Extension of a cube querying system with OLAP operators

A Thesis

submitted to the designated
by the Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Sotirios Zogos

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION
IN ADVANCED COMPUTER SYSTEMS

University of Ioannina
School of Engineering
Ioannina 2022

Examining Committee:

- **Panos Vassiliadis,** Professor, Department of Computer Science and Engineering, University of Ioannina (Advisor)
- **Evaggelia Pitoura,** Professor, Department of Computer Science and Engineering, University of Ioannina
- **Apostolos Zarras,** Professor, Department of Computer Science and Engineering, University of Ioannina

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

GLOSSARY

csv	comma-separated values
GUI	Graphical User Interface
OLAP	Online Analytical Processing
SQL	Structured Query Language
tsv	tab-separated values
UML	Unified Modeling Language

ABSTRACT

Sotirios Zogos, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, February 22
Extension of a cube querying system with OLAP operators
Advisor: Panos Vassiliadis, Professor

Nowadays there is a need for a quick processing of very large amounts of data. This need is covered by Online Analytical Processing (OLAP) systems which manage data using a multidimensional hierarchy model based on Data Cubes. A Data Cube consists of multiple dimensions and each dimension consists of multiple levels. The main purpose of this thesis is to implement the basic OLAP operations, such as Roll-up, Drill-down, Slice and Dice in the Delian Cubes query answering tool [VGK+18]. Also, we introduce the concept of a session, a continuous interchange of information between client and server in the Delian tool, alongside with a Query History Manager, responsible for keeping up all the queries that the user runs in a specific session.

ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ

Σωτήριος Ζώγος, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 22
Επέκταση Συστήματος Επερώτησης Κύβων με Τελεστές OLAP
Επιβλέπων: Παναγιώτης Βασιλειάδης Καθηγητής

Τα συστήματα Online Analytical Processing (OLAP) είναι συστήματα λογισμικού που επιτρέπουν τη διαχείριση της πληροφορίας με βάση το πολυδιάστατο ιεραρχικό μοντέλο δεδομένων. Τα δεδομένα οργανώνονται σε ιεραρχίες επιπέδων και επιτρέπουν πράξεις ιεραρχικής πλοήγησης και φιλτραρίσματος των δεδομένων για να παραχθούν «κύβοι» δεδομένων ως αποτελέσματα των ερωτήσεων που τίθενται επί των λεπτομερών δεδομένων.

Το σύστημα Delian Cubes [VGK+18] είναι ένα σύστημα διαχείρισης κύβων στα πλαίσια μιας συγκρότησης της πληροφορίας σε ένα πολυδιάστατο, ιεραρχικό χώρο δεδομένων. Το αντικείμενο της παρούσας εργασίας αφορά στην υλοποίηση βασικών τελεστών μιας άλγεβρας κύβων και της ενσωμάτωσής του στο σύστημα Delian Cubes, καθώς και την πειραματική τους αξιολόγηση.

CHAPTER 1

INTRODUCTION

1.1 Goals

Online Analytical Processing (OLAP) systems are systems which organize data using a multidimensional hierarchy model. In that model, every dimension has a hierarchy of levels and based on them, data analysts are able to manipulate data (by running Queries) and get the results in a form of Data cube.

The Delian Cubes tool [VGK+18] is a Cube management tool which organizes the data in a multidimensional hierarchically model. Specifically, Delian Cubes is a query engine that receives cube queries, processing them and returns the information on tab-separated text files.

The goal of this thesis is the implementation of the basic cube-algebra operators (OLAP operators) in the Delian Cubes tool and their experimental evaluation. Also, we introduce the concept of a session, a continuous interchange of information between client and server in the Delian tool, alongside a Query History Manager, responsible for keeping up all the queries that the user runs in a specific session.

1.2 Structure of the Thesis

This thesis consists of 5 chapters: In Chapter 2, we review the related work of the thesis. In Chapter 3, we discuss the main operators implemented. In Chapter 4 we report on the experiments, and in Chapter 5 we offer conclusions and thoughts for future work.

CHAPTER 2

RELATED WORK

2.1 OLAP

Online Analytical Processing (OLAP) is a category of software that allows users to analyze information from multiple database systems at the same time [Tayl20]. It is a technology that enables analysts to extract and view business data from different points of view.

Analysts frequently need to group, aggregate and join data. These operations in relational databases are resource intensive. With OLAP data can be pre-calculated and pre-aggregated, making analysis faster. OLAP databases are divided into one or more cubes. The cubes are designed in such a way that creating and viewing reports become easy. OLAP stands for Online Analytical Processing.

2.1.1 OLAP Cubes

At the core of the OLAP concept, is an OLAP Cube. The OLAP cube is a data structure optimized for very quick data analysis. The OLAP Cube consists of numeric facts called measures which are categorized by dimensions. OLAP Cube is also called the **hypercube**.

Usually, data operations and analysis are performed using the simple spreadsheet, where data values are arranged in row and column format. This is ideal for two-dimensional data. However, OLAP contains multidimensional data, with data usually obtained from a different and unrelated source. Using a spreadsheet is not an optimal option. The cube can store and analyze multidimensional data in a logical and orderly manner.

How does it work?

A Data warehouse would extract information from multiple data sources and formats like text files, excel sheet, multimedia files, etc. The extracted data is cleaned and transformed. Data

is loaded into an OLAP server (or OLAP cube) where information is pre-calculated in advance for further analysis.

2.1.2 Basic analytical operations of OLAP

Four types of analytical operations in OLAP are:

1. Roll-up
2. Drill-down
3. Slice and dice
4. Pivot (rotate)

1) Roll-up:

Roll-up is also known as "consolidation" or "aggregation." The Roll-up operation can be performed in 2 ways

1. Reducing dimensions
2. Climbing up concept hierarchy. Concept hierarchy is a system of grouping things based on their order or level.

For example, consider the following diagram (3D cube) that contains information about items that have been sold in a set of Locations (cities) on every quarter of the year.

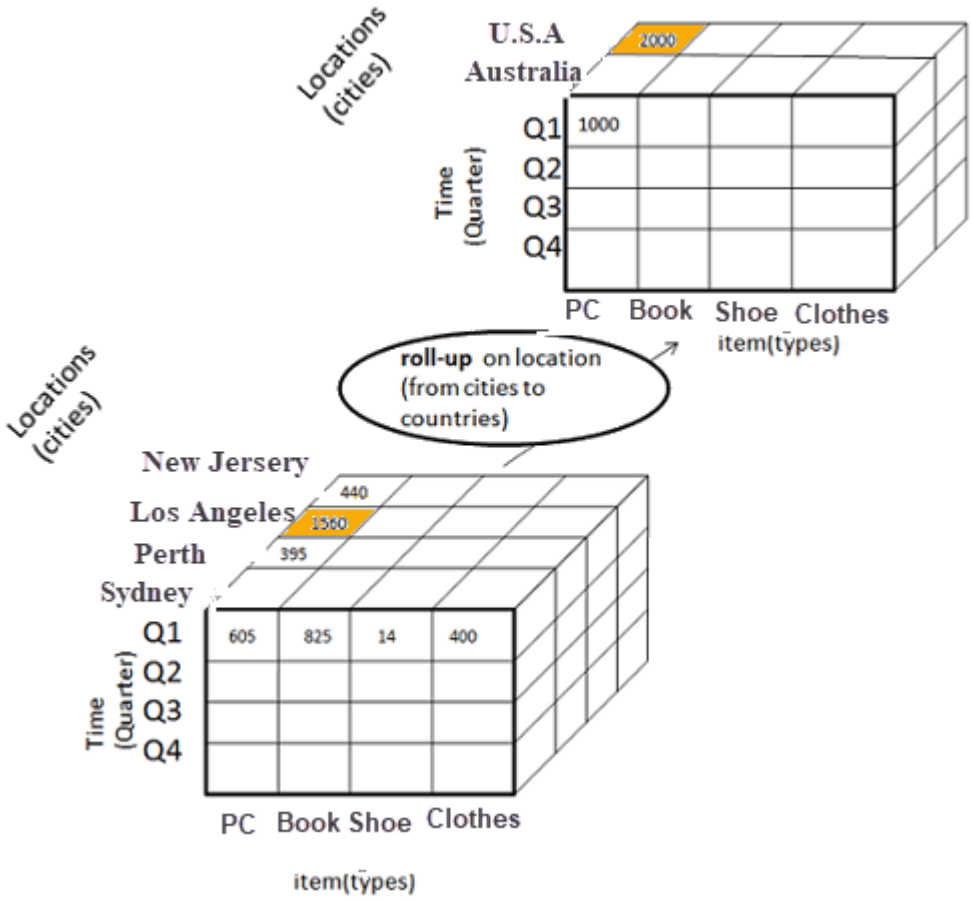

Figure 1: [Tayl20] Roll-up example.

By executing the Roll-up operation for the location attribute on this cube, the dimensions of the cube reduced and the "Location" attribute has jumped in a higher hierarchy (from cities to countries).

2) Drill-down

In drill-down data is fragmented into smaller parts. It is the opposite of the rollup process. It can be done via:

- Moving down the concept hierarchy
- Increasing a dimension

Because of drill-down's behavior is exactly the opposite of the roll-up's behavior, using the previous example now we can drill-down to one (or more) of cube's dimensions. As you can see in the following picture, choosing the "Time" attribute, the drill-down operation moves the parameter one step below on the hierarchy (from quarters of a year to months). This has also as

a result the increase of the dimensions (we used to have 4 quarters and now we are having 12 months).

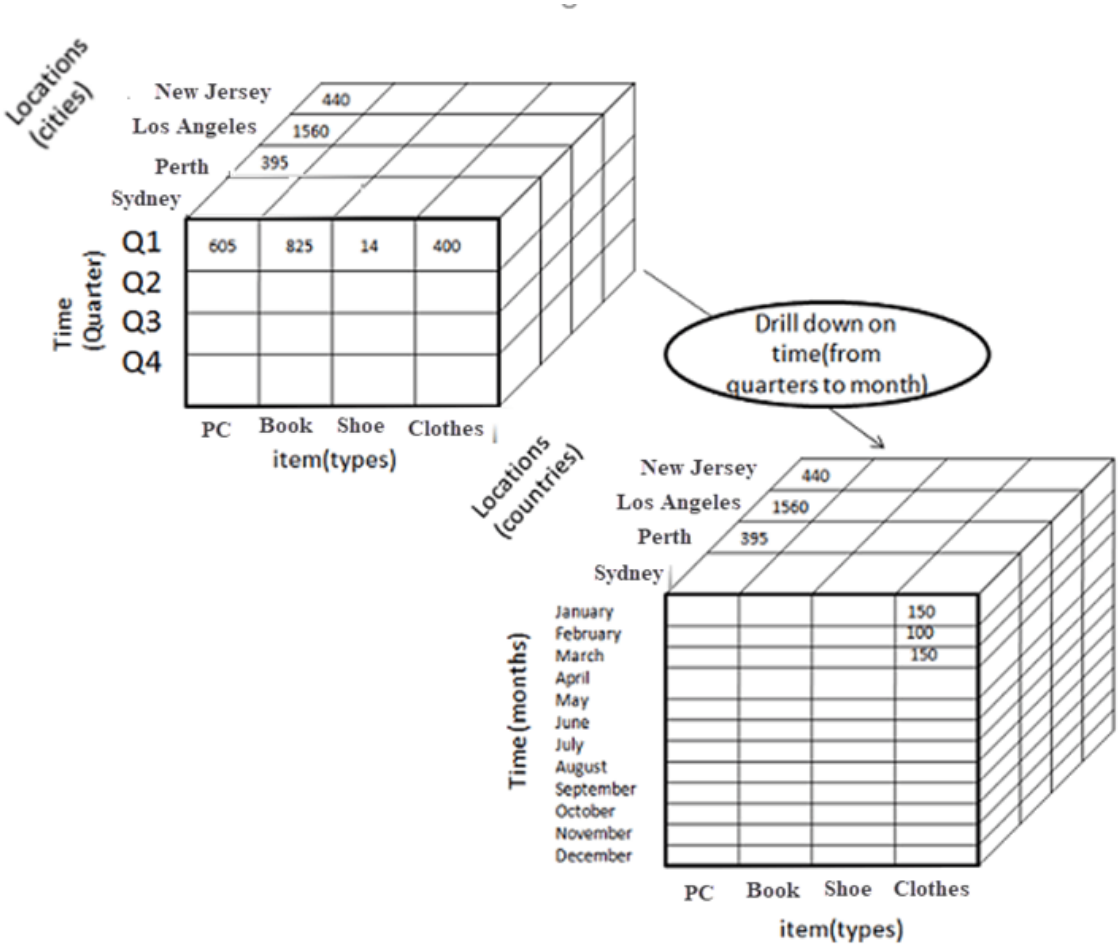

Figure 2: [Tayl20] Drill-down example.

3) Slice:

Here, one dimension is selected, and a new sub-cube is created.

On the slice operation a projection of a cube is created because one of its dimensions has been stabilized and the other dimensions are showing results relevant for the stabilized dimension. For example, suppose that we have the previous example with a cube that contains three parameters: Items, Time, and Locations. This time we need only the number of items that have been sold in these locations but only for the first quarter of the year. By operating slice on the cube, we get the following 2D projection of the cube.

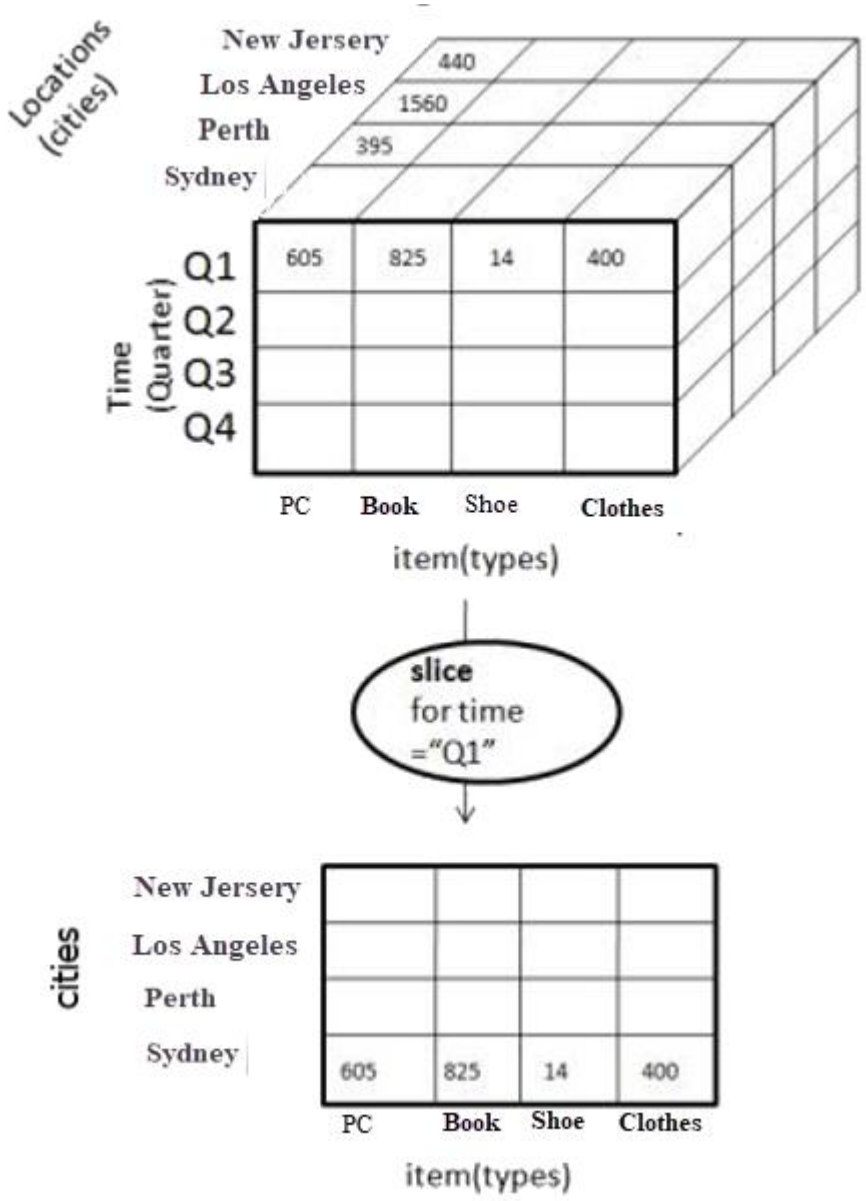

Figure 3: [Tayl20] Slice example.

Now, we have a 2D table which contains the items that have been sold in these locations keeping the time parameter stable.

Dice:

This operation is similar to a slice. The difference in dice is you select 2 or more dimensions that result in the creation of a sub-cube.

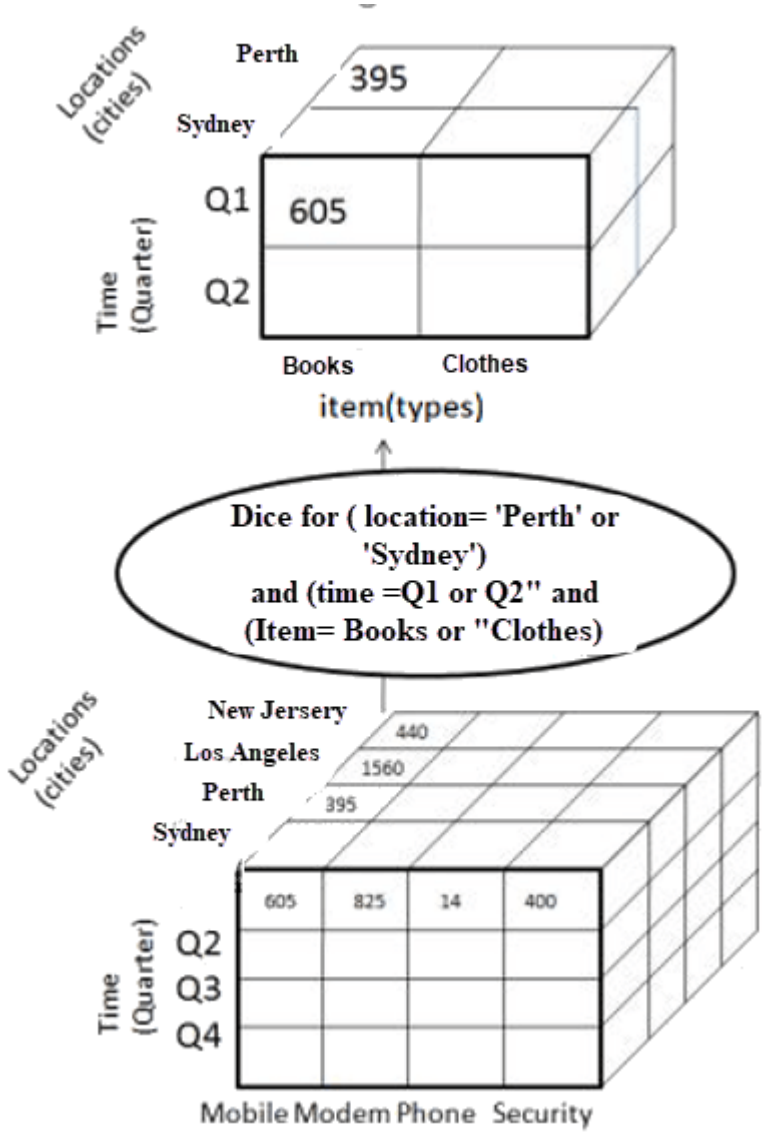

Figure 4: [Tayl20] Dice example.

4) Pivot

In Pivot, you rotate the data axes to provide a substitute presentation of data.

For example, consider a 2D cube (table) containing two data of two parameters: Items, Locations. When the pivot operation is applied to this cube, the cube has been rotated as seen in the picture bellow.

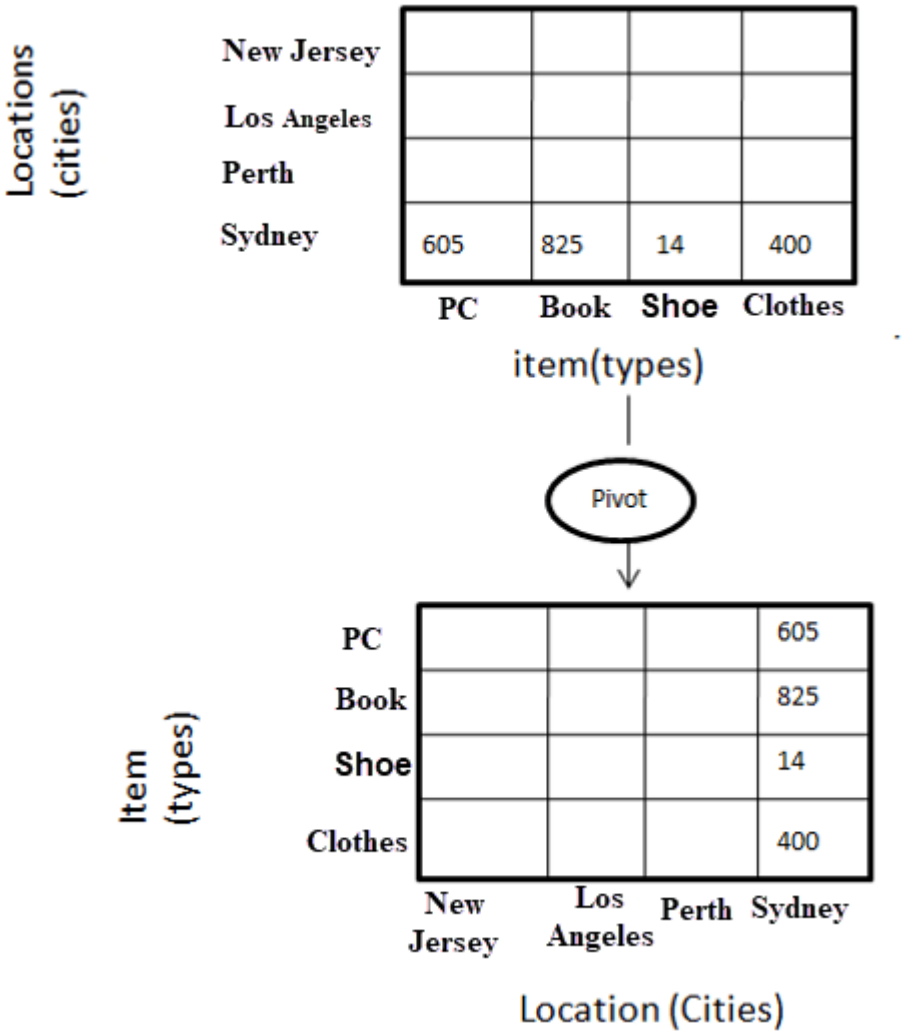

Figure 5: [Tayl20] Pivot example.

2.2 Intentional Analytics Model

[VaMa18] The Intentional Analytics Model brings a new approach to OLAP. With this approach, analysts can explore the information space of a data warehouse by submitting queries in a higher level (user friendlier than before). In the next phase these queries are transformed to traditional OLAP operations, so that the OLAP algorithms can execute the queries and return the corresponding information.

The lack of experience in SQL and other programming languages from inexperienced data analysts was the main reason that data programmers need to create a more interactive and higher-level model for data analysis.

2.2.1 Operations of Intentional Analytics Model

- Analyze: The Analyze operator returns more details for the information that is already presented. The operation is implemented with a drill down OLAP operator.
- Compare: The Compare operator contrasts a cube or cell with its peer cubes/cells. The operation is implemented with a drill across OLAP operator.
- Verify: The Verify operator checks if a pattern you observe happens also at a broader context.
- Abstract: The Abstract operator is responsible for a brief presentation of the requested information. The operation is implemented with a Roll-up OLAP operator.
- Explain: The Explain operator shows which part of the information makes a difference.
- FocusOn: The FocusOn operator constrains the scope of analysis. The operation is implemented with a Slice N Dice OLAP operator.
- Predict: The Predict operator forecasts future values.
- Suggest: The Suggest operator gives the user hints about what operation should perform next.

2.2.2 What is this model offering to data analysis?

[MaLV19] In contrast to the queries thus far which returned as a result a list of tuples, the intentional high-level queries return data stories as results. A data story is a more interactive way to present the results of the queries. Every time that a data story is created, it executes the high-level query, compares the result with similar previous results, analyzes them by highlighting the most important parts of them and summarizes the key highlights into a final summary. For that purpose, in order to retrieve and analyze at such a level, data mining models and engines are required. Also, for the presentation of the results, text generation models and graphical visualization models are quite important.

2.2.3 Intentional Analytics Model statements

The new statements leave some degrees of freedom to the optimizer to decide the best roll-up, algorithm, clustering, and every other parameter that the user has not predefined it. Each of these settings (or options) leads to a different plan and a different result of the model. The optimizer estimates the benefit of each plan in many terms (time, number of insights, etc.) and

picks the best plan. After that, this plan is translated to a sequence of operations and then is executed. The results of every operation with the addition of appropriate graphical models finally produce a data story, which is presented to the user as the final result.

2.2.4 Contributions

The contribution is a vision for a global optimization scheme for Interactive Data Analysis, where:

1. The user expresses in a high-level way an intentional statement without needed to explain and declare the data sources and the operations to apply over them.
2. This statement sketches a data story specification that is a series of actions such as data retrieval, data mining algorithm selection and visualization model selection.
3. An optimizer, which prunes the plans of execution with a cost model, so that we end up with the best plan for the user's needs.
4. The involved execution engines produce all the intermediate results and in the end, all of them are compiled into a data story that is shown to user.

2.2.5 Model Selection Optimization

In [MaLV19], the authors have two basic approaches for optimizing the model selection:

- A rule-based approach: It uses the semantics of the high-level statement and it is based on a set of predefined rules (which derive from the queries and applications of ML algorithms) picks the execution plan.
- A cost-based approach: From the given statement an algorithm generates several execution plans. A function calculates the cost of each execution plan based on the current optimal solution and keeps the best one.

The goal of interactive data exploration is to extract insights from data, so a system that extracts more insights in a given time frame is preferred. With the term "insights" we mean human-digestible pieces of interesting information about data.

2.2.6 Architecture of Intentional Analytics Model

Basically, an Intentional Analytics Model consists of the following six layers.

- a user layer: here the user expresses the statements in an intentional way and waits for the results (data stories).
- a story skeleton layer: here the transformation of the intentional statements to a data story skeleton takes place.
- an optimizer layer: here takes place the translation of the intentional statement into atomic actions picked from a catalog, which are automatically tune and parameterized.
- an execution layer: here happens the execution of queries, the appliance of (applies) models, the extraction of highlights, etc.
- a data layer: this layer holds databases, cubes, flat files, user profiles, user traces, etc.
- a storytelling layer: this layer is responsible for compiling all the intermediate results into meaningful data stories.

2.3 OLAP research directions

From the first introduction of the OLAP concept until now, there has been done a lot of research in this field at many directions. In [RoAb07] the authors felt the need to define a reference algebra for OLAP. They also defined a reference set of operators that could be used to develop design methodologies about OLAP operations. Moreover, in [Sara99] the author is implementing a "DIFF" operator which help the user to obtain in one step summarized reasons for changes observed at the aggregated level. This operator is based on a probability function. In addition, in [Sara00] the same author is implementing an algorithm about guiding the user in to data exploration. This is achieved by maintaining a profile for each user, tracking down all the queries that the user has run and consequently, all the parts of the cube with which they are already familiar. Based on this profile, the algorithm is able to suggest to the user the next steps for the data exploration. Furthermore, in [SaSa01] the authors are proposing a new operator called "RELAX". The operator is responsible for reporting in a single step a summary of all possible maximal generalizations along various roll-up paths of the problem case. The report arms an analyst with the exact extent and scope of the problem so that he can better apply his insight to infer on possible external reasons. Likewise, in [SaAM98] the authors are proposing a "discovery-driven" method of data exploration which guides the user to find any abnormal patterns or

anomalies in the data cube. Finally, in [GoGR14] the authors are introducing the "Shrink" operator in order to achieve a better balance between the data precision and data size. This operator is based on hierarchical clustering and can be applied to a cube resulting from a query to decrease its size while controlling the loss of precision.

Research has been also done in the way of the data presentation. In [GeVa13] and [GeVM15] the authors are introducing the data stories, a memorable way of data visualization summarizing findings and insights. In this implementation, the system provides to the user a text with the highlights of the results, an audio file containing these highlights and a PowerPoint presentation containing all the insights.

2.4 Summary

Lack of data analysts' knowledge was the main reason for a new approach to OLAP operations. In this new approach, an interactive way to present the results of queries is introduced, called data stories. Based on these data stories, the intentional analytics model is offering the contributions. A contribution is an optimization scheme where the user expresses an intentional statement in a high-level way and the query engine produces the results as a data story at the end.

With intentional analytics model the user (commonly a data analyst) is able to execute queries expressed in a high-level way, faster and more efficiently, without having any special SQL skill and get the results in the form of a data story, also expressed in a high-level way, easier for him to analyze and process them.

For this purpose, in Delian Cubes there is the SimpleQueryProcessorEngine which is a high-level query engine based on intentional analytics model. In this SimpleQueryProcessorEngine there are methods that handle SQL queries in a way we have discussed in Chapter 2.

CHAPTER 3

THE IMPLEMENTATION OF A CUBE-BASED QUERY ANSWERING ENGINE

3.1 The Delian Cubes tool

The main tool that is used for the purpose of this thesis is the Delian Cubes tool [VGK+18]. The Delian Cubes tool is a simple query engine that receives cube queries, processing them and returns the information on tsv files. It is written using Java programming language and a MySql database for holding the data that is processing. The main goal of this thesis is the implementation of the OLAP operations on this existing tool.

3.2 System architecture

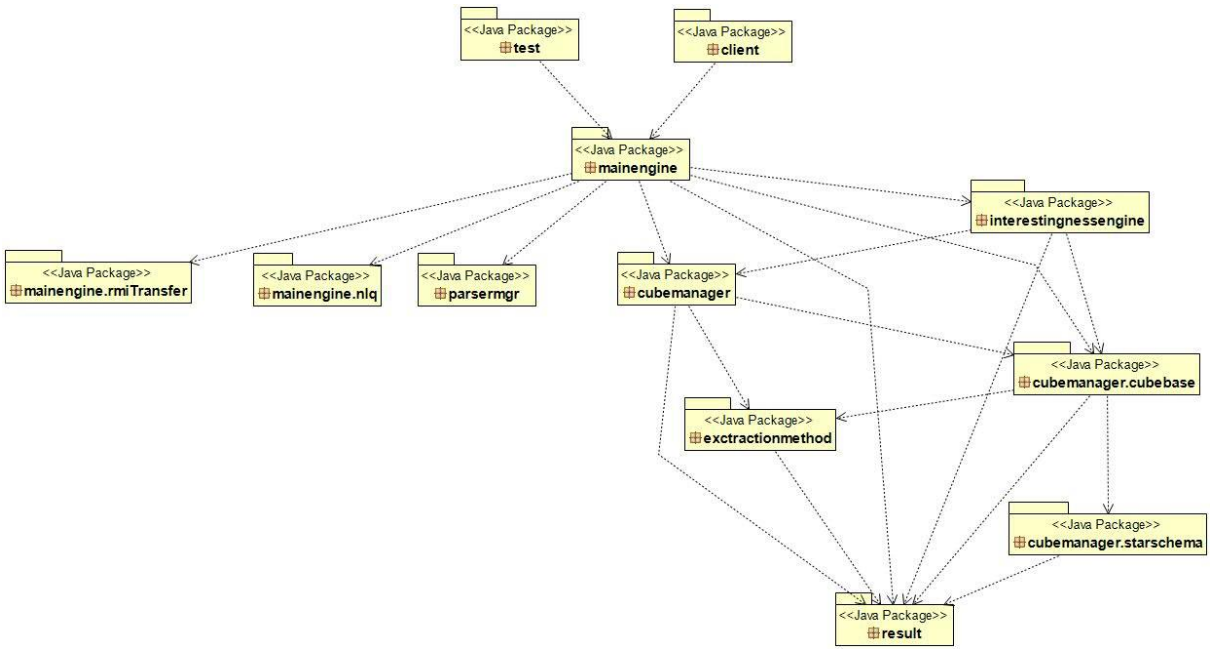

Figure 6: Delian's architecture

Delian's architecture contains all the packages that are shown in the picture above. Below we are going deeper to them and explaining the purpose and functionality of every package.

- *client*: This package contains the classes that are responsible for the creation and handling of the graphical user interface (GUI).
- *mainengine*: The most important package of the system, which is responsible for the initialization of the client and the server as instances and also the initialization of the connection between them.
- *cubemanager*: This package is responsible for managing Cubes. It contains two subpackages the *cubemanager.cubebase* and *cubemanager.starschema* that are containing some of the domain classes of Delian such as the Cube class. Moreover, it is responsible for translating the Delian Cube Queries to SQL Queries.
- *results*: This package is responsible for storing the results of the Cube Queries, aggregate them using many data algorithms and measurements and transform them to tsv files.

3.3 Adding Sessions & Query History at Delian

In this major refactoring, we have introduced the concept of a session. A session is an interactive and continuous interchange of information between client and server. The client now has the ability to run multiple queries in a single session and the server is tracking them down. With this extra feature we have laid the foundation for building the OLAP operations. For that reason, we have created two new classes: The Session class and the *QueryHistoryManager* class. The Session class is responsible for initiating a connection to a client, but we will go deeper into it later in this chapter. The *QueryHistoryManager* class is responsible for keeping up all the queries that the user runs in a specific session. The history gives the advantage to the user to retrieve and use any information of the previous queries. This will be very helpful to OLAP operations, as an operation may rely on a previous one.

The Session class, as we have already mentioned, is responsible for creating a connection between the client and the server. For that purpose, all the functionality about initializing a connection which was on the *SimpleQueryProcessorEngine*, has now been moved to the Session class. Moreover, we have changed the name of the *SimpleQueryProcessorEngine* to *SessionQueryProcessorEngine* as we are now working on a session-based design.

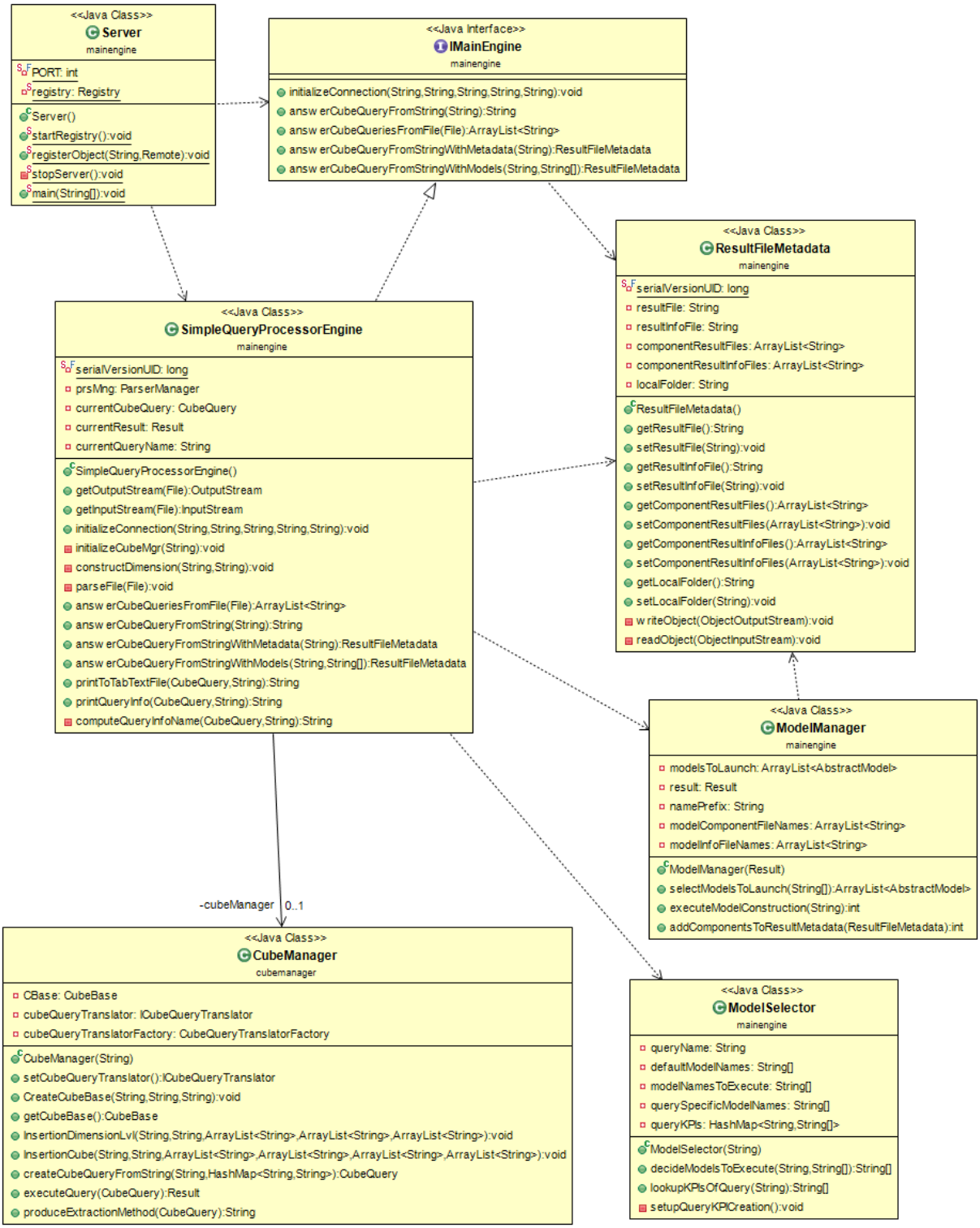

Figure 7: UML Diagram of key classes of the Delian before the refactoring.

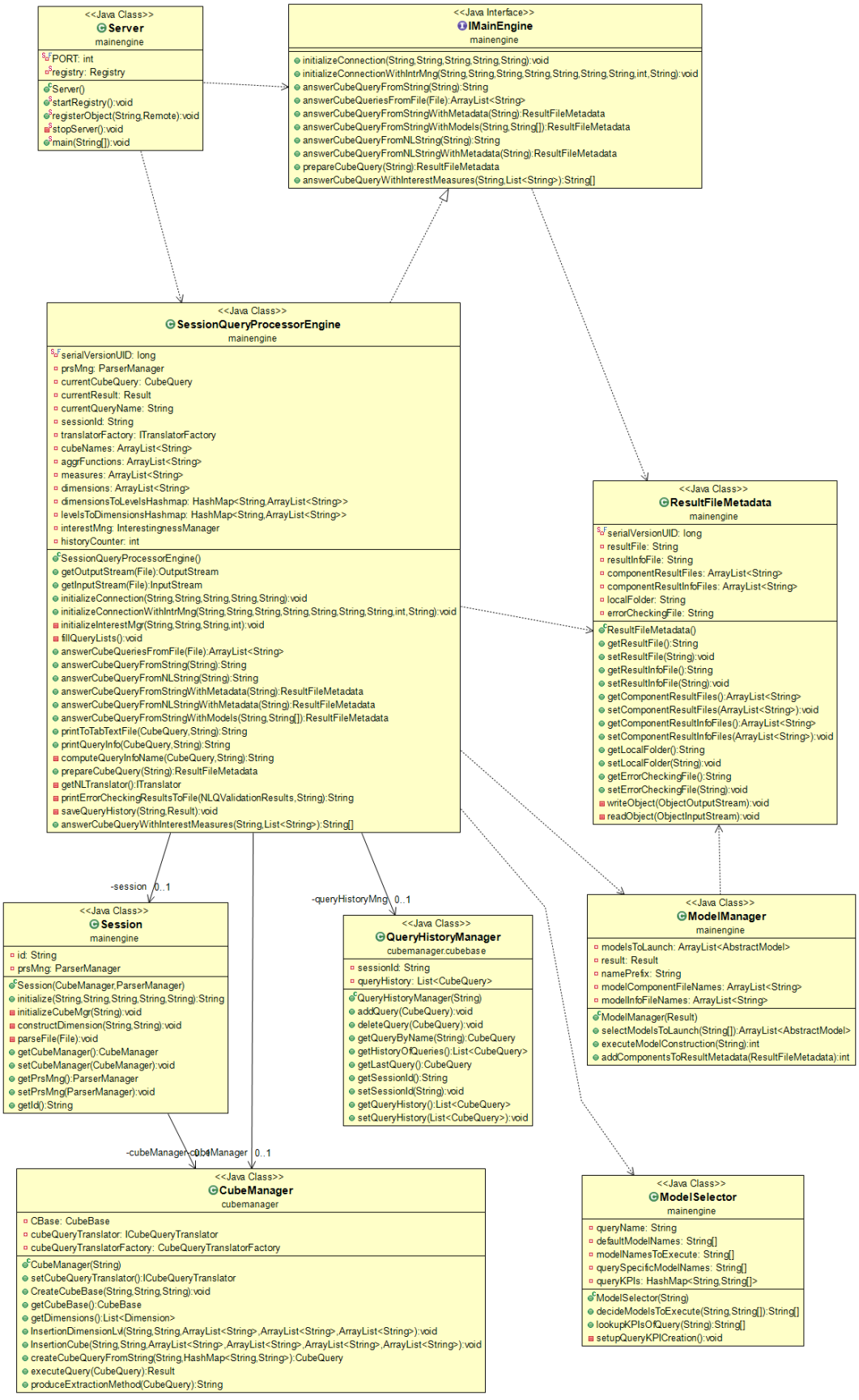

Figure 8: UML Diagram containing the key classes after refactoring.

As a consequence of the refactoring, the program flow has now changed. Until now, the server side was creating a *SimpleQueryProcessorEngine* (a.k.a. *SessionQueryProcessorEngine*) object which was responsible for instantiating objects and initializing all the required services. For this reason, the *SimpleQueryProcessorEngine* class had the following methods:

- *initializeCubeMng*(String *lookupFolder*) : this method was responsible for creating an *CubeManager* instance.
- *constructDimension*(String *inputlookup*, String *cubeName*) : this method was responsible for creating the dimensions of a CubeBase.
- *parseFile*(File *file*) : this method was used by *constructDimension*, for parsing the file which keeps the information of the dimensions of the *CubeBase*.

All the above methods are now in the Session class, as now they are part of each session. From now on, the *SessionQueryProcessorEngine* is initializing a Session object that is unique for each client and the Session class initializes the *CubeManager* as well as every other instance that follows.

We mentioned earlier that we have added a class that tracks all the queries for each session. As you may know, the domain class that is responsible for each query is the *CubeQuery* class. We also need a manager for operating on this class. For that purpose, we introduce the *QueryHistoryManager* class. Every time that the *SessionQueryProcessorEngine* initializes a Session object, it also initializes a *QueryHistoryManager* object and passes on the session id to it. This means that each session has a *QueryHistoryManager* attached to it. On this new class we have implement the following methods:

- *addQuery*(*CubeQuery cubeQuery*) : this method is responsible for inserting a query to the existing List of session queries.
- *deleteQuery*(*CubeQuery cubeQuery*) : this method is responsible for deleting a query from the List of session queries.
- *getQueryByName*(String *name*) : this method is responsible for retrieving information about a query, if the query with that name exists in the history of the session.
- *getHistoryOfQueries*() : this method returns a List of all the queries that have already been executed in that specific session.
- *getLastQuery*() : this method returns the last query that has been executed in that session. Returns null if the query List is empty.

- *getSessionId*() : this method returns the session id of the session that is corresponding to this *QueryHistoryManager* instance.

3.4 Creating the OLAP operations on Delian

Now, let's continue to the most important part of this thesis, which is the implementation of OLAP operations. On the first attempt of this project, we decided to implement the 4 basic OLAP operations which are Roll-up, Drill-down, Slice and Dice (as we have mentioned them before at chapter 2.1.2).

3.4.1 Roll-up implementation

For the roll-up operator we had to construct a method that will take as input a *CubeQuery* object (new or existing one using the *QueryHistoryManager*) and make an aggregation at one of Cube's dimensions. For that purpose, in the arguments of the roll-up operator we must define exactly our *CubeQuery* object alongside with the dimension and the exact level that we want to make the roll-up. After that, the operator checks if the dimension exists on this Cube and if the dimension actually exists, it checks if the level exists on this specific dimension. If everything is good so far and the dimension was not already a grouper, we are able to proceed to the roll-up. Otherwise, if the dimension is already a grouper for the current *CubeQuery*, then there is a last check for the level, to make sure that we are not making a roll-up to a lower or equal level.

After that, if all these checks succeed, we can proceed to the roll-up execution. For that purpose, the program constructs a new *CubeQuery* object as a copy from the object that the method had as input. Then sets at *GammaExpressions* the dimension and level that the method had as input and renames the *CubeQuery* in order for us to know what this new *CubeQuery* is about. In detail, the method sets the *CubeQuery's* name to "Roll-up" + previous *CubeQuery* name. Finally, the program calls the *answerCubeQueryFromStringWithMetadata* function, which is responsible for executing the query (using the new *CubeQuery* object), collecting, and saving the results.

Algorithm 1: Roll-up code implementation

```
@Override
```

```
public ResultFileMetadata rollUp(CubeQuery cubeQuery, String dimension, String level) throws RemoteEx-
ception {
        CubeQuery newCubeQuery = new CubeQuery(cubeQuery);
        boolean dimensionIsAlreadyAGrouper = false;
        if(!dimensions.contains(dimension)) {
                System.out.println("The dimension does not exist");
                ResultFileMetadata resMetadata = new ResultFileMetadata();
                resMetadata.setErrorCheckingFile("The dimension does not exist");
                return resMetadata;
        }
        if(!dimensionsToLevelsHashmap.get(dimension).contains(level)) {
                System.out.println("The level does not exist");
                ResultFileMetadata resMetadata = new ResultFileMetadata();
                resMetadata.setErrorCheckingFile("The level does not exist");
                return resMetadata;
        }
        for( String[] gammaExpression : newCubeQuery.getGammaExpressions() ) {
                if( gammaExpression[0].equals(dimension) ) {
                        dimensionIsAlreadyAGrouper = true;
                        if(    dimensionsToLevelsHashmap.get(gammaExpression[0]).indexOf(gammaExpres-
sion[1]) > dimensionsToLevelsHashmap.get(dimension).indexOf(level) ) {
                                System.out.println("You can't roll up to a lower or equal level");
                                        ResultFileMetadata   resMetadata   =   new   Result-
FileMetadata();
                                resMetadata.setErrorCheckingFile("You can't roll up to a lower or
equal level");
                                return resMetadata;
                        } else {
                                gammaExpression[1] = level;
                        }
                }
        }
        if(!dimensionIsAlreadyAGrouper) {
                ArrayList<String[]> list = new ArrayList<>();
                String[] gamma = { dimension, level };
                list.add(gamma);
                newCubeQuery.setGammaExpressions(list);
        }
        newCubeQuery.setName("Roll-up_" + cubeQuery.getName());
        return this.answerCubeQueryFromStringWithMetadata(newCubeQuery);
}
```

Typical Roll-up call in program:

```
ResultFileMetadata resMetadata = service.rollUp(cb, "account_dim", "lvl3");
```

3.4.2 Drill-down implementation

For the drill-down operation, we follow the same structure and logic as the roll-up. At first, we have to take as input for the drill-down function the *CubeQuery* and the targeted dimension and

level. At the next step, the program runs the checks about the existence of the dimension and about the existence of the level in this dimension (if the dimension actually exists). Also, if the dimension is already a grouper for this *CubeQuery*, a final check applied for the selected level, in order to prevent the program from running a drill-down to a higher or equal level.

The same steps are also followed at the drill-down execution. In the beginning, a new *CubeQuery* object is being constructed as a copy from the object that the method had as input. The name of this *CubeQuery* is changed from the method to "Drill-down" + name of the existing *CubeQuery* in order for us to know what this query was about. The method sets at *GammaExpressions* the dimension and the level that it got as inputs from the user and finally the program calls the *answerCubeQueryFromStringWithMetadata* function in order to execute the query (using the new *CubeQuery* object), collect, and save the results.

Algorithm 2: Drill-down code implementation

```
public ResultFileMetadata drillDown(CubeQuery cubeQuery, String dimension, String level) throws Re-
moteException {
        CubeQuery newCubeQuery = new CubeQuery(cubeQuery);
        boolean dimensionIsAlreadyAGrouper = false;
        if(!dimensions.contains(dimension)) {
                System.out.println("The dimension does not exist");
                ResultFileMetadata resMetadata = new ResultFileMetadata();
                resMetadata.setErrorCheckingFile("The dimension does not exist");
                return resMetadata;
        }
        if(!dimensionsToLevelsHashmap.get(dimension).contains(level)) {
                System.out.println("The level does not exist");
                ResultFileMetadata resMetadata = new ResultFileMetadata();
                resMetadata.setErrorCheckingFile("The level does not exist");
                return resMetadata;
        }
        for( String[] gammaExpression : newCubeQuery.getGammaExpressions() ) {
                if( gammaExpression[0].equals(dimension) ) {
                        dimensionIsAlreadyAGrouper = true;
                        if(    dimensionsToLevelsHashmap.get(gammaExpression[0]).indexOf(gammaExpres-
sion[1]) < dimensionsToLevelsHashmap.get(dimension).indexOf(level) ) {
                                System.out.println("You can't drill down to a higher or equal level");
                                ResultFileMetadata resMetadata = new ResultFileMetadata();
                                resMetadata.setErrorCheckingFile("You can't drill down to an upper or
equal level");
                                return resMetadata;
                        } else {
                                gammaExpression[1] = level;
                        }
                }
        }
```

```
        if(!dimensionIsAlreadyAGrouper) {
                ArrayList<String[]> list = new ArrayList<>();
                String[] gamma = { dimension, level };
                list.add(gamma);
                newCubeQuery.setGammaExpressions(list);
        }
        newCubeQuery.setName("Drill-down_" + cubeQuery.getName());
        return this.answerCubeQueryFromStringWithMetadata(newCubeQuery);
}
```

Typical Drill-down call in program:

```
ResultFileMetadata resMetadata = service.drillDown(cb, "account_dim", "lvl1");
```

3.4.3 Slice implementation

For the slice functionality, we need more specific information. For that purpose, the declaration of the slice method has 5 arguments. As you would expect, the first argument is the *CubeQuery*, based on which the method will execute the slice process. Next, we have the dimension and the level arguments as before, followed by the last two arguments which are the comparison operator and the value. The comparison operator can be anything that is allowed by the SQL as a comparison operator and the value is the searching value in the current dimension and level of the Cube.

In the beginning, this method also checks if the selected dimension exists in the inserted Cube and after that, if it actually exists, the method checks if the selected level exists in this specific dimension. If everything is good so far, then the method creates a new *CubeQuery* object as a copy from the one that the method had as input and changes its name to "Slice" + its current name. Last, the method adds as *SigmaExpressions* of the new *CubeQuery* the dimension, the level, the operator and the value and then proceeds to the execution of the query (using the *answerCubeQueryFromStringWithMetadata* method). Finally, a *ResultFileMetadata* object is produced, which includes the result of this query.

Algorithm 3: Slice code implementation

```
@Override
public ResultFileMetadata slice(CubeQuery cubeQuery, String dimension, String level, String opera-
tor, String value)
                        throws RemoteException {
        CubeQuery newCubeQuery = new CubeQuery(cubeQuery);
        if(!dimensions.contains(dimension)) {
                System.out.println("The dimension does not exist");
```

```
                ResultFileMetadata resMetadata = new ResultFileMetadata();
                resMetadata.setErrorCheckingFile("The dimension does not exist");
                return resMetadata;
        }
        if(!dimensionsToLevelsHashmap.get(dimension).contains(level)) {
                System.out.println("The level does not exist");
                ResultFileMetadata resMetadata = new ResultFileMetadata();
                resMetadata.setErrorCheckingFile("The level does not exist");
                return resMetadata;
        }
        newCubeQuery.setName("Slice_" + cubeQuery.getName());
        newCubeQuery.addSigmaExpression(dimension + "."+ level, operator, value);
        return this.answerCubeQueryFromStringWithMetadata(newCubeQuery);
}
```

Typical Slice call in program:

```
ResultFileMetadata resMetadata = service.slice(queryHistory.getQueryBy-
Name("LoanQuery11_S1_CG-Prtl"), "account_dim", "lvl1", "=", "'Sumperk'");
```

3.4.4 Dice implementation

The last one of the four basic OLAP operators is the Dice. The implementation of Dice has been based on the implementation of Slice, but with one major difference: on Dice we have multiple *SigmaExrpessions* executing at the same query. So, as you can imagine, in the Dice method declaration we also have 5 arguments, starting again with the existing *CubeQuery* that we want to use as input. Now, because of the multiple *SigmaExrpessions*, the rest of the arguments has been changed from Strings (which they are on Slice), to List of Strings in order to carry all this information. So now we have a List of dimensions, a List of levels, a List of operators and a List of values. The idea behind this structure is that the first *SigmaExpression* of the Dice is constructed using the first element of each of these four Lists. The same procedure is used for the construction of the following *SigmaExrpessions*.

At first, this method also checks about the existence of each dimension in the Cube, followed by a check of the existence of the level in each specific dimension (if that dimension exists). In addition, if all checks are passed, the method creates a new *CubeQuery* as a copy from the inputted one and changes its name to "Dice" + old *CubeQuery* name. Then, the method adds to the *SigmaExrpessions* the dimensions, the levels, the operators, and the values using the *addMultipleSigmaExpressions* method which have been constructed in the *CubeQuery* class for that purpose. The *addMultipleSigmaExpressions* method takes as input the four Lists (dimensions, levels, operators, values), constructs each *SigmaExpression* separately and then adds

them all to the current *SigmaExrpessions* of the new *CubeQuery*. Finally, the dice method calls the *answerCubeQueryFromStringWithMetadata* method, which executes the query and returns the results in a *ResultFileMetadata* object.

Algorithm 4: Dice code implementaion

```
@Override
public ResultFileMetadata dice(CubeQuery cubeQuery, List<String> dims, List<String> levels,
List<String> operators, List<String> values) throws RemoteException {
        CubeQuery newCubeQuery = new CubeQuery(cubeQuery);
        for( int i = 0; i < dims.size(); i++ ) {
                if(!dimensions.contains(dims.get(i))) {
                        System.out.println("The dimension does not exist");
                        ResultFileMetadata resMetadata = new ResultFileMetadata();
                        resMetadata.setErrorCheckingFile("The dimension does not exist");
                        return resMetadata;
                }
                if(!dimensionsToLevelsHashmap.get(dims.get(i)).contains(levels.get(i))) {
                        System.out.println("The level does not exist");
                        ResultFileMetadata resMetadata = new ResultFileMetadata();
                        resMetadata.setErrorCheckingFile("The level does not exist");
                        return resMetadata;
                }
        }
        newCubeQuery.setName("Dice_" + cubeQuery.getName());
        newCubeQuery.addMultipleSigmaExpressions(dims, levels, operators, values);
        ResultFileMetadata res = this.answerCubeQueryFromStringWithMetadata(newCubeQuery);
        return res;
}
```

Typical Dice call in program:

```
List<String> dimensions = Collections.unmodifiableList(Arrays.asList("account_dim",
"date_dim"));
List<String> levels = Collections.unmodifiableList(Arrays.asList("lvl1", "lvl3"));
List<String> operators = Collections.unmodifiableList(Arrays.asList("=", "="));
List<String> values = Collections.unmodifiableList(Arrays.asList("'Sumperk'", "1998"));

ResultFileMetadata service.dice(queryHistory.getQueryByName("LoanQuery11_S1_CG-Prtl"), di-
mensions, levels, operators, values);
```

CHAPTER 4

EXPERIMENTS

4.1 Experimental Setup

In order to measure the performance of the four basic OLAP methods that we have previously implement in Delian, there was a need of experimental data alongside with a client. For that reason, we have constructed two database schemas (Loan and Adult) and generated data to populate the databases.

4.1.1 Pkdd99_star Schema

The pkdd99_star Database consists of six tables: account, date, loan, order, payment_reason, status. These tables are divided in two categories, the Dimension tables and the Fact tables. A fact table holds the data to be analyzed, and a dimension table stores data about the ways in which the data in the fact table can be analyzed. The Dimension tables are the tables that they will be used from Delian as dimensions of the OLAP cube. These tables are the account, date, payment_reason and status tables. On the other side, the loan and order tables are Fact tables. Thus, the Loan cube has four dimensions.

4.1.2 Adult_no_dublic Schema

The adult_no_dublic Database consists of nine tables: adult, age, education, gender, marital_status, native_country, occupation, race, work_class. In this database, the Fact table is the adult, and all the other tables are Dimension tables. Thus, the Adult cube has eight dimensions.

4.2 Data generators

In order to measure the performance of the OLAP operations we had to populate the databases with sample data. For that purpose, we have used the *LoanArtificialGenerator* class for generating data for the loan Fact table and the *AdultArtificialGenerator* class for generating data for the adult Fact table. These generators take as input the expected number of rows that we need in the database table and using random generators populate the data, taking values from a pool of permitted values.

4.3 Experiments

In the context of this thesis, we have design and executed two experiments for the evaluation of the OLAP operations.

4.3.1 Assessing performance with respect to data size

The first experiment has to deal with the behavior of the OLAP methods on databases with different scale. For that reason, we used the *LoanArtificialData* class and produced three loan tables: one with 100.000 rows, one with 1.000.000 and one with 10.000.000 rows. Then we created three different copies of the pkdd99_star database and in each of three we loaded one of the previous three loan tables. After that, we picked a Query and ran a client which had a roll-up, a drill-down, a slice and a dice operated in the Query that we had previously picked. While the client was running, we were measuring the time of each OLAP operator. We did this procedure five times for each database copy and we calculated the average time of these five executions for each operator.

Table 1: Analytical results of Roll-up execution time (in seconds) for all three database sizes.

Roll-up						
	1	2	3	4	5	Average
100k	0,1339106 sec	0,1205324 sec	0,1223318 sec	0,1348025 sec	0,1473575 sec	0,13178696 sec
1M	1,0248734 sec	1,2465000 sec	1,1561306 sec	1,5660719 sec	1,2075714 sec	1,2402295 sec
10M	14,8757270 sec	15,2521889 sec	14,3111288 sec	15,3098265 sec	14,17522 sec	14,7848191 sec

Table 2: Analytical results of Drill-down execution time (in seconds) for all three database sizes.

Drill-down						
	1	2	3	4	5	Average
100k	0,176806 sec	0,164903 sec	0,1641364 sec	0,2303439 sec	0,3140372 sec	0,2100453 sec
1M	1,4564023 sec	1,5087474 sec	1,2730509 sec	1,3658397 sec	1,4195323 sec	1,40471452 sec
10M	15,3441143 sec	15,0604922 sec	16,4630303 sec	15,5705939 sec	15,4517125 sec	15,5779886 sec

Table 3: Analytical results of Slice execution time (in seconds) for all three database sizes.

Slice						
	1	2	3	4	5	Average
100k	0,0149513 sec	0,0182567 sec	0,0465964 sec	0,0368517 sec	0,0709311 sec	0,03751744 sec
1M	0,0899326 sec	0,0917714 sec	0,0779583 sec	0,0797755 sec	0,1048079 sec	0,08884914 sec
10M	1,0193446 sec	0,8333888 sec	1,2094172 sec	1,0189816 sec	1,30285886 sec	1,0767982 sec

Table 4: Analytical results of Dice execution time (in seconds) for all three database sizes.

Dice						
	1	2	3	4	5	Average
100k	0,1356711 sec	0,1010337 sec	0,2892847 sec	0,1373736 sec	0,1494659 sec	0,1625658 sec
1M	0,8020527 sec	0,9665567 sec	0,9424599 sec	0,8961333 sec	0,8553486 sec	0,89251024 sec
10M	10,2320879 sec	11,01069 sec	11,1335464 sec	10,3691458 sec	10,7070856 sec	10,69051114 sec

Table 5: Average execution time (in seconds) for all operators for all three database sizes.

	Roll-up	Drill-down	Slice	Dice
100k	0,13178696 sec	0,2100453 sec	0,03751744 sec	0,1625658 sec
1M	1,2402295 sec	1,40471452 sec	0,08884914 sec	0,89251024 sec
10M	14,7848191 sec	15,5779886 sec	1,0767982 sec	10,69051114 sec

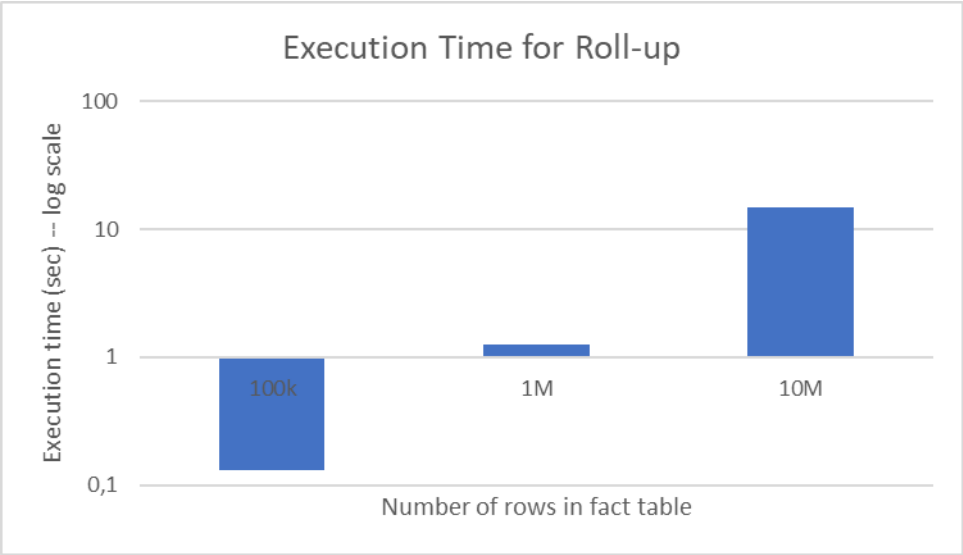

Figure 9: Execution time for Roll-up for all three database sizes.

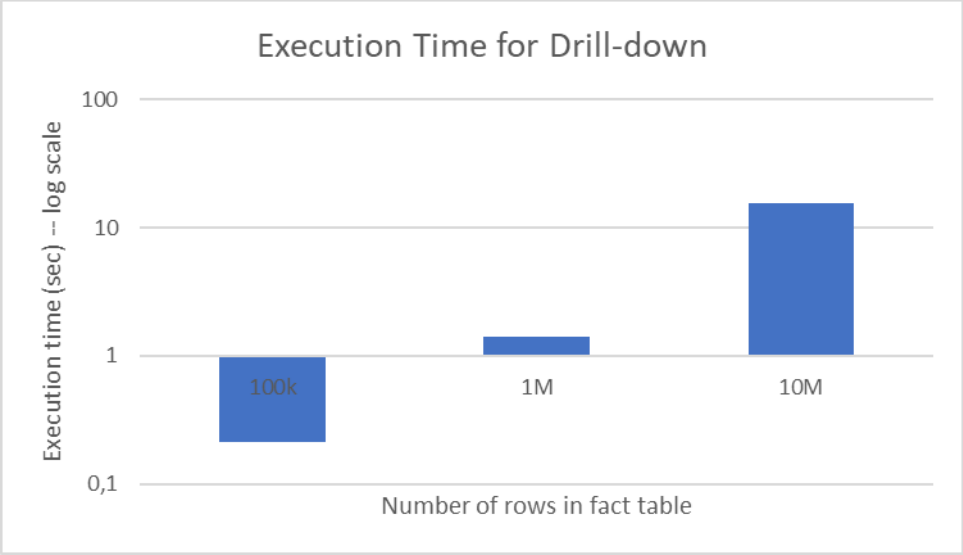

Figure 10: Execution time for Drill-down for all three database sizes.

Figure 11: Execution time for Slice for all three database sizes.

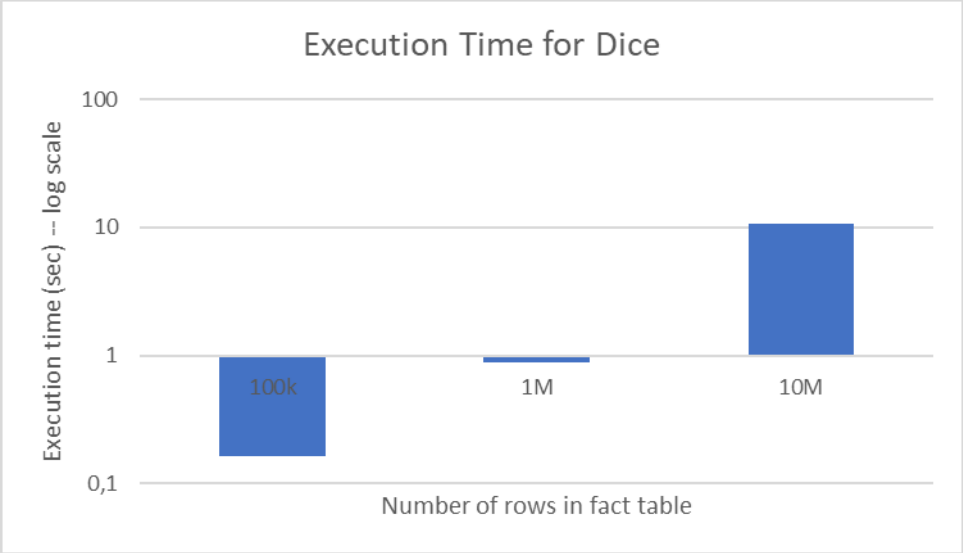

Figure 12: Execution time for Dice for all three database sizes.

In Figures 9, 10, 11, 12 we present the results of the execution time for all four OLAP operators. We measure execution time against the size of the fact tables -- thus, the horizontal axis depicts the size of the fact table in number of tuples and the vertical axis the execution time (in log scale) in seconds. The number reported is the average of 5 executions.

We observe that the execution time is linearly related to the size of the basic cube. Also, we observe that the fastest OLAP operator is always the Slice, and the slowest OLAP operator

is always the Drill-down. Moreover, in the majority of executions, between the Roll-up and Dice, the Dice operator seems to be faster.

4.3.2 Assessing performance with respect to the complexity of hierarchies

The second experiment has to deal with the behavior of OLAP operators in databases with the same size but different number of dimension tables (Cube dimensions). For that purpose, we created a copy of the Adult_no_dublic database and then we generated an adult table with 1.000.000 rows using the *AdultArtificialGenerator* class. After that we picked a similar Query and ran a similar client which had a roll-up, a drill-down, a slice and a dice operated in the Query that we had previously picked. While the client was running, we were measuring the time of each OLAP operator. We did this procedure five times for this database copy and we calculated the average time of these five executions for each operator. Finally, we compared these results with the results that we produced earlier running the client and measuring OLAP operators time in pkdd99_star database with the same size (1.000.000 rows).

There is a significant difference between the Data Cubes that these two databases are producing. The Loan cube (from pkdd99_star database) has four dimensions (account_dim, date_dim, payment_reason_dim, status_dim), but on the other side the Adult cube has double the number of dimensions (age_dim, education_dim, gender_dim, marital_status_dim, native_country_dim, occupation_dim, race_dim, work_class_dim).

Bellow you can see the two Data Cubes and their dimensions, You can find more details about the hierarchy of every dimension table in the Appendix A (section A1 for Loan Cube dimensions and section A2 for Adult Cube dimensions) of this thesis report.

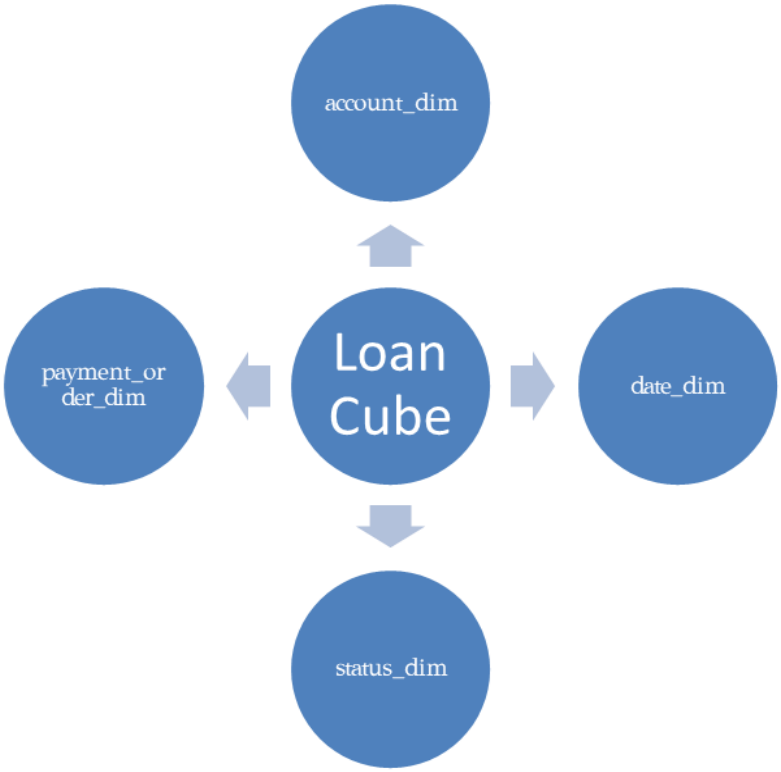

Figure 13: Loan Cube and its dimensions.

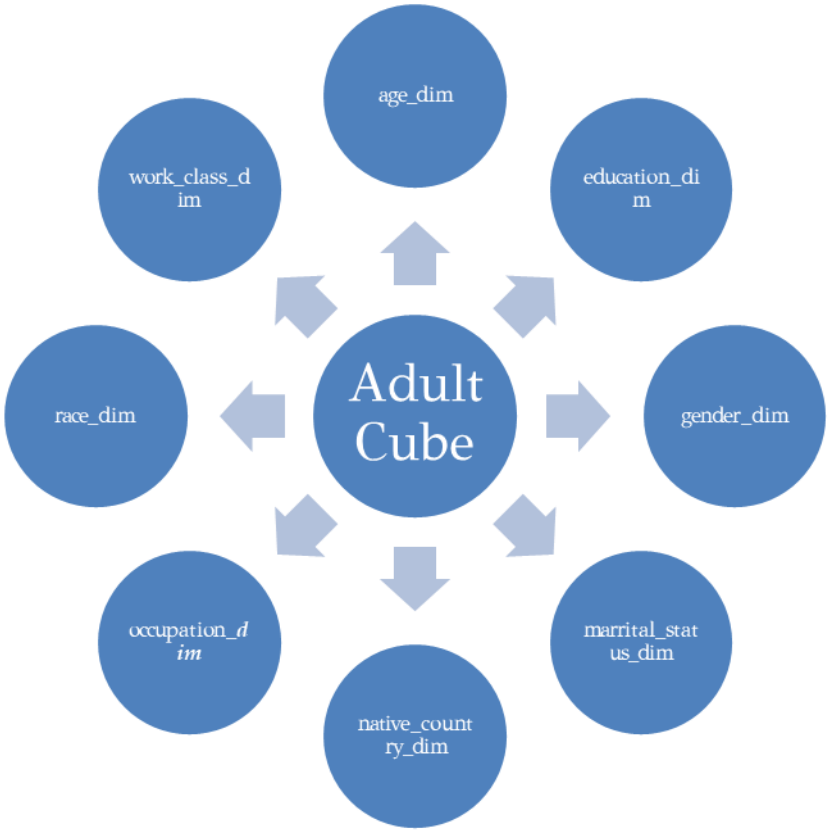

Figure 14: Adult Cube and its dimensions.

Table 6: Analytical results of Roll-up execution time (in seconds) for both loan and adult tables with 1.000.000 rows.

Roll-up						
	1	2	3	4	5	Average
loan 1M	1,0248734 sec	1,2465000 sec	1,1561306 sec	1,5660719 sec	1,2075714 sec	1,2402295 sec
adult 1M	1,7880315 sec	2,1496476 sec	2,1079581 sec	2,4611572 sec	2,0831184 sec	2,1179826 sec

Table 7: Analytical results of Drill-down execution time (in seconds) for both loan and adult tables with 1.000.000 rows.

Drill-down						
	1	2	3	4	5	Average
loan 1M	1,4564023 sec	1,5087474 sec	1,2730509 sec	1,3658397 sec	1,4195323 sec	1,40471452 sec
adult 1M	2,928261301 sec	2,3082004 sec	2,4624654 sec	1,902384699 sec	1,7708388 sec	2,27443012 sec

Table 8: Analytical results of Slice execution time (in seconds) for both loan and adult tables with 1.000.000 rows.

Slice						
	1	2	3	4	5	Average
loan 1M	0,0899326 sec	0,0917714 sec	0,0779583 sec	0,0797755 sec	0,1048079 sec	0,08884914 sec
adult 1M	2,726649 sec	1,7058619 sec	1,732878301 sec	2,1737981 sec	1,9158621 sec	2,05100988 sec

Table 9: Analytical results of Dice execution time (in seconds) for both loan and adult tables with 1.000.000 rows.

Dice						
	1	2	3	4	5	Average
loan 1M	0,8020527 sec	0,9665567 sec	0,9424599 sec	0,8961333 sec	0,8553486 sec	0,89251024 sec
adult 1M	2,3482254 sec	1,804313799 sec	2,902728999 sec	2,2538707 sec	1,8526353 sec	2,23235484 sec

Table 10: Average execution time (in seconds) for all operators for both loan and adult tables with 1.000.000 rows.

	Roll-up	Drill-down	Slice	Dice
loan 1M	1,2402295 sec	1,40471452 sec	0,08884914 sec	0,89251024 sec
adult 1M	2,1179826 sec	2,27443012 sec	2,05100988 sec	2,23235484 sec

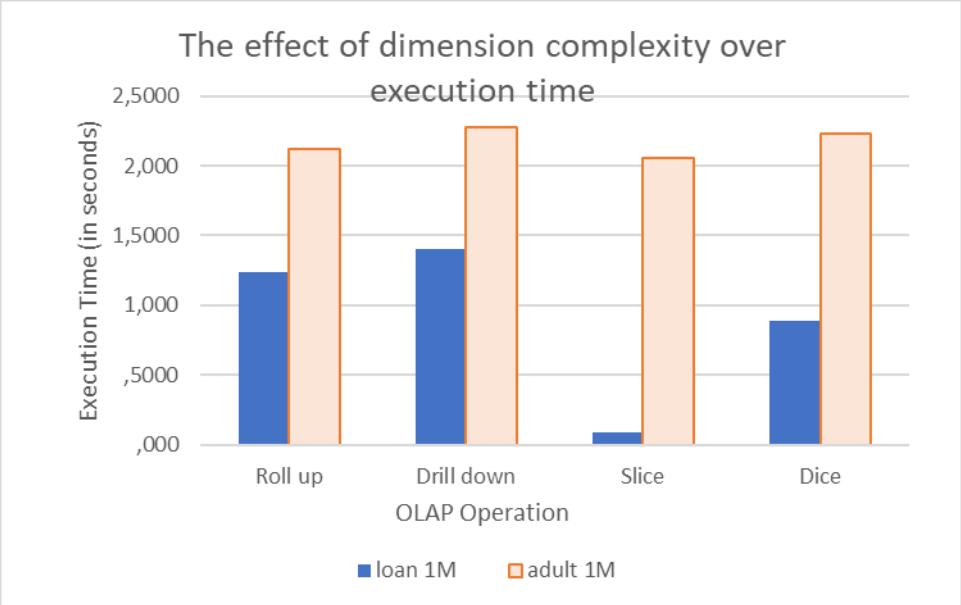

Figure 15: The effect of dimension complexity over execution time.

In Figure 13 we present the results of the execution time for all four OLAP operators. We measure execution time against the dimension complexity of the OLAP cubes -- thus, the horizontal axis depicts the OLAP operations and the vertical axis the execution time in seconds. The number reported is the average of 5 executions.

We observe that the execution time in loan Cube (which has four dimensions) is in the worst-case scenario almost 50% faster than the execution time in adult Cube (which has eight dimensions).

CHAPTER 5

CONCLUSION – FUTURE WORK

5.1 Conclusion

The main purpose of this thesis was to implement the basic OLAP operations such as Roll-up, Drill-down, Slice and Dice in the Delian Cubes tool [VGK+18]. Also, we introduced the concept of a session, a continuous interchange of information between client and server in the Delian tool, alongside with a Query History Manager, responsible for keeping up all the queries that the user runs in a specific session.

5.2 Future work

There is a lot of space left for future work in many directions. One possible future work can be the enrichment of the OLAP operators in the Delian Cubes tool. This can be succeeded by implementing some of the operators of the Section 2.3 such as Pivot, Diff, Relax, Predict, etc. Another possible future improvement can be the evaluation of the OLAP operations on Delian Cubes tool in contrast to any other Data Cube based tool. For this kind of evaluation there has to be research for tools that are similar to the Delian Cubes in order to perform a series of experiments about the performance of the OLAP operators (such as the experiments of Chapter 4). Finally, the graphical representation of the OLAP operations and Data Cube results in the Delian Cubes tool could also be a possible future work. This work could contain the construction of a series of buttons in the GUI of Delian Cubes about every OLAP operator that has already been implemented, but also the construction of a visual representation of the results that were produced after a query execution.

REFERENCES

[Tayl20] D. Taylor, "What is OLAP? Cube, Analytical Operations in Data Warehouse," 2020. https://www.guru99.com/online-analytical-processing.html (accessed Feb. 26, 2022).

[VaMa18] P. Vassiliadis and P. Marcel, "The road to highlights is paved with good intentions: envisioning a paradigm shift in OLAP modeling," 2018.

[MaLV19] P. Marcel, N. Labroche, and P. Vassiliadis, "Towards a benefit-based optimizer for Interactive Data Analysis," 2019.

[RoAb07] O. Romero and A. Abelló, "On the Need of a Reference Algebra for OLAP," 2007.

[Sara99] S. Sarawagi, "Explaining differences in multidimensional aggregates," 1999.

[Sara00] S. Sarawagi, "User-adaptive exploration of multidimensional data," 2000.

[SaSa01] G. Sathe and S. Sarawagi, "Intelligent Rollups in Multidimensional OLAP Data," 2001.

[SaAM98] S. Sarawagi, R. Agrawal, and N. Megiddo, "Discovery-driven Exploration of OLAP Data Cubes*," 1998.

[GoGR14] M. Golfarelli, S. Graziani, and S. Rizzi, "Shrink: An OLAP operation for balancing precision and size of pivot tables," Data and Knowledge Engineering, vol. 93, pp. 19–41, Sep. 2014, doi: 10.1016/j.datak.2014.07.004.

[GeVa13] D. Gkesoulis and P. Vassiliadis, CineCubes: Cubes As Movie Stars with Little Effort. 2013.

[GeVM15] D. Gkesoulis, P. Vassiliadis, and P. Manousis, "CineCubes: Aiding data workers gain insights from OLAP queries," Information Systems, vol. 53, pp. 60–86, Oct. 2015, doi: 10.1016/j.is.2014.12.006.

[VGK+18] P. Vassiliadis, D. Gkitsakis, S. Kaloudis, E. Mouselli, and S. Zogos, "DelianCubeEngine," 2018. https://github.com/DAINTINESS-Group/DelianCubeEngine (accessed Feb. 26, 2022).

APPENDIX A

DIMENSION TABLES

A.1 Explanation of Loan Cube dimension tables

As we had mentioned earlier, the pkdd99_star database consists of six tables. Two of them (loan and order) are fact tables and the other four tables (account, date, payment_reason and status) are dimension tables.

In this appendix we will present you in detail each dimension table so you can have a deeper look into their level hierarchy.

Account table:

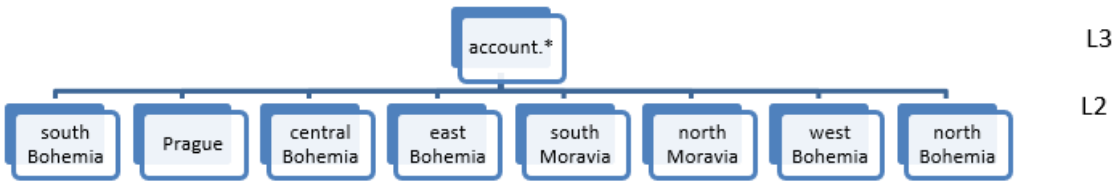

Figure 16: Account dimension table and its levels.

Date table:

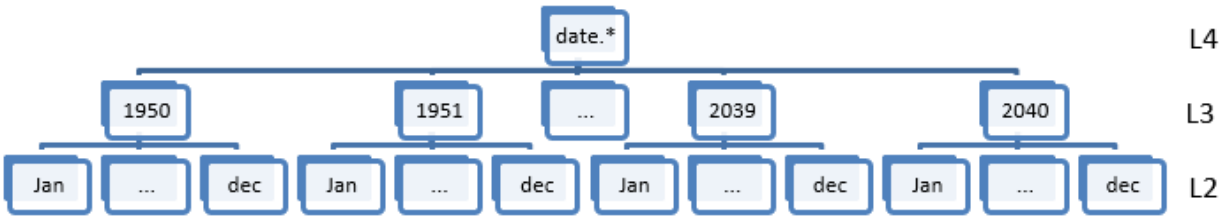

Figure 17: Date dimension table and its levels.

Payment_reason table:

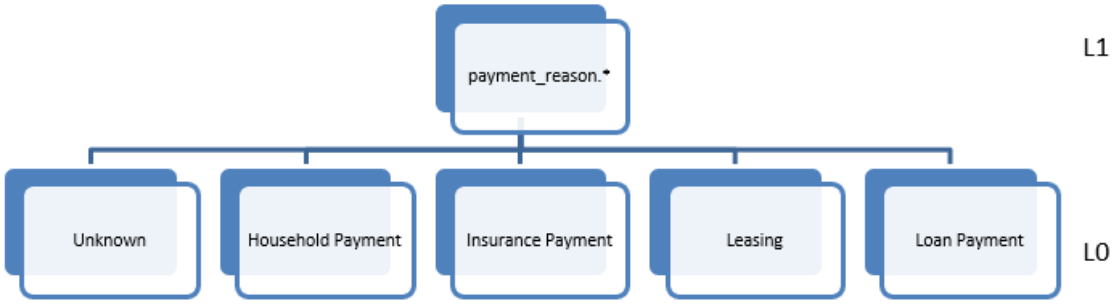

Figure 18: Payment_reason dimension table and its levels.

Status table:

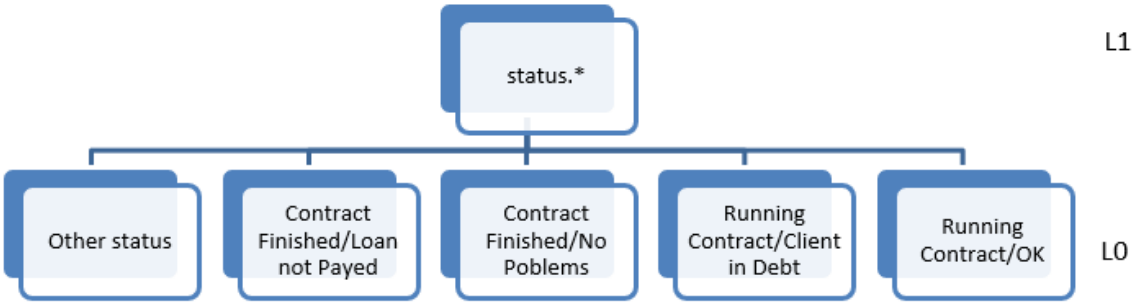

Figure 19: Status dimension table and its levels.

A.2 Explanation of Adult Cube dimension tables

As we had mentioned earlier, the adult_no_dublic database consists of nine tables. One of them (adult) is a fact table and the rest of them (age, education, gender, marital_status, native_country, occupation, race, work_class) are dimension tables.

In this appendix we will present you in detail each dimension table so you can have a deeper look into their level hierarchy.

Age table:

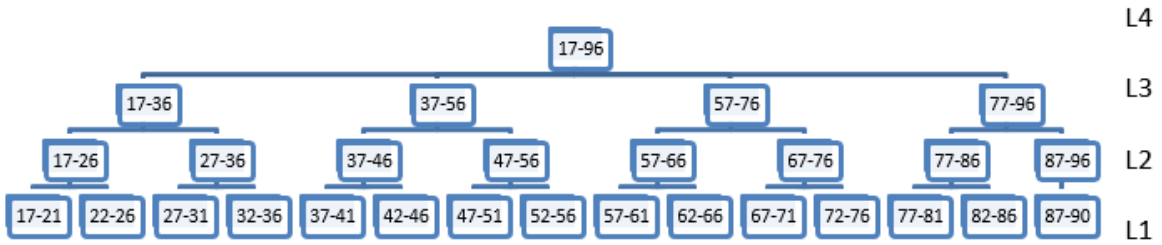

Figure 20: Age dimension table and its levels.

Education table:

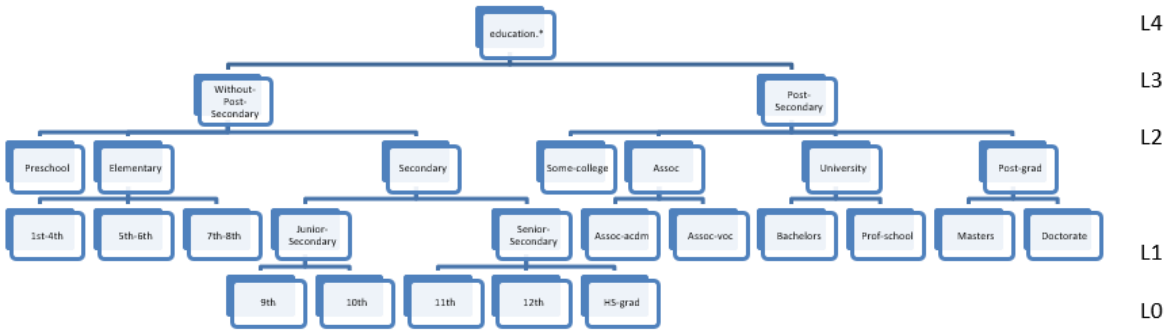

Figure 21: Education dimension table and its levels.

Gender table:

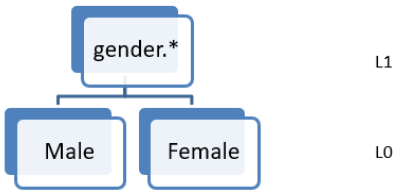

Figure 22: Gender dimension table and its levels.

Marital_status table:

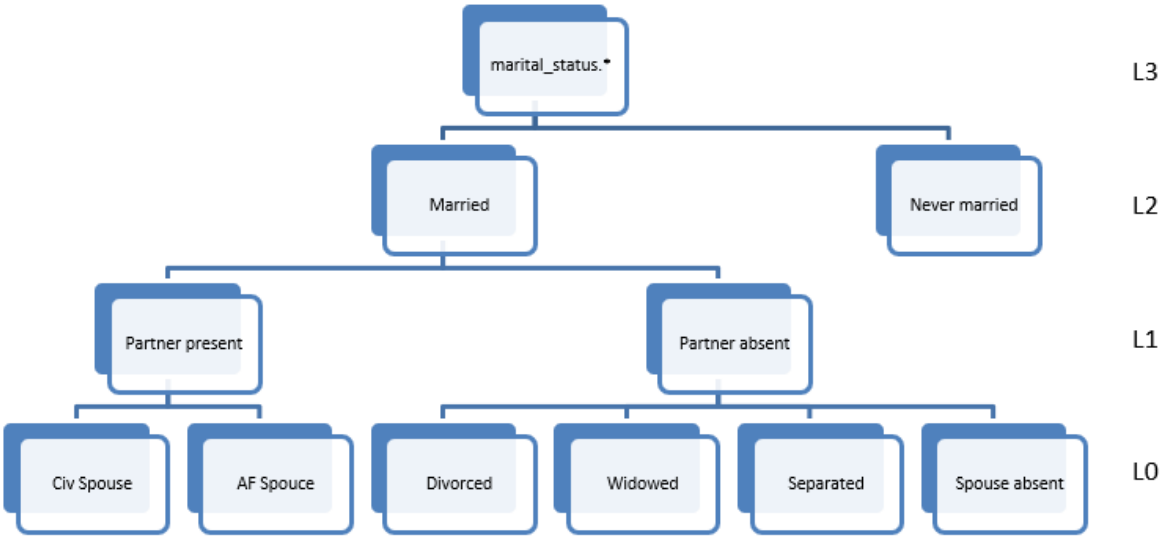

Figure 23: Marital_status dimension table and its levels.

Native_country table:

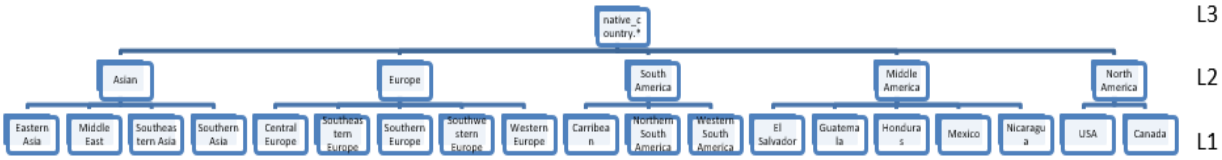

Figure 24: Native_country dimension table and its levels.

Occupation table:

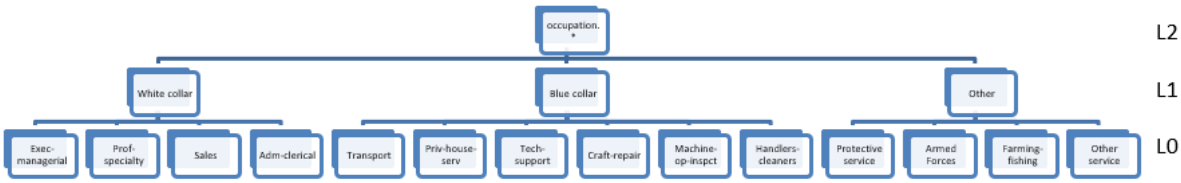

Figure 25: Occupation dimension table and its levels.

Race table:

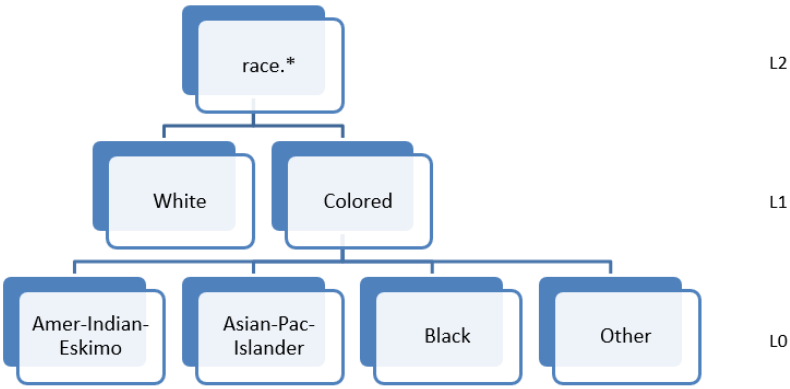

Figure 26: Race dimension table and its levels.

Work_class table:

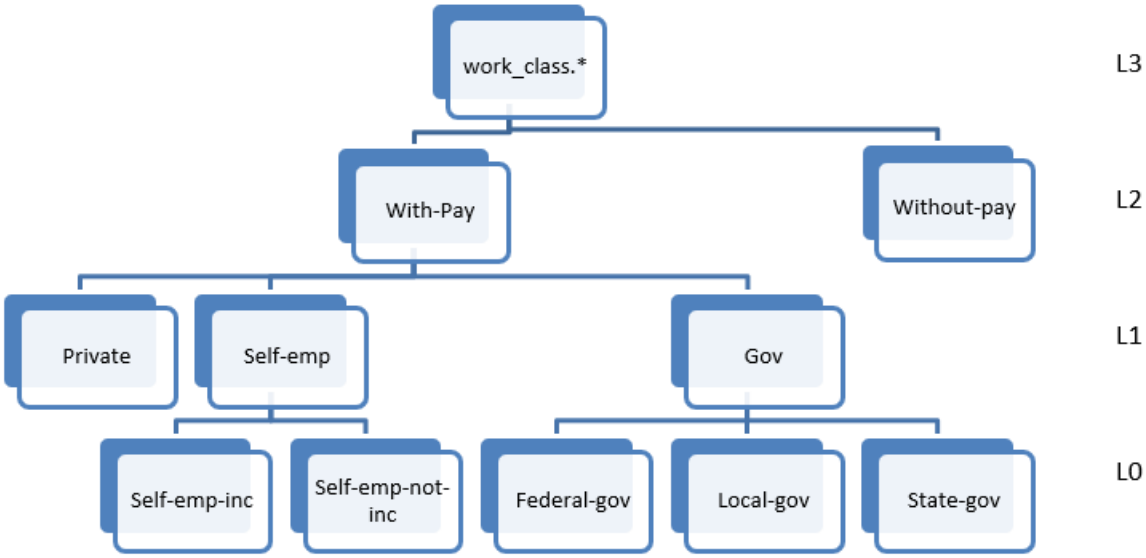

Figure 27: Work_class dimension table and its levels.

SHORT BIOGRAPHICAL SKETCH

Sotirios Zogos was born in Thessaloniki, Greece, in 1997. In 2015 he enrolled in the undergraduate program of Mathematics of the University of Ioannina and earned his Bachelor's Degree in 2019. In 2019 he enrolled in the Graduate Program of the department of Computer Science and Engineering of University of Ioannina, pursuing a Master's Degree entitled "Data and Computer Systems Engineering". Alongside with his Master studies, in 2019 he started working as a Junior Software Engineer in Niki Digital Engineering company here in Ioannina.