Query Processor for the Conceptual Integration Modeling Framework

by

Zhaohong Chen

B. Eng. in Software Engineering, Zhejiang University, 2009

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

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

January 2012

© Zhaohong Chen, 2012
Abstract

A data warehouse is a giant information pot made by integrating data from different data sources. Its schema can easily become complicated in order to hold all historical data. This complexity can confuse end users. The Conceptual Integration Modeling (CIM) framework was proposed to bridge the gap between users and the schema by adding a user-defined conceptual layer. This thesis aims to fulfill the query processor component in the CIM framework. Traditional query languages like SQL require a relatively tough learning curve, which is too dynamic for business users. Instead we propose a new CIM query language and a query interface to work with the visual representation of the CIM framework. This query interface only focuses on user needs and hides unnecessary implementation details to the back end. Thus, it is easy for business users to pose queries. In addition, we also do some optimization on the query processor by selecting views to materialize. Our proposed algorithm can achieve the lower performance bound of 46.7% of the optimal solution.
# Table of Contents

Abstract ......................................................... ii

Table of Contents ........................................ iii

List of Tables ............................................... v

List of Figures ............................................... vi

Acknowledgments ........................................... vii

1 Introduction .................................................. 1
   1.1 Motivation ............................................... 2
   1.2 Proposed Solution ........................................ 4
   1.3 Contributions ........................................... 4
   1.4 Organization of the Thesis ............................... 5

2 Background .................................................. 6
   2.1 Data Warehouse .......................................... 6
      2.1.1 Functionality ....................................... 6
      2.1.2 Schema Design ...................................... 7
      2.1.3 Design Methodology ................................. 10
   2.2 Conceptual Integration Modeling Framework ............. 11
      2.2.1 Motivation .......................................... 11
      2.2.2 Goal ................................................ 12
      2.2.3 Architecture ........................................ 13
      2.2.4 Heterogeneity of Hierarchy ......................... 16
3 Related Work .......................................................... 18

4 Query Processor Framework ........................................... 20
  4.1 CIM Query Language ........................................... 21
      4.1.1 Query Language Design Principles .................. 22
      4.1.2 Query Language Definition ......................... 23
      4.1.3 Visual Query Interface Description ............... 26
  4.2 SQL Query Compilation ....................................... 29

5 Query Optimization ................................................... 35
  5.1 Motivation of our New Solution ............................. 36
      5.1.1 Goal and Input ..................................... 36
      5.1.2 Lattice Framework ................................ 37
      5.1.3 Candidate View Reasoning ......................... 39
      5.1.4 Cost Estimation for Queries ....................... 43
      5.1.5 Space of Views .................................... 45
      5.1.6 Problem Definition ................................ 46
      5.1.7 Benefit Function .................................. 46
  5.2 CIM View Selection Algorithm ............................... 47
      5.2.1 Query Rewriting Rules with Materialized Views ... 47
      5.2.2 Submodularity and CIM ............................. 48
      5.2.3 Algorithm Description ............................. 52

6 Conclusions and Future Work ...................................... 60
  6.1 Conclusions ................................................... 60
  6.2 Future Work .................................................. 61

Bibliography ............................................................. 62
List of Tables

Table 5.1  Attributes and Parent-child Relationships in the Simple Sales Example .......................... 44
Table 5.2  Cost and Space for Attributes and Parent-child Relationships in Simple Sales Example ............ 54
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Star Schema</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Snowflake Schema</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>CIM Architecture Diagram</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>CIM Example [21]</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Visual Query Interface</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Query Processor Architecture</td>
<td>29</td>
</tr>
<tr>
<td>5.1</td>
<td>Lattice Representation of 3 Dimensions</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Query Processor Architecture with View Materialization</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Conceptual Schema for Simple Sales Example</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Lattice for Simple Sales Example</td>
<td>44</td>
</tr>
<tr>
<td>5.5</td>
<td>Conceptual Schema with Costs of Element ID+measures for Simple Sales Example</td>
<td>55</td>
</tr>
</tbody>
</table>
Acknowledgments

First, I would like to express my gratitude for the help and care from my supervisor Dr. Rachel Pottinger. Without her advice and feedback, I would not have done my thesis. However, Rachel is more than a supervisor. She is a friend, and helps students find their interests and provides advice. Her continuous support and trust allowed me to have space and time to regularly reflect on my own, contemplate life and truly grow.

I would also like to thank Dr. Ed Knorr as my second reader. Ed has provided great feedback and raised some very good questions. In addition, it was of great joy to be a TA working with Ed.

The CIM project mentioned in the thesis is a collaborative research project. I appreciate everybody involved in the project for their time and devotion. Without the framework developed, the query processor component would never have been built. I am also glad to have worked with the DMM lab mates. The friendly environment and those casual chats are memorable.

Last but not least, I am truly thankful for being admitted by UBC and spending my best two years in Vancouver. Through these two years I grow a lot and have a much deeper understanding about what I would like to pursue in the future. I appreciate those meaningful talks with all my friends to form my perspective towards life. Of course, I am grateful for the continuous care from my parents, my brother and Jingya.
Chapter 1

Introduction

A data warehouse is a giant information pot used for collecting data from different data sources. Online Analytical Processing (OLAP) is used to answer business analytical queries on top of a data warehouse. OLAP queries are multidimensional in nature, so users may query measures of interest across different dimensions. Consider a company which distributes products from their multiple international locations: a possible query could be to measure the total revenue for each location, or for each product sold. In this scenario, location and product are dimensions of analysis and total revenue is the measure of interest.

An ideal data warehouse is organized with multidimensions spanning from a table of measures to provide convenience for OLAP analysis. However, it is not always possible to have such a simple schema for data warehouses. Data sources, which hold data in traditional Online Transactional Processing (OLTP) databases, have limited storage and generally process many updates that overwrite existing values, often subject to high performance and concurrency goals. In contrast, OLAP queries usually focus on historical data to show trends to business users, which requires the relevant data warehouse to hold that piece of data.

To allow decision makers to make ad-hoc queries without losing data due to data sources’ regular or periodic updates, data warehouses tend to be huge and cover as many aspects of the data sources as possible. Thus the schema of a data warehouse can be very complicated rather than just a simple star or snowflake schema. Tens or even hundreds of tables is the norm for a data warehouse. After
all, the most important goal of a data warehouse is to house data from different
data sources for future analysis.

To overcome the mismatch between the complicated data warehouse schema
and the multidimensional property requirement of OLAP queries, Rizzolo et al.
[23] proposed a Conceptual Integration Modeling (CIM) framework to solve this
problem by providing a semantic layer (also called a conceptual layer interchange-
ably in this thesis) between users and the data warehouse. This semantic layer will
be created by business users, reflecting their needs from the data warehouse. In
addition, different types of users can build their own semantic layers on top of the
same data warehouse. More details of the CIM framework can be found in the next
chapter. In summary, the CIM framework has the following three advantages.

- **CIM can make users focus on their needs.** A data warehouse is usually too
  large for a specific type of user. For example, the sales department may only
  need 10 tables from the data warehouse. While the data warehouse may have
  100 tables, 90 of them do not provide help to the sales department. With a
  semantic layer, users can focus on their needs and not get distracted.

- **CIM can let users express queries from their perspectives.** Schemas in
  the data warehouse aim to provide a universal way to incorporate data from
  multiple data sources, from the schema designers’ perspectives. It may not
  reflect business users’ needs. In such a case, there is a leap for users when
  posing queries on the data warehouse. In CIM, users can create the semantic
  schema in the way corresponding to their query requirements.

- **CIM can provide proper access rights to different users.** A data ware-
  house usually holds all kinds of information. But not all users should have
  the same access rights over the data warehouse. For example, engineers
  should not be exposed to the financial information of a company. Maintain-
  ing a semantic layer can help solve this problem.

### 1.1 Motivation

Before the introduction of the semantic layer, users needed to understand the huge
data warehouse to pose queries. To successfully pose a query to express users’
needs, they needed to understand the schema and find the relevant information. With the help of a semantic layer, users can narrow down the scope of query components from the data warehouse to the semantic layer they create, which is more focused and easy to understand.

Note that the semantic layer is constructed from users’ perspectives. More specifically, we consider the end users, who are business users instead of database experts or programmers. It is quite likely that business users are not familiar with query languages like SQL (assuming our data warehouse is relational). Therefore, SQL as a query language on the conceptual schema does not match the benefit of this semantic layer. Making a convenient query interface for business users, who do not have database or SQL background, to pose queries on this semantic layer can make it more usable to them.

Besides the CIM framework proposed in [23], Shakya [26] defined the data model of the conceptual schema, which focused on creating elements to support the conceptual schema. The query processor for CIM is needed to help users pose queries with ease. Following the design philosophy of the CIM project, which lets business users create the conceptual schema from their perspectives, we would like to make a new query language and query interface that are more natural and usable for users to express queries.

On the other hand, besides usability, we cannot ignore the efficiency of a query processor. A query processor can translate queries from the new query language to SQL and send them to the data warehouse (also called the central database) to execute. However, this data warehouse will be under high pressure with queries. If we analyze the characteristics of these OLAP queries posed by users, we find they are actually multidimensional in nature. Furthermore, optimization is possible because queries of coarser granularity can make use of query results of finer granularity. In this case, the view selection technique in Relational Database Management Systems (RDBMSs) can be helpful if we can select some representative views to materialize, which can shorten query time for other queries.
1.2 Proposed Solution

The last section motivated this thesis. In this section, we clarify the problems we need to solve and summarize the proposed solution. To begin with, we have two major problems to solve.

The first problem is to enable business users to pose queries on this semantic layer and return query results to users. Instead of asking users to write SQL queries on the conceptual schema directly, we think users should only be exposed to a simple query interface asking only minimized details to form a query. Therefore, we propose a CIM query language with a relevant query interface, which is tailored to the conceptual schema that users need to interact with. This query language and query interface will focus on how to represent needs from the users’ perspective. In the back end, we also need an algorithm to translate a CIM query to a SQL query so that the central database can execute it.

The second problem is how to optimize this query processor. We decide to select and materialize some views to speed up query answering, which is a popular approach to query optimization in OLAP. We try to make use of extra local disk space to trade space for efficiency. Instead of proposing some heuristics, we model the problem to conform to some property and propose an approximation algorithm to select a set of views, which can have a minimum performance bound. Compared to heuristics, we think the minimum performance guarantee is more general to different cases.

1.3 Contributions

We summarize the contributions of this thesis as following.

- We propose a query language, as well as a query interface, which is simple and easy for business users to use.

- We successfully translate queries from the CIM query language to SQL that can be executed in the central database.

- We model the view selection problem to satisfy the submodularity property and propose an algorithm to select views in the CIM framework.
We prove that the proposed algorithm can achieve a minimum performance bound of 46.7% of the optimal solution of view selection.

1.4 Organization of the Thesis

We first provide some background knowledge about data warehouses and the CIM framework in Chapter 2. In Chapter 3, we address some related work about query languages and view selection in OLAP. A new query language and the translation algorithm from this query language to SQL are discussed in Chapter 4. Chapter 5 optimizes query answering in CIM by selecting some views to materialize with extra space. Finally, we summarize the work of this thesis and point out some future work in Chapter 6.
Chapter 2

Background

Our work in this thesis is based on previous work on the CIM framework [23]. CIM is a new top-down data warehouse design framework which adds a conceptual schema to express users’ needs. To facilitate readers who are not familiar with a data warehouse and CIM to understand our work, we first give a brief introduction to data warehouses. The goal of this step is to show the classic design of a data warehouse to provide a baseline for comparison with CIM and our work. In the second step, CIM’s architecture and major components will be introduced. We will not go too deep into details in this step. Instead, we let users understand the framework at a high level, which is sufficient to motivate our work.

2.1 Data Warehouse

A data warehouse is a centralized database which is used for reporting and analysis [5]. It integrates data from multiple day-to-day operational data sources, often Online Transaction Processing (OLTP) databases. In contrast to OLTP databases, data warehouses are designed to work with Online Analytical Processing (OLAP) technology to provide decision support.

2.1.1 Functionality

From functionality’s perspective, OLTP databases aim to solve structured, repetitive, transactional and light-weight tasks. Operational time for each task is usually
short and transactions are usually simple and fixed. Such databases exist everywhere to facilitate our daily activities. An automatic teller machine (ATM) for banking, which provides simple operations like deposit and withdrawal is such an example. On the other hand, data warehouses help organize data to serve OLAP applications. OLAP applications include generating reports, providing intelligent suggestions for decision support and data mining, all of which integrate different data. An example of such an application is generating monthly sales report for different cities.

All of these applications are based on queries on data warehouses and thus require data warehouses to store historical and summarized data for analysis. These queries have some unique properties compared to OLTP queries, which include:

- Queries are usually over a long time span. It is common for users to require several years’ data to understand trends well enough to make a decision. In contrast, OLTP queries emphasize immediate transactions.

- Aggregated values are the desired information of OLAP queries. OLAP queries tend to request the aggregated values, which are also called measures, like sums and averages. For example, a sales manager may like to know the total revenue of sales over the last 10 years. These aggregated values can be visualized for further analysis.

- OLAP queries often focus on some facets of aggregated values. We call each facet a dimension. A dimension can be location, time or anything that is related to the aggregated values mentioned above. For example, a query can ask for total revenue by each city in the location dimension. It is common that a specific OLAP query only focuses on some interested dimensions while ignoring others.

2.1.2 Schema Design

A data warehouse is organized as a multidimensional model, which has two main components: fact and dimension. Aggregated data are measures in the fact table. Data in dimensions provide the context for the measures. The reason we call a data warehouse multidimensional is because we usually group dimensions into different
categories to provide enough alternatives for users to navigate the data warehouse. For example, location and time can be two dimensions for the fact table about sales containing the measures: revenue and employee number. In addition, there can be hierarchies in a dimension. A hierarchy describes different granularities in a dimension. An example of such a hierarchy can be day, month and year in a time dimension.

A data warehouse can be implemented on standard or extended relational DBMSs, called Relational OLAP (ROLAP) servers [9].

There are two main types of schema design approaches in ROLAP. The first one is the star schema. A star schema maintains a single fact table and a denormalized dimension table for each dimension. A fact table contains measures and associated foreign keys from each dimension. A dimension table consists of a primary key and relevant attributes for this table. A hierarchy within a dimension is implicit in the star schema because of denormalization. Figure 2.1 shows an example of the star schema.

The second type is the snowflake schema. The snowflake schema is similar to the star schema. The only difference is that it maintains the hierarchical information in a dimension by normalizing it to several tables using foreign key constraints. We call each hierarchical table in the same dimension table a level. Each level represents a granularity of the dimension. Figure 2.2 shows an example of the snowflake schema.

Besides ROLAP, there is another type of multidimensional server, which is called Multidimensional OLAP (MOLAP). MOLAP stores data in cubes, each of which is stored in a special data structure like a multidimensional array or matrix. All subcubes of a data warehouse are pre-calculated. We can always find the aggregated values of any granularity from MOLAP. The advantage of MOLAP is fast query performance because all subcubes are pre-calculated so there is no need to calculate OLAP queries on the fly. But the disadvantage is also clear. It cannot scale well with increased dimensions because the space required to materialize all cubes is exponential in the number of dimensions, even though there are some compression techniques for storing MOLAP.

Since ROLAP is based on relational databases, it can apply the same types of techniques to optimize query processing in ROLAP. There are many well-studied
Figure 2.1: Star Schema

Figure 2.2: Snowflake Schema
techniques in ROLAP that can optimize query performance, such as index selection and view selection.

2.1.3 Design Methodology

A data warehouse is a complicated system. A good design can reduce future modification to the data warehouse schema and can also be easy for end users to understand. In general, a data warehouse can be designed in both bottom-up and top-down approaches. The bottom-up approach, which is known to be supported by Kimball et al. [19], is a popular way to design a data warehouse. It starts by identifying data sources and focuses more on loading data to the data warehouse. This approach is generally done by database experts, thus the schema may not conform 100% to end users’ needs. The advantage of this approach is that the schema of the data warehouse tends to be complete and space efficient because it is designed by database experts. They are good at storing data in an efficient structure. However, there is a mismatch between this schema and the end users. End users usually have specific needs, and it would be better for them to use and pose queries if they can tweak the schema to be more specific and convenient.

In contrast, the top-down approach starts designing a data warehouse from the users’ requirements. End users have the power to decide the schema of the data warehouse instead of data experts. This approach is more user-oriented and starts by designing the schema from the users’ perspective. The main difficulty of this approach is how to create an environment to let business users easily express their needs while hiding unnecessary implementation details from end users.

It is worth mentioning that no matter what design approach we choose, a simple standard star or snowflake schema is hardly the case in a data warehouse. It is quite common that there exists more than one fact table in the data warehouse schema. For example, two departments may focus on different set of measures, which have different dimensions w.r.t. different measures. In addition, schema evolution of the data warehouse can also add complexity to the original simple schema.

Therefore, the schema of a data warehouse is often too complicated for end users. To overcome the mismatch between end users and data experts, the CIM framework is proposed as a top-down design approach based on an existing com-
plicated data warehouse. At the same time, it hides some implementation difficulty in the backend, which is easy enough for business users to use.

2.2 Conceptual Integration Modeling Framework

The Conceptual Integration Modeling (CIM) [23] project is a joint research project between the University of Ottawa and the University of British Columbia. As we have discussed in the previous section, designing a data warehouse is a complicated task. The CIM project aims to solve the design phase problem of a data warehouse. Our work is another component of CIM, which facilitates users querying on the conceptual schema proposed in CIM. In this section, we address the motivation of CIM, provide an overview of the solution and discuss some progress so far.

2.2.1 Motivation

As we have shown, the schema of a data warehouse is quite unlikely to be in a simple star schema or snowflake schema because of the complexity of different data sources and the needs from users.

The first reason to propose CIM is to let business users easily express their needs over the current data warehouse (also referred as the central database) schema. Note that since a data warehouse serves different types of users, it is common for there to be a mismatch between user needs and the data warehouse schema. A business user can be confused and distracted when exposed to the complicated data warehouse schema. The CIM framework adopts a top-down approach to provide a visual environment for users to design conceptual schemas just for their needs, and later maps conceptual schemas to the data warehouse schema. By adding a layer of abstraction, the conceptual schema can hide the complexity of the original data warehouse schema.

Querying ability is the second reason that we need the conceptual layer proposed in CIM. If a user wants to formulate a query using this data warehouse schema, she may need to write a very complicated query. For example, if there are three tables in the data warehouse schema for sales in Canada, Mexico and USA, to query sales in North America, the user needs to combine these three individual tables manually to form the query. Even when the needs from users are
straightforward and semantically clear, expressing such a query on a complicated data warehouse is not easy, which hinders users from posing queries. Instead, if a conceptual schema contains a table of North American data, it becomes more convenient to pose queries directly on this conceptual schema. Such a problem exists when users queries cannot find a direct simple way to form a query in the data warehouse schema.

These two problems are the main motivations for the CIM framework. The framework aims to make data warehouses more usable for business users without requiring expert knowledge.

2.2.2 Goal

The work on the CIM framework has two phases, which we refer to CIM1.0 and CIM2.0. CIM1.0 assumes the data warehouse schema already exists and is in use. We are not changing the schema itself in order not to break existing applications. Therefore, in phase one, Rizzolo et al. [23] proposed a conceptual schema to represent users’ needs. Then this conceptual schema is mapped to the schema of the data warehouse. As a result, users only need to focus on this schema and pose queries on it. Queries can later be automatically translated to SQL queries on the central database.

CIM2.0 emphasizes data warehouse schema creation while preserving the conceptual layer in CIM1.0. It starts from business users creating conceptual schemas. Then, instead of mapping a conceptual schema to the data warehouse schema, CIM2.0 maps a conceptual schema to data sources directly. Based on the mappings created, the data warehouse schema is then created automatically. CIM2.0 is a pure top-down approach, which can remove the step where experts must create the data warehouse schema.

Currently the project is still in CIM1.0. Various papers [23], [26] and [21] describe some of the work done relating to phase one. The following architecture section is about CIM1.0 and this thesis is also based on it.
2.2.3 Architecture

There are four main components in the CIM framework. Before we start describing them one by one, we show a diagram of the relationships among them for future reference in Figure 2.3. The CIM framework has both visual and literal representations for all components. Users are exposed to the visual environment to design the conceptual schema and pose queries on it. Detailed explanations of each component are described as follows:

- **Storage Manager**

  The storage manager is in charge of representing the schema of the central database, both visually and literally. We refer to these two ways as the Storage Visual Language (SVL) and the Storage Data Language (SDL). SVL and SDL are semantically equal but they allow different representations for the user (SVL) and the backend (SDL). The storage manager displays the schema of the data warehouse designed by database experts.

- **Model Manager**

  The model manager is in charge of representing the conceptual schema as
created by users. It also has two representations, named the Conceptual Visual Language (CVL) and the Conceptual Data Language (CDL). The conceptual schema consists of three parts, which are fact tables, levels and parent-child relationships. As the names of the components show, the conceptual schema is quite like a typical snowflake schema. The only difference is that the hierarchy in the conceptual schema is more diverse, which can be either homogeneous or heterogeneous. More details of the hierarchy can be found in [15]. The fact component stores the lowest level dimension ID and aggregated value. A level, which contains a primary key (aka. an ID) and related attributes, represents one level of granularity of a dimension. A parent-child relationship is a two-column table which maps two IDs in a hierarchical relationship.

- **View Compiler**

  The view compiler takes correspondences created by users between the CVL and the SVL and generates a mapping between them in Mapping Definition Language (MDL). The correspondences are just simple lines between the conceptual schema and the storage schema, which sometimes contain restrictions like “greater than”. This mapping maps fact, level and parent-child relationships to SDL by generating views of them over the storage schema. In CIM, the central database is assumed to be relational, therefore this MDL is in SQL, which can translate components in CDL to relevant components in SDL.

- **Query Processor**

  The other use of the conceptual schema is to allow end users to pose queries on it. Then, with the help of the views on the conceptual schema generated by the view compiler, the query processor generates SQL queries over SDL in the backend and executes them. The query processor can simplify the process of posing queries by providing the query environment in the conceptual layer, which allows end users to query in the representation that they have created, and thus best understand. The first part of this thesis is to implement this query processor.
Example 1. Figure 2.4 is an example of components in CIM. The left hand side is the CVL while the SVL is on the right hand side. In the middle there are some user indicated correspondences, which will be compiled into mappings for components’ levels, fact and parent-child relationships in the CVL. The CVL is created by end users and usually directly conforms to their needs whereas the SVL may not. For example, end users may only be interested in total sales including both the east and west sides. Thus, they create a fact table called Sales in CVL. Then they pose queries on Sales, but do not need to pose queries on both SSalesEast and SSalesWest.

On the other hand, end users may need more hierarchy information in the
CanadianLocation dimension. We notice that in CVL, end users have designed level Branch to roll up to both City and Area. In the SVL, they are in the same table SBranch, whose hierarchy information is implicit to users.

In this example, we would like to show some mappings in MDL, which are in SQL. First, we show how a level is mapped to SDL. We take Region in Figure 2.4 as an example and show it below.

```
CREATE VIEW Region AS
SELECT SBranch.regionID AS regionID, SRegion.name AS description
FROM SBranch, SRegion
WHERE SBranch.regionID = SRegion.regionID
```

Next, we show how a parent-child relationship is mapped. A parent-child relationship Province-City is represented as follows:

```
CREATE VIEW Province-City AS
SELECT SBranch.provID AS provinceID, SBranch.cityID AS cityID
FROM SBranch
```

Finally, we show how a fact table is mapped to SDL. The fact table Sales is represented as follows:

```
CREATE VIEW Sales AS
SELECT SSalesWest.totalRevenue, SSalesWest.branchID, SSalesWest.clientID
FROM SSalesWest
WHERE SSalesWest.commission > $20K
UNION
SELECT SSalesEast.totalRevenue, SSalesEast.branchID, SSalesEast.clientID
FROM SSalesEast
WHERE SSalesEast.commission > $10K
```

From the above example, we see that the mapping is a query rewriting from the conceptual schema to the storage schema in SQL.

### 2.2.4 Heterogeneity of Hierarchy

As we know, a conceptual schema can contain hierarchies in each dimension. To adopt different hierarchy requirements from different users, CIM allows heterogeneity in hierarchies. Homogeneity in this sense requires all parent-child relationships to be total. In other words, every element in the child level has to participate in all related parent-child relationships. In a homogeneous hierarchy, a parent
level can follow any single parent-child relationship in a chain to the bottom level connecting the fact table. To better evaluate queries in a homogeneous schema, we just need to pick the most cost-efficient parent-child relationship in a chain. In contrast, since heterogeneous hierarchies can allow different sets of child elements to be involved in different hierarchies, it is not guaranteed that one can find the total answer by following any single hierarchy.

On the other hand, the hierarchies in CIM are assumed to be strict. This means any element in the child level can only roll up to at most one element in any ancestor level. Strictness can make sure that double counting will not happen. For example, if a city $x$ can roll up to two provinces, then summarizing total revenue at the country level, by summing from the province level will double count the total revenue of city $x$.

Shakya [26] analyzed this type of heterogeneous hierarchy and further proposed a disjoin/split operator in CVL. By introducing the disjoin/split operator, any two paths from the same source and target level have no intersection in their instances. This operator can bring some convenience in query optimization mentioned later. In our query processor, we assume this operator is not provided in CVL, thus we need to evaluate queries with heterogeneous hierarchies. Details of the evaluation can be found in Chapter 4.
Chapter 3

Related Work

SQL is often criticized as being ill-suited as a multi-dimensional OLAP query language. Gray et al. [10] claimed that operator GROUP BY cannot fully exploit the semantics in OLAP and proposed an operator called DATA CUBE, which can generalize GROUP BY and provide operations like drill-down and roll-up. Agrawal et al. [6] also proposed a data model for multi-dimensional databases. Their work contained powerful algebraic operators, which can be composed to build OLAP operators like roll-up.

As suggested by the two works above, there is one similarity among most work about the query language in OLAP, which is the demand for more powerful operators to tackle query expression and result formation. In addition, the implicit proposition behind them is that their new OLAP query languages aim to serve OLAP experts.

However, our CIM framework has a different user target, which is business users. Instead of making the query language more powerful, we stress the simplicity in query expression and a highly usable query interface.

We also discuss query optimization in this thesis. View materialization is a common and popular technique in OLAP to speed up query answering.

Harinarayan et al. [14] first proposed a lattice framework and proved that a simple greedy algorithm can achieve a minimum performance bound of 63% of the optimal solution by selecting the same number of views to materialize.

This work was later extended for view selection under a space constraint [12]. In our work we consider both the problems of heterogeneity and the selection of views which satisfies a space constraint.

Gupta et al. [13] also covered the problem of selecting indices for the views. For instance, assume you have a set of candidate views, each with a corresponding set of candidate indices to materialize. The algorithm will only consider materializing indices when the corresponding view will be materialized. In turn, the algorithm must consider the benefit of the indices when considering whether to materialize the view. They have shown that the view selection problem with index selection has an algorithm that achieved a good minimum performance bound of the optimal solution. Our work is modeled similarly to determine which parent-child relationships and attributes to materialize.

Baralis et al. [7] used a similar lattice structure technique with an additional goal of improving performance of the selection process. Since performance is a focus, the search space of candidate solutions is pruned heuristically.

Other work has been done to accommodate view selection in a data warehouse environment. Many methods evaluate which views to materialize in the query plan to benefit multiple queries. Ross and Srivastava [24] and Yang [29] considered common sub-expressions between query plans. These expressions are considered candidates to materialize in addition to the query results. Theodoratos and Sellis [28] presented view selection as a state space search problem which is pruned using heuristics to improve an exhaustive search algorithm.

The literature above focuses on a static set of materialized views. Dynamat [20] and Watchman [17] are two traditional dynamic view selection methods. Both methods used a cache replacement approach where the results of executed queries are cached and replaced based on criteria similar to the linear cost model mentioned above. But our thesis is based on the static state of the CIM framework.
Chapter 4

Query Processor Framework

The view compiler in CIM builds a bridge between a conceptual schema and a storage schema. The bridge itself is a mapping, which is represented as views of components in the conceptual schema over the storage schema in SQL. The goal in generating such a mapping is to allow translation from queries posed on the conceptual schema to SQL queries executable on the storage schema. Before we state the translation process, we first review the mappings we have generated by the view compiler. There are three types of views generated on the conceptual schema:

- Level, which contains the ID and attributes
- Parent-child relationships between levels
- Fact relations

These three types of views cover all components in the conceptual schema. In this chapter, we focus on query processing for the CIM project. The query processor in CIM includes two main parts. The first part is designing a query language for users to easily pose queries on the conceptual schema. This work emphasizes the usability of this query language to users, compared to the original SQL queries. The goal is to enable business users who do not have database background to pose queries. The second part is the backend of the query processor. The conceptual schema only acts as a layer on top of the storage schema, and queries are finally executed in the central database, which is a relational database. The main goal of
this part is to translate queries from a user-defined query language to SQL queries on the conceptual schema. Then the translation of queries from the conceptual schema to the storage schema can be done automatically with the help of views on the conceptual schema.

In the following two sections, we discuss these two tasks.

4.1 CIM Query Language

The query processor’s first task is to design a query language for users to pose queries. Since all levels, parent-child relationships and the fact table are represented as views in SQL, users can just directly write SQL queries over the conceptual schema. However, the conceptual schema in the CIM project aims to help business users pose queries with ease. Writing an SQL query may turn out to be very complicated since it must join all different tables. Also, business users may not already know SQL and SQL has a relatively tough learning curve.

Basically there is a mismatch between query execution and query expression. An RDBMS can understand queries expressed in SQL while SQL is too complicated for users to express their needs. Jagadish et al. [16] discussed the usability issue in current database management systems, pointing out five database pain points that hinder users from using them. The mismatch mentioned above is one of the usability issues in the paper, which the authors called “the painful relation problem”. This problem is caused by the need to explicitly join relations in SQL. One solution to this painful relation problem is to hide unnecessary operators from users. As we will explain in the query formation section later, some of the SQL operators such as JOIN and GROUP BY can be hidden from users and only be executed in the backend. The query language should be concise and straightforward to the business users.

It is worth mentioning that they argued that the logical schema is not user-friendly enough to end users and proposed the use of a presentation layer, which conforms to the perspective of end users and is on top of the logical schema. Our conceptual schema acts as the presentation layer to reflect the needs of business users, as opposed to the storage layer, which is more like a giant pot for different needs over all different kinds of users. Therefore, a new query language is needed
to work with this conceptual schema more naturally.

### 4.1.1 Query Language Design Principles

Before we describe our query language for the conceptual schema in CIM, we first look at some existing query languages for OLAP. Multidimensional Expressions (MDX) [4] is a popular language for query processing in OLAP. MDX is like SQL but adds the functionality to indicate the layout of query results, which can organize data in a tabular form as in Microsoft Excel. MDX is powerful in general, but it is too complicated to write, which makes it a poor choice for the CIM project, which focuses on business users as customers.

First of all, MDX is too flexible to indicate the projections. It can project a whole hierarchy in a dimension, as well as add restrictions to the projection. Sometimes this flexibility can confuse users about what is actually needed to be projected.. We think further decomposition of projection and restriction can be clearer to users to express queries. Too much flexibility can confuse non-expert users and requires a deeper understanding of the query language.

The second feature that adds complexity to the query language is the need to configure the layout of query results. This feature helps users decide how to organize their query results in a tabular format. But at the same time we think too much consideration of layout can hinder users from posing a query. On the other hand, the layout of output can be tuned later instead of at the very beginning, especially when the layout can be tuned with the help of a GUI.

Another reason that MDX does not fit CIM as a query language is the complexity needed to translate it to SQL and to create a visual query language above it. Components in the conceptual schema are expressed as SQL views over the storage schema. In contrast, MDX is a natural fit in MOLAP. Though there is an existing tool called Mondrian [2] to translate MDX queries to SQL queries, it is still complicated to parse the query and do the transition. To summarize, it is not worth the processing time. Therefore, MDX does not fit our requirements.

Before we design our own conceptual query language, we list three principles in designing such a query language:

- **Simplicity in expressing queries.** No one can use SQL or MDX without
training. But we do not want CIM users to spend that much time before being able to pose queries. The query language should be simple enough for them to use without much effort. The goal is for users to be able to express a query with little or no learning curve. We can even sacrifice some flexibility of the query language itself for the simplicity.

- **Not mixing the layout of output and the query itself.** A query should only be responsible for expressing users’ needs. On the other hand, we regard query layout, either literally or visually as another task. In short, the tasks should be decoupled. The reason is that tabular style layout is only one way to show results. Graphs and other visualizations of results are equally important. Integrating the tabular layout to a query like MDX does not solve the problem of all different query visualization formats. A better way is to separate the tasks of the layout and the query needs.

- **Simplicity to visualize the query.** CIM has both visual and literal versions of representation. Users are exposed to the visual environment to design conceptual schemas and draw correspondences. We continue the work on the visual representation and hold that querying visually can help business users better express their ideas. Therefore, we think our query language should be easy to map to the visual representation.

### 4.1.2 Query Language Definition

Before we define our CIM query language schema, we start by analyzing some example query needs from Figure 2.4.

**Example 2.** In this simple sales example, there are two dimensions called *CanadianLocation* and *Client*. Let’s assume a user is interested in knowing the sales information of each city from a big client, John. More specifically, the user would like to know the city names and total revenues per city, purchased by John.

Example 2 describes a query involving two dimensions. If we analyze this query need to extract some critical information, our query language first needs to know what *measure(s)* (e.g., total revenue) this user is interested in. All queries on the conceptual schema include some measures.
The second facet of this query is to decide which measure(s) we are interested in. A measure can be sliced and diced by the dimensions around it. A dimension also has hierarchies on it so that we can further drill down to the appropriate level for its attributes in the dimension. In the above example, name in City is the wanted attribute and level.

The last facet of this query is the restriction applied to it. The restriction is an expression such as equals, greater or less than which confines the scope of either a measure or a level. In the above example, the restriction is the client named John.

The combination of the above three components can fully express any query posed on the conceptual schema. The work to form a formal SQL query can be done in the backend to reduce the burden on users posing queries. Users only need to pick relevant components in the conceptual schema for these three categories to express their query needs.

We use XML as the format to represent queries. The reason is that there are already many existing implementations to read and write XML. On the other hand, as we mentioned above, we will have a visual interface for users to pose queries on. Our query language does not directly interact with end users. The format of this query language just needs to be have enough support and be convenient.

We first describe the definition of the CIM query language and then give the corresponding XML query from Example 2. We use a <Query> element to indicate the beginning of a query. A <Query> element contains three child elements, which are <Fact>, <Projection> and <Restriction> separately.

<Fact> elements contain a name attribute and <Measure> child elements. Each <Measure> contains a name attribute to identify the measure. A <Projection> shows the interested facets of the measures. It has child elements called <Dimension> to indicate which dimension(s) users would like to have with the measures. A <Dimension> has the child element <Level> and a <Level> has the the child element <Attribute>. This <Attribute> element indicates what information to project with the measures. All these XML elements have an attribute called name to identify themselves in the conceptual schema. The last element <Restriction> can have either a <Fact> or a <Dimension> as a child element. Moreover, it adds another child element within <Measure>s and <Attribute>s, which is called <Constraint> with attributes operator and value. Users just need to fill in these
two parts to make a constraint.

We show the formal grammar of the CIM query language as follows:

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>Fact Projection+ Restriction*</td>
</tr>
<tr>
<td>Fact</td>
<td>name Measure+</td>
</tr>
<tr>
<td>Measure</td>
<td>name</td>
</tr>
<tr>
<td>Projection</td>
<td>Dimension+</td>
</tr>
<tr>
<td>Dimension</td>
<td>name Level+</td>
</tr>
<tr>
<td>Level</td>
<td>name Attribute+</td>
</tr>
<tr>
<td>Attribute</td>
<td>name</td>
</tr>
<tr>
<td>Restriction</td>
<td>(FactRes</td>
</tr>
<tr>
<td>FactRes</td>
<td>name (Measure Constraint)+</td>
</tr>
<tr>
<td>DimensionRes</td>
<td>name LevelRes+</td>
</tr>
<tr>
<td>LevelRes</td>
<td>name (Attribute Constraint)+</td>
</tr>
<tr>
<td>Constraint</td>
<td>operator value</td>
</tr>
</tbody>
</table>

We show our CIM query for Example 2 in the following box. As we can see, the components are clear and it is easy to read and write this XML query with the APIs provided such as `getElementById`. More importantly, this query language only asks the needed information from users and it also provides three clear categories to form a query. In the next section, we can see that a query interface can provide more convenience to form this query.

```xml
<Query>
  <Fact name="Sales">
    <Measure name="totalRevenue" />
  </Fact>

  <Projection>
    <Dimension name="CanadianLocation">
      <Level name="City">
        <Attribute name="name" />
      </Level>
    </Dimension>
  </Projection>

  <Restriction>
    <Dimension name="Client">
      <Level name="Client">
        <Attribute name="name">
          <Constraint operator="equal" value="John" />
        </Attribute>
      </Level>
    </Dimension>
  </Restriction>
</Query>
```
In contrast, if we do not define the CIM query language for users, users have to write the SQL query as below. As we can see, this SQL query contains information joining different components in the conceptual schema. Though it is short, it can be hard for business users to write and it is easy to miss some operators like `GROUP BY`.

```sql
SELECT City.name, Sales.totalRevenue
FROM City, City/Branch, Sales, Client
WHERE City.cityID = City/Branch.cityID
AND City/Branch.branchID = Sales.branchID
AND Client.clientID = Sales.clientID
AND Client.name = "John"
GROUP BY City.name
```

Of course, the CIM query language can be represented in other light-weight formats like JSON [1]. We chose XML because it can show more semantic information in the query. Also, we do not expect users to be able to write such an XML query. Instead, they are exposed to the visual query interface, which is shown in the next section. The CIM query language is only used internally to directly represent CIM queries.

The query language we defined above only requires minimal information from users and is simple enough to express users’ needs. In the next section, we will discuss some guidelines for designing a query interface to interact with users.

### 4.1.3 Visual Query Interface Description

So far the query language we described defines the literal representation of queries. Although it is simple and expressive, it can be easy for users to make mistakes. Hand-writing an XML query is not a good idea. We think a visual query interface is a more intuitive way for business users to pose queries. In addition, providing a query interface can make use of the visual representation of the CIM framework.

Notice that in the CIM query language, the deepest element in the XML representation is the attribute of a level or a measure. Of course, a `<Constraint>`
element may be attached to this element. The point is the dimension, level or fact information of this attribute or measure can be traced and be found in the visualized conceptual layer. For example, if a user picks the name of level City in Figure 2.4, we can trace the level name City and the dimension name CanadianLocation from the structure of the visualized conceptual schema. Therefore, users can just focus on the interested measure or attribute, which is an oval in the visualized conceptual schema and assign a category, which is either a measure, projection attribute or restriction to it.

We can achieve this functionality by providing options when the user right-clicks the oval. If it is a constraint, we can further provide operators and values for the selection. Another alternative is to first allocate three boxes of categories visually and then drag and drop ovals from the visualized conceptual schema to these boxes.

Figure 4.1 shows a simple visual query interface along with the visual conceptual layer. In this figure, we can see three boxes representing three categories. To form a query in Example 2, users just need to drag and drop relevant elements from the conceptual schema to the boxes. For example, in this query, users first drag totalRevenue in the oval and drop it into the box named Measures. The same operation applies to the box Projections. But in the case of the Restrictions box, besides dragging and dropping the attribute name from Client, we need to write down the operator and value, which are equal and John by hand. Again, options with each operator can be provided.

From this example, we can see that the query interface is fully integrated with the CVL in the conceptual layer. Users can query visually on the schema that they designed. Once the visual query is formed, it is represented in the CIM query language format on the client side. Then the CIM query, which is in XML, is sent to the server and the server translates it to an SQL query. Together with the views generated as the mapping, the SQL query is executed in the relational central database. The process to translate an XML query to an SQL query is discussed in the next section. The whole query process is shown in Figure 4.2.
Step 1: Drag totalRevenue in Fact Sales and drop it to the box Measures.
Step 2: Drag name in Level City and drop it to box Projections.
Step 3: Drag name in Level Client, drop it to box Restrictions, select the operator = and put the value John.

**Figure 4.1: Visual Query Interface**
4.2 SQL QueryCompilation

In the CIM framework, the view compiler generates three types of views in the conceptual layer as the mapping to the storage layer, indicated in Section 2.2.3. The views are represented in SQL and they are provided as the entry point to the central database. The previous section introduced the query language designed from the users’ perspective and the corresponding query interface. However, the central database is relational and can only understand SQL as a query language. This section aims to bridge the gap between the SQL query language and the CIM query language. The job is to automatically translate CIM queries to SQL queries.
on the conceptual schema, so that the DBMS can understand and execute them.

We start this section by analyzing each component needed in SQL during translation. First of all, note that CIM queries only focus on non-update queries, which only reads data from a database. This observation rules out the use of insertion, update and deletion transactions.

In the following paragraphs, we describe each component of SQL, which is needed to do the translation. These components include SELECT, WHERE, FROM, GROUP BY and UNION. In addition, we give examples for each component.

• **SELECT**

  We first see how to compose the SELECT component in SQL from the CIM query language. SELECT in SQL decides what to project to show users. A measure in CIM query is the first component to consider. We can simply use the format of FACT.MEASURE for this SELECT component by tracing the name of the fact table after the oval of measure is selected in the CVL. As for the aggregate function for the measure, there should be an aggregate function associated with each measure in the conceptual schema. For example, totalRevenue from Figure 4.1 is associated with aggregate function \(\text{sum}\) instead of \(\text{avg}\) since the name of it indicated such an aggregation function. This mapping from the measure to the aggregate function can reside on the server side. Users do not need to explicitly express aggregate functions for each measure.

  In addition, we need to project the wanted information in the level indicated in the first category of the CIM query. This information can be the primary key or an attribute of the indicated level. Since parent-child relationships are also compiled as views in CIM, if the primary key in a level (e.g., level ID) is chosen for projection, we just need to project the primary key from the parent role of the parent-child relationship. For example, if we are interested in cityID, we can project cityID from the parent-child relationship City-Branch. In contrast, if we are interested in an attribute in a level, we have to project the attribute from the level.

  **Example 3.** If we are interested in the total revenue of city ID and city name in Figure 4.1, the projection is written as follows:
In this example, since the attribute name of level City is selected, it does not matter if the primary key cityID is selected from level City or parent-child relationship City-Branch.

• **Join (FROM & WHERE)**

The next component to form an SQL query is about what tables to join. A conceptual schema is like a snowflake schema, which means every level in the schema can find a path to the fact table. The difference is that the path is connected through parent-child relationships from the fact table to each level. To get the measures of a particular level, we have to chain parent-child relationships from that level to the fact table because measures only exist in the fact table. If we only need to project primary keys from parent-child relationships, there is no need to link levels to the chain. Otherwise, we need to link levels to the relevant parent-child relationship, too.

**Example 4.** Let's assume the query is to get the total revenue of each city ID and client name in Figure 4.1. Then the join, combined with SELECT, is represented as follows:

```sql
SELECT City.Branch.cityID, Client.name, sum(Sales.totalRevenue)
FROM City.Branch, Sales, Client
WHERE Sales.branchID = City.Branch.branchID
AND Sales.clientID = Client.clientID
```

From the above example, we can see only the parent-child relationship City/Branch is needed for the CanadianLocation dimension because the query only asks for the primary key cityID. However, the level Client is selected because the query needs to know the attribute name of that level.

• **Restriction (WHERE)**

In our CIM query language, we have a category named Restriction. The functionality of Restriction is to add a scope to filter unwanted information. More specifically, it achieves such a value restriction through the equality
or inequality comparison function. Converted to SQL, such a restriction is represented with the clause \texttt{WHERE}. Restrictions can happen both in the fact table and dimensions. If it is a restriction on measures, it can be reflected in the fact table in SQL. Otherwise, similar to the \texttt{SELECT} case mentioned above, a \texttt{WHERE} clause can be added to either a parent-child relationship or a level, depending on whether it is a primary key in a level.

\textbf{Example 5.} Let’s assume the query is to get the total revenue of city Vancouver and client name in Figure 4.1. Then the \texttt{SELECT}, \texttt{FROM} and \texttt{WHERE} of the SQL query are shown as follows:

\begin{verbatim}
SELECT City.name, Client.name, sum(Sales.totalRevenue)
FROM City, City-Branch, Sales, Client
WHERE Sales.branchID = City-Branch.branchID
    AND City.cityID = City-Branch.cityID
    AND Sales.clientID = Client.clientID
    AND City.name = "Vancouver"
\end{verbatim}

In this example, we added the restriction on level \textit{City} to only filter the sales condition in Vancouver.

- \textbf{GROUP BY}

In the \texttt{SELECT} clause, we selected some primary keys or attributes in levels to show aggregated measures in that specific granularity. In SQL, the \texttt{GROUP BY} statement is used in conjunction with the aggregate functions to group the result-set by one or more columns. Therefore, we need to add the \texttt{GROUP BY} clause corresponding to each component in the projection in the SQL query. In addition, when we choose to project an attribute or the primary key in a level, we will \texttt{GROUP BY} its level primary key anyway. The reason is that two different primary keys may have the same value of an attribute. Grouping by the attribute can mess up the query results. Note that in our thesis to form the \texttt{GROUP BY} clause in SQL, we adopt the Transact-SQL standard [3], which is an SQL extension standard applied in the Microsoft SQL Server. This extension allows projections in the \texttt{SELECT} clause to include columns not existing in the \texttt{GROUP BY} clause. Besides Transact-SQL, extensions on the MySQL Server also support this feature.
Example 6. In this example, we will show the complete SQL query proposed in Example 5 by adding the `GROUP BY` clause.

```sql
SELECT City.name, Client.name, sum(Sales.totalRevenue)
FROM City, City-Branch, Sales, Client
WHERE Sales.branchID = City-Branch.branchID
AND City.cityID = City-Branch.cityID
AND Sales.clientID = Client.clientID
AND City.name = "Vancouver"
GROUP BY City.cityID, Client.clientID
```

- UNION

In our CIM conceptual schema, we proposed to use a heterogeneous hierarchy in Section 2.2.4. For example, `Country` can have two drill-down paths to `Province` and `Region`. The biggest problem of heterogeneity is that following a single parent-child path does not guarantee the complete answer because some child level may not participate in a specific parent-child relationship. Instead, we need to collect all results following all possible parent-child relationships and eliminate the repeated ones. To achieve this feature in SQL, we can use the keyword UNION. The keyword UNION can automatically delete those repeated tuples when calculating results from all paths. The drawback of UNION is the cost can be high because we need to calculate results from all paths. In the next chapter, we will introduce some view selection techniques to alleviate this cost. The following example shows a query with the use of UNION.

Example 7. From Figure 4.1, the query is to find the total revenue by each country.

```sql
SELECT Country-Province.name, sum(Sales.totalRevenue)
FROM Country-Province, Province-City, City-Branch, Sales
WHERE Country-Province.name = Province-City.provinceID
AND Province-City.cityID = City-Branch.cityID
AND City-Branch.branchID = Sales.branchID
GROUP BY Country-Province.name
UNION
SELECT Country-Province.name, sum(Sales.totalRevenue)
FROM Country-Region, Region-Area, Area-Branch, Sales
WHERE Country-Region.name = Region-Area.regionID
```
AND Region−Area.name = Area−Branch.name
AND Area−Branch.branchID = Sales.branchID
GROUP BY Country−Province.name

In the query, we can see that name in the country level is also the primary key for that level. So we can get the country name by reading the parent-child relationship. Also notice that country can be rolled up by both Province and Region, therefore, we need to UNION results from these two paths.

So far we can already translate any CIM queries to SQL queries, component by component. At the same time, if we compare the visual query interface provided to users and the SQL queries generated, we can see that the the join component, the GROUP BY component and the UNION component can be implicitly done in the backend. Users do not need to specify how to form these components in SQL.

When SQL queries over the conceptual schema are generated, they can be sent to the relational central database. Also notice that the views of the conceptual schema are also stored there. Query expansion of views and query execution will happen at this step.

On the other hand, translation from CIM queries to SQL queries can be more efficient if we have some extra space to materialize some views. In the next chapter, we will discuss view selection techniques in CIM.
Chapter 5

Query Optimization

In the last chapter, we described an algorithm to process a query posed on the CVL. The algorithm translates a visual query into an SQL query based on the components of the CVL, which are the levels, parent-child relationships and the fact table. As we know, these components are represented as views over the central database and the views are also stored in the central database. A consequence of this is that each query is processed in the central database.

The central database will be very large and a hot-spot for query answering. It may take a long time to answer queries with it. To speed up query answering, we can materialize some views which can provide a tighter scope to some queries to speed up query answering. For example, if all queries only need the sales condition by each city in Figure 2.4, we can materialize this query result so that queries do not need to drill down to the branch level for results. In short, our strategy is to trade space for time. Harinarayan et al. [14] showed the reason that pre-computing all views is not a feasible alternative due to the exponential space requirement in ROLAP. We must select a subset of candidate views to materialize, assuming there is some space constraint. This problem is challenging as the benefit of any materialized view must be weighed against the space allocation.
5.1 Motivation of our New Solution

Selecting views to materialize in OLAP is not a new problem. Papers in [14] [13] [11] [12] have proposed some approximation algorithms and heuristics to select views for OLAP. Compared to the traditional OLAP model, our view selection problem is based on the CIM framework, which consists of a conceptual layer on top of a complicated database schema like the example in Figure 2.4. Though our conceptual schema is much like a snowflake schema, it doesn’t contain actual instances but rather has virtual views of its components over the central database schema as mappings.

The complexity of mappings can make the original view selection solution perform arbitrarily badly. For example, in a snowflake schema a level contains the primary key and all attributes in a table while in a conceptual schema all attributes can belong to different tables and may need to join several other tables to connect to the primary key. If a certain level is selected to be materialized, the original view selection algorithm can just pick the primary key to materialize to save some space, and queries for attributes can get them through the same table via indices in data warehouse. But the algorithm can perform arbitrarily badly in our CIM conceptual schema because it may take so long to get the attributes through the complicated mapping if multiple tables need to be joined.

Therefore, the performance bound guarantee of the original view selection is not valid in our CIM framework. However, we think the performance bound is important for the view selection technique to be general in different scenarios. To our knowledge, this is the first work to explore view materialization on an OLAP-like conceptual layer with a schema mapping to another storage layer.

5.1.1 Goal and Input

Before we develop a new solution for view materialization in CIM, we need to list some steps to achieve the goal. In this new model, we need to solve the four following subproblems, similar to those for traditional OLAP view optimization:

- What are the candidate components to materialize as views?
- What is the cost model to evaluate queries?
• How can we model the problem in a framework to achieve a guaranteed performance bound?

• What is the algorithm to achieve the bound?

We will talk about these four problems step by step to achieve our goal. At the same time, we also need to show the assumptions of the problem as inputs. Some of these assumptions may need to be pre-processed before deciding our view selection strategy, but we think they are reasonable, which are explained later. Our solution is based on the following inputs:

• A set of virtual views on the conceptual schema as described before, which are levels, parent-child relationships and fact tables in the CDL. These views are compiled from the mappings between CDL and SDL.

• A central RDBMS database, which is the storage layer of SDL.

• Query patterns from users, which is the probability distribution of queries on the CDL. More specifically, we assume each query \( q_i \) is associated with the query frequency \( f_{q_i} \) and this query distribution information can be retrieved through a simple statistics component in the query processor.

• The RDBMS in the central database can make an estimate of the query time for a query. At the same time, the RDBMS can estimate the space of a table from the catalog. Some techniques like sampling may be applied here to get the estimate.

• Queries for business analytics are generally posed to return measures which are aggregated at some level of granularity, as well as some necessary information w.r.t. the level.

5.1.2 Lattice Framework

Before reasoning about candidate views in our problem, we first introduce the lattice framework proposed in [14]. The lattice framework is used to indicate how multiple dimensions are combined to form a view. Figure 5.1 shows how three
dimensions, location, time and customer, are selected and combined together. As we can see, three dimensions can produce $2^3$ elements, which is exponential. A hierarchy, which exists within a dimension, can also fit into this lattice framework as an extension. For example, if location has two hierarchies named city and country, we just need to replicate location twice, then replace them by city and country separately. Then we will have $2 \times 2^3$ different elements. Baralis et al. [7] proposed a heuristic to reduce the search space of the lattice. But our thesis does not focus on search space reduction.

In addition, both dimension and hierarchy have roll-up functions. We need to differentiate them here. Under a dimension scenario, all tuples roll up according to the value of the dimension column while under a hierarchy, the roll-up function needs an extra parent column to decide the roll-up function between two levels. This is the reason we have parent-child relationships in our conceptual schema.

The view selection algorithm [14] picks elements in the lattice to materialize so that it can speed up future query answering with no need to go to a finer granularity.

Figure 5.1: Lattice Representation of 3 Dimensions
More specifically, each element contains the primary key, which we call the ID in the following context, and aggregated values. Attributes and parent IDs can be found in the table in the data warehouse via indices of the level ID.

In the beginning of this chapter, we showed that the complexity of mappings can cause the algorithm perform arbitrarily badly because retrieving attributes through mappings can take a very long time. In the next section, we will talk about the candidate views we can select in our new algorithm.

5.1.3 Candidate View Reasoning

There are three types of virtual views found by the view compilation algorithm: levels, parent-child relationships, and fact relations. In Figure 4.2 we showed the translated SQL query needs to be expanded with the virtual views created as the CVL components are executed in the central database. With view materialization, we can optimize this process by materializing some of the virtual views and change the SQL rewriting process by making use of the materialized views. The query processor architecture with view materialization becomes Figure 5.2.

In this new architecture, the original SQL query can be rewritten to make use of the materialized views. For example, we can materialize all attributes of a level so that any query asking for attributes of this level can directly get them through the materialized table instead of following the virtual view of this level. Before we describe the algorithm, we provide an analysis of our considerations in selecting which views to materialize, based on the virtual views we have. In Section 5.2, we propose our view selection algorithm to choose views from the candidates.

- **Level ID + Measure(s)**

First, we consider the view level in the conceptual layer. A level consists of a single primary key or identifier, which is “ID” most of the time, representing a unique tuple and relevant attributes w.r.t. the primary key. We notice that a level does not contain any aggregated value. As mentioned before, aggregated values are of key interest to the users posing queries. If we choose to materialize a level without materializing relevant aggregated values, queries need to aggregate the value down from the fact table, which is a big cost. On the other hand, aggregated values are generally numeric values, which
do not take much space. Therefore it is always a good idea to materialize measures associated with ID in a level. Of course, the level we materialize has to associate with other levels from other dimensions.

- **Fact Relation with Only Level ID + Measure(s)**

  Second, we consider whether to materialize the fact relation. The fact table in the conceptual schema may be mapped across many tables in the storage layer. If this is the case, materializing the base fact relation can be beneficial to reducing time needed to join these storage layer tables during query answering. The other factor is the space of the fact relation. If the materialized
fact relation is relatively small, the benefit may outweigh the space allocated and the maintenance cost. As a matter of fact, the fact table can be viewed as a specialization of a level discussed above. It just combines everything from the lowest level from all dimensions. So it can be viewed as the most detailed element in the lattice.

- **Attribute in a Level**

Third, we consider attributes in a level. Attributes associated with a given level of granularity have not been considered in cost models of previous work in view selection. We postulate that view selection methods will choose not to materialize attributes to save some space, as attributes can always be accessed with the primary key in the original star/snowflake schema. At the same time, the primary key in a star/snowflake schema has an index on it, which also guarantees quick response to retrieve attributes. However, in our CIM model with additional mappings, the cost to retrieve attributes via mappings can be very expensive, depending on the complexity of the mapping itself. For instance, it could be the case that an attribute associated with a level is mapped to multiple tables in the storage schema, which requires joining these tables to retrieve it. For this reason, we may want to materialize attributes when the mappings to retrieve these attributes from storage are complicated and involve multiple tables. On the other hand, the size of an attribute can itself be another factor. In the original OLAP view selection papers, the cost model proposed evaluating the size of a view or table based on the number of rows. However, in the CIM model the granularity of the cost has the potential to drill down to the column level. If an attribute takes too much space, even if it has to go through a complicated mapping to evaluate the query, we may not materialize it due to the space constraint. In conclusion, attributes in a level can be a candidate view.

- **Parent-child Relationship**

The remaining virtual view is the parent-child relationship. This relation helps the parent level to calculate results based on the child level. In traditional OLAP, this parent-child relationship does not need to be materialized.
The reason is similar to the analysis for attributes. Given a level, it is easy to find its parent in the star/snowflake schema by using an index. In the CIM framework, this can be time-consuming as we have the mapping for the relation. If it is only a simple mapping like the case in the star/snowflake schema, we can choose not to materialize it. Otherwise, it is worth consideration. Some may argue we can directly materialize the parent level with aggregated values instead of this parent-child relation. It is a good point but not necessarily true because the space required for a parent-child relationship can be a lot less than the space required for materializing levels which at least contain ID and measures. Another reason is that since our lattice is multi-dimensional, materializing a parent-child relationship can actually benefit more than one element in the lattice, as long as the element contains the specific level as one dimension. In conclusion, the parent-child relationship can be a candidate view.

To summarize, because of the existence of the mapping between the conceptual schema and the storage schema, we have to consider the influence of such a mapping. In addition, the original cost model in view selection has the tuple-level granularity, which has the potential to drill down to the column granularity considering the intensive space constraint. Based on these observations, we analyze the scope of candidate views and split an element in a lattice to ID+measure, attribute and parent-child relationship from different dimensions. In this new lattice framework, notice that attributes and parent-child relationships can be shared by multiple elements in the lattice. We will materialize them separately from any single element consisting of level ID and measures.

**Example 8.** Figure 5.3 is a simple sales conceptual schema example. In this example, we can see two dimensions named client and branch centering the fact table sales, which has two measures: revenue and profit. In the branch dimension, there is a hierarchy between branch and city. In addition, there are some attributes attached to these levels.

Figure 5.4 is the lattice constructed from the simple sales conceptual schema. This lattice shows 6 potential elements based on the conceptual schema while each element consists of level ID and measures. From this figure, we can see how an
element in the lattice can be generated from another element.

Table 5.1 lists all attributes and parent-child relationships in the simple sales example. These attributes and parent-child relationships can only be materialized if the materialized elements in the lattice contain their corresponding levels. For instance, the population attribute from level City can only be materialized if either element cityID, clientID or element cityID gets materialized. The reason for this is that the level ID needs to be known before materializing its attribute or parent-child relationship. At the same time, we talked about materializing attributes and parent-child relationships in a separate table for sharing among elements. In this example, we can see that attribute population can benefit the elements cityID, clientID and element cityID.

5.1.4 Cost Estimation for Queries

Before we give a formal definition of the problem, we need to provide an estimate cost function for queries.

In the CIM framework, mappings are represented as SQL queries over the central database. In the last chapter, we also showed how a query on CDL can be translated into SQL queries over the central database. By materializing some
Figure 5.4: Lattice for Simple Sales Example

Table 5.1: Attributes and Parent-child Relationships in the Simple Sales Example
views, some queries on CDL are translated in a different way, which makes use of
the views to speed up query answering.

We will define a cost function to estimate the cost of all SQL queries, which
are translated from queries posed on the conceptual schema. There are already
many techniques in query optimization to estimate the cost of SQL. For example,
Selinger et al. [25] discussed processing join sequences in System R to make a cost
estimate and Chaudhuri [8] summarized the techniques used in query optimization
in RDBMSs. There are many query estimation techniques in relational database
management system. In our view selection problem, we are more interested in
deciding what views to materialize based on the cost estimation of queries. The
problem itself focuses more on the view selection strategy rather than the cost
model of SQL queries. We make use of these techniques to indicate the cost of
queries.

Just like Figure 5.2 shows, when we pick which views to materialize, some
queries may have a lower cost by rewriting based on these views, instead of fully
rewriting them based on the storage schema. We will define rules to rewrite those
queries based on selected views in the following section. In this paper, we use
integer values to specify the estimated cost of an SQL query $C(q_i)$, where $q_i$ is a
query $i$.

5.1.5 Space of Views

In this section we discuss the space of the materialized views. As stated before,
the goal of view selection is to materialize a set of useful views within some space
constraint, which makes it important to estimate the space of the materialized view.
First of all, each view can be written as SQL over the SDL. Therefore, the problem
of space estimation of views can leverage space estimation techniques in relational
database management system. Among the techniques in RDBMSs, sampling and
statistics can be heavily applied in this step. In addition, we can estimate the space
with information about the data type. As a result, we make use of these techniques
to do a size estimation of the space of views. Compared to [14], our space estima-
tion of views drills down into the column level, which is due to the granularity of
candidate views.
We use $S(C)$ to denote the space taken by view $C$, which is represented as an integer value. The total space we can allocate for materialized views is $S_{\text{max}}$.

### 5.1.6 Problem Definition

Let $q_i$ be a query, $C(q_i, \emptyset)$ is the cost of answering query $q_i$ without any view being materialized. If $M$ is the set of materialized views, then $C(q_i, M)$ is the cost of answering query $q_i$ with the materialized views $M$. Each $q_i$ is associated with the query frequency $f_{q_i}$ as the input to the problem, which can be obtained from the query distribution among all users. Some data about queries can be collected before running the query optimization process. Let $Q$ be the set of queries which contains $q_1, q_2, ..., q_k$, we denote the total cost to be $TC(Q, M)$ based on view set $M$, where

$$TC(Q, M) = \sum_{i=1}^{k} f_{q_i} C(q_i, M) \quad (5.1)$$

The view selection problem in our CIM project is to select a subset of views $M$ to materialize to minimize the query cost defined in (5.1), with the requirement that the total space taken by $M$, which is $S(M)$ is within the space constraint $S_{\text{max}}$.

### 5.1.7 Benefit Function

When a view gets materialized, we re-evaluate the cost of CDL queries which can be rewritten with the newly added materialized view. We need to quantify how much this newly added view can benefit the relevant queries. Let $C$ be the newly added view. We define the benefit of this view to be the difference between total cost before and after this newly added view. Formally, we have $B(C, M)$, which means the benefit of view $C$ added to the view set $M$. We define

$$B(C, M) = TC(Q, M) - TC(Q, M \cup \{C\}) \quad (5.2)$$

The benefit of this view is the difference between two estimated costs before and after materializing this view.
5.2 CIM View Selection Algorithm

As is stated in the previous chapter, we can always answer a query by drilling down to the fact table and forming an SQL query on the conceptual schema. In the previous section, four types of candidate views, which are level ID+measure, fact relation, attributes and parent-child relationships are proposed in the new CIM view selection solution.

In this section, we focus on two problems. The first one is how to rewrite queries if one of these candidate views get materialized. The second one is the core problem, which is how to select views to materialize while achieving the performance bound.

5.2.1 Query Rewriting Rules with Materialized Views

Let’s first start from the candidate view attribute. If an attribute of a level is materialized, query rewriting is straightforward by just replacing the virtual view, which may join several tables on the fly with the materialized ID attribute pair. The benefit of such a replacement can reduce the processing time to retrieve the interested attribute by just accessing the materialized pair.

Next we look at the candidate view parent-child relationship. Very similar to that of an attribute, query rewriting of a materialized parent-child relationship is straightforward by just replacing the virtual view of the materialized parent-child relationship pair.

Last, we look at the candidate view level ID + measures. As we have discussed before, the materialized fact relation is just a specialized form of level ID + measures. They belong to the same category. So we discuss them together here. If an element of level ID + measures is materialized, any element which depends on this element no longer needs to drill down to the fact table. Instead, it can get the measures from this materialized view. For example, let’s look at Example 6 in the previous chapter. If the element of City, Client+measure is materialized as CityClientM, the SQL query can be rewritten as

```
SELECT City.name, Client.name, sum(CityClientM.totalRevenue)
FROM CityClientM, City, Client
WHERE CityClientM.cityID = City.cityID
AND CityClientM.clientID = Client.clientID
```
AND City.name = "Vancouver"
GROUP BY CityClientM.cityID, CityClientM.clientID

From the example, we can see the measure totalRevenue can be accessed in CityClientM but not from the fact table in the original example. The scope can become much smaller in CityClientM.

5.2.2 Submodularity and CIM

Before describing the view selection algorithm, we need to introduce the submodularity property and model our problem to satisfy this property, with the goal to make use of some good results of this property. Nemhauser et al. [22] first proposed the concept of submodularity and proved its lower bound for approximation with the nonincreasing and nonnegative properties. We first define submodularity here, which is:

**Definition 1.** Let \( S \) be a finite set, \( A \subset B \subset S \) and \( x \in S \). A function \( f \) is submodular if
\[
f(A \cup x) - f(A) \geq f(B \cup x) - f(B).
\]

Intuitively the property means that if an element \( x \) is added earlier, the benefit will be greater than adding it later. It is also proved that the simple greedy algorithm can produce a solution with the lower performance bound of \( (1 - 1/e) = 63\% \) of the optimal solution by selecting the same number of views under time complexity \( O(n^2) \) if the function \( f \) is nonnegative, nondecreasing and submodular. An example of such a problem which fits the above properties is the set cover problem, which is known to be NP-Complete problem. The simple greedy algorithm chooses the current most beneficial set and it is guaranteed that the \( k \) sets chosen in this way will be at least 63\% of the optimal choice. The intuition is that the chosen set at the current stage can cover more space while some of the space will be covered by other sets if chosen later. The official proof can be found in the paper [22].

Later, Khuller et al. [18] and Sviridenko [27] extended the former concept and proved the same bound for the former problem with a space constraint. Take the set cover problem as an example. The problem changes in that each set has a cost value associated with it instead of a unit cost. Instead of choosing \( k \) sets, it is necessary to choose sets whose total size is within some space constraint \( S \). By
modifying the simple greedy algorithm to incorporate an enumeration technique, the lower bound can be maintained with time complexity $O(n^5)$.

In the traditional view selection problem, Harinarayan et al. [14] first adopted the submodularity property in their solution but they used the name monotonicity. Later Gupta et al. [13] extended this property to solve the view selection plus index selection problem. In our view selection algorithm, we decide to leverage the power of the submodularity property and model our problem similarly to [13].

**Observation 1.** According to the submodularity definition, if we define an individual element, which is a set based on a level, consisting of LevelID+measures (must-have), parent-child relationships based on this level (optional) and attributes of this level (optional), the benefit function $B$ of such an element is nonnegative, nondecreasing and submodular.

**Proof.** **First, it is easy to prove that such a benefit function is both nonnegative and nondecreasing.** Remember the definition of our cost function $TC(Q,M)$ is query cost based on the set of views $M$. Suppose the newly added view $v$ causes some queries to be slower. Those queries can just stop using the view $v$. In this case, it is guaranteed that $TC(Q,M+v) \leq TC(Q,M)$ and, of course, the total cost $TC$ is nonnegative.

**Next we need to prove that the benefit function is submodular.** Let’s first discuss the must-have element “level ID and measures” in an individual set. Queries in a certain level have to drill down to a lower level or the current level for measures to aggregate from. We can see from the lattice that “level ID and measures” can benefit query answering in both the current level and above levels in coarser granularity with the query rewriting rules. Let us define two random views $v_1$ and $v_2$, which are ready to be added to a set of materialized views $M$ sequentially. To simplify the proof, we assume all queries are above the highest level $l_M$ in $M$ so that the queries can make use of $M$. The actual scenario is just splitting queries to several partitions by different levels of the views in $M$. To prove such a set satisfies submodularity, we just need to prove $B(v_2,M) \geq B(v_2,M \cup v_1)$.

- If $v_2$ is below $l_M$, then both $B(v_2,M)$ and $B(v_2,M \cup v_1)$ are zero because all queries only need to drill down to $l_M$ to retrieve the relevant measures.
• If $v_2$ is above $l_M$ but below $v_1$, then queries above level $v_1$ do not need to drill down to $v_2$ but can make use of $v_1$, which means $v_2$ cannot provide as much benefit as it is when directly added to $M$, thus $B(v_2, M \cup v_1) < B(v_2, M)$.

• If $v_2$ is above $v_1$ while $v_1$ is above $l_M$, then $B(v_2, M)$ is the accumulated benefits of queries above the level of $v_2$ w.r.t. the original cost based on $l_M$. On the other hand, $B(v_2, M \cup v_1)$ is the accumulated benefits of queries above $v_2$ w.r.t. $v_1$. Since $v_1$ is above $l_M$, then $TC(Q, M + v_1) < TC(Q, M)$, thus we can get $B(v_2, M) \geq B(v_2, M \cup v_1)$.

• If $v_2$ is above $l_M$ while $v_1$ is below $l_M$, as was analyzed before, both $B(v_2, M)$ and $B(v_2, M \cup v_1)$ are the same since queries will just ignore $v_1$ but use $M$.

**Now we add the parent-child relationship to the set.** In the above analysis, we know queries have to drill down to a certain level to aggregate measures. At the same time, we proved that the benefit function of a set containing level ID and measures is submodular. We extend this set by adding the parent-child relationship to it. The benefit of a parent-child relationship is to reduce evaluation time to figure out how to roll up from a current level to a higher level. We reason why adding a parent-child relationship to the set the benefit function is still submodular. The core of submodularity is that adding an element early is guaranteed to provide more benefit than later. In this case, adding a parent-child relationship early can provide a benefit for all the elements combining this level and other different granularities of dimensions in the lattice. Materializing another view before this set will reduce the benefit of this parent-child relationship if this view is the element in the lattice which can roll up with the parent-child relationship. In other cases, adding another view beforehand will not change the benefit of this set. Thus, adding a parent-child relationship to the set satisfies submodularity.

**Lastly, we add the attribute to the set.** The benefit of materializing an attribute is to reduce the repeated calculation by different elements containing the level in the lattice. If the level ID and measures of another element containing the level is materialized, it can make use of the materialized attribute instead of recalculating it. Otherwise, the level ID and measures of this element have to drill down to an element containing either a lower level or finer dimension, and then retrieve
the attribute, which is a two-step process. The only exception that violates this two-step process is that the storage schema mixes attributes and level IDs for both parent and child in the same table. In that case, roll-up function and the retrieval of attributes can be done in the same step. But we argue that this is not the common way since it makes no sense to have level ID to identify these attributes. At the same time, it is too expensive for the storage schema to store that data. If such a case happens, we will suggest normalizing the storage schema first. Therefore, materializing attributes just provides a shortcut for the step of retrieving attributes. Materializing another element’s ID and measures or parent-child relationship beforehand will not change the benefit of materializing attributes, which means it is submodular.

We have proved that a set containing level ID and optional parent-child relationships and attributes satisfies the submodular function. In addition, we can extend the above analysis to a heterogeneous hierarchy too. The proof is similar to the proof of the level ID and measures.

**Observation 2.** Within an element in the lattice when level ID and measures are already materialized, the benefit function of parent-child relationships and attributes of this element is nonnegative, nondecreasing and submodular.

**Proof.** The nonnegative and nondecreasing property are trivial here. We will discuss the submodularity property.

We have two types of candidates within an element: parent-child relationships and attributes. The benefit targets of these two types of candidates are different. Parent-child relationships aim to provide benefit for higher levels to do roll-up functions. On the other hand, attributes aim to avoid repeated calculation of the current level. Therefore, these two types of candidates do not interfere with each other, which means the benefit function is submodular between them.

Next, all attribute candidates are submodular too. When evaluating a query with multiple attributes to project, we will retrieve attributes one by one. Therefore, the sequential property of evaluating a query guarantees that materializing an attribute sooner or later provides the same benefit.

For parent-child relationships, it is the same case. Therefore, the observation holds that we can make use of the lower bound provided by the submodularity
property within an element.

5.2.3 Algorithm Description

The previous section modeled our problem to adopt the submodularity property. As we can see from the above two observations, submodularity applies both between and within the set consisting ID+measures, attributes and parent-child relationships. Similar to the inner-greedy algorithm proposed in [13], we propose our CIM Greedy View Selection Algorithm, which uses greedy selection twice.

The algorithm is shown in Algorithm 1. As we can see from the outer while loop, it selects greedily for the element considering all these three types of candidates. Moreover, it also considers attributes and parent-child relationships for the selected element in case adding them is better than adding a new element. This outer loop actually conforms to the submodularity property of Observation 1 and makes use of the simple greedy algorithm. In the inner while loop, it selects parent-child relationships or attributes in a greedy way, which conforms to Observation 2 and we can also get a bound using greedy algorithm.

Let $n$ be the number of all elements in the lattice and $m$ be the number of all attributes and parent-child relationships. From the outer while loop, we can see it will take at most $O(n^2)$ to both iterate and pick the most beneficial element. More specifically, it can iterate $n$ stages and takes $n$ steps to find the most beneficial element at each stage. The inner while loop is similar, but note that the set of attributes and parent-child relationships are shared among all elements. We maintain a global array to record the existence of such a set to prune the candidates. In addition, those parent-child relationships and attributes that can be retrieved using simple mapping and index with rewritten rules are not considered either. But in the worst case, the time complexity for inner loop in the algorithm is $O(m^2)$. Overall, the time complexity for this CIM Greedy View Selection Algorithm is $O(n^2m^2)$.

Example 9. We provide an example to elaborate our algorithm based on the Simple Sales Example shown in Figure 5.3. Differently from the lattice of Figure 5.4 shown in the Simple Sales Example, we associate the cost of ID+measures and parent-child relationships for all different elements in the lattice, which is shown in Figure 5.5.
Algorithm 1 CIM Greedy View Selection Algorithm

// define M to be the materialized views, consisting of sets we defined above
M = ∅
define A to be the set of parent-child relationships and attributes selected so far
// if the set M is still within the space constraint S
while S(M) < S do
    // C is the best candidate element in the lattice at this stage
    C = ∅
    for each element E_i in the lattice do
        // IS is the element set added in a greedy way
        IS = ID + measures of E_i
        // BC is the value of benefit per unit space
        BC = 0
        // construct the element E_i to be the most beneficial in a greedy way
        while B(IS)/C(IS) > BC do
            let A_{ic} be the parent-child relationship or attribute not in A, but is related
to E_i, that has the maximum benefit per unit space w.r.t. M ∪ IS
            mark A_{ic} already selected in A
            IS = IS ∪ A_{ic}
            BC = B(IS)/C(IS)
        end while
    // select the most beneficial element among all choices in the lattice
    if B(IS,M)/S(IS) > B(C,M)/S(C) or C = ∅ then
        C = IS
    end if
end for
// in case adding parent-child relationships or attributes to the existing element
provides more benefit
for each parent-child relationship or attribute A_{ic} whose element E_i has been
partially materialized do
    if B(A_{ic},M)/S(A_{ic}) > B(C,M)/S(C) then
        C = A_{ic}
    end if
end for
M = M ∪ C
end while
// M is the result set
return M
In the cost estimation section, we indicate the cost of a query is decided by the query estimator in the DBMS. To better explain our algorithm, we simulate the process of query estimation and decompose the cost of a query into three parts: the cost of reading the closest supporting element to roll up, the cost of calculating parent-child functions and the cost of retrieving attributes. In Figure 5.5, the cost next to an element is the cost of reading this element to provide a supporting range. The cost next to the line between elements is the cost to calculate the parent-child relationship. For simplicity, the reading cost of the supporting range in this example is the same as the space taken by this element. Also next to the elements we put the queries posed by users, which are shown for convenience. In Table 5.2, we show the space taken for materialization and the costs before and after materialization for both the parent-child relationships and the attributes.

Suppose we only have four queries which are shown below and they occur at the same frequency:

- Q1: All measures and addresses of branches grouped by branch ID and client ID
- Q2: All measures and names of clients grouped by client ID
- Q3: All measures, names and population of cities grouped by city ID
- Q4: All measures, names and population of cities grouped by city ID and client ID

### Table 5.2: Cost and Space for Attributes and Parent-child Relationships in Simple Sales Example

<table>
<thead>
<tr>
<th>Level</th>
<th>Attribute/Parent-child Relationship</th>
<th>Space</th>
<th>Cost before</th>
<th>Cost after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>address</td>
<td>10</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>Branch</td>
<td>staffNum</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>City</td>
<td>population</td>
<td>5</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>City</td>
<td>name</td>
<td>10</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Client</td>
<td>name</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Client</td>
<td>signupYr</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Branch</td>
<td>branchID-cityID</td>
<td>10</td>
<td>30</td>
<td>5</td>
</tr>
</tbody>
</table>
At each round, our CIM Greedy View Selection Algorithm picks the most benefit per unit space set among all elements in the lattice.

Let’s assume the total space we have is 150. In round 1, we choose to materialize the element of cityID, clientID+measure because it can achieve the maximum benefit per unit space of $(150 \times 3 + 25 \times 2)/50 = 10$. The benefit $150 \times 3$ is from three queries Q2, Q3 and Q4, which no longer need to depend on the storage schema for the aggregate value from but can leverage the element of cityID, clientID+measure. In addition, by materializing this element we can avoid calculating the parent-child relationship of branchID-cityID for both Q3 and Q4, which provides the benefit of $25 \times 2$. But in this round the algorithm will not choose any attribute because none of them can out perform the benefit per unit space ratio.
above. To compare, materializing \textit{branchID}, \textit{clientID+measure} can only provide the ratio of \((110 \times 4)/90 = 4.89\), which is less than the above choice.

Since we still have some extra storage space, our algorithm continues to the next stage. In round 2, we choose to materialize the attribute \textit{population} in the \textit{city} level, which can benefit queries Q3 and Q4 and provides the highest benefit per unit space ratio. The benefit it brings is 30 in total and it only consumes 5 units of space. The ratio is \(15 \times 2/5 = 6\).

In round 3, we choose to materialize \textit{branchID}, \textit{clientID+measure}, as well as the attribute \textit{address} in the \textit{branch} level, which can bring the most benefit per unit of space of \((110 + 30)/(90 + 10) = 1.4\) for Q1. Note that the parent-child relationship of \textit{branchID-cityID} does not provide any benefit because the element \textit{cityID}, \textit{clientID+measure} is already materialized.

The algorithm stops after round 3 because we have exceeded the space limit. We can see that the space taken is slightly more than the space constraint 150 but provides the benefit of \(500 + 30 + 140 = 670\). The solution provided by the algorithm happens to be optimal.

**Theorem 1.** For the lattice of a conceptual schema whose ID+measures, parent-child relationships and attributes can be chosen to be materialized, it is guaranteed our CIM Greedy View Selection Algorithm can produce a solution \(M\) that uses at most \(S + r\) units of space, where \(r\) is the maximum size of an arbitrary element of ID+measures, parent-child relationships and attributes. Only those parent-child relationships and attributes that provide benefits after materialization are considered. The benefit using space \(M\) is at least \((1 - 1/e^{0.63}) = 0.467\) of the optimal benefit using as much space as \(M\).

**Proof.** Let the optimal solution be \(O\) and \(S(O)\) be the space taken by the optimal solution \(O\). The benefit of \(O\) is \(B\). As we know, the optimal solution \(O\) consists of \(k\) elements, where \(k\) is at least 1. Also from what we have discussed before, attributes and parent-child relationships cannot be materialized when ID+measures, which contains ID for their reference, is not materialized, thus providing no benefit. So we split \(O\) into \(k\) disjoint element sets \(O_1, O_2, O_3, ..., O_k\). Each \(O_i\) consists of ID+measures and optional parent-child relationships and attributes of the same element.
Consider a stage at which our greedy algorithm has chosen a set $G_l$ which consists of elements $E_1, E_2, ..., E_l$. The benefit of the set $O \cup G_l$ is at least $B$. So the benefit of the set $O$ w.r.t. $G_l$, $B(O, G_l)$ is at least $B - \sum E_i$.

At the current stage after $G_l$, the worst case is where the $O_1, O_2, ..., O_k$ optimal sets are not selected in $E_1, E_2, ..., E_l$. Due to the submodularity property, the benefit of any individual $O_i$ is not increasing at later stages, which means there is some overlap of benefits among different sets of $O_i$. Also because the cost of a level is not changing, the benefit per unit space of $O_i$ in later stages will not increase either. Therefore, we can guarantee that we can find an $O_i$, such that $B(O_i, G_l)/C(O_i)$ at the current step is no smaller than the average benefit per unit space over all stages, which is $B(O, G_l)/M = (B - \sum E_i)/M$ due to the benefit overlap.

Next, we show the performance of our greedy algorithm when materializing an element, which is denoted as $IS$ in the algorithm. We can show that our algorithm can always achieve 0.63 times the optimal benefit per unit of $O_i$, which shares the same element with $IS$, when $IS$ reaches its maximum benefit per unit space. First, we assume the ID+measures is not selected in $IS$ yet. Thus, we can have

$$B(IS, G_l) = B(IM, G_l) + B(PCA, G_l \cup IM)$$  \hfill (5.3)

In the above equation, $IM$ is ID+measures while $PCA$ is parent-child relationships and attributes. Note in the second part, $B(PCA, G_l \cup IM)$ is selected in a greedy way and stops when the benefit per unit of $IS$ reaches the maximum. According to the submodularity property in Observation 2 and the bound of submodularity proved in [18], by greedily selecting attributes or parent-child relationships we can always achieve 63% of the benefit, given the space requirement of the optimal set.

However we do not know the space taken in the optimal case. That is the reason we cannot guarantee the problem to be solved within space constraint $S$. However if we relax the space constraint to be the space taken by greedily selecting attributes or parent-child relationships until the benefit per unit space reaches a maximum, we can guarantee the benefit is at least 63% of the optimal solution within the space used by the greedy algorithm. The reason is that we can make sure each view selected at its stage satisfies $B(PCA_{i+1}, G_l)/C(PCA_{i+1}) \geq (B - \sum PCA_i)/M$. 

57
On the other hand, notice in the inner while loop, the greedy algorithm stops when the benefit per unit space reaches maximum. If we refer this set to be $IS_{max}$, then we know

$$B(IS_{max}, G_i)/C(IS_{max}) \geq B(IS, G_i)/C(IS)$$

$$= (B(IM, G_i) + B(PCA, G_i \cup IM))/C(IS)$$

$$\geq (B(IM, G_i) + 0.63(O_i - IM, G_i \cup IM))/C(IS)$$

$$\geq 0.63B(O_i, G_i)/C(IS)$$

Therefore, we can prove for each $IS_{max}$, it is guaranteed that the algorithm can achieve $0.63$ of its optimal benefit per unit space of the same element $O_i$ using the new space of $IS$, assuming the element of $IS_{max}$ exists in $O$. Remember that we have proved that must exist one $O_i$, such that $B(O_i, M)/S(O_i) \geq B(O, G_i)/M = (B - \sum E_i)/M$. By greedily selecting the element in the outer loop of the algorithm, we can guarantee such a selected element $C$ is greater than any benefit per unit space of $IS_{max}$. Thus, such a $C$ is guaranteed to be greater than $0.63$ of the $O_i$, which is no smaller than the average benefit per unit space over the following stages. Thus we can get

$$B(C, G_i)/C(C) \geq 0.63(B - \sum E_i)/M \quad (5.4)$$

We also note from the algorithm that we need to choose attributes or parent-child relationships from the existing level. This step is needed to make sure that we are choosing the element that fits Equation 5.4. This step can make sure the selected $C$ in the algorithm is greater than $IS_{max}$ in every element.

Therefore, from above statements, it is always guaranteed that for each inner stage of the algorithm, we can achieve $0.63$ of the optimal solution of an element. According to Lemma 2 in [18], we can achieve

$$B(G_i) = (1 - \prod_{k=1}^{i} (1 - 0.63 \cdot C(O_i)/k)) \cdot B \quad (5.5)$$

The above equation reaches its minimum when each $C(O_i)$ is equal, and then
\[ C(O_i)/k = 1/i. \] Therefore,

\[
B(G_i) \geq (1 - (1 - (1 - 0.63 * 1/i)^i)^i) * B \\
\geq (1 - 1/e^{0.63}) * B \\
= 46.7\% * B
\]

Therefore, we can get the minimum performance bound for the problem, which is 46.7\% of the optimal solution. But remember that we relax the space constraint for the problem to make sure each step is not interrupted by the space constraint. In the inner while loop we will use the space \( r \), which is the maximum level size combined with level ID+measures, attributes and parent-child relationships. Of course, the attributes and parent-child relationships are through complicated mapping even after query rewriting rules. Therefore, within the space \( S + r \), we can guarantee that everything selected with our CIM Greedy View Selection Algorithm conforms to the benefit analysis above, which is at least 46.7\% of the optimal benefit.

\( \square \)
Chapter 6

Conclusions and Future Work

6.1 Conclusions

This paper fulfills the query processor component of the CIM framework. It started by designing a query language accompanied with a query interface which naturally fits the visual representation of the CIM framework. Unlike the traditional query language SQL, this query language is more intuitive and only focuses on user needs while hiding unnecessary implementation details from end users. With the elegant query interface, it is simple enough for business users to pose queries on the conceptual schema. We also showed how to translate this CIM query language to SQL, which can be executed in the central database.

The second part of this thesis focused on optimization of the query processor. Because our conceptual schema in CIM is virtual, the traditional view selection solution in OLAP does not fit the CIM framework to provide a performance bound. Therefore we remodeled the problem to satisfy the submodularity property and developed an algorithm which achieves a performance bound of 46.7% of the optimal solution with some extra space over the original space constraint. Strict proofs and examples were also provided.

In summary, the thesis complements the CIM framework as an important component.
6.2 Future Work

First of all, we focus on some work to complement this thesis. Though in theory, we notice that the traditional view selection solution from OLAP can perform arbitrarily badly with complex mappings of attributes and parent-child relationships, we would like to perform some experiments to verify that our algorithm can outperform the original one. It is also a good idea to show the relationship between the complexity of the mappings and the performance of this algorithm to gain a deeper understanding.

In the long term, there are many interesting directions about view selection in CIM. Dynamics of the view selection is one of them. As we know, one important feature of the CIM framework is the flexibility to change conceptual schemas over the central database. At the same time, the query pattern on conceptual schemas can change over time. The algorithm in the paper focuses more on the static state of the CIM framework, which may bring too much drastic change in the view selection strategy in a dynamic environment. Thus, developing a view selection strategy to work the changing conceptual layer and changing query patterns is practical and important.
Bibliography


