Mobile Based OLAP Using Parallel Processing and Multithreading

By
Kheyali Mitra
B.Tech., West Bengal University of Technology, 2009

Project Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Kheyali Mitra 2016
SIMON FRASER UNIVERSITY
Summer 2016

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, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.
Approval

Name: Kheyali Mitra
Degree: Master of Science (School of Computing Science)
Title: Mobile Based OLAP Using Parallel Processing and Multithreading

Examinig Committee: Chair: Dr. Nick Sumner
Professor

Dr. Wo Shun Luk
Senior Supervisor
Professor

Dr. Ke Wang
Supervisor
Professor

Dr. Binay Bhattacharya
External Examiner
Professor
School of Computing Science
Simon Fraser University

Date Defended/Approved: August 3, 2016
Abstract

This work is committed to discovering the influence of parallel mechanism on I/O heavy applications such as OLAP operations on remote data set running on a resource constrained client. In this research, we have chosen an Android based smartphone as our client. We have implemented a mobile phone based client-centric OLAP application utilizing the potentiality of parallel processing mechanism of Android. We are dividing user request and associated caching tasks into smaller units and distributing them to multiple threads to execute concurrently. While serving user’s requested query data, we are downloading and caching (hash based pre-emptive caching) extra sub-cube data in the background by employing multiple threads running in parallel. This yields better performance for succeeding cross-tab or drill-down actions. We have captured test results by varying number of threads and amount of data size associated with each thread. Result shows that parallel approach has striking performance improvement over sequential approach.

Keywords:  Client-centric OLAP; Client-side caching; OLAP operations; Multithreading; Parallel processing; Android application development
To the almighty

For always showing me the right direction

To my beloved husband, Jay

For his endless encouragements,

And continuous support

To my ma

For everything
Acknowledgements

I would like to thank Dr. Nick Sumner for taking time out of his schedule to serve as the chair of the committee.

A special thanks go to Prof. Dr. Ke Wang for accepting to be my supervisor and Prof. Dr. Binay Bhattacharya for being my external examiner and dedicating their valuable time to review my thesis.

My personal thanks go to my senior supervisor, Prof. Dr. Wo-shun Luk, who offered inestimable supervision and guide to me throughout this work with his expertise, intelligence and knowledge. He always cleared his busy schedules for me whenever I had things to discuss. His valuable advice and direction promoted me to solve any obstacles I met during the whole journey. It would have been impossible for me to complete this quest without his gracious encouragement.

I also want to appreciate the help I received from Bai, the former lab mate. This helped me start with my project.
# Table of Contents

Approval .......................................................................................................................... ii  
Abstract ........................................................................................................................... iii  
Dedication ......................................................................................................................... iv  
Acknowledgements .......................................................................................................... v  
Table of Contents ........................................................................................................... vi  
List of Figures .................................................................................................................. viii  
List of Tables .................................................................................................................. ix  

**Chapter 1. Introduction** .............................................................................................. 1  
1.1. Online Analytical Processing (OLAP) ................................................................. 1  
1.2. Client Side Data Caching .................................................................................... 2  
1.3. Client-centric OLAP ............................................................................................ 2  
1.4. Mobile Client ....................................................................................................... 2  
1.5. Research Motivation and Objective .................................................................. 3  
1.6. Hypothesis .......................................................................................................... 5  
1.7. Related Work ...................................................................................................... 6  
1.8. Thesis Organization ............................................................................................ 8  

**Chapter 2. Overview of Client-Server System** ....................................................... 9  
2.1. Mobile-OLAP Architecture ................................................................................ 9  
    Multidimensional Data and Metadata ...................................................................... 10  
    XMLA ....................................................................................................................... 11  
    Cell Ordinals .......................................................................................................... 11  
    2.1.1. Analytical Operations ................................................................................... 13  
        Cross-tab ............................................................................................................. 13  
        Roll-up .............................................................................................................. 14  
        Drill-down ......................................................................................................... 14  
        A typical user scenario .................................................................................... 14  
    2.1.2. Hash Based Pre-emptive Caching Architecture ........................................... 15  

**Chapter 3. Android based native application** .......................................................... 18  
3.1. Android OS Architecture ................................................................................... 18  
3.2. Android OLAP Application ................................................................................ 19  
3.3. Android Marshmallow ......................................................................................... 20  
3.4. Parallel Processing in Android OS ...................................................................... 21  
    3.4.1. Concurrent Execution of Multiple Threads ................................................. 21  
3.5. Control Over Threading ...................................................................................... 23  

**Chapter 4. Implementation of “Mobile based Parallel OLAP processing”** .......... 26  
4.1. Basic Functionalities ........................................................................................... 26  
4.2. Special Data Structures ....................................................................................... 29  
    4.2.1. Dimensions ..................................................................................................... 29  
    4.2.2. Cache ............................................................................................................. 29
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Three dimensional data cube</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Client centric OLAP system (taken from [4])</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Mobile OLAP Architecture</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>MDX query and its internal representation</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Associated Fact Table</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Cell Table and its internal mapping</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Crosstab on Geography and Date</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Drill down and Roll up operation</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>User selection: Cell table</td>
<td>16</td>
</tr>
<tr>
<td>2.8</td>
<td>Hash Based Pre-emptive Caching Architecture for OLAP system</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Android OS architecture (taken from [22])</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Concurrent execution of multiple threads in Marshmallow 6.0</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Concurrent execution of 17 threads</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Display Screen (starting from top left)</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>Dimension hierarchy selection from tree view</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Cache Structure (HashMap)</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>CellOrdinals</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Mapping between JSON data and user selection</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Dimension Hierarchy: Example</td>
<td>35</td>
</tr>
<tr>
<td>4.7</td>
<td>Application on play</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance Comparison: Set 1</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Performance Comparison: Set 2</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance Comparison: Set 3</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance Comparison: Set 4</td>
<td>47</td>
</tr>
<tr>
<td>5.5</td>
<td>Performance Comparison: Special Case</td>
<td>48</td>
</tr>
<tr>
<td>5.6</td>
<td>Overall comparison of two approaches by varying data size and threads</td>
<td>49</td>
</tr>
<tr>
<td>5.7</td>
<td>Overall comparison: 2 threads and 3 threads</td>
<td>51</td>
</tr>
<tr>
<td>5.8</td>
<td>Overall comparison: 4 threads and 5 threads</td>
<td>51</td>
</tr>
</tbody>
</table>
List of Tables

Table 4.1.   TreeNode class properties and methods ............................................. 29
Table 4.2.   Inflated query: All possible combinations ............................................ 36
Table 5.1.   Test environment details .......................................................................... 43
Table 5.2.   Overall performance gain ......................................................................... 53
Chapter 1. Introduction

1.1. Online Analytical Processing (OLAP)

Decision Support Systems (DSS) helps users to make efficient judgments about present as well as future status of an organization. One of the most dominant technologies for information exploration in DSS is Online Analytical Processing (OLAP). OLAP provides a framework for a wide variety of fundamental business applications especially in sales, marketing analysis and planning. Multidimensional views of OLAP server provides “drill-down”, “roll-up” on dimension hierarchies and “slice and dice” attributes (please refer to Chapter 2 for more details on this). OLAP uses a traditional data model named "data cube" which enables users to view organizational data from diverse prospects. Figure 1.1 demonstrates a three-dimensional data cube from a hypothetical organization. Various planes in the figure signify aggregation at a coarser granularity and each cell in the cube is an aggregated value of one or more measures.

![Three dimensional data cube](image-url)
1.2. Client Side Data Caching

Client-side data caching is a common approach for applications which involve heavy server-client interaction. The concept is to retrieve some extra related information from the server while fetching data for a user request. This extra information is stored in client-side cache for future use. Client-side caching can reduce unnecessary round trip time which is wasted to fetch data from the server for every single request. This, in turn, helps to get better performance.

Client for client-side caching can be a web-based application where the client-side logic is written in JavaScript or some similar scripting language. Web-based applications run on a browser and dependent on internet access. Client-side caching can also be possible in app-based applications. Common examples can be desktop based applications or mobile based native applications. They need to be installed in client side and mostly do not need browsers to run.

1.3. Client-centric OLAP

Our project is based on Online Analytical Processing application and we are implementing a "client-centric" client-side caching on a mobile device. Client-centric applications have a very light-weight OLAP engine running on the client side. It enables users to perform OLAP operations on the data cached on the client side. In other words, if data is available on the client side, this OLAP engine can generate an answer for the user query. Unlike client-centric, a server-centric client always takes a round trip to the server to find out the answer for each user request.

1.4. Mobile Client

Today, smartphones have reached even in the most undeveloped part of the world. According to the statistics from [34], time spent on "Mobile digital media" per adult per day in US is 51% compared to a desktop (42%) (2008-2015). As per Nielsen’s Q1 2013 Cross-Platform Report, "smartphone users spent 87 percent of their “app/web” time using mobile
apps; they spent the remaining 13 percent of their time on mobile web”. On the other hand, in 2014 App Store recorded 85 billion and Google Play recorded 50 billion app downloads [33]. These statistics highlight the bright prospect and popularity of mobile-based applications in the market. In spite of memory size and network bandwidth constraints, a mobile device still can win the race because of its portability and ease of access.

In our project, we have implemented a native application using backbone of Android OS. This system involves a light-weight OLAP processor to perform operations like drill down, roll up and cross-tab on data.

1.5. Research Motivation and Objective

In [11] a “hash based caching mechanism” is proposed for web-based OLAP application. This is a pre-emptive caching approach where data objects are retrieved from the server and cached on the client side even before they are queried by the user. This system tried to predict the most common behavior of user based on user's current drill-down or roll-up operations, dimensions and measures selection. The application generates “inflated query” (a query containing all the leaf level or immediate children level information of all selected dimensions) based on this prediction and wisely utilizes "thinking time" of the user to download "inflated query" data in the background. Therefore, the user is not penalized by this pre-emptive caching on the client side. Results are stored in special key-value paired structure in the client. This hash based structure provides easy and fast access to cache data and consequently provides faster response to user's query. Moreover, this work proposed a special algorithm, called "Sort-Merge" which generates a short and simple MDX query associated with missing user data. Additionally, fetching response in XMLA format helped to avoid unnecessary data transfer over the internet (please refer to Chapter 2 for more details on XMLA). These improved data retrieval time from the server. This approach had shown better performance over no cache mechanism.

However, it has a limitation. This web-based application executes all operations in a serial mode. It lacks true implementation of multithreading and that leads to the employment of only one thread to do all the work. The impact is significant when large data download and processing is involved. This kind of longer running tasks can cause
bad user experience. Because large data retrieval and processing using single thread need a good amount of time to finish. There is a high chance that the total time is greater than "thinking time" of the user and the application may act unexpectedly when the user requests subsequent query while the previous process is still running in the background. This shortcoming motivates us to do further research so that we can overcome this challenge.

One simple solution to this problem is to split the task into multiple sub tasks, assign them to individual threads and execute concurrently. True multi-threading and concurrent execution is possible only if the query processing is done on OS level. Therefore, we decided to select Android-based mobile as our client.

One of the significant part of this project is to migrate from a web-based scheme of [11] to an app-based environment, (Android-based smartphone). We confine our focus on improving overall performance (in terms of execution time) using parallel processing on Android’s multi-core processors. Our "app-based client" [4][6] is client-centric OLAP which always interact with OLAP server by calling dedicated web services and works on request-response mode. We are providing a light-weight, quick responding user-friendly mobile application. A user starts this application by submitting his/her query on selected data cube. The system, in return, brings the result from OLAP server and helps the user to visualize it in the most convenient way (tabular and bar chart fashion). When a request comes from the user, the application has 2 principal responsibilities:

- Fetch requested data (if not in cache), save them in cache and display them to the user

- Fetch data related to children of all selected dimensions and deposit them in the cache.

We can notice that every single request from the user involves a significant amount of data download from the server and display process. Hence we can treat this as an I/O heavy application. That is why we are taking the opportunity to split these tasks into smaller units and assign them to individual thread for parallel execution.

We know that, using multi-core processors and multiple asynchronous threads, n number of tasks can be done concurrently and also in a shorter time than the one with
serial execution. In case of parallel execution, the total time to execute all such tasks will be roughly equal to the task taking maximum time to execute: \( \text{Max}(\text{task}_1, \text{task}_2, \ldots, \text{task}_n) \). Whereas for the sequential execution, it will be the sum of all individual tasks: \( (\text{task}_1 + \text{task}_2 + \ldots + \text{task}_n) \). We are aiming that using this concept, we can improve the user experience.

Now, when we are involving the concept of parallel execution of threads, we have few basic questions in mind and we would like to find answers to all of them from our experiments. Questions which we asked ourselves at the beginning of our work are:

1. Does the application perform better than web based sequential approach when we apply multitasking and parallelism?
2. If we keep adding threads, does that verify the fact that we keep getting better performance? A follow-up question is; how many threads are "optimal" to run in parallel and yield "optimal" performance?
3. Are there any instances found where serial approach fails but parallel survives? We are curious to know about the robustness of this parallel approach.
4. What are the factors which lead to notable performance gain over serial approach?

At the end of our work, we have evaluated performance gain of Android OS based native application over its sequential web equivalent application. Results show that this new approach can perform up to 80-100% faster than the serial approach. This attests that our application performs better than its web equivalent.

1.6. **Hypothesis**

We have considered a couple of hypotheses associated with our work.

- A very common behavior of a user in data analysis is to start visualizing data from specific dimensions and gradually digging deeper into those dimensions for detailed analysis. So, from developer’s point of view, there is a high chance of getting related queries from the user. If a pre-emptive caching is done wisely, we might end up getting the whole piece of data from client side cache. The foremost benefit of this caching is that this can help to reduce the round trip to the OLAP server for every single query. It helps to get better user experience. Based on this assumption, we are claiming that fetching inflated sub cubes associated with user’s current selection will benefit in subsequent queries. However, if user’s next selection does not match with his/her previous choice, we can assume that the user is no longer interested in past selections and try to explore new information from a different set of dimensions and measures. Hence, this is safe to discard the cache items related to previous dimensions.
and fill it in with new selection. Note that this change in selection will be considered as a “miss” and data will be fetched from the server.

- This pre-emptive caching mechanism is effective only when we have valid permission to download immediate next or immediate previous level data of given dimensions from OLAP server. This approach might not be applicable for highly confidential data set.

- Each task in our application refers to a certain amount of data download from the OLAP server, data processing in client side (parsing the response object and mapping each record with its associated CellOrdinal numbers) and finally saving them in the local cache for future use. So execution time of each task includes data download time, data processing time and data saving time in the cache.

- We are measuring the performance of our application in terms of the total execution time of each task. As mentioned before, each execution time involves data download time, data processing and saving time. We are assuming that available internet is fast and network bandwidth is sufficient enough to process multiple requests. We are not concerned about the downloading process. Our main motive is to minimize the data processing time on the client side by applying multiple concurrent threads.

- We are also assuming that there are no dependencies among tasks which are distributed among concurrent threads. In this application, we are splitting tasks in such a way that no two tasks are dependent on each other. This confirms true parallelism. Although we understand that in real life, there could be cases where tasks are connected and cannot be split into an independent component for concurrent execution. These tasks have to be executed in a serial manner.

1.7. Related Work

Online analytical processing is a considerably popular research topic. Over the years, many researchers have broadly investigated numerous aspects of it. The concept of client-side data caching got introduced in late 80’s as a new way of boosting relational database system (RDBMS) performance. In [9] [7] [8], data caching mechanism for OLAP were proposed on the server side to cache a data cube or a subset of that. These caching mechanisms were mainly focused on cache entry and removal strategies. Some researchers even considered data cube caching in a distributed system [5].
In [4], T. Hasio and team proposed a research work based on client-side OLAP running on a restrictive web-based environment. They introduced a query processing data engine in client side to process data already downloaded from the server. The logic of data engine was written in JavaScript. The architecture of this approach is depicted in Figure 1.2. It has server-side component (OLAP server, ADOMD, XMLA and web service) and the client side component made of only JavaScript (JavaScript OLAP Client) and a user-friendly UI to interact.

Figure 1.2.  Client centric OLAP system (taken from [4])

An enhancement of this model to improve user experience is done in [6] by Elah and team. They added a feature to the client side data engine which can help the user to operate on data cube on the client side without any dependency on the server. For that, data engine not only downloads the answer of the user query but also it’s associated data from all leaf nodes of selected dimension hierarchies. For e.g., a user enquires “what is the total internet sale of Canada in the year of 2008?” There is a high chance that user may query more on these dimensions as soon as she/he gets the result; e.g., “what is the total internet sales of British Columbia for the first quarter of 2008?” This sort of related ‘drill-down’ queries could be answered from the client side if the data engine already downloads extra data from the server (instead of only querying Canada and 2008, it may regenerate the query internally to download sales information of all provinces of Canada along with all quarters of 2008) and stores in the cache. Hence, it can save round-
trip time to the server. However, this approach can keep result for only one query. Besides, the user has to face the extended processing time due to downloading all associated drill-down data of given user query. Above all, among all query processing functions, this caching mechanism can only serve drill-down operations. Zheng Xu and team, had implemented the app-based version of the client centric OLAP in an android tablet featuring a multi-core CPU [10]. This work compared the performance of app-based client with web-based and showed that app-based client performs better than web-based client.

A new improved approach was proposed in [11] to overcome challenges faced in [6]. In this work, a hash based caching approach was developed. This work considered average “thinking time” [11] of the user and utilized this interval to download associated “inflated query” data from the OLAP server in the background while the user is busy visualizing the data. So, from that point on, any drill-down, roll-up queries associated with same dimensions can be executed without giving round trip to the server.

Overlapping OLAP system with parallel processing was introduced in [1]. This work was implemented on real-time OLAP system to execute queries in parallel using multi-core processors. This approach improved query processing time from contemporary approaches.

Our work advances the research from [11]. Our intention is to apply parallel processing to do I/O heavy tasks (data download from the server, data processing in the client side and data storage in cache) for OLAP systems. As we have noticed that web-based system is lacking true parallelism and multithreading approach, we preferred Android-based mobile client to implement an OLAP-based system.

1.8. Thesis Organization

This report consists of 6 chapters as a whole. Chapter 2 talks about a brief overview of this application. Chapter 3 explains Android OS and parallel mechanism in details. Chapter 4 describes implementation of “Mobile based parallel OLAP processing”. Chapter 5 covers test results and performance evaluation. Finally, Chapter 6 concludes this report and also discusses future works.
Chapter 2.  Overview of Client-Server System

2.1. Mobile-OLAP Architecture

The architecture of our system is shown in Figure 2.1. There are three major components on the server side: OLAP server, ADOMD middleware and SOAP web service.

OLAP databases have more read than write operations. This system has the advantage of high-speed data access (since it is mainly read operations). The three basic analytical operations performed on OLAP are roll-up, drill-down, and slicing and dicing.

ADOMD is acting as middleware between the server and outside world. This accepts queries related to the data cube in MDX format. The SOAP web service is a separate programmable software entity, which is sandwiched between the ADOMD and the client. The sole purpose of this service is to ease the communication between client and ADOMD. In this project, this web service is processing the data from OLAP server. The response from MDX query is in XML format. This is further processed in web methods hosted under web service. These methods help to transform embedded cell data of the XML document into a simple array like structure which is returned to the client as JSON object.
Android base smartphone is our client and it has a restriction on the resource. The user interface has a tree view like presentation of dimension hierarchies and measures. This provides the platform to communicate between server and client. (Figure 2.1, client side part).

![Mobile OLAP Architecture](image)

**Figure 2.1. Mobile OLAP Architecture**

**Multidimensional Data and Metadata**

Each OLAP database contains one or more fact tables. Each such fact table stores records related to measures associated with dimensions. OLAP server represents fact tables in a form of a multidimensional data cubes. These data cubes store measure values from for each dimension.

Metadata contains information about dimension’s hierarchy. The information basically contains names of each node at every level of the dimension hierarchy tree. After
getting a request from the client, web service retrieves the metadata information from OLAP server and applies ADOMD to send XMLA Discover message to the client.

**XMLA**

XMLA provides standard XML format which defines the representation of multidimensional data types. Typically, XMLA is set of guidelines which need to be followed when passing XML message between client and server using Simple Object Access Protocol (SOAP). Execute and Discover are two major operations of XMLA [35].

Discover contains information related to metadata of the application. Metadata contains information about dimension, hierarchies, and measures.

Execute runs MDX queries on the OLAP server. Result from the execute contains information about the axis, dimensions, measures and their numerical values.

This information except the numerical value is redundant as they are already available at the client when metadata is downloaded. Excluding this information from XMLA response can reduce the volume of the data sent over the network.

**Cell Ordinals**

SOAP web service either can use ADOMD or XMLA to return back the response to the client. Our aim is to reduce the amount of data transferred over the network, and that is why we would like to avoid creating fact table. This guides to select XMLA over ADOMD.

XMLA’s ‘Execute message’ returns the leaves of each dimension’s hierarchy tree. They are described by a list of cell numbers where each cell number is generated by the Cartesian product of all selected dimensions. These cell numbers called CellOrdinal. This CellOrdinal number and aggregated numeric value of measures make a key value pair. We further parse the XMLA response to get only that key-value part discarding rest of the details. In this way, we can eliminate unnecessary data flow over the network channel. This list of numerical values and its association with CellOrdinal creates a table called ‘cell table’. Figure 2.2, 2.3, 2.4 describe a typical MDX query, its associated Fact Table,
corresponding cell table. Appendix A contains a typical example of XMLA request and response.

```
SELECT
    [[Geography], [Geography], [All Geographies], [Canada], children] ON AXIS (0),
    [[Date], [Calendar], [All Periods], [CY 2006], children, [Date], [Calendar], [All Periods], [CY 2008], children] ON AXIS (1)
FROM [Adventure Works]
WHERE [Measures], [Internet Sales Amount]
```

**Figure 2.2.** MDX query and its internal representation

<table>
<thead>
<tr>
<th></th>
<th>Alberta</th>
<th>British Columbia</th>
<th>Brunswick</th>
<th>Manitoba</th>
<th>Ontario</th>
<th>Quebec</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1 CY 2006</td>
<td>$3,805,710.59</td>
<td>$3,805,710.59</td>
<td>$3,805,710.59</td>
<td>$3,805,710.59</td>
<td>$3,805,710.59</td>
<td>$3,805,710.59</td>
</tr>
<tr>
<td>H2 CY 2006</td>
<td>$2,724,632.94</td>
<td>$2,724,632.94</td>
<td>$2,724,632.94</td>
<td>$2,724,632.94</td>
<td>$2,724,632.94</td>
<td>$2,724,632.94</td>
</tr>
<tr>
<td>H1 CY 2008</td>
<td>$9,720,059.11</td>
<td>$9,720,059.11</td>
<td>$9,720,059.11</td>
<td>$9,720,059.11</td>
<td>$9,720,059.11</td>
<td>$9,720,059.11</td>
</tr>
</tbody>
</table>

**Figure 2.3.** Associated Fact Table
Figure 2.4. **Cell Table and its internal mapping**

*Note.* Only the green part (cell table) of the right hand side will be transmitted to client side.

### 2.1.1. **Analytical Operations**

**Cross-tab**

Cross-tabulation (or cross-tab) is an aggregation of a measure based on given dimensions. A Crosstab collects data from fact table or data cube and generates a new table. In simple two-dimensional crosstab, the new table gets row headers from the values
of one dimension. Another dimension is used to generate the column headers. An example of a cross-tab on Geography and Date is displayed in Figure 2.5. The red block is generated from Geography-> All Geography->Canada dimension and the green block from Date->Calendar dimension. Cross-tab produces an M X N matrix. Here M is the number of elements in the first dimension and N is the number of elements in the second dimension.

![Crosstab on Geography and Date](image)

**Figure 2.5. Crosstab on Geography and Date**

**Roll-up**

Roll-up does aggregation on data from a data cube either by rising up a dimensional hierarchy or by a dimension reduction.

**Drill-down**

Drill-down is the opposite of roll-up. It aggregates data either by stepping down a dimensional hierarchy or by entering a new dimension [21].

**A typical user scenario**

A user wants following details:

“what is the total internet sales of Canada in the year of 2008?” He/she request this query using our application and visualizes this result. After seeing this, the user might be curious to get more details of it by asking a further query "what is the total internet sales of British Columbia during the 1st quarter of the year, 2008?". The second query is a “drill-down” from the original one and can be responded immediately from the data cache which we already filled up by downloading inflated query in first place.
Thus, any such subsequent queries can be answered from client side itself without giving a round trip to OLAP server. Figure 2.6, explains such drill-down and roll-up operation.

![Diagram of drill-down and roll-up operation](image)

**Figure 2.6.** Drill down and Roll up operation

### 2.1.2. Hash Based Pre-emptive Caching Architecture

In [11], a hash based algorithm is proposed which is capable of dealing with any shape of cached data and gives better performance for the subsequent drill-down and roll-up operations. In our project, we have also used this approach to implement a native client app in android. Here, we are assigning a unique id to each and every entry of dimension hierarchy starting from root to leaf. We are marking measures list in a similar way.
Thereafter, these numbers can represent user selection and can be useful to create a unique key combination from selected dimensions.

![Figure 2.7. User selection: Cell table](image)

Let us take an example. Suppose a user requests a query by selecting different dimensions. These dimensions are internally identified as X1, X2 and Y1, Y2. If we take cross product of each dimensions, there are 4 tuples: \{X1, Y1\}, \{X1, Y2\}, \{X2, Y1\}, \{X2, Y2\} (as shown in red, yellow green and blue rectangle boxes in Figure 2.7). All these dimensions have two children each (X1.1, X1.2, Y1.1, Y1.2 and so on). If we take cross product of all these leaves, we generate inflated sub-cube related to dimensions \{X1, X2, Y1, Y2\}.

After generating above cell table, we assign a unique id to each of such cell. Once we get the response for the sub cube from the server, we can extract the values and store in each cell. These cell values can be retrieved effortlessly by its associated cell ids.

We maintain a global hash table structure to save key-value pairs, where a key is the cell id and the value is the aggregated value of respective cells. This hash table can be used for future search for data from subsequent queries and it costs only \(O(1)\) time (in average case).
When we get a query which heads to a partial match from this hash table, we can exclude those matched cache cells and regenerate the MDX query containing only un-cached cells along with associated selected measures. We refer those cells as query cells. We are applying ‘Sort-Merge algorithm’ [11] on these query cells to reduce the count. Our goal is to minimize the number of query cells so that we can generate a simple MDX query and it, in turn, helps to improve the performance of the server side by returning only the missing data. In this way, we can achieve faster data retrieval from server. We combine both cached and downloaded data for user display. Our system will also update the hash table with un-cached cells for future use. A typical scenario is presented in Figure 2.8.

**Figure 2.8. Hash Based Pre-emptive Caching Architecture for OLAP system**
Chapter 3. Android based native application

3.1. Android OS Architecture

Android operating system architecture can be roughly divided into five main sections (Figure 3.1). Linux kernel, Libraries, Android Libraries, Android Runtime and Application Framework [22] [23]. Android is built on top of Linux 2.6 Kernel along with few...
architectural modifications by Google. Linux Kernel is responsible for managing process management, memory management, and device management. As the name suggests, libraries, next to Linux Kernel provides various useful libraries. These are java libraries made specifically supporting android operating system. Application Runtime section contains Dalvik Virtual Machine which is a more optimized version of JVM for Android. This utilizes similar functionalities from Linux to manage memory and multithreading and thus helps each android app to run its own process [23].

All mobile devices like smartphones have some restriction on memory size. In Andriod mobiles, Android’s Dalvik virtual machine allocates a certain level of heap memory per application. Hence, it is crucial for mobile based native applications to consume less memory.

Our mobile client is a native android Java app and it communicates with the server using ksoap library. ksoap is an open source library which provides effective SOAP-based communication between client and server for the Android platform [24]. However, unlike the client-centric architecture, this client does not have a data engine. It only has a visualization engine which is used to present the results from the application server to the user.

3.2. Android OLAP Application

Mobile applications are divided into three different categories: native, hybrid, and web apps [17], [18], [19].

- Native apps need to be installed from an application store (For e.g. Google Play). The user has to launch it every time before he/she can use it. The user experience in these apps is rich and these are capable of utilizing OS level features (like multithreading, parallel processing of Android OS). However, it carries the extra cost of platform-specific development.

- Mobile web apps are the ‘mobile version’ of the original desktop website. These apps are executed by a web browser and mostly utilizes HTML5. One benefit of these
kinds of apps is, there is no need to install them into the mobile and that is why user never faces space crunch issue. Almost, in all the cases, these apps rely on the internet and they are slower than native apps. Thus, a web app might not be a good option for a complicated functionality implementation.

- Hybrid apps are in between native and web apps. They are comparatively easy to build than native apps and perform better than web apps. Hybrid apps have options for native codes through which special features of the device can be accessed. Like native apps, these apps also require being installed. At the same time, they also need the browser to be embedded within the app. Hybrid apps is still not an option for more complex features.

We have chosen native application option for our case. Using this, we are utilizing Android OS level features to analyze data from OLAP server.

Any latest Android device (smartphone, tablet) has CPUs with multiple cores. This attests the opportunity that resource intensive jobs could be executed concurrently using multiple threads where each thread gets executed on a single processor. This can help to expedite the entire operation [16]. With the help of Linux-like process and thread management system, Android provides a single thread of execution environment for every single application. Android has a rich collection of inbuilt classes and Interfaces which can help to customize multithreading.

Using these features, we can concurrently download a large set of query data in the background while processing user’s request. Total time to complete all these in parallel will be \( \text{Max} (\text{task}_1, \text{task}_2 \ldots \text{task}_n) \) where \( \text{task}_1, \text{task}_2 \ldots \text{task}_n \) are time taken to complete each job. This expedites the whole process.

### 3.3. Android Marshmallow

Android 6.0 - Marshmallow is the most recent OS version of Android family. This version added few special features like fingerprint support, better battery life, flexibility on permission levels. It provides “Improved application performance and lower memory
overhead for faster multi-tasking” [31]. However, from a developer’s point of view, it is similar to its older version, Lollipop 5.0. Like its former version, Android “M” holds a rich and smooth powerful computing environment. It is 64 bit compatible and supports on ARM, x86, and MIPS architectures. Lollipop 5.0 introduced an exclusive runtime environment, ART runtime which is a combination of ahead-of-time (AOT), just-in-time (JIT), and interpreted code [32] [36]. ART helps to collect garbage efficiently and thus improves GC events’ performance. It optimizes performance for foreground users by dynamically moving memory. According to [36], this OS helps to gain 4 times better performance than ever. Plus, it compresses background apps and services to allow user do multiple tasks simultaneously. Developer can build Java based application and deploy it as native apps.

3.4. Parallel Processing in Android OS

Android started supporting multiple cores from Honeycomb OS. Multi-core processors assure performance improvement of the native applications. In Android environment, any application has the UI thread which is the main thread liable for controlling all the UI events.

Recent Android OS restricts permission to manipulate the UI to only the main thread. That means, additional threads (if any) in the application must pass data back to the main thread and this will eventually update the UI. It is a good practice not to overload the main thread with any time-consuming tasks (downloading huge amount of data, network operation, I/O operations etc.) The reason behind this is, this can hang the application for a while and yields bad user experience. The conclusion is, this should be managed in a separate background thread. Any native application can parallelize tasks using various parallelization methods provided by Android. [26] [27] [28].

3.4.1. Concurrent Execution of Multiple Threads

Android provides numerous interfaces and abstract classes which helps different tasks to execute without blocking the UI Thread of an application. For instance, abstract class AsyncTask helps to get an easy use of the UI thread. This class allows to perform background operations asynchronously and publish results on the UI thread as and when
required. However, this class is ideal for short operations (a few seconds at the most.) [29]. There are dedicated APIs like Executor, ThreadPoolExecutor and FutureTask which are used for time consuming operations.

In our application, we have utilized Executor class to execute and populate inflated query data in cache [16]. This utilizes a thread pool for tasks execution. The thread pool creates a certain number of ready to use threads (worker threads) and employ them when a request comes to the pool for a task execution. This is supported by a special queue structure of size 10 which is called "delayed queue". A task can wait in this queue before it gets access to any worker thread.

Earlier Android versions supported pool size of only 1. Hence, no parallel execution was possible. From Donut (1.6) the pool size has been increased to up to 5. Android 3.0 and above allows custom thread pool executor to configure the delayed tasks queue size. So, a customized number of threads can run in parallel. However, it is not advised to change the default settings due to certain application errors like InterruptedExecution and IllegalStateException [29].

In order to understand how Android handles parallel execution, we have done an experiment with 14 threads which is described below:

**Task:** We employed these threads to download small amount of data from the server (<5 KB in size) and process them on the client side.

We ran this application in Android Marshmallow 6.0 OS with quad code processor. We have captured the timestamps of all tasks from the internal log and analyzed the overall execution time of all these tasks. We observed following behaviors of the system.

- At time $t = 0$ (approximately), 5 tasks begin their execution at the same time. Rest 9 tasks remain in a queue and wait to get a free worker thread.
- At $t = k$, one of the first 5 tasks finishes, and its associated worker thread is available. It picks a task from the queue and starts executing. This repeats its cycle of selecting and executing jobs from queue until the queue is empty.
- At $t = n$, all assigned tasks complete their execution.
Figure 3.2 captures one of the instances during the execution of these 14 threads. This experiment confirms the fact that at any certain instance, there are 5 worker threads either running in multiple core or waiting for tasks to be assigned. If there are more tasks than available worker threads, those extra tasks can wait in delayed queue. All assigned tasks will eventually get chance to be executed using one of these worker threads.

![Diagram of Concurrent Execution](image)

Figure 3.2. Concurrent execution of multiple threads in Marshmallow 6.0

### 3.5. Control Over Threading

In the previous example, we have seen there were 14 tasks to be executed in parallel and witnessed how Android executes them. At a certain instance, there could be 5 tasks executing concurrently engaging 5 worker threads and rest 10 jobs can wait in delayed queue. We are curious to see how the system reacts when there more than 10 waiting threads, i.e., there are more than 15 jobs to be executed.
In order to simulate this scenario, we have created 17 jobs (same task like above experiment) and assigned them to 17 different threads. Now, we want to execute them in parallel in Android environment. Just like above, we have captured the timestamps from system log. We plotted these timestamps to generate a Gantt chart. X axis shows execution time; Y axis shows threads in execution. This is shown in Figure 3.3.

**Figure 3.3. Concurrent execution of 17 threads**

**Observation:**

- Average time to complete each task is 325 milliseconds.
- Out of 17, first 5 threads start right away.
- Execution of rest 12 threads follows exact path as mentioned in 3.4.1 section.
• Thread 5 took maximum time (519 ms) to complete the job. Hence, as per the convention, time to complete all these tasks should be

\[ \text{Max(tasks\,1,\,tasks\,2,\,tasks\,3\,\ldots\,\,tasks\,n) = 519\,ms.} \]

Note that total time to execute 17 thread is not equal to 519 ms. Rather, the total time to complete all these 17 tasks is 1219 ms. In fact, it is 
\[ \text{Max(tasks\,1,\,tasks\,2,\,tasks\,3\,\ldots\,\,tasks\,n) + C. \, C = some\, constant\, value.} \]

From Figure 3.3, we notice that all 12 threads are waiting to get free worker threads for first 182 ms. After this, another set of 5 threads gets the chance for execution and rest 7 still wait in the queue. Thread 17 has to wait till 1006 ms to get a free worker thread. This observation helps to realize that C is some numeric value imposed by context switching between threads. We can also conclude that the more the number of threads are, the bigger the value of C is. It is still better than the sequential execution of all 17 tasks which would take around 5518 ms to complete. Although, when the number of threads is very large, the overall execution time could be worse than its serial equivalent.

**Conclusion:**

1. Android executes parallel threads exactly the way mentioned in 3.4.1; be the number is 1 or 128 (maximum allowed thread). Please note that number of worker threads changes based on number of processors available. In our test environment, it is 5.

2. It is clear that applying a large number of threads is not the solution to get faster execution time. Some times it even works in reverse fashion. There is a trade-off between the number of threads involved in an application and its overall completion time. When thread execution involves finite resource (CPU, memory, I/O) consumption, concurrent usage can limit the performance of a system. It increases context switching which in turn worsens the performance. Usually, application with **number of cores + 1** or fewer concurrent threads can improve the execution time. We have witnessed from the previous experiment that Android with quad core allows 5 worker threads (4 + 1) to run 5 different tasks concurrently. If we adhere to this number, no more threads need to wait in the queue and consequently will not charge additional waiting time [37]. Although, this is not the only factor which controls better performance. We have discussed this in Chapter 5.
Chapter 4. Implementation of “Mobile based Parallel OLAP processing”

4.1. Basic Functionalities

Our system works as follows:

• User selects dimensions and measures from the UI and submits the selection.

• The system assigns a number (unique) for each of them based on their position in original dimension tree (hierarchy id) and measure list (measure id).

• These numbers are treated as unique keys. We generate all combinations of those keys.

• An MDX query containing user selection is generated and sent to the server (assuming no match found in cache).

• At the same time, we internally generate inflated MDX queries related to user selection. This mostly includes immediate parent and immediate children records of user selected dimensions. Using parallel mechanism, these queries also get executed along with user query. In most cases, the amount of data downloaded and processed using inflated queries are notably larger than user query data.

• The result from the server of these MDX queries come as a list of key-value pair in JSON format (cell ordinal number, value). We map those combinations with cell ordinal numbers and store the corresponding value in our hash table (cache).

• Any subsequent queries from the user can be checked against this hash table first and if any matches found (partial or full) those selections can be excluded. If it is a full match, we have the answer ready in the local cache. Else, we generate a filtered and shorter version of the original query consisting only uncached keys and fire that to fetch data from the server. We get the answer to the user query by combining data from both the cache (if match found) and data from server.

• This answer is displayed using Google chart API.

Figure 4.1 shows a typical user activity using our mobile application.
Figure 4.1. Display Screen (starting from top left)
Note: This OLAP client does not load the whole content of every dimension hierarchy. This is because the complete dimension hierarchy data is excessively large to be downloaded at once. Instead, we are loading first two level data at first and generate the tree view. Further data related to a specific dimension will be downloaded only when the user expands that specific node. Figure 4.2 explains the activity:

![ParallelOLAPProcessing](image)

**Figure 4.2. Dimension hierarchy selection from tree view**

The application loads up to two levels of dimensions namely Root (we call it here “Dimension” to distinguish root from other levels) and its immediate children. For example, ‘Geography’ is one of its children. Initially it does not have further levels inside it. When user clicks on Geography, it fetches all records for its descendants and populates as tree view inside it. Thus user can see Geography-> Geography-> All geography-> Canada-> Alberta-> Calgary-> <some postal code> The arrows in the figure imitates user clicks on every level.
4.2. Special Data Structures

4.2.1. Dimensions

In our project, we have used a special tree structure ‘TreeNode’ [14] to store dimension hierarchy. We have added 2 more properties and associated methods in it. TreeNode has following properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent of type TreeNode</td>
<td>This is the immediate parent of current node</td>
</tr>
<tr>
<td>Children of type List&lt;TreeNode&gt;</td>
<td>This is the immediate children list of current node</td>
</tr>
<tr>
<td>Reference of type generic</td>
<td>This is the data part.</td>
</tr>
<tr>
<td>Hierarchy Name of type string</td>
<td>This stores the complete hierarchy name of current node</td>
</tr>
<tr>
<td>Node counter of type integer</td>
<td>This is the unique integer number associated with current node</td>
</tr>
</tbody>
</table>

Table 4.1. TreeNode class properties and methods

It has various methods like `getParent()`, `getChildren()`, `getNodeCounter()`, `getHierarchyName()`, `addChildNode()`, `getReference()` etc. which are used to create and manage the tree view.

4.2.2. Cache

We are using a hash table to cache records in memory. It is a key-value paired data storage where key is a combined key generated from Cartesian product of selected dimensions. This key is associated with each CellOrdinals. Value is another set of hash table where key is measures and value is the aggregate number downloaded from the server. The structure looks like:

```
HashMap<String, HashMap<Integer, Long>> CachedDataCubes
```
Below is a snapshot of our growing cache (Figure 4.3) while user is using the application. The area in red border in the image is one such key-value pair. For key “799#1977”, the cache has two measure values (one for #0 and other for #3). When we expand the “Value” part, we can see aggregated value of respective measures in it. Now, if a user wants value for “799, 1977 dimensions with measure “3” or “0”, we can get the value from cache.

Figure 4.3. Cache Structure (HashMap)

4.3. External libraries

4.3.1. Connecting web service using kSOAP2

We are using the kSOAP2 library to communicate between the Android device and the SOAP-based web service. kSOAP2 has three components; XML parser, de/serializer, and a transport layer. In this project, we are using HttpTransportSE as our transport layer
Please refer to [40], it is a code repository of our application. The source code can be accessed from there.

4.3.2. Data display using Google chart API

We are using Google chart API for data visualization. This API is very rich and provides interactive user interface [20]. This API is supported by Android application. We are using WebView component. Please refer to [40] for accessing the code.

4.4. Data Split - CellOrdinal Rewrite

The response from the server comes in JSON format. It is a two-dimensional array containing a CellOrdinal number and aggregated value in each sub-array. CellOrdinals are continuous integer numbers starting from 0 and they are associated with all possible combinations of all selected dimensions and its leaves (if MDX query includes leaf level records) per measure. These sequence numbers must match with CellOrdinal numbers generated by XMLA. Without exact mapping between cell Ordinal and cell in cache, we can not be able to fetch records associated with each cell. Before querying to the server, we generate these CellOrdinal values and store them in a global list. After parsing the JSON response object, we populate measure values based on CellOrdinal numbers in the local cache.

Figure 4.4 shows a grid example consisting of only 2 dimensions, country and date. Country has 5 leaves (Canada, US, France, UK, Germany) and date has 4 leaves (2008: H1, H2, 2010: H1, H2). Cartesian product of these dimensions generates 20 cells. CellOrdinal values of all these are mentioned in each cell starting from 0 (top, left). Once we map the values of the response from the server, the value of cell 0 or CellOrdinal 0 will refer to the value of Canada - 2008:H1. Henceforth, we access cell 0 to fetch record for Canada - 2008:H1 quarter.
Figure 4.4. CellOrdinals

We maintain an internal map between these CellOrdinal numbers and dimension and measure names. This helps to fetch records for user query. Figure 4.5 shows an example of this. Hash table (top right in Figure 4.5) is the main cache which keeps all records. Rest are supporting data structures which help to maintain the mapping between user selection and internal representation.
4.5. Inflated Query Caching

We are forming four separate MDX queries from user's selection; three of them consist of immediate children and fourth one consists of immediate parents of each selected dimensions. These are four inflated queries associated with each user query. The details of these inflated queries are discussed in Chapter 5.
To cite an example, we can recall the MDX query from Chapter 2 (Figure 2.2).

```
SELECT {[Geography].[Geography].[All Geographies].[Canada].CHILDREN, [Date].[Calendar].[All Periods].[CY 2006].CHILDREN, [Date].[Calendar].[All Periods].[CY 2008].CHILDREN} ON AXIS (0), FROM [Adventure Works] WHERE [Measures].[Internet Sales Amount]
```

Its corresponding inflated MDX query containing immediate children will be:

```
SELECT 
    {[Geography].[Geography].[All Geographies].[Canada].[Alberta].CHILDREN, [Geography].[Geography].[All Geographies].[Canada].[British Columbia].CHILDREN, [Geography].[Geography].[All Geographies].[Canada].[Brunswick].CHILDREN, 
    [Geography].[Geography].[All Geographies].[Canada].[Manitoba].CHILDREN, [Geography].[Geography].[All Geographies].[Canada].[Ontario].CHILDREN, 
    [Geography].[Geography].[All Geographies].[Canada].[Quebec].CHILDREN} ON AXIS (0),
    {[Date].[Calendar].[All Periods].[CY 2006].[H1 CY 2006].CHILDREN, 
    [Date].[Calendar].[All Periods].[CY 2006].[H2 CY 2006].CHILDREN, 
    [Date].[Calendar].[All Periods].[CY 2006].[H1 CY 2008].CHILDREN, 
    [Date].[Calendar].[All Periods].[CY 2006].[H2 CY 2008].CHILDREN} ON AXIS (1)
FROM [Adventure Works] where [Measures].[Internet Sales Amount]
```

In order to split a bigger piece of job into smaller tasks, we are splitting this MDX query into three parts and generating small and simple MDX queries. We further employ three separate threads to handle these three MDX queries concurrently.

### 4.5.1. Inflated Query Data Fetch Strategy

We have taken a new strategy to fetch and process inflated query data.

First of all, we are avoiding large data storage in the cache (the reason is explained in 4.6 section). That is why we are only considering the immediate next and immediate previous level data of selected dimensions. At this moment, if the user performs a roll-up or drill-down operation on immediate parents or children of the same dimensions, we already have the data and we can display it from the cache.
Second, we are also trying to fetch most common combination of records so that it can raise the percentage of hits from the cache. For example, let us consider a case where a user selects A, B and C, D dimensions in his/her query with measure value as sales amount. Dimension hierarchy of A, B, C and D are described in Figure 4.6.

Now, when we generate an inflated query for its next level we also include current dimensions with that. If subsequent query drills down only in 1 dimension keeping rest unchanged, we still can have a chance of retrieving records from the cache. The query content looks like:

A1, A2, A3, B1, B2, B3, A, B on Axis 1
C1, C2, C3, D1, D2, D3, C, D on Axis 2
Corresponding cell table looks like Table 4.2:

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>A1, C1</td>
<td>A2, C1</td>
<td>A3, C1</td>
<td>B1, C1</td>
<td>B2, C1</td>
<td>B3, C1</td>
<td>A, C1</td>
<td>B, C1</td>
</tr>
<tr>
<td>C3</td>
<td>A1, C3</td>
<td>A2, C3</td>
<td>A3, C3</td>
<td>B1, C3</td>
<td>B2, C3</td>
<td>B3, C3</td>
<td>A, C3</td>
<td>B, C3</td>
</tr>
<tr>
<td>D1</td>
<td>A1, D1</td>
<td>A2, D1</td>
<td>A3, D1</td>
<td>B1, D1</td>
<td>B2, D1</td>
<td>B3, D1</td>
<td>A, D1</td>
<td>B, D1</td>
</tr>
<tr>
<td>D2</td>
<td>A1, D2</td>
<td>A2, D2</td>
<td>A3, D2</td>
<td>B1, D2</td>
<td>B2, D2</td>
<td>B3, D2</td>
<td>A, D2</td>
<td>B, D2</td>
</tr>
<tr>
<td>D</td>
<td>D, A1</td>
<td>D, A2</td>
<td>D, A3</td>
<td>D, B1</td>
<td>D, B2</td>
<td>D, B3</td>
<td>A, D</td>
<td>B, D</td>
</tr>
</tbody>
</table>

Table 4.2. Inflated query: All possible combinations

If user selects A, B on Axis 1 and drills down on C1, D1 on Axis 2 in succeeding query, it is still a hit. We can answer any combinations from green cells along with all combinations from grey cells. This approach shows good experimental result in terms of number of query hit from cache.

### 4.6. Cache Deletion Mechanism

All smartphones have limited resources available to use. A user may select multiple dimensions along multiple axes. If we combine all descendants of all dimensions, the number of CellOrdinals will be excessively large and so does the downloaded data. Our global cache size will grow by collecting all uncached records from all these queries. If we are not mindful of this size, the caching mechanism may lead to system failure. That is why we have decided to download only immediate next level children of selected dimensions. We are also downloading immediate parent level records. This helps us to increase the percentage of the hit from the cache. We are also maintaining a cache deletion policy to control over the size. As mentioned in Chapter 1 under Hypothesis section, caching in our application is based on the most common behavior of a user. If a user stays with his/her previously selected combinations and if the successive queries are either drill-down or roll-up on same dimensions and measures, chances are high to get respective answers from the cache. However, if previously selected dimensions do not match with user’s current selection, we can assume that the user is not interested in old
dimension records anymore and wants to explore different entries from different dimensions. So, we can delete past entries related to old selection. This concept is analogous to LRU (least recently used) approach. This also helps to maintain the cache size. The implementation logic is explained in next section:

- We are maintaining a dimension selection history by tracking root dimension names of every selected dimension.
- Every time the user requests for data, we are comparing user selection with user selection history.
- If we found a match, we keep existing records in the cache.
- On the other hand, if no match found, we can safely delete records from the cache associated with old selection.

For example, say, a user selects Geography-> All Geographies->Canada->British Columbia as dimension along with sales as measure. So its root dimension is Geography. We store Geography in our record. Next, the user selects Geography-> All Geographies->France. Its root dimension is still Geography. We check with history and keep existing cache records. If the user selects Products->All Products->Bikes, the root dimension is Product which is not matching with Geography. If Product entry is not in history records, i.e., user never searched for Products before, we can assume that user is deviating from previous selections and growing interest in a different dimension. Therefore, we can remove entries related to Geography and update history as well as the cache with Product related records.

Subsequent queries on the same dimension might not always give 100 % hit from the cache. It could be a partial match (along with few new dimension entries) or even no match (same dimensions but new measures). In all these cases, we keep existing records in the cache and add missing records for future use.

We are also maintaining a threshold size of 10 MB for the cache. If the cache size exceeds this threshold, we are flushing all the records and storing new data in it. This might cause cache miss but this assists us to control the cache size and in turn, helps to use limited resources wisely.
4.7. Running All Tasks in Parallel

The most important part of our project is to improve I/O of the application by using multithreading. We split our tasks into two parts:

- Fetch records for user requested query (the part which is not cached)
- Create corresponding inflated query to fetch records from the server and populate cache

We are using multiple threads (in our experiment up to 5) to do these two jobs. In Android native app, there is always the main thread running the application. In a multi-threaded application, like ours, this main thread branches out separate threads to do specific tasks. Figure 4.7 shows a snapshot of our application working on multiple threads.

For parallel execution of threads, we are using an Executor framework provided by Android. This helps to maintain a pool of threads with a job queue. Using ThreadPoolExecutor, multiple threads can execute tasks in parallel. The Executor interface is the fundamental component of the Executor framework. Its execute() method
executes the command passed to a thread. This interface provides a way of redefining task submission by including minute details like how each task will be run, details of thread use, scheduling, etc.
Chapter 5. Experimental Results and Discussion

We started this project with 4 specific questions in mind which are discussed in Chapter 1. At the end of the work, we try to evaluate if we get satisfactory answers for all of these questions.

5.1. Experimental Results

As discussed in Chapter 1 under Research Objective, in [11], the user's request and corresponding inflated query data download are done sequentially. This can work well for smaller data size (less than 2 MB). In the case of larger data processing (more than 5 MB), there can be a situation where the user might have to wait for a while before the inflated sub-cubes are available in client-side. This process will slow down the application and yield bad user experience. Moreover, we have also witnessed cases where system running on serial mode freezes in the above situation.

In our approach, we tried to overcome these limitations by employing multiple threads to perform data downloads in parallel. We are utilizing user's waiting time for his/her own requested data by firing all queries asynchronously and concurrently. By the time user visualizes requested data and fires subsequent drill-down or roll-up operation, the inflated query data is ready in client side.

5.1.1. Serial vs Parallel Approach

[11] was a web-based OLAP application. Ours is a native application running on Android OS. Hence, it is not justifiable to compare the performance of a web-based app with a native app. Keeping this in mind, we have implemented a web equivalent native app in Android which runs on a single thread. We also have created 4 different versions of same application by engaging 2 threads, 3 threads, 4 threads and 5 threads. Implementation logic behind all these applications is exactly same. They differ only by the number of threads. These applications are tested over Adventure Works Dataset. The intention is to compare the performance of parallel execution with serial one. We have
tested over a wide range of data size (from 4 KB to 6.8 MB). Each MDX query (user query and its associated augmented or inflated query) is executed in two different approaches, one sequential and other in parallel. We ran the same set of test scenarios for all 5 approaches and collected the time taken to execute each test case. We collected all test results from system log and analyzed them to generate Gantt chart of each test scenario.

Note:

• All User Query task includes generating associated MDX query, downloading it (if not cached) from the server, process it and display it using Google chart API.

• All Inflated Query task includes generating associated MDX query, downloading it from the server, process it and store it in cache.

• All time is measures in milliseconds. We have repeated same test cases (for both the approaches) 3 times and calculated average time for each operation.

**Performance Factor**

In this experiment, we are focusing on application efficiency. Each test result focuses on capturing the answer of the question "how fast an operation can be executed?". Hence we are considering only one parameter, namely total execution time which is measured in milliseconds.

**Data Set**

AdventureWorks is a sample database provided by Microsoft SQL Server. The data is related to a fictitious large, bicycle manufacturing company [39]. We are using this data warehouse as our data set.

**Task**

Each of these approaches performs the following tasks:

1. For an instance, let us assume that user requested following dimensions for a certain measure:
Let us also assume that A1, A2, B1, B2, C1, C2, D1, D2 are children of A, B, C, D respectively. We are generating 3 different inflated query to download data for future reference.

Query1: Generate query with 1 level down for half of the selected dimensions, keeping rest unchanged.

Query2: Generate query with 1 level down for half of the selected dimensions, keeping rest unchanged.

Query3: Generate query with 1 level down for all of the selected dimensions.

Query4: Generate query with 1 level up for all of the selected dimensions.

The reason behind generating all these queries are twofold, one, data size from 1 level below of current dimensions can be immense (as per our results shows, it can be few MB). We want to split this kind of big task into small jobs and assign these jobs to individual threads. We are trying to balance the load of each thread (as much as possible). Thus, each thread will be responsible for a controllable amount data. Two, in this way, we are covering all possible combinations of current dimensions and its immediate children.

We did not split query 4 since we have witnessed from the result that in most of the case (especially in our example) amount of data associated with immediate parents of current dimensions are not huge. Hence, one thread can manage the whole work.
**Test Environment**

<table>
<thead>
<tr>
<th>Client</th>
<th>Android OS</th>
<th>Marshmallow 6.0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM Heap Size Limit</td>
<td>201 MB</td>
<td></td>
</tr>
<tr>
<td>Allocated VM Memory</td>
<td>16 MB (16 MB ~ 60 MB)</td>
<td></td>
</tr>
<tr>
<td>VM Heap Size</td>
<td>26 MB (26 MB ~ 38 MB)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Back-end Server</th>
<th>Dedicated Server for this Testing:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Windows Server 2008 R2 64bit</td>
</tr>
<tr>
<td></td>
<td>48 GB Memory</td>
</tr>
<tr>
<td></td>
<td>Intel Xeon X5650 @ 2.67GHz, 2 processors, 24 cores total</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OLAP Server</th>
<th>VANPGC13B11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SQL Server Analysis Service 2005,</td>
</tr>
<tr>
<td></td>
<td>Windows Server 2003 64-bit SP2</td>
</tr>
</tbody>
</table>

| Dataset                    | Adventure Work                      |

<table>
<thead>
<tr>
<th>Network Bandwidth</th>
<th>The client will be simulated to have the upload and download limits of a typical ADSL connection, 1Mbits/sec upload and 10Mbits/sec download.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Latency</td>
<td>The front-end machine will be simulated to have approximately 30/90/150 ms roundtrip latencies to the back-end servers.</td>
</tr>
</tbody>
</table>

**Table 5.1. Test environment details**

Note: Our application is also tested in Tablets with Lollipop 5.1 OS. This application can be downloaded and installed in Tablets without any major change in code.

**5.1.2. Evaluation**

Figure 5.1 shows comparison of the two approaches. Here, parallel execution is using 2 concurrent threads to process 4 KB data. In case of parallel execution, both the tasks (user query and inflated query) has started almost at the same time and the total time to complete whole operation is **491 ms**. Total Time can be expressed as

\[
T(n) = \max(\text{task}_1, \text{task}_2) + C
\]

where C is some constant time dedicated for context switching
On the other hand, in case of sequential execution, user query gets executed first and displayed to user and then inflated query download starts. It executes one after the other. Serial approach takes \textbf{810 ms} time to complete same amount of task of parallel approach. Total time for this case is

\[ T(n) = \text{sum (task}_1\text{+task}_2) + C. \]

where \(C\) is some constant time dedicated for internal operations done by Android OS

It is obvious from this result that our approach outperforms old approach with sequential execution. This claim also applicable for any size (we are restricting ourselves up to few MB since our client is a mobile device and it has its own resource limitation) of data download. We have done this comparison by varying the data size and number of threads and we always achieved better performance than serial approach.
This answers our first question “**Does the application perform better than web based sequential approach when we apply multitasking and parallelism?**” The answer is yes. It does.

In order to answer the second question, we started experimenting with varying number of threads. Figure 5.2, 5.3 and 5.4 display comparison between sequential and parallel execution on a bigger amount of data set (medium to large data set size with respect to increasing number of threads).

![Parallel Execution Data Size: 308 KB](image1)

![Serial Execution Data Size: 308 KB](image2)

**Figure 5.2. Performance Comparison: Set 2**

Here, we have employed 3, 4 and 5 threads respectively and observed that it is certainly beneficial to employ more threads. Therefore, the answer to the second question “**If we keep adding threads, does that verify the fact that we keep getting better performance?**” is also yes when the number of threads are less than the optimal number. As per the convention, in quad core processor like ours, we can achieve optimal performance by employing 5 (4+1) threads in parallel (please refer to Chapter 3 for more details on this). However, as we have noticed in Chapter 3, if we keep adding threads beyond the optimal number, we might lose performance gain and even end up getting worst performance for large number of concurrent threads. So, there is a tradeoff between
associated issues with an increasing number of threads and faster execution of tasks. One has to find the optimal number of threads which will help to accelerate the overall process with minimal amount of context switching and minimal amount effort for thread handling.
In our case, we have limited our whole experiment up to 5 threads since we noticed that

Figure 5.3. Performance Comparison: Set 3

Figure 5.4. Performance Comparison: Set 4
Figure 5.5 shows a special case where the sequential application stopped working. In this case, the comparison is between 3 threads running parallel vs sequential execution where the data size is 3.4 MB. This is the distinct case which shows that serial approach is not good for longer running operations which involve heavy data download and processing. The serial execution reveals sensitive behavior after a certain amount of data size. By sensitive, we mean this approach unexpectedly stops working in some instances and abruptly closes the application. We have observed system failure in various instances when the total amount of data size is around 3 MB or more (witnessed failure at 2.7 MB, 3.1 MB, 5MB, 6 MB and beyond). The tentative cause is discussed in section 5.1. Although, the exact reason behind is still unexplored.

Approach with parallel threads also exhibits above behavior but it supports bigger data size than the serial one. We have witnessed this behavior when the data size reached 14 MB or more. This number varies based on the number of threads involved in the application (14 MB for 4 and 5 threads, it is little less than 14 MB for 3 threads and less than 10 MB for 2 threads) and amount of resource allowed by the client (Virtual memory, heap size etc). This special case answers question number 3 “Are there any instances found where serial approach fails but parallel survives?”
5.2. Overall Picture

In the end, we tried to get an overall picture of our experiment. We want to know how each of these 4 approaches (2 threads, 3 threads, 4 threads and 5 threads) is performing against serial thread or no threaded approach. In order to visualize the comparison, we used previous results and plotted as line chart. Figure 5.6 portrays this comparison over a variation of data.

![Time vs. Data (For all 5 Approaches)](chart)

Figure 5.6. Overall comparison of two approaches by varying data size and threads used previous results and plotted as line chart. Figure 5.6 portrays this comparison over a variation of data.

In this line graph we plotted total execution time taken over various data size. Y axis represents total time (in milliseconds) per user operation (user query download, data display, associated inflated query data download, processing and storage in cache). X axis represents data size in KB. Figure 5.7 and 5.8 give a comparison between serial execution and 2 threads, 3 threads, 4 threads and 5 threads respectively. Figure 5.7 and 5.8 are extracted from Figure 5.6. Figure 5.7 contains two line graphs, one shows a
comparison between 2 threaded and serial approach and another shows 3 threads and serial approach. 2 threaded approach does not provide much performance gain over serial approach. However, 3 threaded approach shows better performance than 2 threaded one. Like Figure 5.7, Figure 5.8 focuses on 4 threaded and 5 threaded approach. As depicted in the figure, switching from 3 threads to 4 has given a notable performance gain. This also followed in 5 threaded approach.
Figure 5.7. Overall comparison: 2 threads and 3 threads

Figure 5.8. Overall comparison: 4 threads and 5 threads
5.2.1. **Overall Observation**

The following are few important observations we made from all these test results.

- Execution time varies from few milliseconds to few seconds.

- We can see from the result that serial execution of tasks always takes more time than all others which use the parallel execution of tasks. When the data size is small, overall performance gain is not significant. The difference gets prominent when data size and the number of threads are getting larger.

- Among all, the one with 5 threads yields the best performance.

- The more we add the number of threads, the better the overall performance is. We can keep increasing number of threads to split the task but we have to consider an optimal number and the reason is discussed in details in Chapter 3.

- If we look at the overall performance, we notice that this depends on the number of threads involved, the amount of data, the number of dimensions, and most importantly equal distribution of tasks among threads.

- If the number of dimensions is high, (Table 5.2, case 5,6 with 7 dimensions), parallel performs significantly better than serial. In fact, serial fails to download and process all those combinations within time, whereas parallel can complete in a reasonable amount of time. So, this special case can be considered as if it is 100 % gain over serial version.

  When we distribute the tasks almost equally among the threads, we gain better performance (case 4 of Table 5.2). However, this not the only factor which controls the performance. It is also controlled by rest of the factors as mentioned above. Table 5.2 gives a glance of overall performance gain. To draw a supportive example, we can compare case 4 and 5. Case 5 gains better performance than case 4 even though case 4 has distributed the task equally among all threads.
<table>
<thead>
<tr>
<th>Case No</th>
<th>Total Data Size</th>
<th>Threads</th>
<th>Task handled per thread (%)</th>
<th>No. of dimensions involved</th>
<th>Performance gain over serial approach (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>940 KB</td>
<td>Inflated Query1</td>
<td>2.90</td>
<td>4</td>
<td>8.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query2</td>
<td>4.68</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query3</td>
<td>89.79</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query4</td>
<td>2.64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>652 KB</td>
<td>Inflated Query1</td>
<td>1.53</td>
<td>5</td>
<td>39.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query2</td>
<td>24.85</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query3</td>
<td>72.24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query4</td>
<td>2.93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4.1 MB</td>
<td>Inflated Query1</td>
<td>0.24</td>
<td>5</td>
<td>47.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query2</td>
<td>24.39</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query3</td>
<td>75.61</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query4</td>
<td>0.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6.8 MB</td>
<td>Inflated Query1</td>
<td>25.00</td>
<td>4</td>
<td>58.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query2</td>
<td>25.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query3</td>
<td>25.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query4</td>
<td>25.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.43 MB</td>
<td>Inflated Query1</td>
<td>0.70</td>
<td>7</td>
<td>81.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query2</td>
<td>23.43</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query3</td>
<td>67.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query4</td>
<td>8.58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2.7 MB</td>
<td>Inflated Query1</td>
<td>0.17</td>
<td>7</td>
<td>system crushes for serial version ~ 100 % gain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query2</td>
<td>25.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query3</td>
<td>68.93</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inflated Query4</td>
<td>5.65</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.2. Overall performance gain**

Note: We are not including the thread related to user request here as this thread is taking care only user requested data and compared to threads involved in inflated data processing, the % or task for this thread is very low (most of the cases it is < 10%).

The last observation sheds some light on the factors responsible for performance gain (as mentioned in question 4: “What are the factors which lead to notable performance gain over serial approach?”). Although, we are skeptical to conclude on this. We believe further research must be conducted to identify all factors (if there are more) responsible for performance tuning.
Chapter 6. Conclusion and Future work

6.1. Conclusion

This project investigated and evaluated OLAP application in app-based client where the resource is a constraint but the parallel execution in multi-core CPU is possible.

We implemented the web equivalent in Android OS (Marshmallow 6.0) and applied concurrent execution of multiple threads. Since we have resource restrained client, we have implemented the policy to release cache content and make space for new data. Further, we restricted ourselves to download only immediate children and immediate parents of the current selection.

Despite the fact that web-based client is popular, its performance is not that satisfactory when it is compared with its equivalent native app.

As per the experimental result, this serial approach demonstrated inferior result to the parallel approach.

In Chapter 5, we acknowledged all questions which we discussed in Chapter 1. Overall completion time of our approach is invariably better than the serial execution of tasks. When the data size is small (< 1 MB) the performance gain might not be significant but our approach stands out when data size is large (few MB). The test result shows cases where performance gain over serial approach reach even 80 - 100%. We have also discussed the determinants which influence performance gain. We have witnessed that task distribution among all threads is one of the important factors which controls performance.

Above all, our new strategy sustains during large inflated data download in the background whereas its equivalent sequential mechanism fails to even complete the whole process.

Hence, we can conclude that distributing jobs among multiple threads and running them in parallel can result in better execution time. It is true that web-based clients fail to
provide real parallelism. That is why when it comes to decide which approach to choose for tasks similar to ours, app-based client with parallelism wins the game.

6.2. Future work

There are several possible areas which may further influence our results.

First, in our approach we have downloaded only up to 1 level below and 1 level up data of current dimensions. We can continue the experiment for various levels, e.g., 2 level below and 2 level up data and observe the performance.

Second, we can investigate cases where the system fails to perform operations. In the case of serial execution threshold is much lower than parallel approach. We believe if we get the better insight of the reason and factors behind it, we can apply the knowledge and tune up the overall performance.

Third, if we know all the factors which are controlling performance gain, we can improve our task distribution policy and this might help to get a better result. A detailed investigation is needed in this area too.

Finally, the user interface part of our native application on Android platform can be improved further. At present, we are lacking fast user interaction in terms of capturing user selection (user needs to drill down each and every level of dimension hierarchy to select the desired dimension and finally submit the query; this takes a couple of seconds). That is why the user is unable to find a considerable difference between two approaches except the case where serial approach halts the system. A good user interface in the foreground along with parallel execution of multiple threads in the background not only enhance the overall user experience but also strengthen the quality of the application. Hence, if we can provide a rich user interface, this can accelerate user interaction and user can sense the difference between parallel and serial execution of background tasks.
References


[19] “Choosing Between a Native, Hybrid, or Web App”, www.crew.co/how-to-build-an-online-business/native-hybrid-web-app-differences/


[27] “Running in a Background Service”, developer.android.com/training/run-background-service/index.html


[37] “Why using more threads makes it slower than using less threads”, 2013, unix.stackexchange.com/questions/80424/why-using-more-threads-makes-it-slower-than-using-less-threads/80425#80425


[40] “ParallelOLAPProcessing”, 2016, github.com/kheyalimitra/ParallelOLAPProcessing.git
Appendix A.

XMLA Request

```xml
<Execute xmlns="urn:schemas-microsoft-com:xml-analysis">
  <Command>
    <Statement>
      SELECT {[Geography].[Geography].[All Geographies].[Canada].children} on axis(0),
      {[Date].[Calendar].[All Periods].[CY 2006].children,[Date].[Calendar].[All Periods].[CY 2008].children} on axis(1)
      FROM [Adventure Works]
      WHERE [Measures].[Internet Sales Amount]
    </Statement>
  </Command>
  <Properties>
    <PropertyList>
      <Format>Multidimensional</Format>
      <AxisFormat>TupleFormat</AxisFormat>
    </PropertyList>
  </Properties>
</Execute>

<Discover xmlns="urn:schemas-microsoft-com:xml-analysis">
  <RequestType>MDSCHEMA_CUBES</RequestType>
  <Restrictions>
    <RestrictionList>
      <CATALOG_NAME>Adventure Works DW 2008R2</CATALOG_NAME>
    </RestrictionList>
  </Restrictions>
  <Properties>
    <PropertyList>
      <Catalog>Adventure Works DW 2008R2</Catalog>
      <Format>Tabular</Format>
    </PropertyList>
  </Properties>
</Discover>
```

XMLA Response

```xml
<Cell CellOrdinal="0">
  <Value xsi:type="xsd:double">3805710.7571</Value>
  <FmtValue>$3805710.7571</FmtValue>
</Cell>
<Cell CellOrdinal="1">
  <Value xsi:type="xsd:double">3805710.7571</Value>
  <FmtValue>$3805710.2513</FmtValue>
</Cell>
<Cell CellOrdinal="2">
  <Value xsi:type="xsd:double">3805710.7571</Value>
  <FmtValue>$3805710.2513</FmtValue>
</Cell>
```

59
<table>
<thead>
<tr>
<th>Cell Ordinal</th>
<th>Value</th>
<th>FmtValue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3805710.7571</td>
<td>$3805710.7571</td>
</tr>
<tr>
<td>2</td>
<td>3805710.7571</td>
<td>$3805710.7571</td>
</tr>
<tr>
<td>3</td>
<td>3805710.7571</td>
<td>$3805710.7571</td>
</tr>
</tbody>
</table>

......