicer axis (*where* clause) only allows retrieval of one measure. In many occasions, multiple measures are desirable. For instance: visualizing the results in a scatter plot, or performing calculations using multiple measures. Second, we mentioned in the previous section that in some cases, fetching every single leaf member is not possible, as it takes too long for the OLAP server to perform partial-aggregation and the JavaScript client cannot digest that many document objects.

We modify the sub-cube retrieval routine and generalize it to the following:

```
Select {measure1,... measurem} on Axis(0),
Descendants([dimension1].[hierarchy1].[root], distance1, Leaves) on Axis(1),
...
Descendants([dimensionn].[hierarchyn].[root], distancen, Leaves) on Axis(n),
From [cube]
```

Instead of using the measure as the slicer, we treat measures as dimensions and create a set of desired measures on axis 0. Subsequent axes are then bound to the desired dimensions. Since we want to answer queries at various levels of granularity with respect to each dimension, we need to fetch members from as far down the hierarchy tree as possible. By specifying the absolute distance and the keyword *Leaves*, the *descendants* function returns the set of member nodes that are of distance $distance_i$ from [$dimension_i$].[$hierarchy_i$].[*root*] and, in addition, leaf nodes that are closer than $distance_i$ from the root. In trees such as the one in Figure 19 that are unbalanced, this approach effectively captures short branches. Without the leaf members from shorter branches, a cross-tab on high level would generate incorrect aggregation.

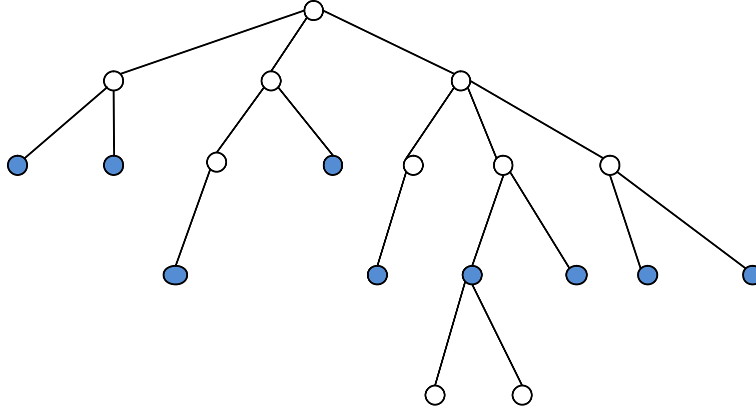


Figure 19. Member node selection with *descendants*(*root*, *3*, *leaves*)

The multidimensional result set is a sub-cube of [*cube*], of which cells are addressed as *CellOridinals* in a row-major format within a sub-cube [29]. Conceptually, a *CellOrdinal* value is a number assigned to a cell as if the result set is a *p*-dimensional array where *p* is the number of axes. If axis *k* has U_k members, the ordinal number of a cell whose tuple ordinal are (s_0, s_1,s_2,… s_{p-1}) is:

$$\sum_{i=0}^{p-1} s_i \cdot e_i \text{ where } e_0 = \perp \text{ and } e_i = \prod_{k=0}^{i-1} U_k$$

Figure 20. CellOrdinal calculation

To minimize modifications required on the JavaScript OLAP Client so that it can perform aggregation as if the multidimensional result set is a fact table, we transform the sub-cube into a fact table on the RWS Layer. Consider the following excerpt from an XMLA response to the query in Figure 18:

```
<CellData>
  <Cell CellOrdinal="0">
    <Value xsi:type="xsd:double">1.35E3</Value>
    <FmtValue>$1,354.59</FmtValue>
  </Cell>
  …
  <Cell CellOrdinal="1793">
    <Value xsi:type="xsd:double">2.4567E2</Value>
    <FmtValue>$345.67</FmtValue>
  </Cell>
</CellData>
```

Each CellOrdinal is associated with a tuple/cell in the sub-cube space. A CellOrdinal number is calculated by the formula in Figure 20. Conversely, if we know the CellOrdinal number of a tuple, the number of axes, and the cardinality of the member set for every axis, we can determine the tuple ordinal ($measure_1, \ldots measure_m, s_1, \ldots s_n$) where s_i = [$dimension_i$].[$hierarchy_i$]. For instance, a value of 345.67 is associated with CellOrdinal=1793, which could mean the first member in the first axis, the twenty-second member in the second axis, and the fifty-eighth member in the third axis. With the axis member set in the XMLA response, this particular tuple translates to "$345.67 is the [*Measures*].[*Amount*] of all purchases of [*Product*].[*All Products*].[*Generic Keyboard*] in [*Store*].[*All Stores*].[*California*]&[*Irvine*], that is delivered on [*Delivery Date*].[*Date*].[*2007/12/18*], from the [*Sales*] cube". We move the first *m* elements that denote the measures to the end of the *n* dimensions. We insert this tuple as a row into a sub-cube fact table with the reduced *UniqueMemberName* as column values. We perform reductions on the *UniqueMemberName* for two obvious reasons: first, to minimize the amount of data transmitted, and second because the fully qualified name is redundant, as we already know the dimension and hierarchy from the column header. The dimension and hierarchy names of the axes serve as the headings of the sub-cube fact table. Table 2 shows portions of a sub-cube table by transforming the multidimensional result set.

[Product].[All Products]	[Store].[All Stores]	[Delivery Date].[Date]	[Amount]
Generic Mouse	[BC]&[Vancouver]	2007/12/15	123.45
Generic Keyboard	[Ontario]&[Ottawa]	2007/12/16	44.15
Generic Keyboard	[Ontario]&[Toronto]	2007/12/16	533.21
Generic Keyboard	[California]&[Irvine]	2007/12/18	345.67

Table 2. Multidimensional result set transformation

The potential size of the sub-cube fact table is worth noting. Theoretically, if the [*Sales*] cube is extremely dense, the multidimensional result set would return a CellOrdinal for nearly every combination of members in every hierarchy requested. This implies the size of a *k*-axis sub-cube fact table is equal to:

$$\prod_{i=1}^{k} | M(A_i) |$$

where A_i is the *dimension.hiearchy* selection on axis *i* and $M(A_i)$ is the descendant member set in A_i.

After the RWS creates the sub-cube fact table, it serializes it to a JSON string before sending it to the client. Similar to the metadata download, the JavaScriptOLAP Client uses *JSON.parse* and directly instantiates the XmlHttpRequest response string into a 2-dimensional Array Object.

3.6.3 Data Caching

In contrast to an Online Transaction Processing application (OLTP), where transactions frequently update the database, an OLAP application optimizes data retrieval data for responsive query answering. In other words, data tend to stay static until a periodic scheduled update. It then makes sense to retain a copy of the previously fetched dataset and store it locally at the server.

Caching can be implemented on the RESTful Web Service Layer, the JavaScript OLAP Client, or both. Although current browsers already support local/web persistent storage in the HTML5 draft, the disk quotas are limited to three to ten megabytes per domain, which is easily exceeded. The limit is set in stone unless we modify an open source browser. Another issue with a client-side cache is that when the JavaScript client receives a new dataset, it needs to decide whether to perform caching and, in the event that it has reached the quota limit, it needs to determine which datasets to swap out. Furthermore, when the user wants to fetch a sub-cube, the JavaScript client needs to validate the local dataset version with the RESTful wrapper. To alleviate the JavaScript client from these overheads, we experimented with server-side caching.

When the JavaScript OLAP Client sends a request to the RESTful Web Service Layer to fetch a sub-cube, the sub-cube's uniqueness is determined by a few properties: the number of axes in the dataset, the hierarchies specified for those axes, the depth of the hierarchy retrieved, and the measure selections. These properties can be derived from the MDX statement in the query. When the RESTful web service has this information, it uses a hash function to obtain a file name and check for the existence of that file in a repertoire of datasets on disk. The sub-cube is returned if it has been previously cached; otherwise, the web service requests a new sub-cube from the OLAP server. When the OLAP server returns a result set, the RWS sends a transformed and compressed sub-cube fact table to the client and caches an identical copy, identified by the file name produced by the hash function.

Note that although we do not explore such options in this research, the cached sub-cube fact table does not necessarily have to be an exact match of the client request. As long as the measure selection is identical, a cached dataset that is a superset of the client request is sufficient;

however, the RWS needs to decompress the cached copy, remove and reorder columns to match the client request, and recompress the sub-cube fact table before returning it to the client.

3.6.4 Dataset Compression

Compressing the dataset before sending it to the client has several advantages. First, energy consumption associated with sending and receiving data is a major concern for mobile devices and usually trumps the overhead incurred by client side decompression cost. Second, dedicated efforts to minimize network usage add up to drive down business operating cost. Finally yet importantly, the reduction in transmission time is favourable from a user experience perspective.

On the premise that the server is responsible for compression, there are three ways of decompressing the dataset in a web-based OLAP application:

i. Leave compressed dataset in browser memory to minimize memory footprint, and decompress individual records as required by aggregation on the fly.

ii. Decompress the whole dataset as soon as JavaScript receives it.

iii. Decompress with the browser before passing dataset to JavaScript.

JavaScript does not handle binary data efficiently in comparison to programming languages that are closely tied to physical hardware [30]. Although we do not investigate in this research, it is very likely that (i) and (ii) will incur further cost to aggregation time that outweighs any benefits and makes client-side aggregation a moot point. On the other hand, web browsers natively support two compression schemes: *gzip* and *deflate*. If a web server returns a response to the client and indicates that the *http-content-encoding* is *gzip* or *deflate*, the client will automatically decompress the content at the browser level, which of course is pre-compiled binary. In fact, compressing http content at the browser level is the norm nowadays for any type of web application. Figure 21 illustrates this process with a *gzip* compression.

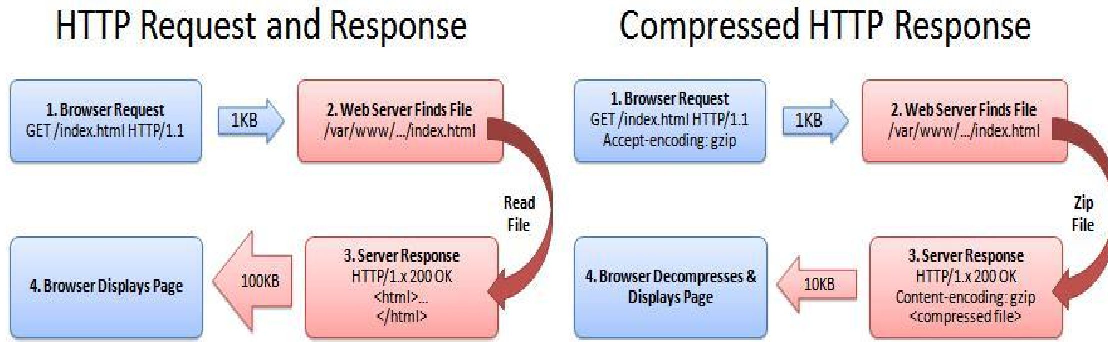


Figure 21. Compression at HTTP level [31]

We chose the third approach and adapted *zlib* [32], an implementation of *deflate* as the sub-cube compression scheme. *Deflate* is a lossless general purpose compression method that combines the *LZ77* algorithm and *Huffman Coding* [33]. The following figures show the result of compressing the datasets.

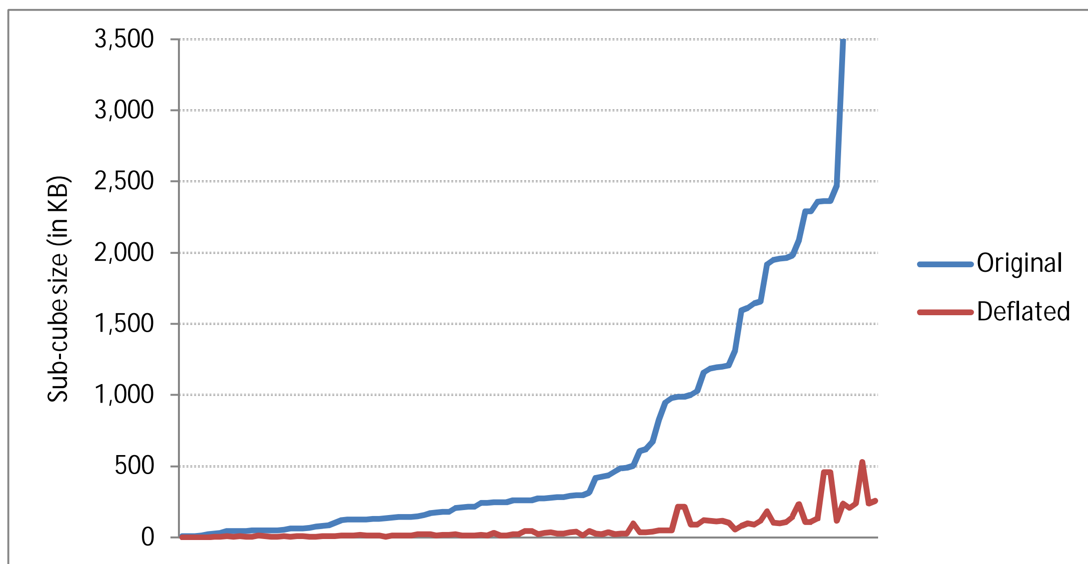


Figure 22. Original versus deflated sub-cube size

Throughout this research, we generated 111 unique sub-cubes from two synthesized datasets and one real-life dataset. In this graph, the horizontal axis represents the sub-cubes sorted according to their original sizes. The vertical axis shows the effect of deflating the sub-cubes before the RWS sends them to the client. The average compression rate is 9.37% of the original sub-cube size with $\sigma = 4.36$. Strong evidence shows that sub-cube compression is highly effective on the nature of all the datasets we use in this research. We achieve our goal of reducing dataset transmission time and bandwidth consumption while incurring only an initial minimal decompression cost.

3.7 Lightweight OLAP User Workflow

We compare the performance of a client-centric system on basic OLAP operators such as cross-tabs, random drill-downs, and roll-ups. These operations are common and frequently executed by a lightweight OLAP user. We run 74 OLAP operations where each operation is repeated at least five times per network latency setting. Workflow operations are categorized by measure groups, which are sets of related dimensions and measures, and are not mutually exclusive. Table 3 shows the testing environment.

<table>
<tr><th></th><th>JavaScript OLAP Client (JSOC)</th><th>Server-based Client (SBC)</th></tr>
<tr><td>Browser Platform</td><td colspan="2">Firefox 3.6.10</td></tr>
<tr><td>Front-end Machine</td><td colspan="2">Core 2 Duo E6750 @ 2.66GHz,
3.25GB of Ram,
Windows XP Professional 32-bit SP3</td></tr>
<tr><td>Back-end Server</td><td>Core 2 Duo E6750 @ 2.66GHz,
3.25GB of Ram,
Windows XP Professional 32-bit SP3</td><td>Windows Server 2008 R2 64bit
48 GB Memory
Intel Xeon X5650 @ 2.67GHz, 2 processors, 24 cores total</td></tr>
<tr><td>OLAP Server</td><td colspan="2">Windows Server 2003 64-bit SP2
SQL Server Analysis Service 2005
12 GB Memory
Intel Xeon E5320 @ 1.86Ghz, 2 processors, 8 cores total</td></tr>
<tr><td>Dataset</td><td colspan="2">REAL Warehouse Sample V6 MT</td></tr>
<tr><td>Network Latency</td><td colspan="2">The front-end machine will be simulated to have approximately 30/90/150 ms roundtrip latencies to the back-end servers.</td></tr>
</table>

Table 3. Testing environment

In addition to the load time and script time we described in section 2.4, note that load time is a one-time cost for the JavaScript OLAP Client during the sub-cube download before a cross-tab is generated; it is incurred on a server-centric model per OLAP operation. Script time is measured for our JavaScript OLAP Client and is assumed negligible in the case of the server-centric model.

```
C:\Documents and Settings\C5143631\Desktop\Dummynet\binary>ping -n 4 10.165.27.24

Pinging 10.165.27.24 with 32 bytes of data:

Reply from 10.165.27.24: bytes=32 time<1ms TTL=127
Reply from 10.165.27.24: bytes=32 time<1ms TTL=127
Reply from 10.165.27.24: bytes=32 time<1ms TTL=127
Reply from 10.165.27.24: bytes=32 time<1ms TTL=127

Ping statistics for 10.165.27.24:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
######################################################################
## Setting delay to 15ms for both incoming and outgoing ip packets ##
######################################################################

C:\Documents and Settings\C5143631\Desktop\Dummynet\binary>ipfw pipe 3 config delay 15ms mask all

C:\Documents and Settings\C5143631\Desktop\Dummynet\binary>ipfw add pipe 3 ip from any to any
00100 pipe 3 ip from any to any

C:\Documents and Settings\C5143631\Desktop\Dummynet\binary>ping -n 4 10.165.27.24

Pinging 10.165.27.24 with 32 bytes of data:

Reply from 10.165.27.24: bytes=32 time=29ms TTL=127
Reply from 10.165.27.24: bytes=32 time=30ms TTL=127
Reply from 10.165.27.24: bytes=32 time=31ms TTL=127
Reply from 10.165.27.24: bytes=32 time=30ms TTL=127

Ping statistics for 10.165.27.24:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 29ms, Maximum = 31ms, Average = 30ms
```

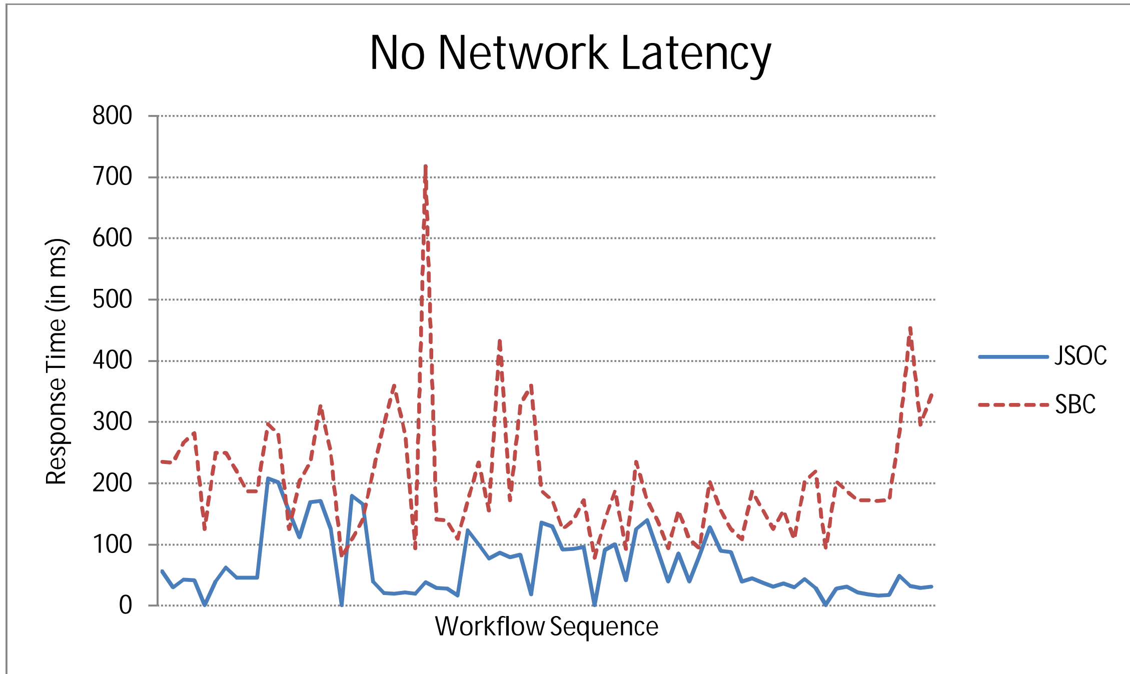
Figure 23. Traffic shaping with *Dummynet* [34]

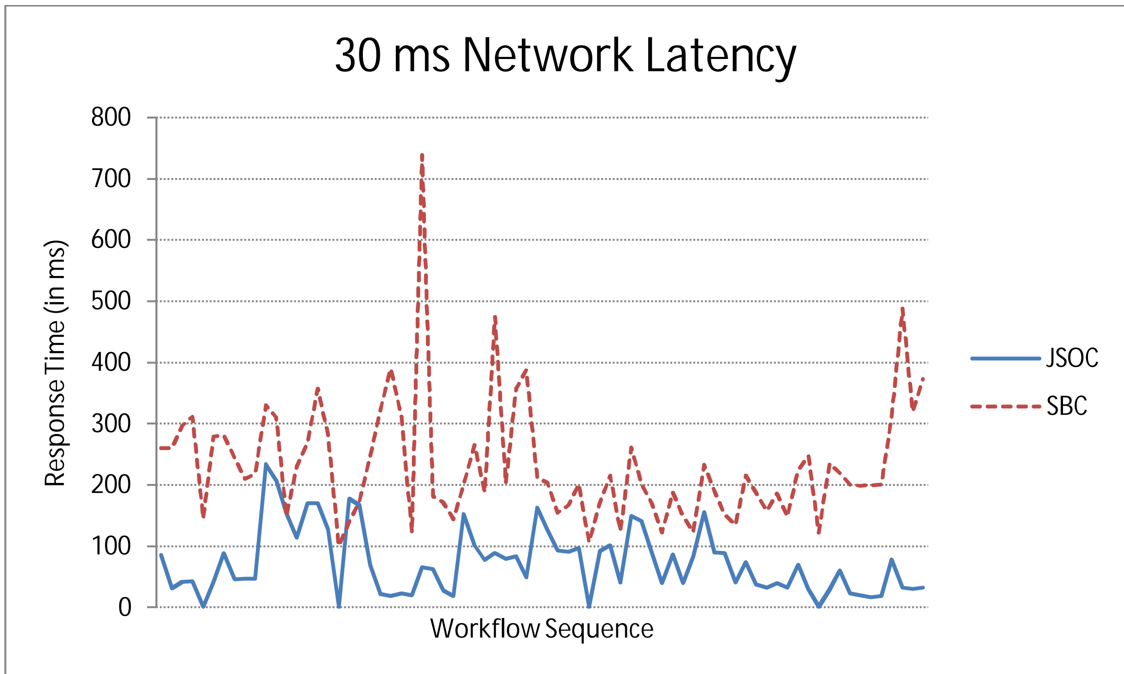
Network traffic is shaped using *Dummynet* and is verified by sending a ping request to the backend OLAP server before and after lag injection, as seen in Figure 23. Details of the workflow operations in each measure group can be found in Appendix D and the dataset used can be found in Appendix C. We summarize the average response time of the test results. As expected, incremental network latency affects the server-centric model by a constant amount because every OLAP operation request is sent to the server. The parameter has a lesser impact on the JavaScript OLAP Client as the only time the client needs to communicate with the server is during the initial sub-cube retrieval.

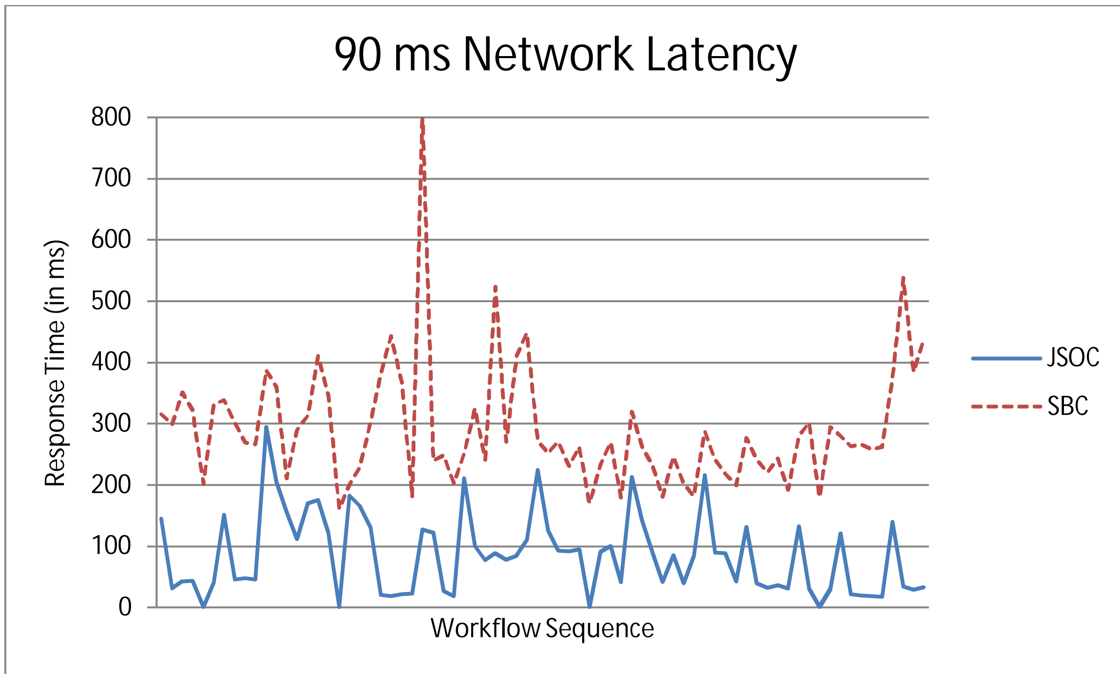
	Network Latency (in ms)							
	0		30		90		150	
Measure Group	JSOC	SBC	JSOC	SBC	JSOC	SBC	JSOC	SBC
Store Sales	75	229	80	258	93	313	110	374
Item Vendor	81	265	90	296	107	353	146	417
Distribution Center Inventory	87	144	91	174	100	236	116	298
Store Inventory	29	197	39	235	50	294	63	354

Table 4. Average performance time on model comparison

The following set of figures shows the response time of the two models with respect to every OLAP operation in the workflow sequence:







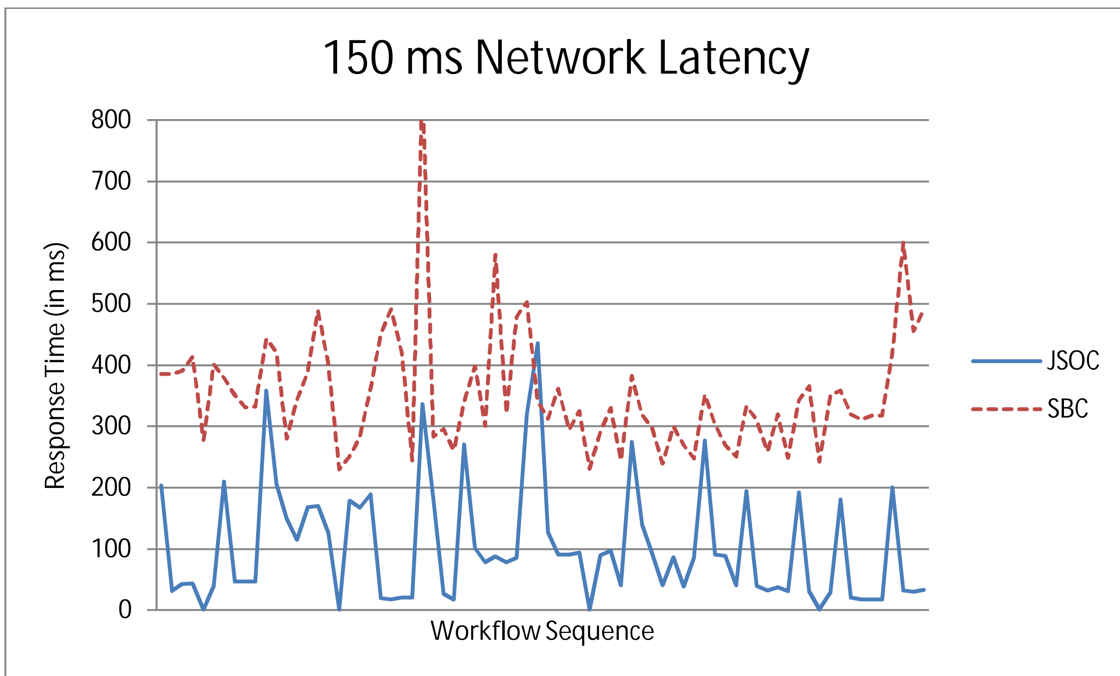


Figure 24. Performance time comparison on workflow sequence

These figures show that, for the server-centric model, the server performance remains constant, but the lines shift upward as the network latency increases. In the case of the JavaScript OLAP Client, we observe spikes that are affected by injected latency. This phenomenon results from the order of executing the OLAP operations and the way sub-cubes are retrieved. In a way, this is a worst case scenario where a fetch is done before every cross-tab generation. In an ideal

case, a user can combine the dimensions that he/she is interested in and download them in one sub-cube. Such a sub-cube is capable of generating various combinations of cross-tabs, and facilitates drill-down and roll-up operations without fetching additional data. In our experiment, we force the client to download a new sub-cube before each and every cross-tab in order to avoid such bias, which would otherwise reduce the number of spikes in the JavaScript OLAP Client series.

It is also worth noting that when the RESTful Web Service component requests the sub-cubes from the OLAP server, there is a cost to generate the dataset. For example, there are 15 sub-cubes retrieved in this workflow sequence. The average time for the OLAP server to generate these partial pre-aggregated results is 27.9 seconds. This may seem like a steep cost that outweighs the advantages of client-side aggregation; however, recall that one of the functionalities implemented in the RWS layer is caching. When we repeat the workflow sequence using simulated network latency of 0/30/90/150 milliseconds, the sub-cubes are retrieved from a disk cache that takes 1 millisecond or less. In a real world application, there are at least three ways to relieve this overhead from the user. First of all, modern OLAP servers have the option of optimizing multidimensional data by pre-aggregating at various levels of granularity, or by not aggregating at all. A server optimized at leaf level pre-aggregation would complement our sub-cube retrieval method. Second, an OLAP server is updated at regular intervals, after each data integration; it can preemptively compute, transform, and store frequently asked sub-cubes and store it on a server hard disk. Third, as mentioned in section 3.6.3, the RWS can be modified to fetch a cached sub-cube (of which the dimension/hierarchy and measures are a superset of the requested sub-cube), splice and rearrange the order of columns, and return it without sending a request to the OLAP server.

With regards to network consumption, we track the total amount of data transmitted between the two models in each measure group in Table 5 and conclude that the front-load cost of the JavaScript OLAP Client does not necessarily consume more network bandwidth compared to the server-centric model. As a matter of fact, with more operations performed on a sub-cube, the overall network usage is considerably less than that of a server-centric model.

	Network Usage (in KB)	
Measure Group	JSOC	SBC
Store Sales	259.9	383.5
Item Vendor	94.3	176.4
Distribution Center Inventory	344.2	219.6
Store Inventory	65.4	254.0

Table 5. Total data size transferred

3.8 Performance Degradation with Respect to Sub-cube Size

Recall the prototype engine in section 2.3. The query processing method of the prototype is similar to that of a main-memory OLAP engine running on a server: perform table scan of the fact tables. While a server-side main-memory OLAP engine can digest massive table sizes, not all browsers can efficiently handle table sizes anywhere near a million rows using the same method. Given that fact tables can grow by thousands or more records from daily business operations, performance degradation of fact table linear scanning is inevitable in a browser environment.

In our proposed sub-cube fact table approach, it does not matter how large the database is in terms of number of tuples in the fact table. What really matters is the size of the downloaded sub-cube. There are two ways in which the sub-cube size may explode to a point that the JavaScript OLAP Client using table scan query processing cannot outperform the server-centric model. The objective in this section is to evaluate such performance degradation.

Sub-cube Size Explosion by Introducing Additional Slices

Consider workflow sequence number 21 to 25 from Appendix D. Without simulating network latency as a parameter, the following table shows the response time of the server-centric model to perform the five OLAP operations and the performance of the JavaScript OLAP Client with respect to different sub-cube sizes retrieved from the RWS.

		JavaScript OLAP Client							
		Sub-cube size							
Workflow	SBC	5,263	7,395	12,575	16,772	26,660	31,192	36,975	75,247
21	219	37	53	90	119	192	243	275	431
22	297	21	30	49	66	105	134	155	287
23	359	20	30	48	65	104	134	154	287

Workflow	SBC	5,263	7,395	12,575	16,772	26,660	31,192	36,975	75,247
24	281	22	32	49	66	103	141	162	292
25	94	20	30	46	63	106	120	149	288

Table 6. Performance degradation with respect to sub-cube size

One may wonder why the sub-cube size can increase by an order of magnitude or more, which reveals the drawback of letting the user to slice the cube at will. When the user downloads more dimensions than necessary to answer his cross-tab query, the size of the dataset increases because of amplified granularity. To answer the above workflow sequence, the minimal sub-cube only requires two dimension slices from the entire cube: the [*Time*] dimension diced to year 2004, and the [*Item*] dimension diced to "Fiction" category. However, if the user intentionally or inadvertently takes slices of additional dimensions or dices a larger set of members, the number of tuples in the sub-cube fact table increases as shown in Table 7.

Number of Tuples	Dimension/Hierarchy Change
5,263	Minimal sub-cube download
7,395	Add [*replen strategy*].[*strategy type*] hierarchy
12,575	Add [*customer*].[*years since purchased*] hierarchy
16,772	Add [*replen strategy*].[*title type*] hierarchy
26,660	Add [*store*].[*store status*] hierarchy and [*vendor*].[*vendor type*] hierarchy
31,192	Add [*store*].[*store*] hierarchy
36,975	Add [*vendor type*].[vendor type] hierarchy and [*replen strategy*].[*strategy type*] hierarchy
75,247	Include all book instead of "Fiction" category

Table 7. Sub-cube size explosion

Sub-cube Size Explosion due to Increased Member Set

Another possibility for retrieving a sub-cube of larger size is to include additional members in the sliced dimensions. Referring back to the five OLAP operations in the previous section, the default retrieval method is to fetch descendants of the [*Item*].[*By Category*] hierarchy down to the [subcategory] level, and not to the leaf level, which contains a set of 43,951 members. The rationale for this is described in section 3.6.1. Consider the scenario if we fetch a subset of the 43,951 members. By increasing the number of members in this dimension, the sub-cube size grows extremely fast as the member set size in the [*Item*] dimension is multiplied by the cross product of the member set sizes of other sliced dimensions. This has a profound impact on

the size of the sub-cube unless the original cube is sparse, in which case the non-zero cells would be relatively low.

		JavaScript OLAP Client				
		Sub-cube size				
Workflow	SBC	5,263	8,575	14,772	27,193	51,247
21	219	41	62	110	229	367
22	297	22	40	59	129	243
23	359	22	39	61	125	229
24	281	25	39	60	119	235
25	94	23	37	61	128	232

Table 8. Increased member set size

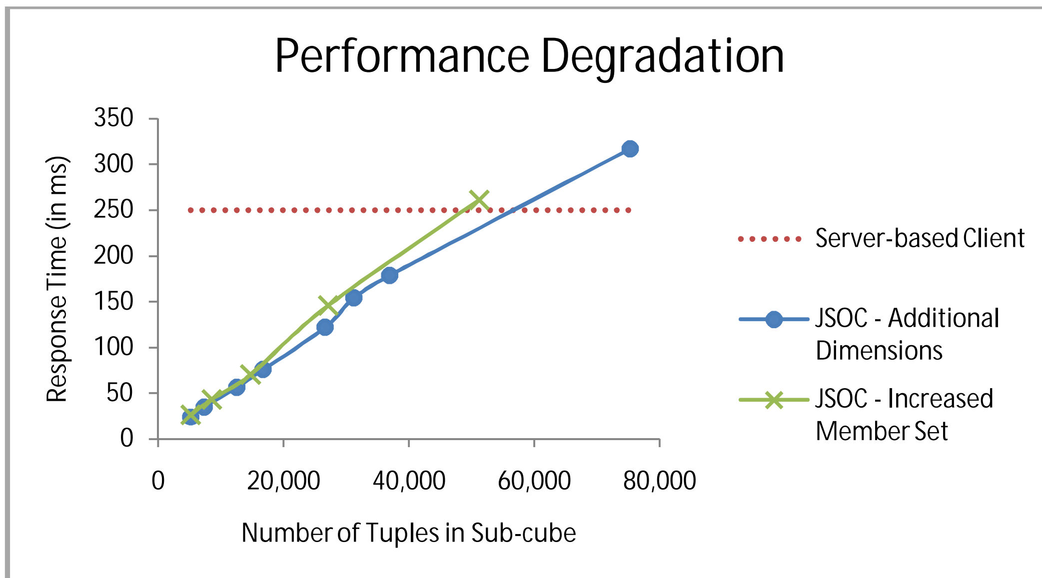


Figure 25. Performance degradation

We average out the time over these five operations and plot out the three series: server-based client, JavaScript OLAP Client: sub-cube size increase by addition dimensions and sub-cube size increase by larger member set. Figure 25 shows the points at which the same query takes longer for the JavaScript OLAP Client to perform local aggregation than a server-centric model. In this case, client-side aggregation will incur a performance penalty as the current algorithm performs a linear scan on the tuples in the sub-cube, while server-side aggregation only focuses on members explicitly specified in a query. Although the threshold varies depending on the actual query executed, dataset and metadata used, and client/server hardware, these experiments show similar results, with the degradation being slightly faster in the case of

increased member set size. We believe this is because when we expand the member set by taking more members from deeper levels of the hierarchy tree, it takes slightly more time to determine the ancestor member at the aggregation level.

4 Calculations and Data Visualization

While lightweight OLAP users mainly generate cross-tabs, OLAP power users often demand queries that involve calculations such as top-k percentage contribution over m-time span, and m-day moving average. In this chapter, we explain these two calculations, algorithms used to carry out these calculations, and the data visualizations applied. With extensive testing, our goals are (i) to compare the JavaScript OLAP Client's performance against a simulated server-centric web-based OLAP application to perform these calculations; and (ii) to show that performing successive calculations against a server-centric model would introduce jitter in the presentation layer, if not completely overwhelm the OLAP server.

In addition to these two calculations, section 4.4 introduces a novel client-side aggregation technique in common data visualization – a scatter plot. We implement this technique on the conventional vector-graphic based library and the new HTML raster-based library, and demonstrate that HTML5 makes a JavaScript-based Rich Internet Application a strong candidate as an OLAP client.

4.1 Data Visualization in Traditional Web-based OLAP

We do not use run tests against SAP's server-based client because even though it has the functionality to send calculation queries, we cannot achieve the second goal. In current generation of web-based OLAP clients, the server renders graphical data visualization in real-time and sends it as a static image to the client. As shown in previous sections, current JavaScript engines are powerful enough to render graphics on the browser. To compare apples to apples, it would be more interesting to test our client-centric model against a model where the server performs the calculations and the client renders the results using the same data visualization. To that end, we simulate a web application that has minimal script execution on the browser side and that uses the OLAP Data Access layer used by the server-centric model to fetch computed results from the same OLAP server. Common across all tests, all clients and servers are on the same network where the network latencies between any entities are no more than one millisecond. In the top-k percentage contribution and m-day moving average calculation tests, we inject three

levels of network latency to simulate clients connecting to a remote server and observe the impact to query response time.

4.2 Top-*k* Percentage Contributions Over *m*-time Span

This calculation is essentially a ranking problem. We introduce another function by modifying g*etMatrixAnswer* to perform group-by on one axis rather than two. During scanning the fact table for tuples that fall within the *m*-time span, we perform a group-by on the single axis on the specified measure. Consider the following example: a user requests the top ten item subcategories and their percentage contribution to the total sales in the past two weeks. In this case, we retrieve a sub-cube that contains three dimensions: [*Item*] at the [*Subcategory*] granularity, [*Time*] at the [*Calendar Date*] granularity, and the measure [*Sales*]. Using *getMatrixAnswer* to group-by on the [*Item*].[*Subcategory*] axis, filtered by tuples with [*Time*].[*Calendar Date*] in the past two weeks. When the aggregation ends, we obtain an object *Answer* that contains a list of <*item subcategory*: *sale amount*> as its <*property*: *value*>, as well as a running total of the sale amount of every item subcategory. Then we select the top 10 item subcategories ranked by their sale amount using *SelectTop*(10, *Answer*) in Figure 26.

```
Algorithm SelectTop (k, Answer)

Array TopKAnswer;
For (each property in Answer)
    TopKAnswer.insert({property, Answer[property]}, k)
return TopKAnswer;
```

Figure 26. Top-*k* selection

In this example, the algorithm maintains a static array of size ten. It loops through the properties of *Answer* and insert the <item subcategory, sale amount> into this array using the following insert custom *insert* method.

```
Array.insert = function (object, k) {
   For i = 0 to Array.length – 1
      If (object.value >= Array[i].value) {
         // insert at i and shift all elements after i
         Array.splice(i, 0, object);
         break;
   End For
   //return only the first k elements
   return this.slice(0,k);
}
```

Figure 27. Custom array insertion method

This method inserts the *object* passed in as a parameter at the appropriate position according to its *value*, and shifts all elements thereafter by one. It then truncates the array to the first k elements. The running of the top-k selection algorithm is $O(nk)$ where n is the number of members.

Finally, we compute the aggregated sales amount of the ten item subcategories as a percentage of the total sales amount of every item subcategory and display them in a HTML table as well as a browser rendered pie chart using *rGraph*, a *Raphaël* SVG library [35]. We use a slider bound to [*Time*].[*Calendar Date*] to allow the user to change the value of m successively, providing real-time visualization to the calculation result. Although possible with a server-centric model, in the presence of multiple simultaneous users, as well as network latency, the result would be jerky. Therefore, offloading these types of intensive calculations to the client is beneficial from the perspective of the user and the server.

To benchmark this calculation against a server-centric model, we run an experiment with three values of k and a slider control bound to the [*Time*] dimension. Each slider movement re-computes the top-k contribution and re-renders the output. We run the calculation 500 times with random start and end dates for each model. For the JavaScript OLAP Client, the sub-cube downloaded is [*Item*].[*Subcategory*], [*Time*].[*Calendar Day*].[*2004*], and [*Measure*].[*Sale Amt*]. The sub-cube is 471KB with 77,629 tuples and takes 7/40/98/159 milliseconds to download with injected latencies of 0/30/90/150 ms, respectively. We observe that the JavaScript OLAP Client performs the calculation and renders the result in the range of 75 to 85 ms regardless of the value k. While the server-centric model also computes the result in comparable time (75~91 ms),

simulated network latency adds an overhead to the response time and prevents the browser from rendering visualization seamlessly.

	JavaScript OLAP Client				Server-centric OLAP Client			
	Network Latency (in ms)				Network Latency (in ms)			
Parameter *k*	0	30	90	150	0	30	90	150
10	76	76	75	76	75	104	163	226
20	81	82	83	82	85	118	174	243
30	83	82	84	85	91	123	174	242

Table 9. Top-*k* percentage contribution performance

4.3 *m*-day Moving Average

Another common calculation is the *m*-day moving average of a measure for a set of members. For example, the user may be interested in the past 7-day moving average sales amount of all the stores in Seattle in 2004 plotted as a line graph. First, we use our sub-cube retrieval method to obtain dataset consists of the pre-aggregated sales amount of all *n* stores in Seattle by individual dates in 2004. We perform a group-by on individual dates, and then by store, and sort the dataset according to the date on a ascending order. We compute the 7-day moving average of each day in January and plot the initial result on a line graph visualization using *flot*, a Canvas library [36]. We add a "playback" function so that the graph iterates through each day in 2004, calculates the 7-day moving average for the next day, and presents a "moving" average animation as a different way for the user to observe data trends.

Given an *n*-day simple moving average for yesterday for $store_i$, the cost of calculating the *n*-day simple moving average for today is inexpensive:

$$SMA(today, store_i) = SMA(yesterday, store_i) + \frac{Sales(today, store_i)}{n} - \frac{Sales(today - n, store_i)}{n}$$

Because the JavaScript client has the aggregated dataset in memory, it is not necessary to perform a linear scan to determine $Sales(date, store_i)$. Cases such as the *n*-day moving average best exemplify the benefits of delegating work to the client-side, especially when requests are frequent and the server needs to recalculate every time.

For the *m*-day moving average calculation, we retrieve a sub-cube for the JavaScript OLAP Client – [*Store*].[*By Geography*].[*Seattle*], [*Time*].[*Calendar Day*].[*2004*], and [*Measure*].[*Sale Amt*]. The moving average sale of each store in Seattle is computed for January to form the series and to initialize a line graph. When the user initiates the animation, the

JavaScript OLAP Client computes the moving average of each store for the next day and continuously updates the visualization until it reaches the end of year 2004. To make a fair comparison, the server-centric model also runs an MDX query to fetch the initial plot. When the user initiates the animation, the browser fetches the moving average of the next day, instead of the whole period from the OLAP server. When the browser receives the data for the next day, it updates the visualization locally. Table 10 shows the performance time of both models, including rendering time.

	JavaScript OLAP Client				Server-centric OLAP Client			
	Network Latency (in ms)				Network Latency (in ms)			
Parameter *m*	0	30	90	150	0	30	90	150
7	88	87	87	87	159	194	249	309
14	89	90	89	88	166	193	259	317
21	90	91	91	92	172	204	258	320
30	90	93	92	93	173	204	261	329

Table 10. *m*-day moving average performance

In general, human visual perception will not notice flicker at thirty frames per second or more. This is equivalent to about 33 milliseconds for the script time plus rendering time. Although the performance our current implementation is slower than that threshold, it is fair to say that the JavaScript OLAP Client still does better than the server-centric model in three ways. First, we shorten the amount of time it takes to display the pie chart by bridging the gap on network latency. Second, we reduce overall network traffic and server load, so that the server can service more clients. Finally, the smoother visualization provides a better user experience.

4.4 Scatter Plot Aggregation on Mouse Selection

Up to this point, we have been primarily concerned with the aggregation performance aspect of a web-based OLAP client. From the experiments in section 2.4, we learn that dynamically manipulating the HTML content through Document Object Model is not to be trivialized. Often times, OLAP analysis can produce massive datasets. If the results are always shown in a data grid, rendering time alone on a web browser may bog down the application. Even if rendering time is not an issue, analysts can miss potentially interesting trends when they are overwhelmed by the sheer amount of data in a large spreadsheet, as pointed out by Shrader [37]. Graphical visualization tools come into play to overcome such a challenge [38]. A scatter plot is a type of mathematical diagram using Cartesian coordinates to display a set of data. The *x*

and *y* axes represent two measures of a dataset; for each tuple in the dataset, a point is plotted according to its two measures as the *x* and *y* coordinates. While a data grid can potentially span several screens, a scatter plot, on the other hand, can scale to the width and height of the screen, providing the ability to display hundreds of thousands of data points in confined space.

In a traditional server-centric model, a scatter plot of the data is pre-rendered at the server and delivered to the browser as a static image file. Thus, the analyst cannot change the scatter-plot interactively in order to drill-down on a region of the plot with a mouse (i.e. using the mouse to create a rectangle on the plot). As a result, the JavaScript OLAP Client is clearly superior to a server-centric OLAP client. However, to generate scatter plots at the client-side using the dataset in-memory is a challenging problem.

We experiment with generating scatter plots at the client-side using the datasets in-memory. Scalable Vector Graphics, or SVG, has been around for over a decade. In the past, web developers have used this mark-up to create dynamic and interactive graphic components for the web browser. HTML5 introduces the Canvas element tag that brings a new world of possibilities with client-side rendered graphics. Rather than a straightforward graph presentation, we explore a new technique that is tailored to client-side OLAP.

Suppose that the user creates a scatter plot with a measure on the *x*-axis, another on the *y*-axis, and a third measure on the *z*-axis. We summarized *x*, *y* and *z* coordinates of data points in the mouse selection region on the graph. This type of aggregation brings a completely new spectrum of data analysis. It has the potential to aggregate a group of visible outliers, or quickly determine the contribution of a set of points relative to other selected regions. Furthermore, as the user drags the mouse selection, the aggregation updates itself in real-time. Such technique is nearly impossible in a server-centric model where the client informs the server of the user selection. First, the server would need to track the entire dataset that the user currently has on the workspace. Second, even if the server was capable of that, it would not be able to keep up with returning aggregation results on every *onMouseDrag* event in the presence of network latency, let alone the server process time. If the client takes a step back, and responds only to the *onMouseUp* event, the user would need to do repetitive mouse selections to see the difference of selecting neighbour points. For these reasons, we explore the scatter plot aggregation on mouse selection on the client-centric model.

There are two methods of graphic rendering in a web application as described in the top-*k* contribution and the *m*-day moving average calculations: Scalable Vector Graphics and Canvas. When the number of rendered elements is small, the performance difference is negligible. With

SVG, every vector graphic that appears on the canvas is an object in the Document Object Model. The benefit of SVG is that event listeners can be bound to each object; and the drawback is that retrieving an object requires DOM traversal. After we plot the graph, mouse region selection is implemented by drawing a panel in the scatter plot using *Protovis*, a SVG library [39]. On the other hand, the Canvas approach is quite different as it is a raster-based graphic library. The entire canvas is treated as an image, although there can be data associated with the image. We plot the tuples and store them in the order they are fed, in a base canvas. We implement the region selection ourselves by stacking another layer of Canvas element on top of the base. This way, the base layer keeps a static rendition of the scatter plot and remains intact while the user interacts with the top layer. When the user clicks on the canvas, the *onMouseDown* event triggers a rectangular box on the top layer: as the cursor moves across the layer, the top layer is constantly erased and redrawn. The *onMouseDrag* event handler is fired continuously as the selection change, and loops through the set of points in the graph. If a point falls within the selection region by comparing the x and y coordinates of the rectangle, we use the index of the point, look up the tuple in the dataset without linear scanning, and aggregate the three measures. We are able to circumvent linear scanning because the tuples in the dataset and the set of points in the graph are strictly ordered.

The goal in this step is to evaluate the potential of the new Canvas element introduced in HTML5 as a candidate for rendering data visualization. In this research, we introduced a novel client-side aggregation technique: summarization of n data points in a mouse selection region on a scatter plot. We experiment with both methods of rendering a scatter plot from OLAP data, and observe their performance as the user interacts with the data point.

The result of this test in Table 11 shows a dramatic difference between SVG and Canvas. These two libraries have a fundamental difference. Recall that web documents conform to a Document Object Model. Every HTML tag is an element object in a DOM. Event listeners can be bound to DOM elements even during runtime, thus making them interactive. JavaScript can dynamically update the content of an HTML document by inserting, updating, and deleting elements from the DOM. It is well known that DOM traversal to find elements is very costly. However, no research has been done on its performance to extend data point aggregation in addition to DOM traversal. SVG objects are also part of the DOM, which explains why the scatter plot aggregation performs so poorly under SVG as there are thousands of objects. When the selection box is dragged, thousands of DOM traversals are initiated, and the browser bogs down. The Canvas element, on the other hand, is one DOM object. The change in mouse

selection does not rely on DOM traversal to determine which points are selected. Rather, the data associated with the object is searched, and as there is no need to determine ancestor-descendant relationship during aggregation, the Canvas approach is capable of calculating several orders of magnitude more tuples than for example, a cross-tab.

	SVG	Canvas
n	Avg. Time (ms)	Avg. Time (ms)
5,000	398	1
10,000	-	1
50,000	-	3
100,000	-	7
200,000	-	14
300,000	-	18
400,000	-	26

Table 11. SVG versus Canvas scatter plot aggregation

Our goal is not to claim that Canvas has far superior performance to SVG. Rather, through this experiment, we understand that Canvas has great potential to provide new data exploration techniques, in which data visualization and OLAP aggregation overlap. Therefore, it complements SVG to deliver a graphical-rich OLAP client tool.

5 Conclusion and Future Work

5.1 Conclusion

While disk-based and heavily-indexed OLAP servers have been around for decades, main-memory OLAP servers are taking off with cost of memory drastically lower than it has ever been. In this research, we propose an in-memory client-side OLAP engine written in JavaScript. The web-based OLAP application runs in the context of a restrictive browser. We experiment with the performance of the JavaScript OLAP engine against various parameters and observe challenges with a standalone system. We enhance the baseline JavaScript OLAP Client, which in conjunction with a server component, overcomes many of the challenges we observed. The OLAP architecture is different from traditional client-server OLAP systems where the client is merely a presentation layer.

Through careful experimentation, we demonstrate as a proof-of-concept that client-side aggregation can outperform a server-centric OLAP model, in which network latency can contribute a considerable proportion to the response time. We devised a set of OLAP operations commonly found in a lightweight OLAP user workflow and benchmarked the performance of the JavaScript OLAP Client against a real-world OLAP system using a real-life dataset.

Furthermore, we strengthen our claim by showing the capability of the client-centric model to perform calculations commonly used by OLAP power users. We compare our results to a simulated server-centric model and show that a client-centric model not only performs better, but is also capable of rendering graphical visualization by itself, reducing overall network transmission, and offloading server load to the client-side so the server can serve additional requests. We also contribute to the research community by demonstrating a novel technique for a user to explore his/her data by interacting with the visualization and performing client-side aggregation on the fly, which would otherwise be impossible in a server-centric model.

5.2 Future Directions

As there are very few studies in the area of client-side OLAP, particularly in a web-based environment, we feel there are still many aspects of this research worth exploring, both from academic and industrial perspectives. JavaScript engines are constantly improving in terms of

speed. Browsers are adopting more and more features that facilitate the development of web applications parallel to desktop applications.

From a query processing perspective, our JavaScript OLAP engine uses an in-memory table scan algorithm to perform aggregation; it would be interesting to try out other index algorithms and compare their differences in a browser setup. In fact, it would also be worthwhile to explore persistent local storage to keep the dataset on disk rather than in-memory because many browsers have embedded the SQLite engine to simulate a small relational database. In other words, it is possible to let the SQLite engine do the group-by operations after data is retrieved from a server.

Dataset compression can also be investigated to determine the performance trade off between holding more compressed tuples in main memory and decompressing subsets of the data on the fly. Part of our research involves looking into the ability of JavaScript to work with compressed data. Preliminary results show that JavaScript is efficient with binary format as it is not a primitive type in the language [30].

On the topic of Rich Internet Applications, it would be interesting to see the JavaScript OLAP engine ported to Flex or Silverlight and compare the performance bottlenecks. A Flex or Silverlight version could consume our web service layer easily and would enable a lot more possibilities with hardware accelerated graphics libraries.

The RESTful Web Service layer can also be extended to enhance its caching strategy: pre-generation of commonly requested sub-cubes. In addition, if user profiling is incorporated into this layer, it can reduce the size of sub-cubes returned by fetching dimensions at higher granularity instead of leaf level based on the user's query history. Alternatively, when a user requests a sub-cube with many dimensions, the RWS can select an optimal combination of granularities.

Appendices

Appendix A: Weather Dataset Specification

Dimension 1: Brightness Hierarchy: Brightness-Level Level: Brightness (2) Level: Brightness Value (3)	Dimension 6: Measures Hierarchy: Measures Level: Reading (1)
Dimension 2: Day Hierarchy: Day-Level Level: Day (2) Level: Day Value (31)	Dimension 7: Present-weather Hierarchy: Present-weather-Level Level: Present-weather (2) Level: Present-weather Value (102)
Dimension 3: Hour Hierarchy: Hour-level Level: Hour (2) Level: Hour Value (9)	Dimension 8: Solar-altitude Hierarchy: Solar-altitude-Level Level: Altitude Band (2) Level: Altitude Sub-band (9) Level: Altitude Value (187)
Dimension 4: Latitude Hierarchy: Latitude-Level Level: Zone (2) Level: Latitude (9) Level: Latitude Value (160)	Dimension 9: Station-id Hierarchy: Station-Level Level: Station Group (2) Level: Station Sub-group (86) Level: Station ID (7122)
Dimension 5: Longitude Hierarchy: Longitude-Level Level: Zone (2) Level: Longitude (19) Level: Longitude Value (370)	Dimension 10: Weather-change-code Hierarchy: Weather-change-code-Level Level: Weather-change-code Group (2) Level: Weather-change-code Value (11)

Appendix B: Partial XMLA Response


```
<return xmlns="urn:schemas-microsoft-com:xml-analysis">
…
<OlapInfo>
  <CubeInfo>
    <Cube>
      <CubeName>Sales</CubeName>
    </Cube>
  </CubeInfo>
  <AxesInfo>
    <AxisInfo name="Axis0">
      <HierarchyInfo name="[Shippers].[Shipper Name]">
        <UName name="[Shippers].[Shipper Name].[MEMBER_UNIQUE_NAME]" type="xsd:string" />
        <Caption name="[Shippers].[Shipper Name].[MEMBER_CAPTION]" type="xsd:string" />
        <LName name="[Shippers].[Shipper Name].[LEVEL_UNIQUE_NAME]" type="xsd:string" />
        <LNum name="[Shippers].[Shipper Name].[LEVEL_NUMBER]" type="xsd:int" />
        <DisplayInfo name="[Shippers].[Shipper Name].[DISPLAY_INFO]" type="xsd:int" />
      </HierarchyInfo>
    </AxisInfo>
    <AxisInfo name="SlicerAxis">
      <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>
      …
      <Tuple>
        <Member Hierarchy="[Time].[Year]">
          <UName>[Time].[Year].[All]</UName>
          <Caption>All</Caption>
          <LName>[Time].[Year].[(All)]</LName>
          <LNum>0</LNum>
          <DisplayInfo>65545</DisplayInfo>
        </Member>
      </Tuple>
      …
    <CellData>
      <Cell CellOrdinal="0">
        <Value xsi:type="xsd:double">1.35445859E6</Value>
        <FmtValue>$1,354,458.59</FmtValue>
      </Cell>
      <Cell CellOrdinal="1">
        <Value xsi:type="xsd:double">4.0775082E5</Value>
        <FmtValue>$407,750.82</FmtValue>
      </Cell>
      <Cell CellOrdinal="2">
        <Value xsi:type="xsd:double">3.7398319E5</Value>
        <FmtValue>$373,983.19</FmtValue>
      </Cell>
      <Cell CellOrdinal="3">
        <Value xsi:type="xsd:double">5.7272458E5</Value>
        <FmtValue>$572,724.58</FmtValue>
      </Cell>
      <Cell CellOrdinal="35">
        <Value xsi:type="xsd:double">2.262985E5</Value>
        <FmtValue>$226,298.50</FmtValue>
      </Cell>
    </CellData>
  </root>
</return>
```

Appendix C: REAL Warehouse Sample V6 MT Dataset Specification

Dimension 1: Buyer
  Hierarchy: Buyer
    Level: Buyer Index (26)
    Level: Buyer Name (585)
  Hierarchy: Buyer ID
    Level: Buyer ID (585)
  Hierarchy: Buyer Name
    Level: Buyer Name (585)

Dimension 2: Customer
  Hierarchy: Auto Renew Ind
    Level: Auto Renew Ind (3)
  Hierarchy: By Geography
    Level: Country (2)
    Level: State (6)
    Level: City (91)
    Level: Customer Name (218405)
  Hierarchy: Card Status Code
    Level: Card Status Code (5)
  Hierarchy: City
    Level: City (91)
  Hierarchy: Country
    Level: Country (2)
  Hierarchy: Cust Info Code
    Level: Cust Info Code (13)
  Hierarchy: Customer
    Level: Customer Index (2139)
    Level: Customer Name (218405)
  Hierarchy: Customer Index
    Level: Customer Index (2139)
  Hierarchy: Customer Name
    Level: Customer Name (218405)
  Hierarchy: Email Addr
    Level: Email Addr (133826)
  Hierarchy: Expiration Date
    Level: Expiration Date (63)
  Hierarchy: Member ID
    Level: Member ID (218405)
  Hierarchy: Primary Member ID
    Level: Primary Member ID (2)
  Hierarchy: Purchase Date
    Level: Purchase Date (1239)
  Hierarchy: State
    Level: State (6)
  Hierarchy: Years Since Purchased
    Level: Years Since Purchased (5)
  Hierarchy: Zip Code
    Level: Zip Code (124)

Dimension 3: DC Vendor
  Hierarchy: Active

Level: Active (3)
Hierarchy: Vendor
Level: Vendor (12687)
Hierarchy: Vendor Num
Level: Vendor Num (12687)

Dimension 4: Distribution Center
Hierarchy: Distribution Center
Level: Distribution Center (6)
Hierarchy: Distribution Center Code
Level: Distribution Center Code (6)

Dimension 5: Item
Hierarchy: Author
Level: Author (19831)
Hierarchy: Author Index
Level: Author Index (16968)
Hierarchy: Bargain Catalog Ind
Level: Bargain Catalog Ind (4)
Hierarchy: BN Retail
Level: BN Retail (837)
Hierarchy: BN Retail Grouping
Level: BN Retail Grouping (6)
Hierarchy: By Category
Level: Product (3)
Level: Subject (19)
Level: Category (123)
Level: Subcategory (373)
Level: Item (43951)
Hierarchy: By DC Vendor
Level: DC Vendor Alpha (26)
Level: DC Vendor (1306)
Hierarchy: By Dept
Level: Dept (12)
Hierarchy: By Original Purchase Vendor
Level: Original Vendor Index (25)
Level: Original Purchase Vendor (300)
Hierarchy: By Purchase Vendor
Level: Purchase Vendor Index (24)
Level: Purchase Vendor (232)
Hierarchy: By Return Code
Level: Return Code Status (4)
Level: Return Code (6)
Hierarchy: By Return Vendor
Level: Return Vendor Index (25)
Level: Return Vendor (297)
Hierarchy: By Source Vendor
Level: Source Vendor Index (23)
Level: Source Vendor (190)
Hierarchy: By Status
Level: Current Status (4)
Level: Status Detail (14)
Hierarchy: By Subject
Level: Subject (19)
Level: Category (123)

Level: Subcategory (373)
Level: Item (43951)
Hierarchy: Category
Level: Category (123)
Hierarchy: Class Code
Level: Class Code (21)
Hierarchy: Current Status
Level: Current Status (4)
Hierarchy: Current Status Code
Level: Current Status Code (14)
Hierarchy: DC Vendor
Level: DC Vendor (1306)
Hierarchy: Dept
Level: Dept (12)
Hierarchy: Display
Level: Display (241)
Hierarchy: EAN
Level: EAN (43928)
Hierarchy: Edition
Level: Edition (1575)
Hierarchy: Imprint Index
Level: Imprint Index (33)
Hierarchy: Imprint Name
Level: Imprint Name (5885)
Hierarchy: ISBN
Level: ISBN (42771)
Hierarchy: Item
Level: Item (43951)
Hierarchy: Language Code
Level: Language Code (55)
Hierarchy: Large Print Ind
Level: Large Print Ind (3)
Hierarchy: Long Item Desc
Level: Long Item Desc (43928)
Hierarchy: Original Purchase Vendor
Level: Original Purchase Vendor (300)
Hierarchy: Product
Level: Product (3)
Hierarchy: Publish Date
Level: Publish Date (4702)
Hierarchy: Publish Year
Level: Publish Year (70)
Hierarchy: Publisher Index
Level: Publisher Index (32)
Hierarchy: Publisher Name
Level: Publisher Name (4547)
Hierarchy: Purchase Vendor
Level: Purchase Vendor (232)
Hierarchy: Retail Amt
Level: Retail Amt (853)
Hierarchy: Return Code
Level: Return Code (6)
Hierarchy: Return Code Status
Level: Return Code Status (4)
Hierarchy: Return Vendor

Level: Return Vendor (297)
Hierarchy: Source Vendor
Level: Source Vendor (190)
Hierarchy: Status Detail
Level: Status Detail (14)
Hierarchy: Subcategory
Level: Subcategory (373)
Hierarchy: Subject
Level: Subject (19)
Hierarchy: Total Num Pages
Level: Total Num Pages (1243)
Hierarchy: Total Num Pages Grouping
Level: Total Num Pages Grouping (6)
Hierarchy: Years Since Published
Level: Years Since Published (10)

Dimension 6: Measures
Hierarchy: Measures
Level: MeasuresLevel (19)

Dimension 7: Periodicity
Hierarchy: Measure
Level: Measure (2)

Dimension 8: Replen Strategy
Hierarchy: Replen Strategy
Level: Strategy Type (4)
Level: Strategy (8)
Level: Title Type (14)
Hierarchy: Strategy
Level: Strategy (8)
Hierarchy: Strategy Type
Level: Strategy Type (4)
Hierarchy: Title Type
Level: Title Type (14)

Dimension 9: Store
Hierarchy: Ad Area
Level: Ad Area (6)
Hierarchy: City
Level: City (126)
Hierarchy: Close Date
Level: Close Date (75)
Hierarchy: District
Level: District (6)
Hierarchy: Division
Level: Division (2)
Hierarchy: Geography
Level: Division (2)
Level: Region (3)
Level: District (6)
Level: City (126)
Level: Store (285)
Hierarchy: Grand Open Date
Level: Grand Open Date (2)

Hierarchy: Linear Ft
Level: Linear Ft (66)
Hierarchy: Linear Ft Grouping
Level: Linear Ft Grouping (5)
Hierarchy: Market Area
Level: Market Area (7)
Hierarchy: Mgr Name
Level: Mgr Name (135)
Hierarchy: Months Since Opened
Level: Months Since Opened (6)
Hierarchy: Open Date
Level: Open Date (222)
Hierarchy: Region
Level: Region (3)
Hierarchy: Square Feet
Level: Square Feet (147)
Hierarchy: Square Feet Grouping
Level: Square Feet Grouping (6)
Hierarchy: Status Desc
Level: Status Desc (5)
Hierarchy: Store
Level: Store (285)
Hierarchy: Store Phone
Level: Store Phone (140)

Dimension 10: Time
Hierarchy: Calendar
Level: Calendar Year (20)
Level: Calendar Qtr (74)
Level: Calendar Month (219)
Level: Calendar Week (1132)
Level: Day (6616)
Hierarchy: Day Of Week
Level: Day Of Week (8)
Hierarchy: Fiscal
Level: Fiscal Year (21)
Level: Fiscal Qtr (75)
Level: Fiscal Period (220)
Level: Fiscal Week (947)
Level: Day (6616)

Dimension 11: Vendor
Hierarchy: Vendor
Level: Vendor_Index (26)
Level: Vendor Name (1536)
Hierarchy: Vendor Index
Level: Vendor Index (26)
Hierarchy: Vendor Name
Level: Vendor Name (1536)

Dimension 12: Vendor Type
Hierarchy: Vendor Type
Level: Vendor Type (6)

Appendix D: Workflow Sequence

Measure Group – Store Sales

1. Generate cross-tab on [Store].[Geography].[West].[Washington], [Time].[Calendar].[Quarters].[2004] and [Measure].[Sale Amt]
2. Drill-down on [Store].[Geography].[West].[Washington].[Seattle]
3. Drill-down on [Time].[2004].[Q3]
4. Drill-down on [Time].[2004].[Q4]
5. Roll-up [Time].[2004].[Q3]
6. Drill-down on [Time].[2004].[Q4].[November]
7. Generate cross-tab on [Store].[Geography].[West].[Washington], [Time].[Calendar].[Month].[2004] and [Measure].[Mark Down Amt]
8. Drill-down on [Time].[2004].[January]
9. Drill-down on [Time].[2004].[February]
10. Drill-down on [Time].[2004].[March]
11. Generate cross-tab on [Store].[Geography].[Barnes & Noble], [Item].[By Category] , and [Measure].[Sale Qty]
12. Drill-down on [Item].[By Category].[Book]
13. Drill-down on [Item].[By Category].[Book].[Art]
14. Drill-down on [Item].[By Category].[Book].[Art].[Graphic Design]
15. Drill-down on [Store].[Geography].[West]
16. Drill-down on [Store].[Geography].[West].[Washington]
17. Drill-down on [Store].[Geography].[West].[Washington].[Seattle]
18. Roll-up on [Store].[Geography].[West].[Washington].[Seattle]
19. Drill-down on [Store].[Geography].[Central]
20. Drill-down on [Store].[Geography].[Central].[Illinois]
21. Generate cross-tab on [Time].[2004].[Quarters], [Item].[By Category].[Book].[Fiction] and [Measure].[Item Coupon Amt]

22. Drill-down on [Item].[By Category].[Book].[Fiction].[ADV/ESPIONAGE]
23. Drill-down on [Item].[By Category].[Book].[Fiction].[ADV/ESPIONAGE].[ADV/ESPIONAGE]
24. Drill-down on [Item].[By Category].[Book].[Fiction].[Horror]
25. Drill-down on [Item].[By Category].[Book].[Fiction].[Horror]. [Asia]
26. Generate cross-tab on [Item].[By Category].[Book].[Literature], [Customer].[Customer].[Customer Index] and [Measures].[Member Disc Amt]
27. Generate cross-tab on [Store].[Geography].[Barnes & Noble], [Customer].[By Geography] and [Measures].[Discount Amt]
28. Drill-down on [Customer] .[By Geography].[USA]
29. Drill-down on [Customer] .[By Geography].[USA].[MN]

Measure Group – Item Vendor

30. Generate cross-tab on [Item].[By Category].[Book], [Vendor].[Vendor].[Vendor Index] and [Measure].[Item Vendor Count]
31. Drill-down on [Item].[By Category].[Book][Military History]
32. Drill-down on [Item].[By Category].[Book][Military History].[Weapons of War]
33. Drill-down on [Item].[By Category].[Book][Military History].[Weapons of War].[Weapons of War]
34. Drill-down on [Vendor].[Vendor].[Vendor Index].[F]
35. Drill-down on [Vendor].[Vendor].[Vendor Index].[P]
36. Generate cross-tab on [Vendor Type].[Vendor Type] .[Vendor Type], [Item].[By Category].[Subcategory] and [Measure].[Item Vendor Count]

Measure Group – Distribution Center Inventory

37. Generate cross-tab on [Distribution Center].[Distribution Center].[Distribution Center], [Item].[By Category] and [Measures].[DC Days In Stock], [Time][Calendar][2004]
38. Drill-down on [Item].[By Category].[Book]
39. Drill-down on [Item].[By Category].[Book].[Study Aids]
40. Drill-down on [Item].[By Category].[Book].[History]

41. Drill-down on [Item].[By Category].[Book].[Westerns]

42. Drill-down on [Item].[By Category].[Book].[History]

43. Drill-down on [Item].[By Category].[Book].[Fiction]

44. Drill-down on [Item].[By Category].[Book].[Business]

45. Drill-down on [Item].[By Category].[Book].[Business].[Management]

46. Generate cross-tab on [Distribution Center].[Distribution Center] .[Distribution Center], [Time].[Calendar].[Month].[2004] and [Measures].[Total DC Qty], [Item].[By Category].[Book]

47. Drill-down on [Time].[Calendar].[2004].[October]

48. Drill-down on [Time].[Calendar].[2004].[September]

49. Drill-down on [Time].[Calendar].[2004].[September].[Week 36]

50. Drill-down on [Time].[Calendar].[2004].[August]

51. Drill-down on [Time].[Calendar].[2004].[August].[Week 35]

52. Drill-down on [Time].[Calendar].[2004].[November]

53. Generate cross-tab on [Distribution Center].[Distribution Center] .[Distribution Center], [Item].[By Category].[Product] and [Measures].[Available Qty], [Time].[Calendar].[2004]

54. Drill-down on [Item][By Category].[Pre-Pack]

55. Drill-down on [Item][By Category].[Pre-Pack].[Little Gift Bks]

56. Drill-down on [Item][By Category].[Pre-Pack].[Little Gift Bks].[Mini Kits]

    Measure Group – Item Inventory

57. Generate cross-tab on [Replen Strategy].[Replen Strategy].[Strategy Type], [Store].[Geography].[District] and [Measures].[On Hand Qty], [Time].[Calendar].[2004]

58. Drill-down on [Replen Strategy].[Replen Strategy].[Strategy Type].[Backlist]

59. Drill-down on [Replen Strategy].[Replen Strategy].[Strategy Type].[Backlist].[Modeled]

60. Drill-down on [Replen Strategy].[Replen Strategy].[Strategy Type].[Frontlist]

61. Drill-down on [Replen Strategy].[Replen Strategy].[Strategy

Type].[Frontlist].[Store Managed]

62. Generate cross-tab on [Item].[By Category].[Subject], [Buyer].[Buyer].[Buyer Index] and [Measures].[Model Qty]
63. Drill-down on [Buyer].[Buyer].[Buyer Index].[B]
64. Roll-up on [Buyer].[Buyer].[Buyer Index].[B]
65. Drill-down on [Buyer].[Buyer].[Buyer Index].[C]
66. Generate cross-tab on [Time].[Calendar].[Calendar Week].[2004], [Store].[Geography].[District] and [Measures].[Return Qty], [Item].[By Category].[Book].[Fiction]
67. Drill-down on [Time].[Calendar].[Calendar Week].[2004].[30]
68. Drill-down on [Time].[Calendar].[Calendar Week].[2004].[31]
69. Drill-down on [Time].[Calendar].[Calendar Week].[2004].[32]
70. Drill-down on [Time].[Calendar].[Calendar Week].[2004].[33]
71. Generate cross-tab on [Time].[Calendar].[Calendar Month].[2004], [Item].[By Category].[Category].[Business], and [Measures].[On Order Qty]
72. Drill-down on [Item].[By Category].[Category].[Business].[Investing]
73. Drill-down on [Item].[By Category].[Category].[Business] .[Investing].[Day Trading]
74. Drill-down on [Item].[By Category].[Category].[Business].[Spiral Jrnl]

References

[1] E. Thomsen, OLAP solutions : building multidimensional information systems, New York ; Chichester: Wiley, 2002, pp. 661.

[2] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *SIGMOD Record*, vol. 26, pp. 65-74, 1997.

[3] A. Witkowski et al., "Spreadsheets in RDBMS for OLAP," in *Proceedings of ACM SIGMOD*, 2003.

[4] N. Pendse, "Commentary: What in-memory BI 'revolution'?" Website, 2010. http://www.bi-verdict.com/fileadmin/FreeAnalyses/Comment_InMemBI.htm.

[5] L. Qiao, V. Raman, F. Reiss, P. Haas and G. Lohman, "Main-Memory Scan Sharing for Multi-Core CPU's," in *Proceedings of VLDB Conference*, pp. 610-621, 2008.

[6] S. Jobs, "Thoughts on Flash," Website, 2010. http://www.apple.com/hotnews/thoughts-on-flash/.

[7] D. Hachamovitch, "HTML5, Native: Third IE9 Platform Preview Available for Developers," Website, 2010. http://blogs.msdn.com/b/ie/archive/2010/06/23/html5-native-third-ie9-platform-preview-available-for-developers.aspx.

[8] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan and S. Sarawagi, "On the computation of multidimensional aggregates," in *Proceedings of The International Conference on Very Large Databases*, pp. 506-521, 1996.

[9] R. Agrawal, A. Gupta and S. Sarawagi, "Modeling Multidimensional Databases," in *Proceedings of International Conference on Data Engineering (ICDE'97)*, pp. 232-243, 1997.

[10] L. Cabibbo and R. Torlone, "A logical approach to multidimensional databases," *Advances in DB Technology - EDBT 98*, pp. 183-197, 1998.

[11] C. Chan and Y.E. Ioannidis, "Bitmap Index Design and Evaluation," in *Proceedings of ACM SIGMOD*, pp. 355-366, 1998.

[12] J. Gray, A. Bosworth, A. Layman, D. Reichart and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," in *Proceedings of International Conference Data Engineering (ICDE'96)*, pp. 152-159, 1996.

[13] H. Gupta, V. Harinarayan, A. Rajaraman and J.D. Ullman, "Index Selection for OLAP," in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 208-219, 1997.

[14] T.B. Pedersen and C.S. Jensen, "Multidimensional Data Modeling for Complex Data," in *Proceedings of International Conference on Data Engineering (ICDE '99)*, 1999.

[15] J. Pei, "A general model for online analytical processing of complex data," in *Proceedings of the 22nd International Conference on Conceptual Modeling (ER'03)*, pp. 13-26, 2003.

[16] Adobe Systems Incorporated, "Flex Data Visualization Developer's Guide - AdvancedDataGrid Controls and Automation Tools," Website, 2010. http://livedocs.adobe.com/flex/3/html/Part2_adv_data_grid_API_1.html.

[17] W. Luk, "ADODA: A Desktop Online Data Analyzer," in *Proceedings of DASFAA*, 2001.

[18] C. Lang, B. Bhattachuarjee, T. Malkemus, S. Padmanabhan and K. Wong, "Increasing Buffer-locality for Multiple Relational Table Scans through Grouping and Throttling," in *Proceedings of International Conference on Data Engineering (ICDE)*, pp. 972-984, 2007.

[19] P.M. Deshpande, J.F. Naughton and Y. Zhao, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates," *ACM SIGMOD*, pp. 159-170, 1997.

[20] jQuery UI Team, "jQuery UI," Website, 2010. http://http://jqueryui.com/.

[21] YUI Team, "YUI 2: TreeView," Website, 2010. http://developer.yahoo.com/yui/treeview/.

[22] F. Gingras and L.V.S. Lakshmanan, "nD-SQL: A Multi-dimensional Language for Interoperability and OLAP," in *Proceedings of the 24th VLDB Conference*, pp. 134-145, 1998.

[23] XML for Analysis Council, "XML for Analysis," Website, 2010. http://www.xmla.org/.

[24] E. Melomed, "Configuring HTTP Access to SQL Server 2005 Analysis Services on Microsoft Windows Server 2003," Website, 2010. http://technet.microsoft.com/en-ca/library/cc917711.aspx.

[25] Microsoft Corporation and Hyperion Solutions Corporation, "XML for Analysis Specification," Website, 2010. http://msdn.microsoft.com/en-us/library/ms977626.aspx.

[26] R. Bouman, "xmla4js," Website, 2010. http://code.google.com/p/xmla4js/.

[27] R.T. Fielding, D. Software and R.N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology*, vol. 2, pp. 115-150, 2002.

[28] L. Richardson and S. Ruby, RESTful web services, Farnham: O'Reilly, 2007, pp. 419.

[29] Microsoft Corporation, "Calculating CellOrdinal," Website, 2010. http://msdn.microsoft.com/en-us/library/ms713648%28VS.85%29.aspx.

[30] D. Crockford, JavaScript: The Good Parts, O'Reilly Media, Inc., 2008.

[31] K. Azad, "How To Optimize Your Site with GZIP Compression," Website, 2010. http://betterexplained.com/articles/how-to-optimize-your-site-with-gzip-compression/.

[32] J. Gailly and M. Adler, "zlib," Website, 2010. http://www.zlib.net/.

[33] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," 1996.

[34] M. Carbone and L. Luigi Rizzo, "Dummynet Revisited," *ACM SIGCOMM Computer Communication Review*, vol. 40, pp. 12-20, 2010.

[35] D. Baranovskiy, "Raphaël - JavaScript Library," Website, 2010. http://raphaeljs.com/.

[36] O. Laursen, "flot," Website, 2010. http://code.google.com/p/flot/.

[37] M. Schrader, Oracle Essbase & Oracle OLAP : the guide to Oracle's multidimensional solution, New York: Oracle Press/McGraw-Hill, 2010, pp. 498.

[38] A. Maniatis, P. Vassiliadis, S. Skiadopoulos and Y. Vassiliou, "Advanced Visualization for OLAP," in *Proceedings of ACM 6th International Workshop on Data Warehousing and OLAP (DOLAP 2003)*, 2003.

[39] Stanford Visualization Group, "Protovis," Website, 2010. http://vis.stanford.edu/protovis/.