Query processing for cube queries via query usability

A Thesis

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

by

Iliana Filippou

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
IN DATA AND COMPUTER SYSTEMS ENGINEERING
WITH SPECIALIZATION
IN DATA SCIENCE AND ENGINEERING

University of Ioannina

June 2026

Examining Committee:

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

CONTENTS

CONTENTS i

LIST OF FIGURES iv

LIST OF TABLES vi

ABSTRACT vii

ΕΚΤΕΤΑΜΈΝΗ ΠΕΡΊΛΗΨΗ ix

CHAPTER 1 Introduction 1

1.1 Goals ... 1

1.2 Structure of the Thesis ... 2

CHAPTER 2 Related Work 5

2.1. Query Containment and Usability ... 5

2.1.1. Query Containment ... 5

2.1.2. View Usability ... 6

2.1.3. Query Rewriting ... 8

2.1.4 Limitations & Summary ... 9

CHAPTER 3 Preliminaries 10

3.1. Relational Database ... 10

3.2. Multidimensional Database Model ... 10

3.3. Business Intelligence (BI) ... 11

3.4. Data Warehouse ... 11

3.5. OLAP (On-Line Analytical Processing) ... 12

3.6. OLAP Cubes ... 12

3.7. Hierarchies and Levels ... 13

3.8. OLAP Operations ... 13

3.8.1. Roll-up ... 14

3.8.2. Drill-down ... 14

3.8.3. Slice 15
3.8.4. Dice 15
3.8.5. Pivot 15
3.9. Star Schema 16
CHAPTER 4 Usability 18
4.1. Introduction 18
4.2. Cube Query Model 19
4.2.1. Detailed dataset: 19
4.2.2. Selection condition – sigma (σ) 20
4.2.3. Grouper expression – gamma (γ) 21
4.2.4 Aggregate measures and distributive functions 22
4.3. Signatures 24
4.4. Query History and Sessions 25
4.5. The Usability Problem 26
4.5.1. Perfect Rollability 26
4.5.2. The six conditions of usability - Usability theorem 28
4.5.2.1. Condition 1: Same underlying Dataset 28
4.5.2.2. Condition 2: Compatible Dimensions and Distributive Measures 29
4.5.2.3. Condition 3: One Sigma Atom per Dimension 29
4.5.2.4. Condition 4: Perfect Rollability 29
4.5.2.5. Condition 5: New Grouper levels are Ancestors of Base Grouper levels 30
4.5.2.6. Condition 6: Signature Subset 30
4.5.3. Result Derivation from a Usable Base – Algorithm 9 34
CHAPTER 5 Implementation 37
5.1. Introduction 37
5.2. Pre-existing Query Execution Architecture 37
5.2.1 Overview 37
5.2.2 The Pre-existing Query Execution Flow 38
5.2.3 Limitations of the Baseline Approach 40
5.3. Design of the Usability Subsystem 40
5.3.1 UsabilityOptimizer 41

5.3.2 CubeQueryUsabilityChecker 42
5.3.3 CubeQueryUsabilityExecutor 42
5.4. Implementation of the Usability Execution Path 43
5.4.1 The Extended Entry Point 43
5.4.2 Execution Flow 43
5.4.3 Result Derivation Algorithm 46
5.5. Integration into the Existing Architecture 47
5.6. Summary 48
CHAPTER 6 Experiments 50
6.1. Experimental Setup 50
6.1.1. Experimental Goal and Evaluation Metrics 50
6.1.2. Competitor 51
6.1.3. Datasets 52
6.1.4. Measurement methodology 54
6.2. Effect of Usability on Performance 54
6.2.1. Effect of Data Scaling on Usability Performance 54
6.2.2. Effect of Usability Check Overhead on Usability Performance 58
6.2.3. Effect of Usability Queries Coverage on a Query Session 61
6.2.4. Effect of Query History Size on Usability Performance 63
6.2.5. Effect of Usability Query Position on Query History 67
6.2.6. Effect of Query History Growth on Usability Coverage and Performance 71
6.2.7. Effect of Query Complexity (Gamma/Sigma Anatomy) on Usability 73
6.2.8. Conclusions – Observations 80
CHAPTER 7 Conclusion and Future Work 83
REFERENCES 87
SHORT BIOGRAPHICAL SKETCH 88

List of Figures

Figure 1: A relational table containing sales information 6
Figure 2: A View containing total sales by country & year 7
Figure 3: Example of Hierarchies, Dimension Levels, Roll-up and Drill down 16
Figure 4: Example of a Star Schema with a fact table (Sales) at the center, and 4 dimension tables surrounding it (Time, Department, Product, Location) containing their respective levels 17
Figure 5: A detailed representation per day for Time, Location, Product and Sales count 20
Figure 6: Selection condition on [fig. 5] for Year = 2024 and City = Athens. It keeps only the relevant rows 21
Figure 7: Grouper expression on [fig. 5]. Grouping by Time_dim.Month 22
Figure 8: Grouper expression on [fig. 5]. Grouping by Time_dim.Year 22
Figure 9: Result after calculating average sales for 2024 on [fig. 5] 23
Figure 10: Example for demonstrating that the average result cannot always be correctly computed by aggregating partially averages. The table from [fig 5] has been used 23
Figure 11: A sales cube D with Country, Year, Product and Sales 24
Figure 12: A selection for year 2024 on cube D 25
Figure 13: Example of Total Monthly Revenue (group by Month), derived from result [fig 5] 32
Figure 14: Example of Total Quarterly Revenue (group by Quarter), expected from result [fig 5]. Since there is no data yet for months July and after we assume that Q3 and Q4 would contain null data and for convenience are omitted 32
Figure 15: Algorithm 9, reproduced from Vassiliadis [Vas23]: Answer a Cube Query from a Pre-Existing Query Result 35
Figure 16: Class diagram of Delian before usability implementation 39
Figure 17: answerCubeQueryFromStringWithUsability: execution flow, from receiving the raw query string to returning the output 45
Figure 18: Class diagram of Delian after usability implementation 48

Figure 19: pkdd99_star star schema (loan_cube and orders_cube), showing the fact tables, dimension tables, and their conformed dimension hierarchies 53
Figure 20: Effect of scalability in total execution time with 20% usability coverage 56
Figure 21: Effect of scalability in total execution time with 30% usability coverage 56
Figure 22: Usability Improvement Rate among various database sizes 57
Figure 23: Usability Check Overhead compared to DB execution time 59
Figure 24: Average Usability Check Overhead compared to average DB execution time 60
Figure 25: Comparison between usability check and usability execution time for usable queries 60
Figure 26: Comparison of total direct DB usability time and total usability time in various coverage scenarios 62
Figure 27: Usability time Speed up across various coverage scenarios 63
Figure 28: Effect of History Size on Usability at 10% coverage 65
Figure 29: Speed-up effect as history size increases 65
Figure 30: Effect of history size on usability at 40% coverage 66
Figure 31: Comparison between usable query position and total usability execution time for 1M 68
Figure 32: Speed-up effect by usable query position for 1M 69
Figure 33: Comparison between usable query position and total usability execution time for 10M 70
Figure 34: Speed-up effect by usable query position for 10M 70
Figure 35: Usability coverage percentage and total time as a function of accumulated history size 72
Figure 36: Usability execution time across increasing gamma complexity 77
Figure 37: Usability execution time across increasing sigma complexity 77
Figure 38: Usability execution time across increasing combined gamma & sigma complexity 78
Figure 39: Usability speed-up (direct execution time / usability execution time) across the six query complexity types comparing gamma-only, sigma-only, and combined gamma and sigma complexity variation 78

List of Tables

Table 1: Scalability experiment with 20% usability coverage 55
Table 2: Scalability experiment with 30% usability coverage 56
Table 3: Decomposed query execution time for each of 10 sessions 59
Table 4: Usability session times with different coverage percentages 62
Table 5: Effect of query history size on usability performance, with coverage held constant at 10% 64
Table 6: Effect of query history size on usability performance, with coverage held constant at 40% 66
Table 7: Effect of usable query position within a fixed 50-query history on 1M dataset 68
Table 8: Effect of usable query position within a fixed 50-query history on 10M dataset 69
Table 9: Usability coverage and execution time per batch as query history grows 72
Table 10: Usability speed-up and usability execution time for the six gamma-complexity query types (P1–P6) 76
Table 11: Usability speed-up and usability execution time for the six sigma-complexity query types 76
Table 12: Usability speed-up and usability execution time for the six pair types with gamma and sigma complexity varied simultaneously 76

ABSTRACT

Iliana Filippou, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, June 2026
Query processing for cube queries via query usability
Advisor: Panos Vassiliadis, Professor

Modern data analysis systems often rely on Online Analytical Processing (OLAP) techniques, where large volumes of data are organized into multidimensional structures (data cubes) to support efficient decision-making. A major challenge in such environments is determining whether previously computed analytical results can be reused to answer new queries, avoiding expensive recomputation.

This work is based on the concept of cube query usability, which extends traditional database view usability to hierarchical multidimensional environments. Cube query usability determines whether an existing cube query contains sufficient information to answer a new query without executing it but based only on its syntactic definition and by examining certain factors. Factors that affect the usability concern selection conditions, grouping levels, aggregation functions and the ability to perform valid OLAP operations between different levels of abstraction. Emphasis is also given to the concept of perfect rollability, which ensures the transformations between aggregation levels preserve the correctness of the requested results.

The main objective of this thesis is the implementation and the evaluation of a theoretical cube query usability framework, already introduced in the literature,

through a series of experimental scenarios. The developed system tests the theoretical conditions required for query usability and examines the cases in which an existing multidimensional query can successfully support a new analytical request without re-executing it. The results provide an empirical assessment of the applicability of the proposed theory and investigates its potential for improving query optimization and data reuse in OLAP based systems.

ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ

Ηλιάνα Φιλίππου, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2026
Επεξεργασία ερωτήσεων για ερωτήματα-κύβων με επαναχρησιμοποίηση αποτελεσμάτων προηγούμενων ερωτημάτων.
Επιβλέπων: Παναγιώτης Βασιλειάδης Καθηγητής

Τα σύγχρονα συστήματα ανάλυσης δεδομένων βασίζονται σε μεγάλο βαθμό στις τεχνικές Online Analytical Processing (OLAP), στο πλαίσιο των οποίων μεγάλοι όγκοι δεδομένων οργανώνονται σε πολυδιάστατες δομές (data cubes) για την υποστήριξη αποτελεσματικής λήψης αποφάσεων. Βασική πρόκληση σε τέτοια περιβάλλοντα αποτελεί ο προσδιορισμός του κατά πόσο προηγουμένως υπολογισμένα αποτελέσματα αναλυτικών queries μπορούν να επαναχρησιμοποιηθούν για την απάντηση νέων, αποφεύγοντας έτσι την εκ νέου εκτέλεσή τους έναντι της βάσης δεδομένων.

Η εργασία αυτή βασίζεται στην έννοια του cube query usability, η οποία επεκτείνει το παραδοσιακό view usability σε ιεραρχικά πολυδιάστατα περιβάλλοντα. Το cube query usability καθορίζει εάν ένα ήδη εκτελεσμένο ερώτημα περιέχει επαρκείς πληροφορίες για την απάντηση ενός νέου ερωτήματος, αξιοποιώντας αποκλειστικά τα αποθηκευμένα αποτελέσματά του, χωρίς πρόσβαση στα ακατέργαστα δεδομένα και χωρίς την εκτέλεσή του στην βάση δεδομένων. Οι παράγοντες που επηρεάζουν τη χρηστικότητα αυτή αφορούν τα selection conditions, τα grouping levels, τα aggregate functions και τη δυνατότητα έγκυρων OLAP μετασχηματισμών μεταξύ διαφορετικών επιπέδων αφαιρετικότητας. Ιδιαίτερη

έμφαση δίνεται στην έννοια του perfect rollability, το οποίο διασφαλίζει ότι οι μετασχηματισμοί μεταξύ επιπέδων διατηρούν την ορθότητα των αποτελεσμάτων.

Κύριος στόχος της παρούσας διπλωματικής εργασίας είναι η υλοποίηση και η αξιολόγηση του θεωρητικού πλαισίου του cube query usability εντός του συστήματος Delian Cube Engine, μέσω μιας σειράς πειραματικών σεναρίων. Το αναπτυχθέν σύστημα ελέγχει τις θεωρητικές συνθήκες που απαιτούνται για το usability και εξετάζει τις περιπτώσεις στις οποίες ένα υπάρχον πολυδιάστατο ερώτημα μπορεί να υποστηρίξει επιτυχώς ένα νέο αναλυτικό αίτημα χωρίς επανεκτέλεση. Τα αποτελέσματα παρέχουν εκτεταμένη αξιολόγηση της εφαρμοσιμότητας της προτεινόμενης θεωρίας και διερευνούν τις δυνατότητές της για τη βελτίωση του query optimization και της επαναχρησιμοποίησης δεδομένων σε OLAP συστήματα.

CHAPTER 1

INTRODUCTION

1.1 Goals

Modern decision-making increasingly relies on the analysis of large volumes of data through Business Intelligence (BI) systems. At the core of these systems lies the OLAP (On-Line Analytical Processing) paradigm, in which data is organized into multidimensional structures, commonly referred to as data cubes, that allow analysts to view a single set of facts from many different perspectives. A typical OLAP session does not consist of a single, isolated query; rather, an analyst formulates a sequence of related questions, drilling down into a hierarchy for more detail, rolling up for a broader summary, or narrowing and widening a selection condition, all over the same underlying dataset. This exploratory, iterative pattern of querying is what distinguishes analytical workloads from simpler transactional ones, and it is precisely this pattern that motivates the present work.

In a conventional query execution engine, every query submitted by an analyst, regardless of how closely it resembles a question asked moments earlier, is parsed and executed directly against the underlying database. This baseline approach is functionally correct but computationally wasteful: in an interactive OLAP session, the result of a previously executed query frequently already contains, in aggregated or filtered form, all the information necessary to answer the current one. Re-executing such a query against the database wastes resources that could otherwise be avoided entirely by reusing results that have already been computed and cached.

The question of whether an already computed result can be reused to answer a new query, instead of recomputing it from scratch, is not new to database research. It has been studied for decades under related problems such as query containment, view usability, and query rewriting, each of which examines, from a different angle, whether an existing computation can be exploited to answer a new request. This thesis builds on this line of research and extends it to the multidimensional, hierarchical setting of OLAP cubes through the notion of **cube query usability** [Vas23] [Vas23b], which determines whether a previously executed cube query can answer a new one using only its cached, aggregated result, without granting access to the raw detailed data and without resubmitting any query to the database.

The work presented in this thesis **implements** and **evaluates** the cube query usability framework within the Delian Cube Engine [Del25], a Java-based OLAP query engine. Three new components, an *optimizer* that searches the query history for a reusable base query, a *checker* that verifies whether the six structural and hierarchical conditions of usability hold, and an *executor* that derives the new result from the base query's cached cells in memory, are integrated into the engine's existing execution pipeline without disrupting its pre-existing architecture. When a usable base query cannot be found, execution falls back transparently to the standard database-backed path, so correctness is preserved in every case. The resulting system is evaluated experimentally across a range of dataset sizes and query workload configurations in order to quantify the performance benefits of usability-based query reuse and to characterize the overhead it introduces.

1.2 Structure of the Thesis

The remainder of this thesis is organized into seven further chapters, as follows:

- Chapter 2: Related Work

  Reviews the research problems of query containment, view usability, and query rewriting, addressing their limitations as well.

- Chapter 3: Preliminaries

  Reviews the database foundational background and research areas most relevant to this thesis, namely relational and multidimensional data models, business intelligence and data warehousing, OLAP and the cube model, as well as hierarchies, levels, and the standard OLAP operations.

- Chapter 4: Usability

  Formalizes the cube query model used throughout the thesis, including the sigma (selection) and gamma (grouping) expressions and the notion of distributive aggregate measures, introduces signatures, as the vocabulary for reasoning about subsets of the multidimensional space, and develops the theoretical foundation of cube query usability, including the notion of perfect rollability and the six conditions whose simultaneous satisfaction determines whether a base query can answer a new query without re-execution.

- Chapter 5: Implementation

  Describes the pre-existing query execution architecture of the Delian Cube Engine, the design of the new usability subsystem, namely the *UsabilityOptimizer*, *CubeQueryUsabilityChecker*, and *CubeQueryUsabilityExecutor* classes, the implementation of the usability execution path and its result derivation algorithm, and the integration of the subsystem into the existing engine.

- Chapter 6: Experiments

  Presents the experimental setup, datasets, and evaluation metrics, and reports the results of five experiments that measure the effect of dataset scaling, usability check overhead, query coverage, query history size, reusable query position, the growth of usability coverage as a query history accumulates, and the structural complexity of queries, on the performance of usability-based execution relative to direct database execution.

- Chapter 7: Conclusion and Future Work

  Summarizes the contributions of this thesis and the main findings of the theoretical and experimental work. Outlines directions for extending the present work, including evaluation under more complex and dynamic workloads, improved organization and indexing of the query history, support for partial usability across multiple base queries, and cost-based selection among multiple usable candidates.

CHAPTER 2

RELATED WORK

2.1. Query Containment and Usability

A fundamental challenge in database systems is the efficient processing of analytical queries over large volumes of data. In many scenarios previously executed queries or intermediate results contain information that can be reused to answer new queries. Avoiding recomputation has therefore been a long-standing objective in database research.

Three closely related database research problems have been studied in this context: query containment, view usability and query rewriting. These problems explore the theoretical foundation for determining whether an existing computation can be exploited to answer a new query, and the work of Vassiliadis [Vas23] extends them to hierarchically structured multidimensional spaces, building further upon them.

2.1.1. Query Containment

The query containment problem examines whether the result produced by one query is always included in the result produced by another query. Given two queries Q1 and Q2, Q2 is considered to contain Q1 if every tuple returned by Q1 is also returned in Q2, for every possible database instance [Vas23]. The problem has been studied extensively since Chandra and Merlin's seminal result on the NP-completeness of deciding containment for conjunctive queries [CM77].

For example, considering a relational table containing sales information:

PRODUCT	COUNTRY	YEAR	REVENUE
Laptop	Greece	2023	5000
Phone	Greece	2023	2000
Laptop	Italy	2023	3000

Figure 1: A relational table containing sales information

The query
SELECT * FROM Sales WHERE Country='Greece'
returns all sales performed in Greece.

A second query:
SELECT * FROM Sales WHERE Country='Greece' AND Product='Laptop'
returns only laptop sales in Greece.

Since every laptop sale in Greece is also a sale in Greece, the second query is contained in the first:

$$Q_{Laptop,Greece} \subseteq Q_{Greece}$$

2.1.2. View Usability

The database view is a stored result of a previously executed query. So instead of storing raw data only, databases can store the answer of queries. Views are widely used in analytical environments because they allow computations to be reused [Vas23]. The problem of deciding whether a query can be answered using a view, also known as view usability, has been surveyed in depth by Halevy [Hal01].
For example, let's consider the following view:

SELECT Country, Year, SUM(Sales)
FROM Sales
GROUP BY Country, Year

The view stores:

COUNTRY	YEAR	TOTAL SALES
Greece	2023	100000
Greece	2024	120000
Italy	2023	80000

Figure 2: A View containing total sales by country & year

Now let's suppose a user requests:
SELECT Country, SUM(Sales)
FROM Sales
GROUP BY Country

This query can be answered using the view because the stored result contains information at a finer granularity. The yearly values can be aggregated again:

$$Sales(Greece) = Sales(Greece, 2023) + Sales(Greece, 2024)$$

However, the reverse operation is impossible. If only yearly totals are stored, the original monthly or daily information cannot be reconstructed. Therefore, usability depends on whether the stored representation preserves the information required by the new query

2.1.3. Query Rewriting

When a view is determined to be usable, the database system must construct a new query that contains the requested result using the available view. This process is known as query rewriting [Vas23]. The goal is to replace a query over the original data source with an equivalent query over a previously computed representation, an idea whose first algorithmic treatment for conjunctive queries is due to Levy et al. [LMSS95].

For example, let's assume that a system maintains the following materialized view:

```
CREATE VIEW YearCountrySales AS
SELECT Country, Year, SUM(Sales) as TotalSales
FROM Sales
GROUP BY Country, Year
```

A user requests:

```
SELECT Country, SUM(Sales)
FROM Sales
GROUP BY Country
```

The rewritten query becomes:

```
SELECT Country, SUM(TotalSales)
FROM YearCountrySales
GROUP BY Country
```

The database avoids accessing the original detailed data.

2.1.4 Limitations & Summary

These concepts originate mainly from relational database theory, where data is represented as tables rather than multidimensional cubes. None of these methods offer a principled way to decide whether a value expressed at one level of a hierarchy (for instance, a year) implies, contains, or is implied by a value expressed at a different level of the same hierarchy (for instance, a month or a quarter).

Therefore, are effective when queries operate over independent attributes, but they do not fully address the additional complexity introduced by multidimensional databases, where dimensions are organized into hierarchies and queries operate at different abstraction levels. As Vassiliadis [Vas23] observes, the only prior work able to reason about implications between selection atoms defined at different levels of a hierarchy is the one of Vassiliadis and Skiadopoulos [VS00], which motivates the comprehensive treatment of cube usability for hierarchical multidimensional spaces that this thesis adopts and implements [Vas23].

CHAPTER 3

PRELIMINARIES

3.1. Relational Database

A relational database is a structured collection of data organized into tables consisting of rows and columns, where relationships between data are established through primary and foreign keys. Introduced by Edgar F. Codd in 1970 [JPT10], the relational model became the foundation of modern database systems.

It is optimized for transactional processing (OLTP), ensuring data consistency and integrity, using Structured Query Language (SQL) for data manipulation and retrieval. They serve as the operational backbone for most enterprise systems.

This structure, while effective for transactional systems, led to the development of alternative models better suited for analytical processing, such as the multidimensional database model.

3.2. Multidimensional Database Model

The multidimensional database model (MDDB) was developed to support analytical rather than transactional processing. Data is organized around facts (quantitative data such as sales or revenue) and dimensions (descriptive categories which characterize these facts, such as time, location, or product).

This model allows users to view data from multiple perspectives and perform aggregations efficiently, enabling fast analytical queries and supporting decision-making processes.

Building upon this model, organizations began integrating analytical processes into broader systems of decision support, giving rise to the field of Business Intelligence (BI).

3.3. Business Intelligence (BI)

Business Intelligence (BI) contains the technologies, applications, and practices used to collect, integrate, analyze, and present business information. Its primary goal is to support better enterprise decision-making by transforming raw data into meaningful insights. BI systems rely heavily on **data warehouses** for consolidated data storage and **OLAP** tools for multidimensional analysis.

Through dashboards, reports, and visualization tools, BI provides organizations with a comprehensive view of their performance, enabling them to identify tendencies & correlations, measure key performance indicators (KPIs), and develop data-driven strategies. In essence, BI bridges the gap between data storage and strategic decision-making.

To enable such analytical capabilities, BI systems depend on a solid data foundation, which is provided by the data warehouse.

3.4. Data Warehouse

A data warehouse is a large, centralized repository designed to store integrated, historical data collected from multiple heterogeneous sources within an organization. Its primary function is to support data analysis and reporting rather than daily operations.

Data warehouses often use multidimensional modeling to facilitate complex queries and trend analysis, forming the backbone of business intelligence systems.

3.5. OLAP (On-Line Analytical Processing)

OLAP refers to a class of systems designed for interactive and fast exploration of multidimensional data. OLAP systems allow users to perform various operations on data cubes, enabling dynamic exploration of data across different dimensions. Such operations will be explained in detail later.

These systems support decision-making by providing quick responses to complex analytical queries. They are widely used for identifying, patterns, and insights within large datasets and presenting data in easily interpretable formats such as cubes, charts, and dashboards.

3.6. OLAP Cubes

At the core of On-Line Analytical Processing (OLAP) lies the OLAP cube, a multidimensional data structure designed for fast and efficient analytical processing. Unlike traditional two-dimensional spreadsheets that organize data in rows and columns, an OLAP cube allows data to be represented across multiple dimensions, enabling far more complex and flexible analysis [JPT10].

Although the term cube implies three dimensions, in practice, it can include any number of dimensions, hence the term hypercube. Each dimension represents a unique perspective for examining data, such as time, location, or product, while the quantitative values being analyzed are referred to as measures. This multidimensional organization enables users to aggregate, compare, and explore data in ways that are not feasible with simple tabular models.

In real-world applications, data is extracted from various heterogeneous sources (such as databases, spreadsheets, and text files) and consolidated within a data

warehouse, where it undergoes cleaning and transformation processes before being loaded into an OLAP server. Once processed, the data is pre-aggregated and pre-calculated, allowing for rapid query execution and interactive analysis across multiple dimensions.

A deeper understanding of cube organization involves examining hierarchies and levels, which indicate how data is structured and aggregated.

3.7. Hierarchies and Levels

Each dimension within a cube serves two main purposes: it allows for the selection of data (filtering specific subsets) and the grouping of data at varying levels of detail. Dimensions are typically structured into hierarchies, which organize data into multiple levels of granularity [JPT10]. A hierarchy represents a "containment" relationship, where higher levels summarize or group data from lower levels.

For example, in a Time dimension, the hierarchy might progress from Day → Month → Quarter → Year. A specific data point such as March 15, 2025 belongs to the Day level, which rolls up into the Month level (March), then into Quarter 1, and finally into Year 2025. This structure enables analysts to view and aggregate data at any desired level of detail, such as daily sales, monthly revenue, or annual performance.

These hierarchical structures play a crucial role in the functionality of OLAP operations, providing users with the option to navigate and analyze data across different dimensions.

3.8. OLAP Operations

On-Line Analytical Processing (OLAP) systems provide a set of fundamental operations that enable users to explore and analyze multidimensional data

interactively. The primary OLAP operations (roll-up, drill-down, slice, dice, and pivot) allow analysts to summarize, refine, filter, and reorganize data dynamically.

Together, these operations make it possible to examine data from multiple perspectives, supporting complex analytical queries and decision-making processes.

3.8.1. Roll-up

The roll-up operation, also known as aggregation or consolidation, summarizes data by moving up a concept hierarchy or by reducing the number of dimensions. It provides a higher-level view of the data by grouping detailed information into broader categories [JPT10].

For example, consider a sales data cube that stores sales by city. The sales figures for New Jersey (440 units) and Los Angeles (1,560 units) can be aggregated to the country level, resulting in a total of 2,000 units for USA. In this case, the city dimension is removed, and the data is consolidated at a higher level of the geographic hierarchy.

3.8.2. Drill-down

The drill-down operation is the reverse of roll-up. It increases the level of detail by moving down a concept hierarchy or adding additional dimensions. This operation allows analysts to explore more granular data for deeper insights [JPT10].

For example, if sales data is initially aggregated at the quarter level (e.g., Q1), drilling down reveals the corresponding months (January, February, and March) along with their individual sales figures. This enables a more detailed understanding of trends and variations within the quarter.

3.8.3. Slice

The slice operation selects a single dimension value, creating a sub-cube that focuses on a specific subset of the data. This helps isolate particular aspects of the dataset for closer examination.

For example, in a cube with the dimensions Time, Location, and Product, applying a slice on Time = Q1 produces a smaller cube containing only data from the first quarter, across all products and locations. This allows analysts to concentrate solely on performance during that period.

3.8.4. Dice

The dice operation is similar to slicing, but involves selecting two or more dimension values to create a sub-cube, much like an "extended" slice. It allows for multidimensional filtering of data for more targeted analysis.

For example, if a data cube includes sales data across multiple countries and time periods, applying a dice with Time = Q1 and Location = USA results in a sub-cube that shows sales only for the first quarter in the United States. This operation provides a more specific and contextualized view of the data.

3.8.5. Pivot

The pivot (or rotation) operation reorients the multidimensional view of data by rotating its axes to provide an alternative presentation. This helps users visualize data relationships from different perspectives.

For example, in a sales cube showing Products across Regions, performing a pivot could swap these axes to display Regions across Products, offering a new viewpoint on the same data. Such reorganization enhances interpretability and aids in comparative analysis.

To sum up all of the above concepts, let's assume the following example [Figure 1]. Suppose we have 3 dimensions in the data warehouse: Time, Location and Product.

We can further move up or down to each of the dimension levels, depending on our analysis needs.

Time dimension levels can be (from higher level to lowest level of detail): Year → Quarter → Month → Day, and by rolling up or drilling down we move from bottom to top or vice versa, respectively, as shown in the figure below:

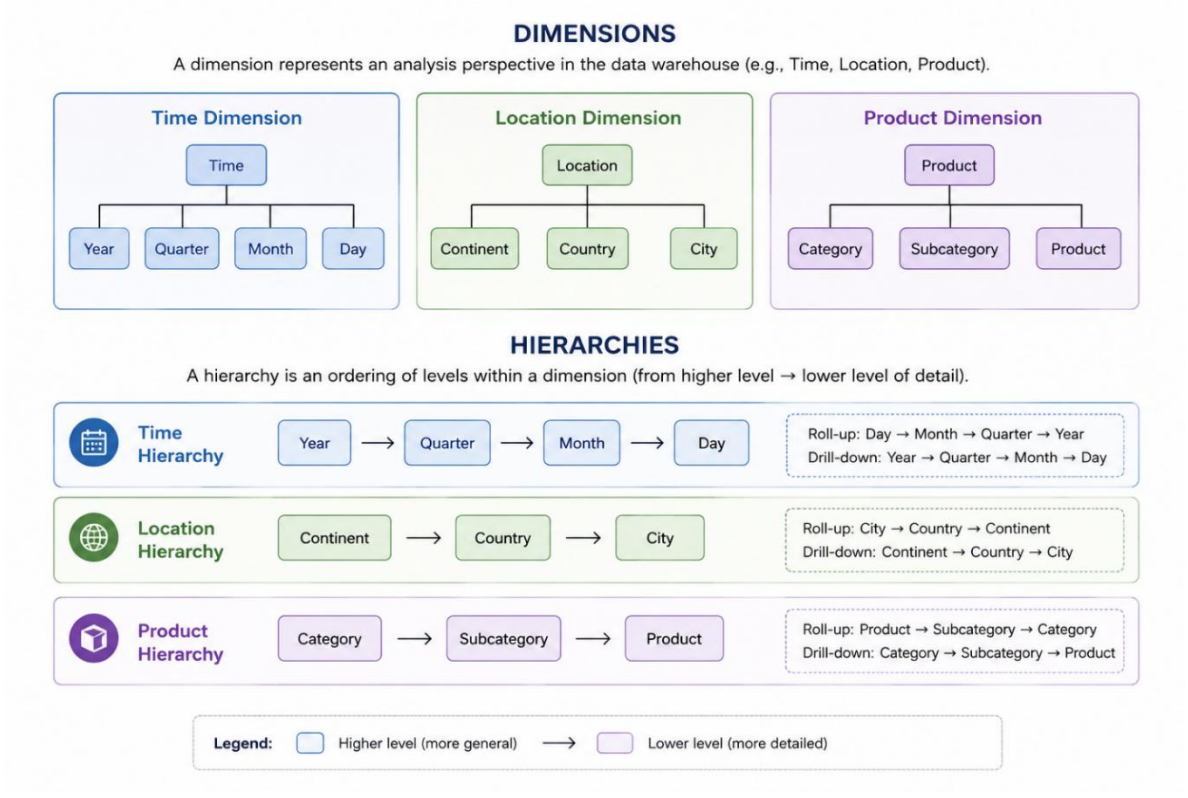


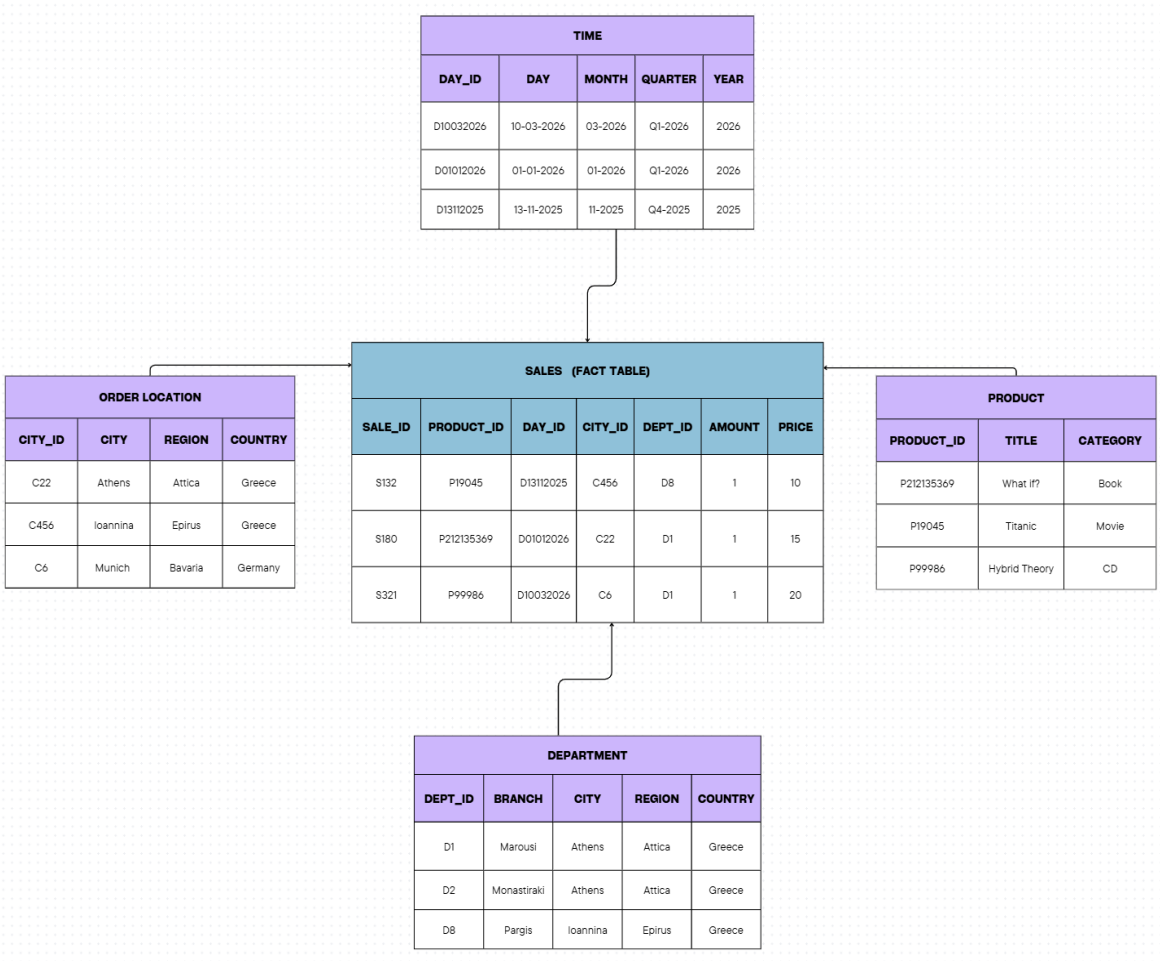
Figure 3: Example of Hierarchies, Dimension Levels, Roll-up and Drill down

3.9. Star Schema

For each dimension a star schema has one dimension table, containing the key column and one column for each level of that specific dimension. In case the levels

have extra properties, then the dimension table may also have a column dedicated to each of them.

A star schema also has a fact table that contains a column for each measure. In the row this contains the measure value. Of course, the fact table also has one column for each dimension. These hold the foreign keys that reference the dimension tables' primary keys [JPT10].



TIME

DAY_ID	DAY	MONTH	QUARTER	YEAR
D10032026	10-03-2026	03-2026	Q1-2026	2026
D01012026	01-01-2026	01-2026	Q1-2026	2026
D13112025	13-11-2025	11-2025	Q4-2025	2025

ORDER LOCATION

CITY_ID	CITY	REGION	COUNTRY
C22	Athens	Attica	Greece
C456	Ioannina	Epirus	Greece
C6	Munich	Bavaria	Germany

SALES (FACT TABLE)

SALE_ID	PRODUCT_ID	DAY_ID	CITY_ID	DEPT_ID	AMOUNT	PRICE
S132	P19045	D13112025	C456	D8	1	10
S180	P212135369	D01012026	C22	D1	1	15
S321	P99986	D10032026	C6	D1	1	20

PRODUCT

PRODUCT_ID	TITLE	CATEGORY
P212135369	What if?	Book
P19045	Titanic	Movie
P99986	Hybrid Theory	CD

DEPARTMENT

DEPT_ID	BRANCH	CITY	REGION	COUNTRY
D1	Marousi	Athens	Attica	Greece
D2	Monastiraki	Athens	Attica	Greece
D8	Pargis	Ioannina	Epirus	Greece

Figure 4: Example of a Star Schema with a fact table (Sales) at the center, and 4 dimension tables surrounding it (Time, Department, Product, Location) containing their respective levels

Therefore, if we put the fact table at the center and draw the dimension tables around it then the picture resembles a star. That is why it is called a "star schema".

CHAPTER 4

USABILITY

4.1. Introduction

This chapter lays the theoretical foundation of the usability theory [Vas23], on which the rest of the thesis is built. The theoretical results of this chapter have been presented in [Vas23] and [Vas23b].

Section 4.2 introduces a formal model of cube queries, defining their parts, the detailed dataset, the selection condition, the grouper expression, and the aggregate measures, and establishes the notion of distributive aggregate functions, which later proves essential to the correctness of query reuse.

Section 4.3 builds on this model to define query signatures, a mechanism for characterizing the region of the multidimensional space that a query's result actually covers.

Section 4.4 introduces the notion of query history and sessions, the context within which previously executed queries can later be considered for reuse.

Finally, Section 4.5 formalizes the central problem addressed by this thesis, the usability problem, namely, determining whether a previously executed query can answer a new query without re-executing it against the underlying database.

Together, these sections establish the vocabulary and formal tools used throughout the remainder of the thesis to define, reason about, and exploit cube query usability.

4.2. Cube Query Model

The proposed framework represents analytical operations through a formal cube query model.

A cube query consists of four components
q = (dataset, selection, grouping, aggregation) or more formally [Vas23]:

$$q =< DS^0, \varphi, [L_1, L_2, \dots, L_n, M_1, M_2, \dots, M_m], [agg_1(M_1^0), agg_2(M_2^0), \dots, agg_m(M_m^0)] >$$

Where

- DS^0 represents the detailed dataset.
- φ is a multidimensional conjunctive selection condition
- $L_1, L_2, \dots, L_n$ are the grouper levels
- $M_1, M_2, \dots, M_m$ are the aggregated measures
- $agg_1, agg_2, \dots, agg_m$ are aggregate functions applied to measures

In terms of the SQL analogy over a star schema, the cube query would be:

SELECT L_1, L_2, ..., L_n, $agg_1(M_1^0)$ AS M_1, $agg_1(M_1^0)$ AS M_2, ..., $agg_1(M_1^0)$ AS M_m
FROM DS^0
NATURAL JOIN D_1 NATURAL JOIN D_2 ... NATURAL JOIN D_n
WHERE φ^0
GROUP BY L_1, L_2, ..., L_n

4.2.1. Detailed dataset:

The detailed dataset represents the lowest level of information from which all queries are derived.

For example:

DATE	CITY	PRODUCT	SALES
01/01/24	Athens	Laptop	1000
02/01/24	Athens	Phone	500
15/02/24	Ioannina	Book	20
10/03/24	Ioannina	Phone	600
20/07/24	Athens	Laptop	1100
10/07/24	Ioannina	Printer	50
03/09/24	Ioannina	PC	400
10/09/24	Athens	Printer	70

Figure 5: A detailed representation per day for Time, Location, Product and Sales count

This detailed representation allows higher-level summaries to be generated.

4.2.2. Selection condition – sigma (σ)

A sigma (selection) expression set specifies: for each dimension, a selection predicate that restricts which dimension members are considered, meaning it defines which part of the multidimensional space participates in the query. This creates a subspace.

Each selection atom (σ - sigma) consists of the following parts: $(D_i.L_i{}^{\varphi}, operation, value)$ meaning "restrict dimension D_i to members of level L_i while applying filter φ".

The operator is typically equality (=) or a membership set (IN), while the rest of inequality comparison operators still hold.
L_i^{φ} is called the sigma level of the respective dimension D_i.

For example, considering the table from *[Figure 5]* again, the expression:
$(Year = 2024) \wedge (City = Athens)$
means that only sales performed in Greece during 2024 will be considered.

DATE	CITY	PRODUCT	SALES
01/01/24	Athens	Laptop	1000
02/01/24	Athens	Phone	500
20/07/24	Athens	Laptop	1100
10/09/24	Athens	Printer	70

Figure 6: Selection condition on [fig. 5] for Year = 2024 and City = Athens. It keeps only the relevant rows

4.2.3. Grouper expression – gamma (γ)

Grouping levels determine the output granularity of the query result. Each grouper expression is of the following form: gamma(D_i, L_i^g) meaning "Group the data by the members of level L_i of dimension D_i". The result will contain one cell per distinct combination of member values across all grouper levels, following the cube query model of [Vas23].

For example, considering the same table from *[Figure 5]*, grouping by Month produces the following detailed result

MONTH	SALES
JANUARY	1500
FEBRUARY	20
MARCH	600
JULY	1150
SEPTEMBER	470

Figure 7: Grouper expression on [fig. 5]. Grouping by Time_dim.Month

Grouping by Year produces a more general result like below:

YEAR	SALES
2024	3740

Figure 8: Grouper expression on [fig. 5]. Grouping by Time_dim.Year

4.2.4 Aggregate measures and distributive functions

A set of aggregate measures pairs a numeric attribute with an aggregate function agg(m) where m is the name of the measure (the fact-table attribute to aggregate).

An aggregate function f is distributive if and only if it satisfies the following property: Given a partition of a set S into subsets S_1, S_2,..., S_k it holds that

$f(S) = f(f(S_1), f(S_2),..., f(S_k))$.

The functions *SUM*, *COUNT*, *MIN* and *MAX* are all distributive. In contrast, *AVG* (arithmetic mean) is not distributive because the average of averages is not generally equal to the overall average, unless the subsets have equal cardinality.

Considering the example table from *[Figure 5]* and applying the AVG in order to calculate the average sales for the whole year 2024, the correct number would be 467.50

YEAR	AVG(SALES)
2024	467.50

Figure 9: Result after calculating average sales for 2024 on [fig. 5]

However, if we first calculate the average sales for the level "month" in 2024 and then try to aggregate it up to the average sales for the whole year 2024, the result is going to be 748, which is incorrect.

MONTH	AVG(SALES)
JANUARY	750
FEBRUARY	20
MARCH	600
JULY	575
SEPTEMBER	235

YEAR	AVG(SALES)
2024	748

Figure 10: Example for demonstrating that the average result cannot always be correctly computed by aggregating partially averages. The table from [fig 5] has been used

Usability relies on distributives functions because deriving a coarser result from finer cached cells requires re-aggregating partial results [Vas23].

4.3. Signatures

The signature of a query with respect to dimension D is the set of member values at D's grouper level L^g that are selected by the query's sigma predicate. Formally, if the sigma selects members at sigma level L:

$$sig(q, D) = \pi_{L^g}(\sigma_{L^\varphi \in V}(D))$$

The signature is obtained by rolling down the sigma values from the sigma L^φ to the finer grouper L^g.

In simpler words, the signature characterizes which cells will actually appear in the query result. It represents the boundary of the multidimensional region covered by the query. A query does not just return a set of values, it defines a region inside the cube. The signature potentially allows the system to compare whether two queries refer to the same region or whether one region includes another [Vas23].

For example, let's consider the following sales cube D:

COUNTRY	YEAR	PRODUCT	SALES
Greece	2024	A	100
Greece	2025	A	150
Italy	2024	B	200

Figure 11: A sales cube D with Country, Year, Product and Sales

And the requested question: “Total sales by Country for Year = 2024”.

The selection is: $\sigma_{Year=2024}(D)$

Then the result is

COUNTRY	YEAR	PRODUCT	SALES
Greece	2024	A	100
Italy	2024	B	200

Figure 12: A selection for year 2024 on cube D

Then the projection on grouping level L^g = {Country} is:

$sig(q, D) = \pi_{Country}(\sigma_{Year=2024}(D))$ = {Greece, Italy}

4.4. Query History and Sessions

The query history of a user is the chronological record of all cube queries submitted by that user, together with their syntactic definitions and, where available, their already-computed result cells. A user’s query history is built up over time, session after session: the history at any point is the list of queries obtained by concatenating all of the user’s sessions so far, in the order in which they occurred.

A session is an ordered list of cube queries issued by a single user in the course of one continuous analytical task. Within a session, an analyst typically poses a sequence of related queries, drilling down for finer detail, rolling up for a coarser summary, or adjusting the selection condition to explore a different part of

the data. A session is the natural unit over which query reuse and usability are evaluated, since the queries within a single session are the most likely to overlap with one another.

4.5. The Usability Problem

The main focus of [Vas23] is the extension of traditional view usability theory to multidimensional databases. In a data warehouse environment, users frequently execute analytical queries that overlap with previous computations. Given that executing a cube query typically requires a multi-table join between the fact table and the dimension tables, followed by a group by aggregation, potentially scanning millions of rows, it is wasteful to repeat this expensive database operation in case a previously computed result already contains all the data needed to answer the new query. Instead of recomputing results from the detailed dataset, a system can attempt to reuse the already computed cube.

The usability problem can be defined as follows:
Given an existing query q^b whose result is already known, and a new query q^n, determine whether q^b contains enough information to answer q^n, without actually executing it.

The existing query is considered usable if it can be transformed into the requested query while preserving correctness.

4.5.1. Perfect Rollability

A fundamental concept for determining usability is rollability. Perfect Rollability describes whether a cube can be transformed from one aggregation level to another, without losing information and indicates whether the selection and grouping work well together, producing meaningful results.

A query is said to be perfectly rollable with respect to a dimension D if its sigma level L^{σ} is an ancestor (or equal to) its grouper level L^{g} in the dimension hierarchy. Formally [Vas23]: $L^{\gamma} \preccurlyeq L^{\sigma}$.

Intuitively, a query is perfectly rollable if the selection (sigma) predicate operates at a granularity that is at least as coarse as the grouping (gamma) granularity. This ensures that for every member of the grouper level its entire lineage at the sigma level is either fully included or fully excluded by the sigma predicate, meaning there are no partial groups.

Let's consider the following example for further understanding [Vas23b], having selection on year = 2020 and grouping by month. The selection condition is defined at a higher level of the grouper level, then all the months must be present, from 01-2020, 02-2020 all the way up to 12-2020, and all the days from 01-01-2020 up to 31-12-2020. Selection year=2020 and group by month brings all the months of 2020.

Once the filter is applied and then the grouper is also applied, the resulting values include the entire set of detailed values at the lower level of the grouped ones. Therefore, perfect rollability holds.

Supposed that the most detailed level consisted of dates from 26/6/2020 to 5/10/2020, then perfect rollability does not hold.

This can work for the opposite direction as well [Vas23]: $L^{\sigma} \prec L^{\gamma}$, although it is a less common practice.

For example, let's consider the inverse, where the selection condition sits lower than the level of the grouper. Then we must assert that all values participate. For gamma and sigma combination where selection is Month from 01-2020 to 12-2020 and grouping is by year, then it must be guaranteed that the year includes the entire set of month values.

In more simple words, a query is perfectly rollable when every requested result can be obtained exactly from the available information. Perfect rollability is a necessary prerequisite for usability, because partial groups can not be reliably reaggregated. Behind a perfectly rollable query, the intuition is that the selection predicate does not split any group created by the grouping operation. Every group must contain either all of its descendants or none of them. Therefore, the selected region must preserve complete hierarchical units.

4.5.2. The six conditions of usability - Usability theorem

Can the old query answer the new one?

The usability optimization implemented in this thesis is grounded in a formal theory of cube query usability, as formalized in Theorem 9.1. [Vas23]. Informally, a previously executed query q^b (the *base query*) is said to be *usable* for a new query q^n if and only if the result of q^n can be derived entirely from the cached result of q^b through in-memory filtering, roll-up, and re-aggregation, without any recourse to the database.

Formally, q^b is usable for q^n when the following six conditions are simultaneously satisfied:

4.5.2.1. Condition 1: Same underlying Dataset

Both q^n and q^b must reference the same underlying detailed cube (data source). This ensures that both queries operate over identical raw data, making the cells of q^b semantically comparable to those required for q^n.

4.5.2.2. Condition 2: Compatible Dimensions and Distributive Measures

Both queries must share the same set of *dimensions* in their grouper expressions and the same set of aggregate *measures*.

Furthermore, all aggregate functions must be *distributive*, that is, they must belong to the set {SUM, COUNT, MIN, MAX}, so that partial aggregates can be re-combined without loss of accuracy.

4.5.2.3. Condition 3: One Sigma Atom per Dimension

Each query's selection condition must contain exactly one sigma atom per dimension appearing in its grouper expression. This condition, evaluated independently for both q^n and q^b, guarantees that the selection predicate is a simple, dimension-wise conjunction amenable to cell-level filtering.

4.5.2.4. Condition 4: Perfect Rollability

For each of the queries q^n and q^b independently the following must hold:

$$D.L^g \preccurlyeq D.L^\varphi$$

A query is *perfectly rollable* with respect to a dimension D if, for that dimension, the level at which the sigma selection is applied is an ancestor of (or equal to) the level at which grouping is performed. That is, the sigma level must be no finer than the grouper level.

Both the new query q^n and the base query q^b must be perfectly rollable. This ensures that the grouper levels of both queries are properly aligned with their corresponding sigma predicates, preventing any result cell from crossing a sigma-

boundary and ensuring that the selection and grouping levels are hierarchically consistent within each query.

4.5.2.5. Condition 5: New Grouper levels are Ancestors of Base Grouper levels

For every dimension, the grouper level (gamma) of q^n must be an ancestor of (or equal to) the grouper level (gamma) of q^b.

This guarantees that q^n operates at a coarser or equal granularity than q^b, making it possible to derive q^n's cells by rolling up q^b's cells in the hierarchy.

$$D.L^{g^b} \preccurlyeq D.L^{g^n}$$

If q^n required a finer granularity than q^b, the cached cells would lack the detail necessary to answer the query.

4.5.2.6. Condition 6: Signature Subset

This condition ensures coverage. Every dimension member that q^n needs must already be present in q^b's result. If q^n requires a member value that q^b did not produce (because q^b's sigma predicate excluded it), the cached result is incomplete for q^n's purposes.

For this purpose, the *signature* of q^n with respect to dimension D, defined as the set of dimension member values at q^b's grouper level that fall within q^n's selection range, must be a subset of the *grouper domain* of q^b, which is the set of dimension member values actually covered by q^b's result cells.

This ensures that every value needed to answer q^n is present in q^b's cached result.

Formally, for each dimension D_i the following must hold:

$$sig(q^n, D_i) \subseteq gdom(q^b, D_i)$$

These conditions ensure that the source query contains all necessary information for reconstructing the target query.

Let's assume the following two queries over a Sales cube, with dimension *Time* (hierarchy Day → Month → Quarter → Year) and measure *Revenue* aggregated with *SUM*.

Base query q^b :

```
SELECT
    Month,
    SUM(Revenue) AS TotalRevenue
FROM Sales
WHERE Year IN (2024, 2025)
GROUP BY Month;
```

New query q^n:

```
SELECT
    Quarter,
    SUM(Revenue) AS TotalRevenue
FROM Sales
WHERE Year = 2024
GROUP BY Quarter;
```

MONTH	TOTAL REVENUE
January	100
February	200
March	300
April	150
May	0
June	180

Figure 13: Example of Total Monthly Revenue (group by Month), derived from result [fig 5]

QUARTER	TOTAL REVENUE
Q1	600
Q2	330

Figure 14: Example of Total Quarterly Revenue (group by Quarter), expected from result [fig 5]. Since there is no data yet for months July and after we assume that Q3 and Q4 would contain null data and for convenience are omitted

Let's verify the six conditions of Theorem 9.1 in order to determine whether q^b is usable for q^n.

Check 1 – Same data source:

Both queries are defined over the same underlying cube, *Sales*. Therefore $DS(q_b) = DS(q_n)$, so condition 1 is satisfied.

Check 2 – Compatible dimensions and measures:

Both queries use only "Time" dimension and aggregate the same measure, "*SUM(Revenue)*", with *SUM* being a distributed function.
Condition 2 is satisfied.

Check 3 – One atom per dimension:

The selections are:
For q^b: $\sigma_b = Year\ IN\ (2024, 2025)$
For q^n: $\sigma_n = Year = 2024$
Therefore, the selection condition of **both** queries contains exactly one atom for the Time dimension and none for any other dimension. Check 3 is satisfied.

Check 4 – Perfect rollability

The base query q^b groups by Month and has selections by year.
The new query q^n groups by Quarter and has selections by year.
The hierarchy is Day → Month → Quarter → Year
So **both** queries are perfectly rollable since the sigma level (*Year*) is an ancestor of the grouper level (*Month*) for q^b and, the sigma level (*Year*) is likewise an ancestor of the grouper level (*Quarter*) for q^n.
Check 4 is satisfied.

Check 5 – Compatible grouping levels

The base query q^b has the following gamma: $\gamma_b = (Month)$
The new query q^n has the following gamma: $\gamma_n = (Quarter)$

Since Quarter is an ancestor of *Month* in the *Time* hierarchy, the transformation is valid because the q^n is aggregating at a higher (coarser) or equal granularity level than q^b.

Check 6 – Signature coverage

The grouper domain of base query q^b consists of $gdom(q^b, D_i) = [(2024, Month), (2025, Month)]$, which is the set of *Month* values actually covered by its result. So,

$gdom(q^b, Time)$ = {Jan 2024, Feb 2024, ..., Dec 2024, Jan 2025, Feb 2025, ..., Dec 2025}

The signature of new query q^n expressed at the q^b groupers (which is *Month*) consists of $sig(q^n,\ D_i) = (2024, Month)$, which the set of months implied by q^n's selection condition (Year = 2024).

$sig(q^n,\ Time)$ = {Jan 2024, Feb 2024, ..., Dec 2024}

Therefore:

$sig(q^n, D_i) = (2024, Month) \subseteq [(2024, Month), (2025, Month)] = gdom(q^b, D_i)$

Check 6 is satisfied.

Since all of checks have been satisfied the query q^n can now be calculated using the results of base query q^b without executing it.

But how exactly?

4.5.3. Result Derivation from a Usable Base – Algorithm 9

When all six conditions above are satisfied, the result of q^n is derived from q^b's already computed cells through the following procedure [Vas23]:

Algorithm 9: Answer Cube Query from a Pre-Existing Query Result

Input: A new query expression q^n and a previously computed query q^b along with its result $q^b.cells$
Output: The result of q^n, $q^n.cells$

begin
  $q^n.cells \leftarrow$ compute q^{n^+} and for every coordinate, create a new cell with all measures initialized to $\varnothing$
  if q^b *and* q^n *satisfy all conditions of Theorem* 9.1 **then**
    forall *dimensions* D_i **do**
      $\alpha_i^{n@b} \leftarrow$ the transformed atom of the new query at the schema grouper level L_i^b of q^b
    end
    $\phi^{n@b} = \wedge\, \alpha_i^{n@b}$
    $q^{n@b}.cells \leftarrow$ apply $\phi^{n@b}$ to $q^b.cells$
    q^{n^G} = group the cells of $q^{n@b}.cells$ according to q^{n^+}
    forall *measures* M_j **do**
      $q^n.cells.M_j \leftarrow$ apply agg_j^F to the j-th measure of the members of the groups of q^{n^G}
    end
  end
  return $q^n.cells$
end

Figure 15: Algorithm 9, reproduced from Vassiliadis [Vas23]: Answer a Cube Query from a Pre-Existing Query Result

Step 1: Sigma filter expansion

The sigma predicates of q^n are expressed via drill down at q^b's grouper level. This is necessary when q^n's sigma level is coarser than q^b's grouper level.

Step 2: Cell filtering.

The cells of q^b are scanned and those that do not satisfy the sigma filter expansion are discarded, and the rest are kept, producing a set of surviving cells.

Step 3: Result calculation

The surviving cells are grouped by the groupers of q^n for every dimension. Within each group, measures are aggregated using the corresponding distributive aggregation functions.

Using the example above, let's see how we could calculate the results of q^n using the existing results of q^b.

Step 1 – Sigma filter expansion:

The sigma filter expansion of q^n contains the sigma values of q^n grouped by the gamma values of q^b. Basically, q^n's sigma predicate is re-expressed at q^b's grouper level (*Month*) by descending from *Year* to *Month* in the *Time* hierarchy. Year 2024 unfolds into its twelve constituent months. More specifically:

$\sigma_n: Year = 2024$

$\gamma_b: Time = Month$

So, the Sigma filter expansion would be:

[January 2024, February 2024, ..., December 2024].

Step 2 – Cell filtering

The cells of q^b are scanned and only those result cells that are in the sigma filter expansion of q^n [January 2024, February 2024, ..., December 2024] are kept as the surviving cells and will be used for q^n's result derivation.

Since q^b's result contains the twenty-four months of 2024 and 2025, this step discards the twelve cells belonging to 2025 and keeps the twelve surviving cells belonging to 2024.

Step 3 – Result calculation

Since we have the results for each month of 2024 (as we kept only those who were on [January 2024, February 2024, ..., December 2024] set) we can now group by q^n's groupers (which are $\gamma_b: Time = Quarter$) to the surviving cells of q^b while applying the aggregate SUM function, yielding the quarterly totals that constitute the result of q^n.

Because every cell of q^n was derived from cells already present in the base query's cached result, this computation is performed entirely in memory, without resubmitting q^n to the database.

CHAPTER 5

IMPLEMENTATION

5.1. Introduction

One of the central objectives of this work is to augment the Delian Cube Engine with a query usability mechanism, enabling the system to derive the result of a new analytical query from the cached result of a previously executed one, without resubmitting a query to the underlying database. This optimization, grounded in the theoretical framework of cube usability as formalized in Theorem 9.1[Vas23], offers a significant reduction in query execution latency for workloads in which consecutive queries share structural and dimensional properties.

This chapter provides a detailed explanation of *(i)* the architecture of the Delian Cube Engine as it existed prior to this work, *(ii)* the theoretical conditions that govern the applicability of usability-based execution, and *(iii)* the full design and implementation of the usability subsystem introduced in this thesis, including the new classes, their responsibilities, and their integration into the existing execution pipeline.

5.2. Pre-existing Query Execution Architecture

5.2.1 Overview

The Delian Cube Engine is a Java-based OLAP query engine that processes cube queries expressed in a dedicated query language. The engine is built around a

layered architecture in which a remote interface, a session processor, and a set of specialized managers collaborate to handle client requests.

At the outermost layer, the class *SessionQueryProcessorEngine*, which extends *UnicastRemoteObject* and implements *IMainEngine*, serves as the RMI-accessible entry point for all client interactions. It maintains references to the session's active *CubeManager*, *QueryHistoryManager*, and other contextual objects, and it delegates all substantive processing to a suite of internal managers via a Director-based mechanism.

5.2.2 The Pre-existing Query Execution Flow

Prior to the introduction of usability, the primary method for answering a cube query supplied as a raw string was *answerCubeQueryFromString()*. Its execution proceeded through the following steps:

1. **Entry point.** The client invokes *SessionQueryProcessorEngine.answerCubeQueryFromString(String queryRawString)* The method packages the raw string into a parameter map and forwards the request via *delegateExecution("answer_from_string", params)*, which dispatches it through a Director to the appropriate manager.

2. **Routing.** The *QueryExecutionManager.routeCommand()* method receives the command alias *"answer_from_string"* and routes it to *QueryExecutionManager.answerCubeQueryFromString(String, CubeManager, QueryHistoryManager)*.

3. **Parsing.** The raw query string is parsed by *CubeManager.createCubeQueryFromString()*, which constructs a fully populated *CubeQuery* object encoding the query's selection conditions (sigma

expressions), grouping levels (gamma expressions), aggregate measures, and reference cube.

4. **Database execution.** The parsed *CubeQuery* is passed to *executeCubeQuery()*, which submits the query to the underlying database via *CubeManager.executeQuery().* This produces a *Result* object populated with the retrieved cells.

5. **Output and history.** The result is serialized to a tab-delimited text file under the *OutputFiles/* directory. The executed query is added to the session's *QueryHistoryManager*, and the path to the output file is returned to the caller.

In this baseline architecture, every query, regardless of its structural similarity to previously executed queries, incurs a full round-trip to the database. No mechanism existed to exploit the results of prior queries, even when those results contained all the information necessary to answer the new request.

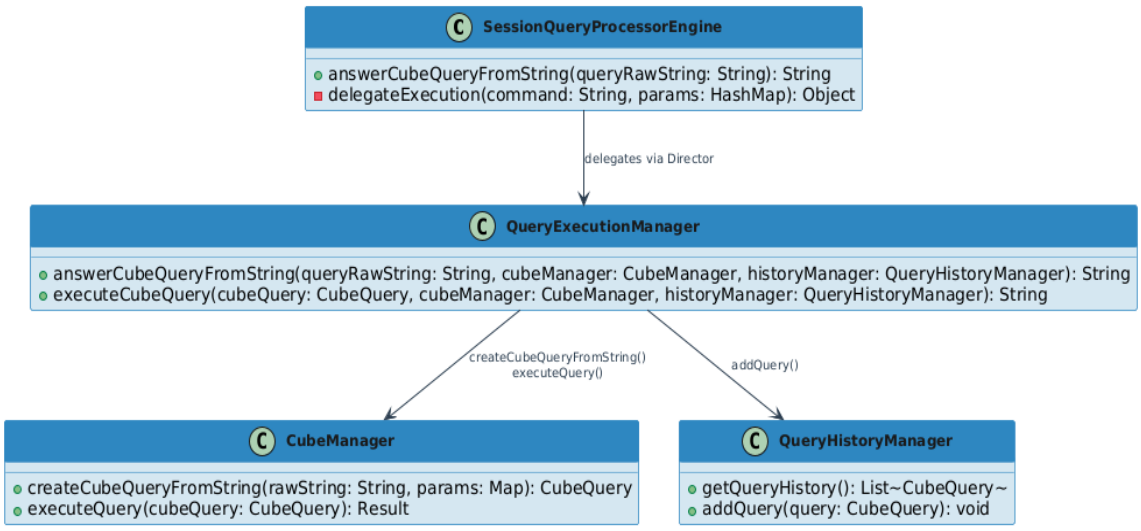


Figure 16: Class diagram of Delian before usability implementation

5.2.3 Limitations of the Baseline Approach

The baseline approach contains no functional mechanism for executing usability-based analysis, where users typically issue sequences of closely related queries, drilling up or down within a hierarchy, narrowing or broadening a selection condition, or examining different aggregations over the same data space. In such sessions, the result of a previously executed query may be a superset of the cells required to answer the current one, rendering a new database query redundantly.

The absence of a usability mechanism forces the engine to re-execute every query against the database, suffering unnecessary I/O and processing overhead.

5.3. Design of the Usability Subsystem

To implement the theoretical framework described above, three new **classes** were introduced in the package *cubemanager.usability*:

- *UsabilityOptimizer*: the orchestrator, responsible for searching the query history and coordinating the usability check and execution.

- *CubeQueryUsabilityChecker*: the condition evaluator, responsible for verifying the six usability conditions for a candidate pair (q^n, q^b) and deciding whether the new query can indeed be answered from the base query

- *CubeQueryUsabilityExecutor*: the result calculator, responsible for deriving the result of q^n from the cached cells of q^b when a usable base is found and therefore usability has been confirmed.

The three classes are designed according to a clear separation of concerns: the *optimizer* searches and orchestrates, the *checker* validates, and the *executor* computes. Both *UsabilityOptimizer* and *CubeQueryUsabilityChecker* follow the

Singleton pattern, as a single shared instance is sufficient within a session and avoids redundant instantiation. *CubeQueryUsabilityExecutor* is instantiated once by *CubeQueryUsabilityChecker* at construction time and held as a final field.

5.3.1 UsabilityOptimizer

UsabilityOptimizer is the entry point for the usability subsystem. It holds a reference to the singleton *CubeQueryUsabilityChecker* and exposes the method *tryExecuteWithUsability(CubeQuery newQuery, QueryHistoryManager historyManager, CubeBase cubeBase)*, which returns a *boolean* indicating whether the usability-based execution succeeded.

Internally, it delegates to *findUsableBase()*, which retrieves the session's query history through *QueryHistoryManager.getQueryHistory()*, reverses the list to examine the most recently executed queries first, and iterates over the candidates. For each candidate with a non-null cached result, it invokes *CubeQueryUsabilityChecker.setQueries()* to configure the checker with the current pair (q^n, candidate), then calls *CubeQueryUsabilityChecker.checkUsability()*. The **first** candidate for which all six conditions are satisfied is returned as the usable base query.

If a usable base is found, *UsabilityOptimizer* proceeds to call *CubeQueryUsabilityChecker.executeCubeQueryWithUsability(newQuery),* which delegates to the executor.

Timing metrics are recorded for both the checking phase (*lastCheckTimeMs*) and the execution phase (*lastAnswerTimeMs*), enabling experimenting for performance analysis later.

The computed result is subsequently retrievable via *getComputedResult()*.

5.3.2 CubeQueryUsabilityChecker

CubeQueryUsabilityChecker holds the pair of queries under comparison (q^n as *newQuery* and q^b as *baseQuery*) along with the shared *CubeBase* instance.

Its primary method *checkUsability()* evaluates the six conditions in sequence using short-circuit *boolean* evaluation: if any condition fails, the remaining checks are skipped.

The six condition-checker methods, *checkCondition1SameDS()* through *checkCondition6SignatureSubset()* are package-private, facilitating unit testing without exposing them as part of the public API.

Condition 6 requires resolving dimension member values across hierarchy levels. *CubeQueryUsabilityChecker* provides two auxiliary methods: *rollDownValuesToLevel(),* which expands a set of coarser-level values to a finer level by executing a targeted and low in cost *"SQL SELECT DISTINCT"* query against the dimension table, and *buildSigmaFilter()*, which parses the sigma expressions of a query into a map from dimension name to the set of permitted values, handling equality (=) and membership (IN) operators.

5.3.3 CubeQueryUsabilityExecutor

CubeQueryUsabilityExecutor is responsible for the in-memory derivation of q^n's result from q^b's cached cells. It holds a reference to the *CubeQueryUsabilityChecker* (from which it reads the base query and cube base) and stores the derived result in the field *computedResult*. The derivation is performed by the method *executeCubeQueryWithUsability(CubeQuery currentCubeQuery),* which proceeds in six steps described in detail in Section 4.5.

5.4. Implementation of the Usability Execution Path

5.4.1 The Extended Entry Point

The usability execution path is triggered by the method *answerCubeQueryFromStringWithUsability(String queryRawString)* in *SessionQueryProcessorEngine* (SQP). Structurally parallel to the preexisting *answerCubeQueryFromString()*, this method packages the raw query string into a parameter map and delegates execution via *delegateExecution("answer_from_string_with_usability", params)*, routing it through the Director to *QueryExecutionManager.answerCubeQueryFromStringWithUsability (String, CubeManager, QueryHistoryManager).*

5.4.2 Execution Flow

The full execution flow of the usability is as follows:

Step 1- Query Parsing. The raw query string is parsed into a *CubeQuery* object via *CubeManager.createCubeQueryFromString()*, exactly as in the baseline path.

Step 2 - Usability Check. The parsed query is submitted to *UsabilityOptimizer .tryExecuteWithUsability().* The optimizer searches the session history (most-recent first) for a candidate base query whose cached result satisfies all six usability conditions with respect to the new query.

This search is performed entirely in memory, with the exception of the SQL lookups required by Conditions 5 and 6 for cross-level value resolution.

Step 3a - Usability-Based Execution (usability: success). If a usable base is found, the optimizer invokes the executor to derive the result of q^n from q^b's cached cells in memory (see Section 4.5.3). The derived result is stored in the

QueryExecutionManager, serialized the output file, and the new query is added to the session history. The path to the output file is returned.

Step 3b - Fallback to Standard Execution (usability: failure). If no usable base is found, either because the history is empty, no candidate satisfies all six conditions, or the candidate's cached result is unavailable, the method falls to *executeCubeQuery(),* the same database-backed execution method used by the baseline path. This ensures correctness in all cases: the usability mechanism is just an optimization that is bypassed when inapplicable.

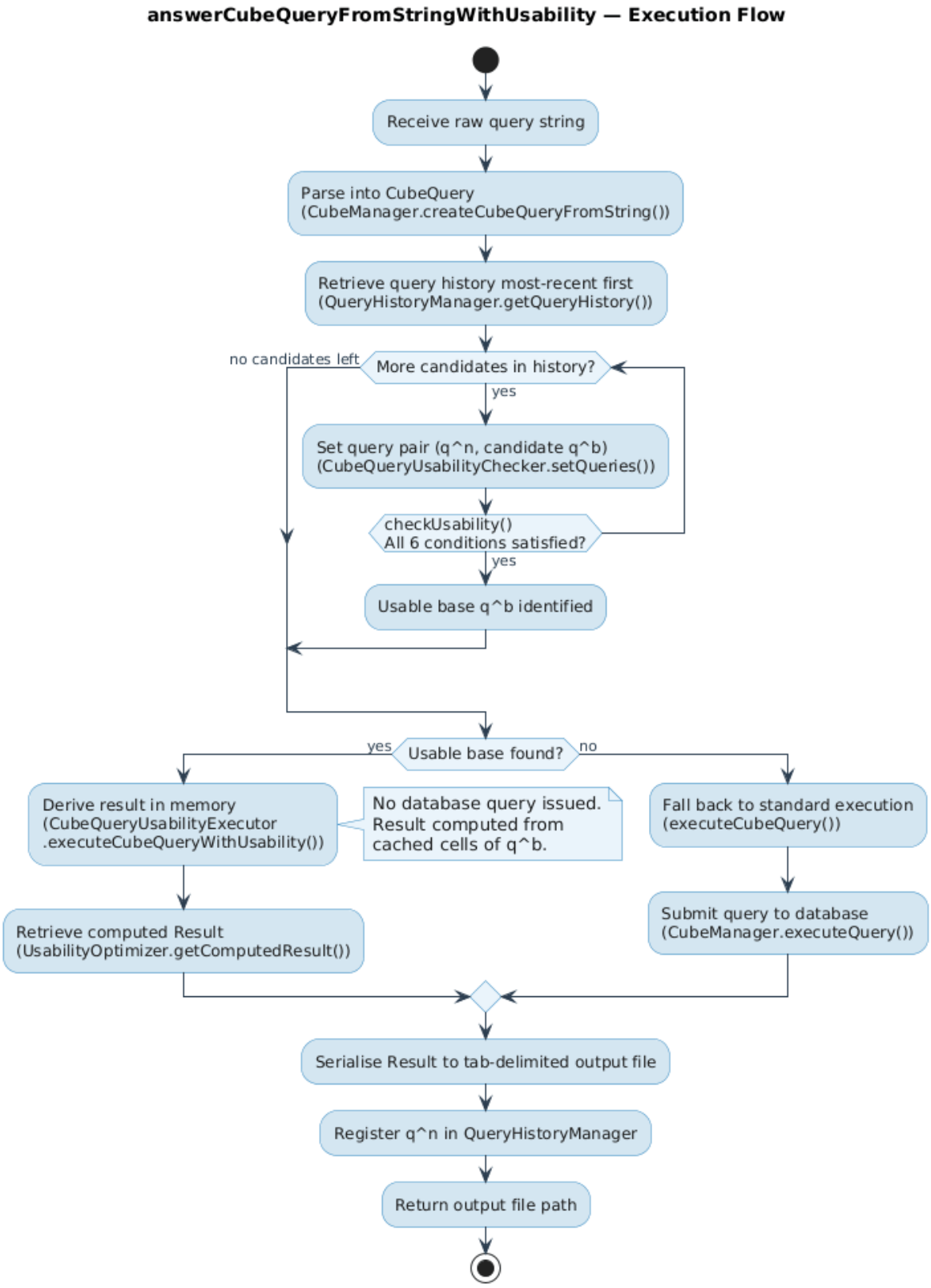


Figure 17: answerCubeQueryFromStringWithUsability: execution flow, from receiving the raw query string to returning the output

5.4.3 Result Derivation Algorithm

When a usable base q^b has been identified, the executor derives the result of q^n through the following six-step algorithm:

Step 1 - Index Construction. A mapping from dimension name to column index in q^b's result cells is constructed, enabling efficient lookup of a cell's member value for any given dimension. Simultaneously, maps from dimension name to sigma level name and grouper level name are built for both q^n and q^b.

Step 2 - Sigma Filter Expansion. The sigma filter of q^n (the set of permitted member values per dimension) is expanded to operate at q^b's grouper level. When q^n's sigma level is coarser than q^b's grouper level, for instance, when q^n selects at the year level while q^b's cells are at the month level, the permitted values are rolled down via *rollDownValuesToLevel(),* which queries the dimension table to enumerate all finer-level members that fall within the coarser selection.

Step 3 - Cell Filtering. Each cell in q^b's cached result is tested against the expanded sigma filter. A cell survives if, for every dimension in the filter, its member value is an element of the permitted set. The surviving cells form the subset of q^b's result that is relevant to q^n.

Step 4 - Roll-Up Lookup Construction. When q^n's grouper level is coarser than q^b's grouper level (as permitted by Condition 5), the surviving cells must be rolled up before aggregation. For each such dimension, a lookup map from fine-level values to coarse-level values is pre-computed via a single SQL query against the dimension table, avoiding repeated database calls within the inner loop.

Step 5 - Grouping and Aggregation. The surviving cells are grouped by their member values at q^n's grouper levels, with roll-up applied as needed using the lookup maps from Step 4. Within each group, the aggregate measures are combined

according to their respective functions: accumulated for SUM and COUNT, and compared for MIN and MAX. The count of underlying detailed cells is also summed per group.

Step 6 - Result Construction. A new *Result* object is constructed from the grouped data. Column names and labels are populated from q^n's gamma expressions, with dimension column names resolved to their database attribute names via *resolveLevelColumn().* A *measure* column and a *countOfDetailedCells* column are appended. The *result* array (a two-dimensional string matrix with column names at row 0, labels at row 1, and data from row 2 and on) is assembled and attached to the result.

5.5. Integration into the Existing Architecture

The usability subsystem was designed to integrate into the Delian Cube Engine with minimal disruption to the existing architecture. The key integration points are as follow:

- **QueryExecutionManager** acquires a static reference to the *UsabilityOptimizer* singleton at class load time. The new method *answerCubeQueryFromStringWithUsability()* is co-located with the existing *answerCubeQueryFromString(),* sharing the same parsing step and reusing the existing *executeCubeQuery()* as its fallback, thereby avoiding code duplication.

- **SessionQueryProcessorEngine** exposes the new method as an RMI accessible operation, registering the command alias *"answer_from_string_with_usability"* in the router of *QueryExecutionManager* alongside the pre-existing command aliases.

- **QueryHistoryManager** is used unmodified. The usability subsystem reads from it during the base search and writes to it upon successful execution, exactly as the baseline path does.

- **The cubemanager.usability package** encapsulates all new logic cleanly, with no modifications required to *CubeManager*, *CubeBase*, *CubeQuery*, or *Result*.

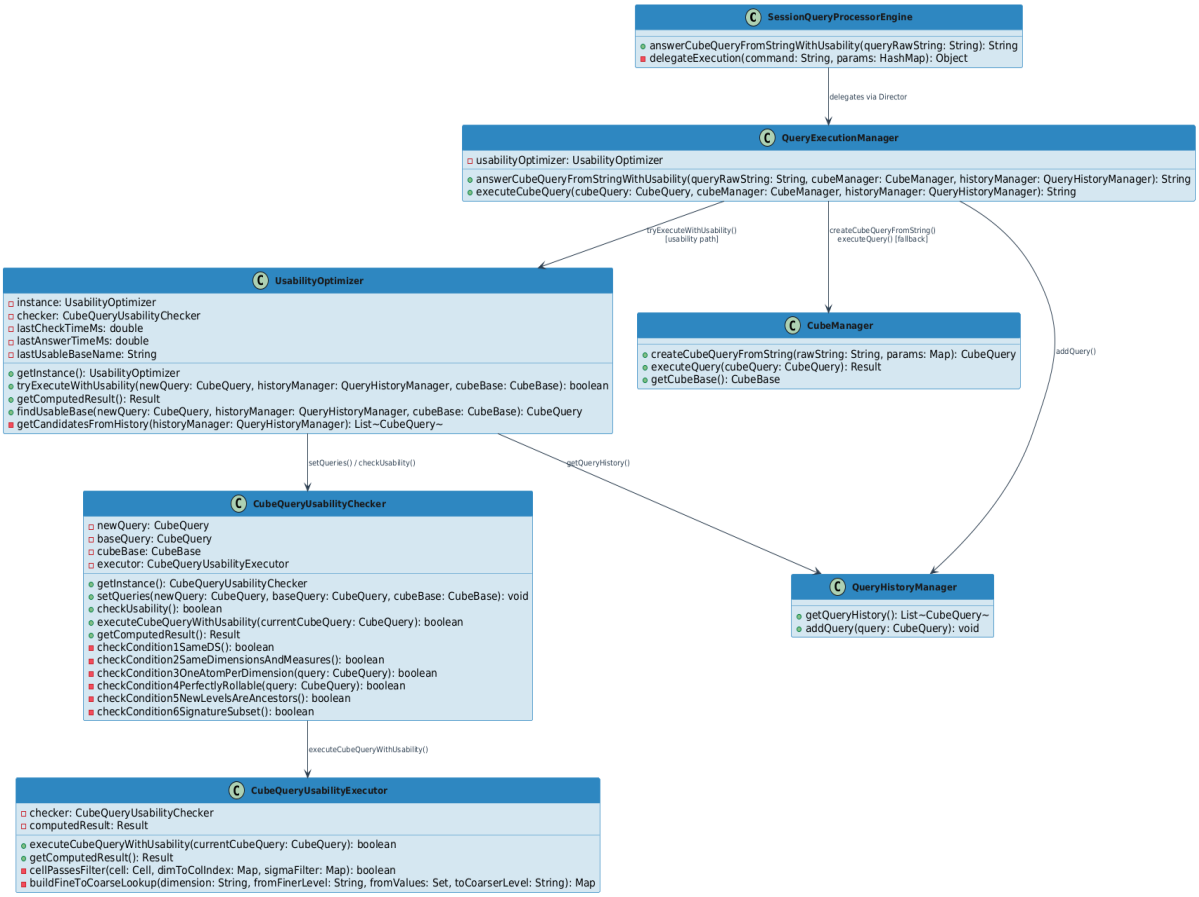


Figure 18: Class diagram of Delian after usability implementation

5.6. Summary

This chapter has described the implementation of the query usability mechanism into the Delian Cube Engine Environment. Prior to this work, every cube query submitted to the engine was executed directly over the underlying database, with no mechanism to exploit the results of previously executed queries. The baseline

execution path followed a straightforward parse-execute-output pipeline attached to *answerCubeQueryFromString()*.

The usability subsystem introduced in this thesis extends this pipeline with a history-aware optimization layer. The three new classes, *UsabilityOptimizer*, *CubeQueryUsabilityChecker*, and *CubeQueryUsabilityExecutor*, implement the theoretical conditions of Theorem 9.1 [Vas23], enabling the engine to derive the result of a new query in memory from a cached base query when the predescribed six structural and hierarchical conditions are satisfied. The new execution flow, exposed through *answerCubeQueryFromStringWithUsability(),* is fully backward-compatible: when no usable base is found, execution falls back to the standard database execution path, guaranteeing correctness regardless of session state.

The implementation adheres to the existing architectural conventions of the Delian Cube Engine, the Director-based dispatch, the manager layer, the Singleton pattern for shared services, and introduces no breaking changes to any pre-existing class.

The result is a self-contained, well-isolated subsystem that enhances the engine's query processing capabilities while preserving the integrity and maintainability of the existing codebase.

CHAPTER 6

EXPERIMENTS

This section presents an experimental evaluation of the proposed theory and algorithm, focusing on their execution time under various data settings and query workload scenarios.

The experiments are conducted using Delian Cubes system [Del25], which is a cube query tool developed to serve as a cube query answering engine. The implementation of this thesis extended the usage of the Delian Cube to support the usability workflow, as described in the previous sections.

All algorithms and testing experiments were implemented in Java within the Delian Cube environment, while dedicated server was utilized as the backend database system. The experiments evaluating the usability-based query execution implementation were run on a dedicated server provisioned in 2026 and running Ubuntu 24.04. The machine is equipped with an Intel Core i9-14900 processor, 256 GB of RAM, and a storage configuration combining a 10 TB HDD with a 256 GB SSD, complemented by an MSI GeForce RTX 5090 GPU with 32 GB of video memory; the server is reachable on the internal network at IP address 10.7.3.177.

6.1. Experimental Setup

6.1.1. Experimental Goal and Evaluation Metrics

The experimental evaluation aims to investigate the practical performance of the cube query usability framework for multidimensional query reuse. While the theoretical model determines whether a previously executed query can answer a new question,

and how it can answer it, if so, the experiments that follow focus on evaluating the efficiency and scalability of the proposed approach under different workload conditions.

The evaluation examines several factors that may influence the effectiveness of usability-based query execution. Factors that may have an effect on usability include the size of the underlying dataset, the computational overhead introduced by usability checking, the percentage of covered queries within a workload, the size of the query history, the position of the reusable queries within that history, the rate at which usability coverage evolves as a query history accumulates over time, and the structural complexity of the queries themselves, in terms of the number of grouping dimensions and selection conditions involved. Through these experiments the goal is to examine and determine whether query usability can provide performance benefits while maintaining acceptable execution overhead.

In the experiments that follow, a query is labeled as covered if, at the time it is submitted, a base query already exists in the session history that satisfies the six usability conditions with respect to it, and can therefore answer it without a new database execution. The base query is called usable (or reusable). Conversely, a query is non-covered if no such qualifying base query is present in the history, in which case it must be answered through direct database execution regardless of the usability mechanism being enabled. Note that this is a property of the incoming query relative to a specific history and session, rather than an inherent property of the query itself; the same query may be covered in one session, where a suitable base query precedes it, and non-covered in another, where it does not. This notion of reusability is what determines the coverage percentage referenced throughout the experimental workloads (e.g., Sections 6.2.3–6.2.4).

6.1.2. Competitor

We compare usability execution against regular direct database execution. Since no prior implementation of cube query usability exists for direct comparison, the

baseline against which the proposed mechanism is evaluated is the standard, unmodified execution path of the Delian Cube Engine, i.e., direct database query execution with no usability checking or query reuse. In this baseline, every query in a workload is parsed and executed independently against the underlying database, regardless of whether an equivalent or related query has been previously answered. This plain execution mode therefore serves as the reference point throughout the experiments, allowing the performance of usability-enabled execution to be expressed as a relative improvement (or overhead) over conventional, reuse-free query answering.

6.1.3. Datasets

The **pkdd99_star** database is a star-schema rendering of the PKDD'99 bank financial dataset, including four dimension tables: *account*, *date*, *status*, *payment_reason* and two fact tables: *loan* and *orders*.

The identical structure is reused unchanged across the **pkdd99_star_100K**, **_1M**, **_10M**, and **_100M** variants, which differ from the base database only in the target database name and most importantly in the *volume* of fact data they hold for scalability testing.

At the center of this schema is the loan fact table, which records one row per bank loan contract and captures the core lending measure **amount** alongside duration and payments, while foreign keys tie each loan to the account that holds it, the calendar day on which it was granted, and its current contract status.

These three foreign keys are exposed analytically through the loan_cube OLAP cube defined in *loan.ini*, which sums the measure *amount* and lets it be sliced and rolled up along three conformed dimensions:

- **account_dim**, with its hierarchy from individual account up through district_name and region to the grand total All_account

- **date_dim**, with its calendar hierarchy from SK_day up through day, month, and year to All_date
- **status_dim**, whose flat hierarchy of SK_status to status to All_status reflects the five possible loan states (other, finished-unpaid, finished-ok, running-in-debt, running-ok).

The loan cube allows questions such as total or average loan amount, duration, and payments to be answered at any combination of account/district/region, day/month/year, and loan-status granularity, while the sibling orders_cube reuses the same account_dim together with a separate reason_dim to analyze standing payment orders, giving the schema two complementary fact perspectives (lending and payments) over a shared customer/geography dimension.

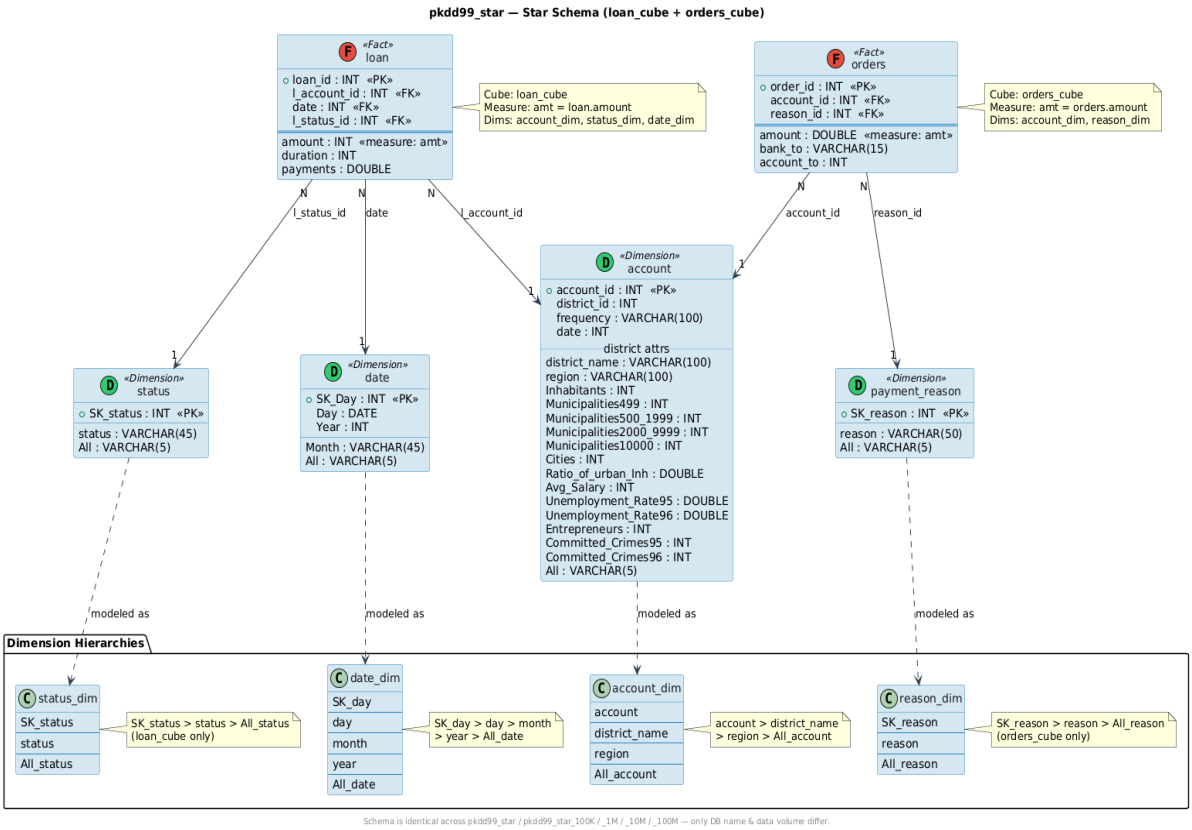


Figure 19: pkdd99_star star schema (loan_cube and orders_cube), showing the fact tables, dimension tables, and their conformed dimension hierarchies

6.1.4. Measurement methodology

To safeguard against transient system noise (e.g., OS scheduling, JVM warm-up, MySQL caching effects) skewing any individual measurement, every batch and session in every experiment described below was executed five times under identical conditions, and the timing values reported for each experiment correspond to the arithmetic mean of these five independent repetitions rather than to any single run.

6.2. Effect of Usability on Performance

6.2.1. Effect of Data Scaling on Usability Performance

The first experiment investigates the scalability of the usability mechanism with respect to the size of the underlying dataset. The objective is to evaluate whether the benefits of query reuse become more (or less) significant as the amount of stored data increases.

The experiment consists of multiple query execution sessions, where each session is performed over a different dataset size. For each dataset a fixed workload of identical queries is executed, allowing the comparison of execution behavior under different data volumes, while keeping the query workload and schema constant.

By varying only the dataset size, this experiment evaluates whether the usability approach remains efficient as the cost of finding, accessing and processing the original data increases due to higher volume.

The expected outcome is that larger datasets could provide greater potential for optimization, since avoiding computation over large amounts of data accumulates and becomes increasingly beneficial.

Datasets:

The datasets used in this experiment are based on the pkdd99_star schema and differ only in their size. Five dataset instances were used, containing approximately 1,000 rows, 100 thousand rows, 1 million rows, 10 million rows, and 100 million rows, respectively. These datasets are denoted as pkdd99_star, pkdd99_star_100k, pkdd99_star_1m, pkdd99_star_10m, and pkdd99_star_100m.

Query Workloads:

The workload consists of a single fixed batch of 20 OLAP queries over the loan_cube, executed twice — once configured so that 20% of the batch (4 queries) is covered with respect to the query history, and once so that 30% of the batch (6 queries) is covered. Each of these two batches is run unmodified against all five physical databases (pkdd99_star, pkdd99_star_100K, pkdd99_star_1M, pkdd99_star_10M, pkdd99_star_100M), which share an identical schema and differ only in the cardinality of the loan and orders fact tables, allowing the fact-table size to be isolated as the sole independent variable across runs.

Each experimental session was executed five times, and both the total execution time and the average execution time per query were collected. The measurements were performed for two execution strategies: direct query execution against the database and execution using the proposed usability-based query reuse mechanism. The collected results allow a comparison of the performance behavior of both approaches under increasing data volumes.

The following section presents the experimental results and analyzes the impact of dataset scaling on the effectiveness of the usability framework.

Table 1: Scalability experiment with 20% usability coverage

Dataset	SessionID	Total Queries	AVG – Total Direct Time (ms)	AVG – Total Usability Time (ms)	Avg Direct Time (ms/query)	Avg Usability Time (ms/query)	AVG – Overall speed-up (direct / usability)	Usability Hits/Coverage Number	Usability Hits/Coverage Percent (%)
pkdd99_star	1	20	14,76	17,294	0,74	0,86	0,82	4	20
pkdd99_star_100k	2	20	54,512	21,63	2,73	1,08	2,56	4	20
pkdd99_star1M	3	20	444,312	51,164	22,22	2,56	8,75	4	20
pkdd99_star_10M	4	20	4310,48	345,368	215,52	17,27	12,48	4	20
pkdd99_star_100M	5	20	179326	147095,5	8966,30	7354,77	1,22	4	20

Table 2: Scalability experiment with 30% usability coverage

Dataset	SessionID	Total Queries	AVG – Total Direct Time (ms)	AVG – Total Usability Time (ms)	Aug Direct Time (ms/query)	Aug Usability Time (ms/query)	AVG – Overall speed-up (direct / usability)	Usability Hits/Coverage Number	Usability Hits/Coverage Percent (%)
pkdd99_star	1	20	13,334	19,27	0,67	0,96	0,71	6	30
pkdd99_star_100k	2	20	53,526	19,294	2,68	0,96	2,79	6	30
pkdd99_star1M	3	20	432,584	40,55	21,63	2,03	10,85	6	30
pkdd99_star_10M	4	20	4219,448	257,022	210,97	12,85	16,45	6	30
pkdd99_star_100M	5	20	177748,8	145618,1	8887,44	7280,90	1,22	6	30

From the tables above we obtain the following results.

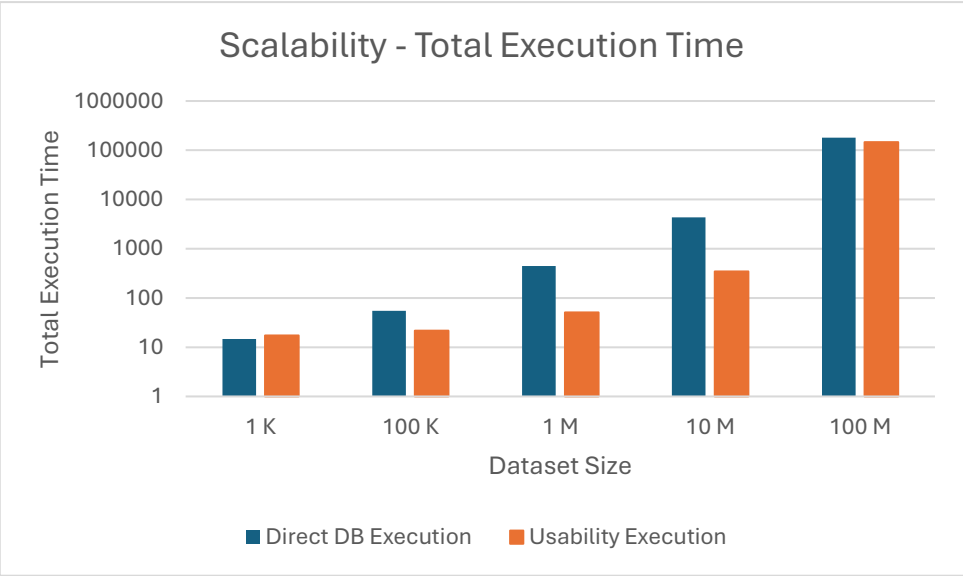


Figure 20: Effect of scalability in total execution time with 20% usability coverage

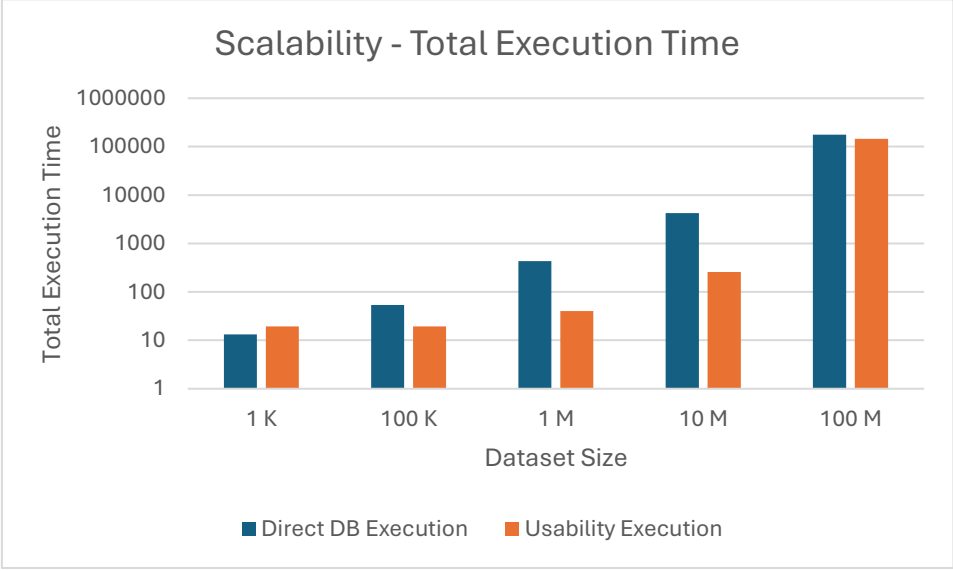


Figure 21: Effect of scalability in total execution time with 30% usability coverage

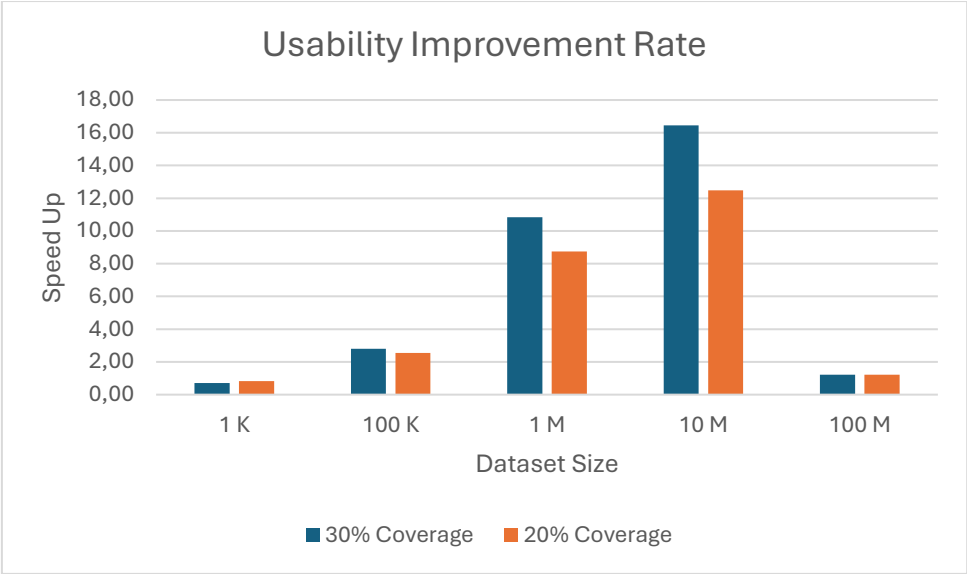


Figure 22: Usability Improvement Rate among various database sizes

Conclusion – observations

1) Usability almost always seems to outperform the direct query execution, and in 1M and 10M rows it reaches over 15 times faster execution than the direct database execution (including the non usable queries) (figure 2)
2) The only exception is the smallest available dataset, which usability is slower than direct database execution, but still the difference is negligible (approximately less than 6 milliseconds).
3) There is a noticeable threshold though when we approach 100m rows of data, the usability is still faster just with a lower rate though. The phenomenon could be explained by the fact that as the database size grows but the usable queries remain the same, the speed up ratio will eventually reach the limit of 1, because the amount of time it takes for the non-covered queries to be executed becomes so extremely large, that it becomes impossible for the few usable queries to make a huge difference and greatly impact the total execution time. In simple words, the cost of the non-usable queries in a workload eventually dominates the total execution time, regardless of how much faster the covered queries are answered.

6.2.2. Effect of Usability Check Overhead on Usability Performance

The second experiment focuses on breaking down the query execution time components and analyzes the contribution of those different components involved in the implemented usability-based query execution.

The main purpose is to understand the performance overhead introduced by the usability mechanism, observe any “bottleneck” and determine whether the additional processing required for query analysis affects (and how much, if so) the overall execution time.

The execution is decomposed into three main components:

1. **Usability checking time:** The time required to determine whether a previously executed query can answer the new query. Checking for usability is performed for every new query.
2. **Database query execution time:** the time required to execute the query directly against the database when reuse is not possible.
3. **Usability-based answer generation time:** the time required to derive the requested result from a previously stored usable query in case usability is applicable.

This experiment evaluates whether the cost of checking query usability adds up significantly to the overall time or remains negligible compared to the cost of executing analytical queries directly, and whether successful query reuse provides measurable performance improvements.

Query workload:

The workload is composed of 10 independent batches of 20 queries each, where each batch is constructed with a different ratio of usable to non-usable queries. The experiment was run against the pkdd99_star_1M dataset.

Table 3: Decomposed query execution time for each of 10 sessions

SessionI	Total Queries	Usability Hits/Coverage Number	Usability Hits/Coverage Percent	Total Usability Check Time (ms)	Avg Usability Check Time (ms / query)	Total DB execution Time (ms)	Avg DB execution Time (ms / query)	Total Usability Answer Time (ms)	Avg Usability Answer Time (ms / query)	Usability Check Overhead (non usable queries) (%)
1	20	1	5.0%	3,56	0,18	694,32	36,54	0,02	0,02	0,51
2	20	2	10%	0,21	0,01	1652,91	91,83	0,02	0,01	0,01
3	20	2	10%	12,70	0,64	2037,74	113,21	2,86	1,43	0,62
4	20	2	10%	6,13	0,31	2229,19	123,84	0,02	0,01	0,27
5	20	1	5%	3,09	0,15	512,31	26,96	0,02	0,02	0,60
6	20	2	15%	1,26	0,06	1138,87	63,27	0,03	0,01	0,11
7	20	2	10%	7,60	0,38	1595,43	88,64	3,20	1,60	0,48
8	20	4	20%	30,38	1,52	551,97	34,50	18,65	4,66	5,50
9	20	1	5%	7,65	0,38	3983,59	209,66	3,49	3,49	0,19
10	20	2	15%	12,59	0,63	1884,18	104,68	8,77	4,38	0,67
AVG					0,43		89,31		1,56	0,90

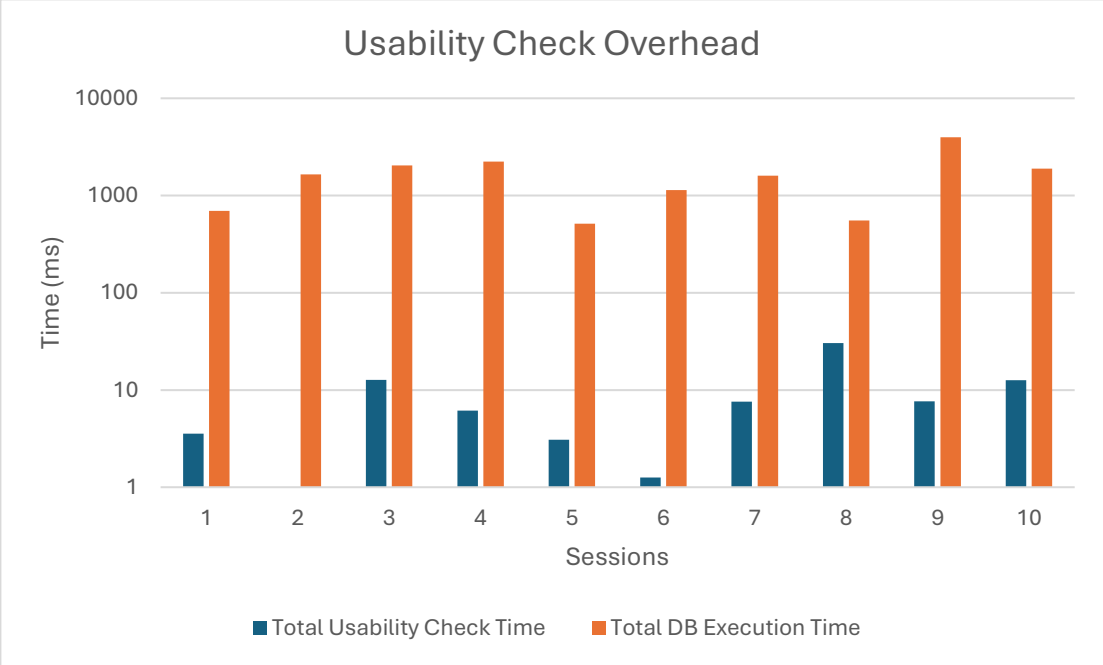


Figure 23: Usability Check Overhead compared to DB execution time

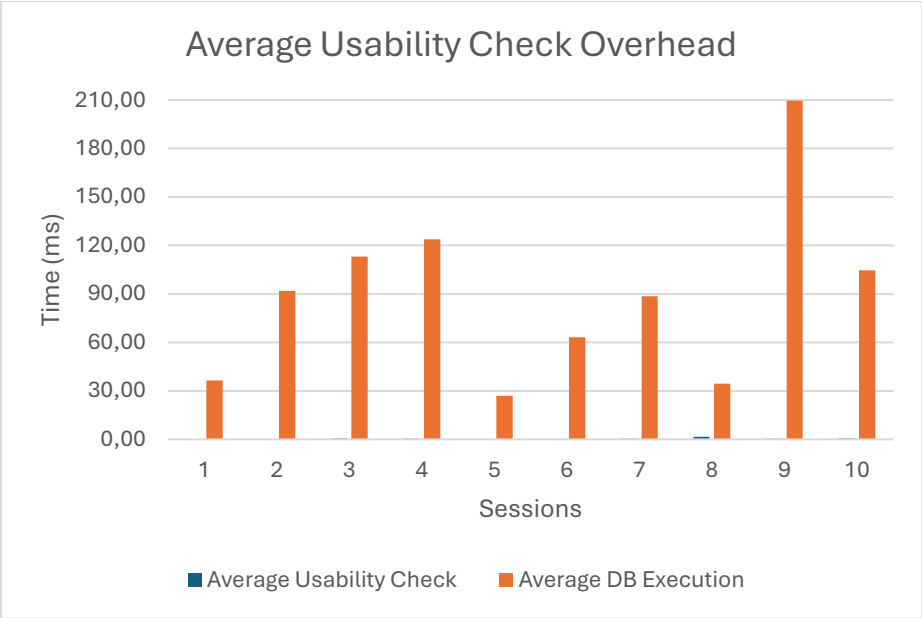


Figure 24: Average Usability Check Overhead compared to average DB execution time

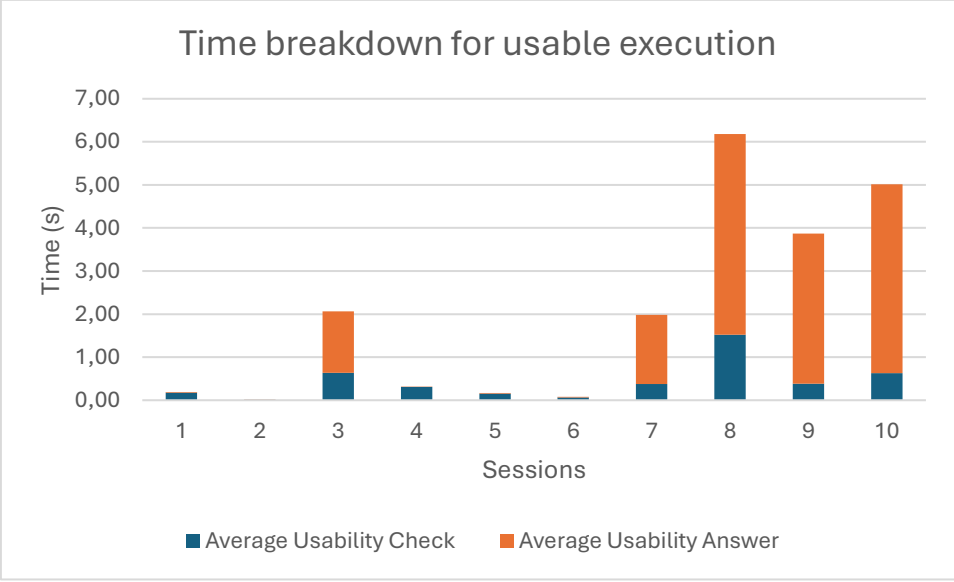


Figure 25: Comparison between usability check and usability execution time for usable queries

Conclusion – observations

Without usability we would directly execute the queries to the database. When implementing usability we had to check first. But how much extra costs does this add up? And is it worth it?

This experiment proves that it takes negligible cost to check for usability as it add up less than 1% of time overhead to the total direct execution time (for non usable queries).

Even in the worst case scenario with no usable queries found and every query is executed directly it costs almost nothing to check anyway.
Usability answer time could be 3000 times faster than actual database query execution time, so it is clear that usability is preferred and the gain outweighs the tiny checking cost.

6.2.3. Effect of Usability Queries Coverage on a Query Session

The third experiment evaluates how the number of covered (and usable) queries within a workload affects the overall execution performance of a query session.

A query session on this experiment consists of a fixed number of cube queries, while the proportion of queries that can be answered through usability is gradually increased. The experiment therefore compares workloads with different levels of usability coverage, ranging from mostly non-covered queries to workloads containing a large percentage of covered ones.

The purpose is also to determine the point at which cube query reuse begins to significantly increase performance. It evaluates whether increasing the availability of reusable queries leads to a corresponding reduction in total execution time.

Query workload:

The workload consists of 10 sequential sessions, each containing the same 16 queries, in which the proportion of usable and covered queries is progressively increased from session to session: session 1 contains 1 covered query (6.25% coverage), and in each subsequent session one additional non-covered query is replaced by a covered one, until session 10 contains 10 usable queries out of 16 (62.5% coverage). This experiment run against the pkdd99_star_1M dataset.

Table 4: Usability session times with different coverage percentages

SessionID	Total Queries	Usability Hits/Coverage Number	Usability Hits/Coverage Percent	Total Direct Time (ms)	Avg Direct Time (ms/query)	Total Usability Time (ms)	Avg Usability Time (ms/query)	Overall speed-up (direct / usability)
1	16	1	6,25	442,65	27,67	43,78	2,74	10,11
2	16	2	12,5	438,18	27,39	42,14	2,63	10,40
3	16	3	18,75	426,97	26,69	32,87	2,05	12,99
4	16	4	25	423,20	26,45	31,72	1,98	13,34
5	16	5	31,25	422,77	26,42	34,44	2,15	12,28
6	16	6	37,5	416,55	26,03	22,34	1,40	18,64
7	16	7	43,75	407,38	25,46	23,73	1,48	17,16
8	16	8	50	407,04	25,44	29,70	1,86	13,71
9	16	9	56,25	408,57	25,54	40,54	2,53	10,08
10	16	10	62,5	740,13	46,26	49,97	3,12	14,81

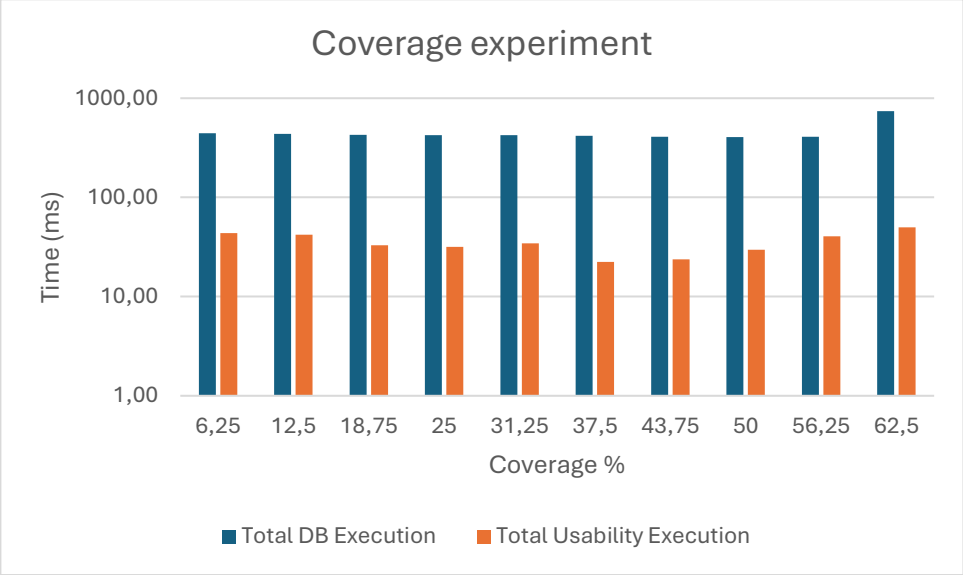


Figure 26: Comparison of total direct DB usability time and total usability time in various coverage scenarios

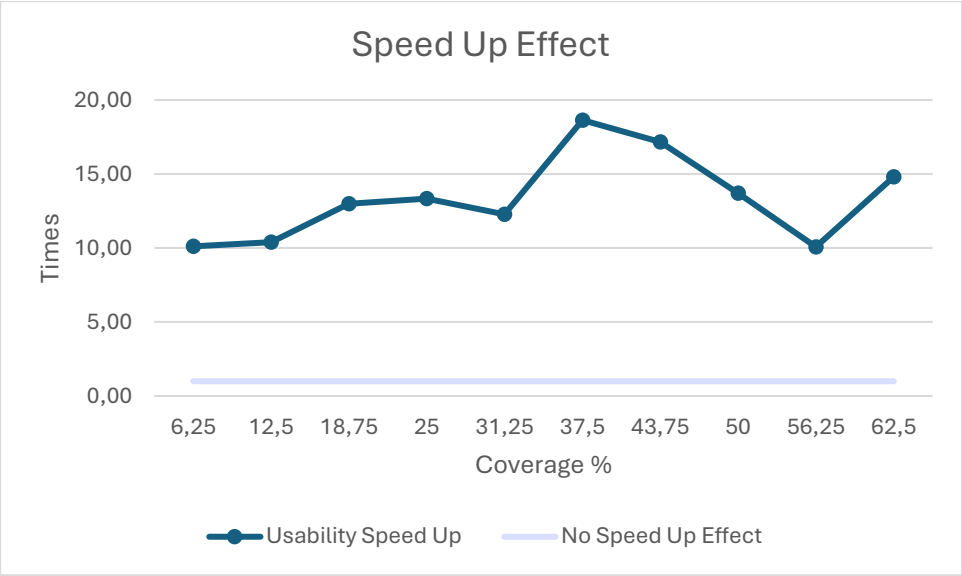


Figure 27: Usability time Speed up across various coverage scenarios

Conclusion – observations

The usability execution time produces continuously over 10 times faster total query execution time against the direct DB execution time (which is 1 magnitude) which proves that manipulation of the coverage percentage does affect the usability execution time. In contrast, we observe that generally higher availability in reusable queries lead to even better usability performance.

6.2.4. Effect of Query History Size on Usability Performance

The fourth experiment examines whether the number of previously executed queries stored in history influences the performance of usability.

The query history represents the collection of past executed queries that the system can search through in order to find reusable results. In this experiment, the usability coverage is kept constant at 10% and 40% levels while the size of query history is gradually increased.

By evaluating different history sizes, this experiment investigates whether searching through a larger collection of previously executed queries perhaps stalls and introduces additional overhead or affects in any way the ability of the system to efficiently identify usable base computations.

Query workloads:

The workload consists of 5 batches, each maintaining a fixed usability coverage of 10% but differing in overall size: the first batch contains 10 queries with 1 usable query, the second contains 20 queries with 2 usable queries, and so on up to the fifth batch, which contains 50 queries with 5 usable queries, so that the absolute number of usable queries scales together with the batch size while the coverage ratio is held constant. The new and base queries were relatively uniformly distributed within the history. The experiment run against the pkdd99_star_1M dataset.

Table 5: Effect of query history size on usability performance, with coverage held constant at 10%

SessionID	Total Queries	Total Direct Time (ms)	Total Usability Time (ms)	Avg Direct Time (ms/query)	Avg Usability Time (ms/query)	Overall speed-up (direct / usability)	Usability Hits/Coverage Number	Usability Hits/Coverage Percent
1	10	16,072	15,902	1,61	1,59	1,01	1	10%
2	20	4,514	5,574	0,23	0,28	0,81	2	10%
3	30	20,452	20,686	0,68	0,69	0,99	3	10%
4	40	19,67	26,068	0,49	0,65	0,75	4	10%
5	50	46,446	53,224	0,93	1,06	0,87	5	10%

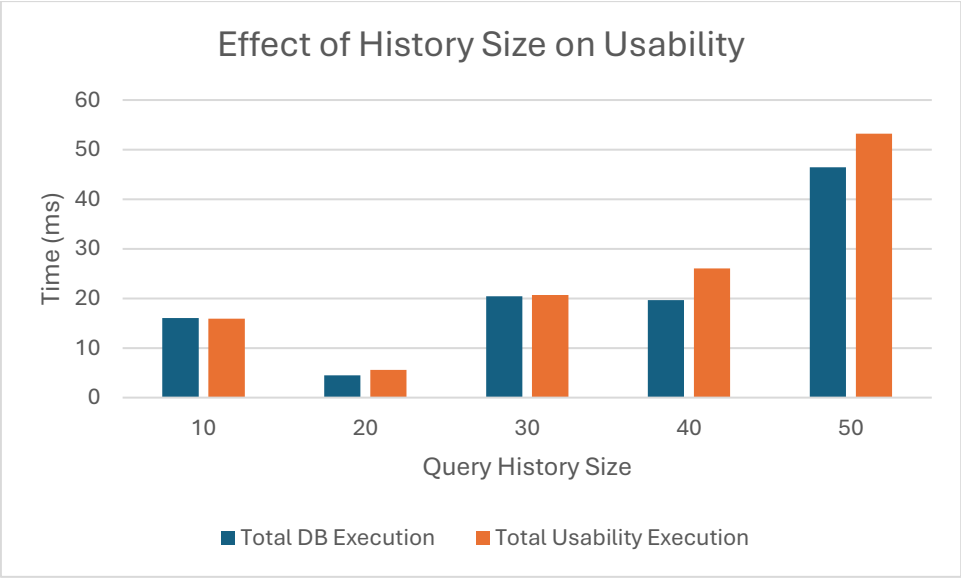


Figure 28: Effect of History Size on Usability at 10% coverage

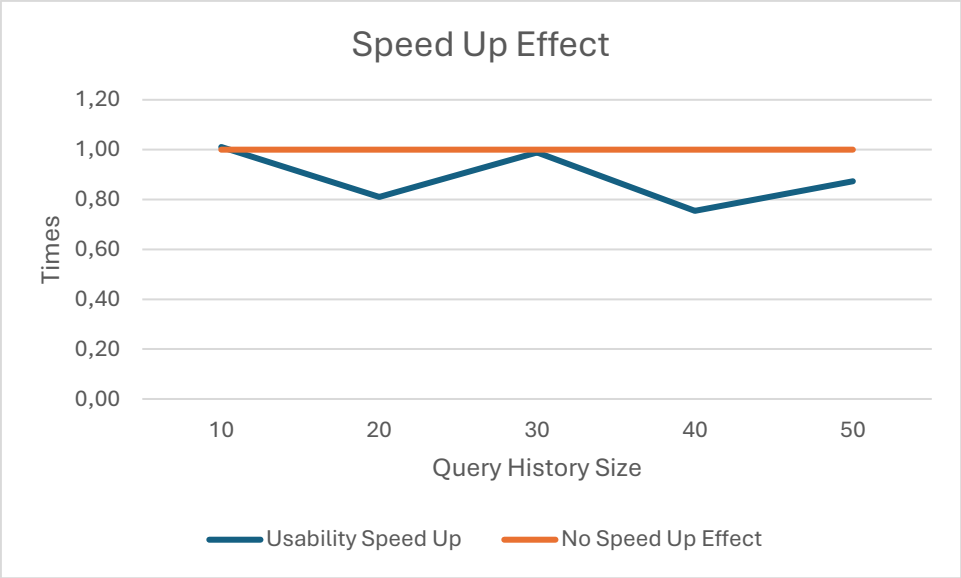


Figure 29: Speed-up effect as history size increases

To examine whether this small overhead at low coverage persists at a higher coverage ratio, the same experiment was repeated with the usability coverage held constant at 40% instead of 10%, again across the same five history sizes (10, 20, 30, 40, and 50 queries), so that the number of usable queries in each batch scales proportionally to four times of its 10%-coverage counterpart (4, 8, 12, 16, and 20 usable queries respectively, out of 10, 20, 30, 40, and 50 total queries).

Table 6: Effect of query history size on usability performance, with coverage held constant at 40%

SessionID	Total Queries	Total Direct Time (ms)	Total Usability Time (ms)	Avg Direct Time (ms/query)	Avg Usability Time (ms/query)	Overall speed-up (direct / usability)	Usability Hits/Coverage Number	Usability Hits/Coverage Percent
1	10	828,26	418,65	82,83	41,87	1,89	4	40%
2	20	813,64	430,36	40,68	21,52	1,91	8	40%
3	30	424,99	26,73	14,17	0,89	15,90	12	40%
4	40	2641,05	968,34	66,03	24,21	2,73	16	40%
5	50	3793,42	1398,18	75,87	27,96	2,71	20	40%

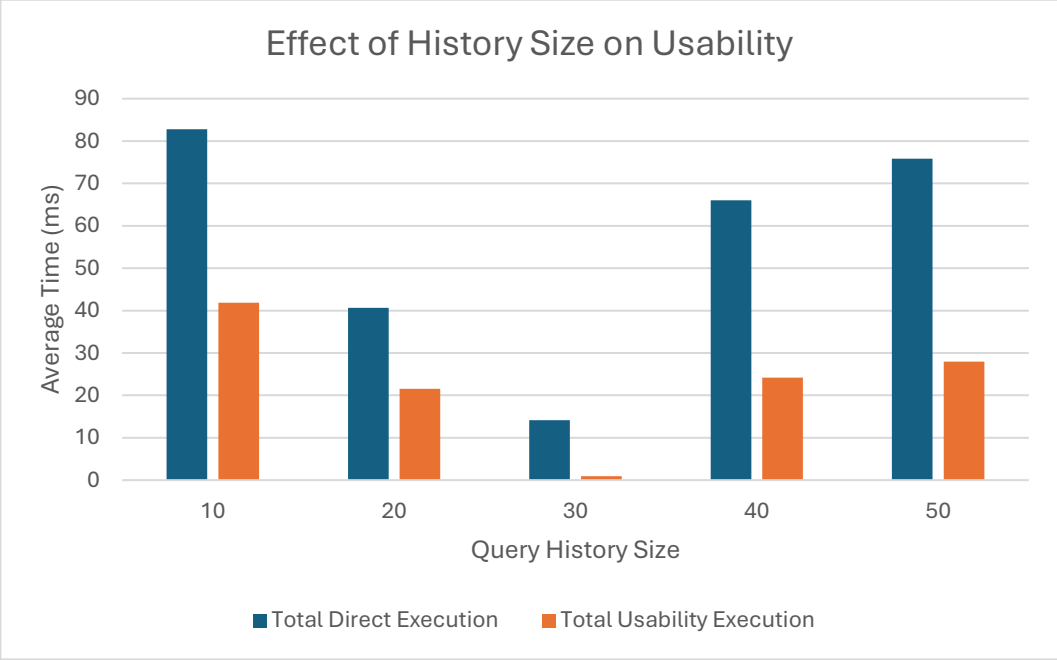


Figure 30: Effect of history size on usability at 40% coverage

Conclusion – observations

A small drop in performance is noticeable as the history size increases when coverage is held at 10%. At this coverage level, the usability speed-up falls just below 1, meaning that usability execution is occasionally slightly slower than direct database execution. However, this drop wavers between 0.85 and 1, so usability execution is never more than roughly 0.85 times slower than direct execution even in the worst case observed.

When the same experiment is repeated at a higher, 40% coverage level, this picture changes substantially: usability execution is faster than direct execution at every history size tested, with the speed-up ranging from 1.89x at the smallest history (10 queries) up to 15.90x at a history of 30 queries, and settling around 2.7x at the two largest history sizes tested (40 and 50 queries). This indicates that the small overhead observed at 10% coverage is specific to the low-coverage regime, where the fixed cost of searching the history for a usable match is not compensated by the very few available actual reuse opportunities. Once coverage rises to 40%, that same search cost is offset by a substantially larger number of queries that benefit from reuse, and the net effect of growing the history is strongly positive rather than mildly negative.

Taken together, the two coverage levels suggest that the effect of history size on usability performance is not fixed, but depends on the coverage ratio of the workload: a larger history is essentially neutral (a barely measurable slowdown) when few queries in it are reusable, and increasingly beneficial as the proportion of reusable queries grows.

6.2.5. Effect of Usability Query Position on Query History

The fifth experiment investigates whether the position of a usable(reusable) query within the query history affects the execution time.

Since the usability mechanism searches the history to identify whether an existing query can answer a new question, the location of the matching query may increase the number of comparisons required before a usable query is found and influence the total time. Intuitively, A reusable query appearing early in the history requires fewer checks compared to one located towards the end of the history.

To evaluate this effect multiple query sessions with a fixed history size are executed while changing only the position of the reusable query. The experiment

examines whether increasing the distance between the new query and its matching base (historical) query results in additional execution effect, and if so, to what extend.

Query workload:

The workload consists of 6 sessions that all share the same fixed history of 50 queries and the same base usable query placed at the first position of that history. What varies across sessions is solely the position of the new incoming query (q^n) relative to that base query, with the new query placed at the 2nd position in session 1, the 10th in session 2, the 20th in session 3, the 30th in session 4, the 40th in session 5, and the 50th in session 6. This experiment run against the pkdd99_star_1M and pkdd99_star_10M datasets.

Table 7: Effect of usable query position within a fixed 50-query history on 1M dataset

SessionID	Total Queries	position of usable query in history	Total Direct Time (ms)	Total Usability Time (ms)	Avg Direct Time (ms/query)	Avg Usability Time (ms/query)	Overall speed-up (direct / usability)	Usability Hits/Coverage Number	Usability Hits/Coverage Percent
1	50	2	4550,528	4538,384	91,01	90,77	1,00	1	2%
2	50	10	4548,898	4546,892	90,98	90,94	1,00	1	2%
3	50	20	4544,758	4545,286	90,90	90,91	1,00	1	2%
4	50	30	4543,84	4545,782	90,88	90,92	1,00	1	2%
5	50	40	4543,09	4544,06	90,86	90,88	1,00	1	2%
6	50	50	4541,522	4544,848	90,83	90,90	1,00	1	2%

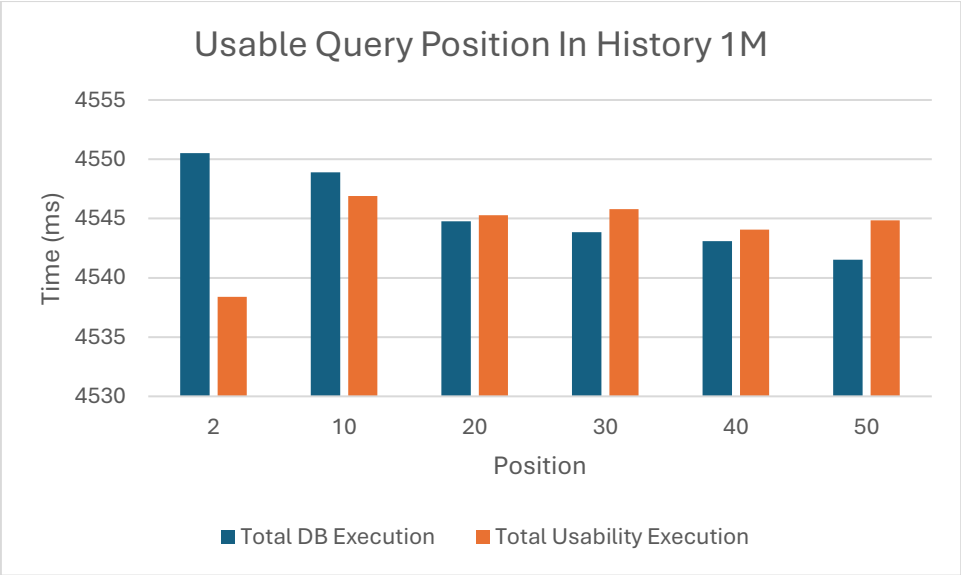


Figure 31: Comparison between usable query position and total usability execution time for 1M

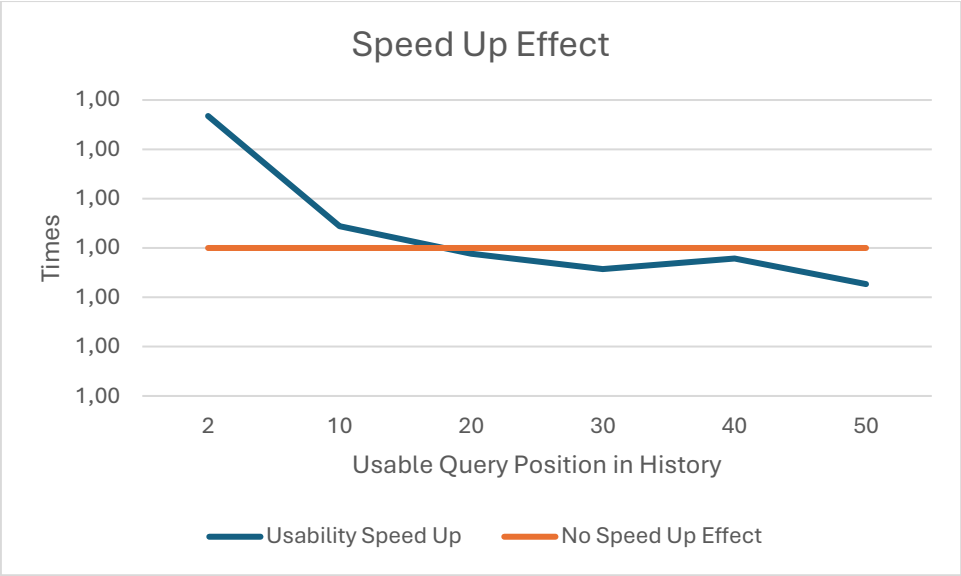


Figure 32: Speed-up effect by usable query position for 1M

To check whether this result still holds once the underlying dataset is an order of magnitude larger, the same six sessions were re-executed unchanged against the pkdd99_star_10M dataset, again with a fixed 50-query history and the same base/new query positions (2, 10, 20, 30, 40, and 50).

Table 8: Effect of usable query position within a fixed 50-query history on 10M dataset

SessionID	Total Queries	position of usable query in history	Total Direct Time (ms)	Total Usability Time (ms)	Avg Direct Time (ms/query)	Avg Usability Time (ms/query)	Overall speed-up (direct / usability)	Usability Hits/Coverage Number	Usability Hits/Coverage Percent
1	50	2	51647,69	51389,06	1032,95	1027,78	1,01	1	2%
2	50	10	51404,54	51363,37	1028,09	1027,27	1,00	1	2%
3	50	20	51397,01	51321,54	1027,94	1026,43	1,00	1	2%
4	50	30	51336,63	51324,97	1026,73	1026,50	1,00	1	2%
5	50	40	51357,82	51353,71	1027,16	1027,07	1,00	1	2%
6	50	50	51530,97	51498,7	1030,62	1029,97	1,00	1	2%

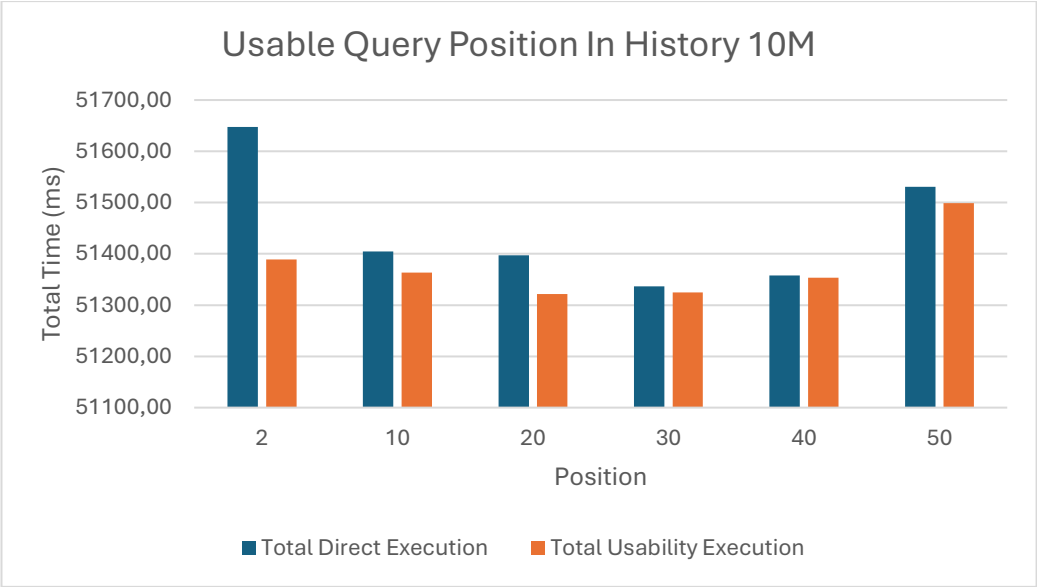


Figure 33: Comparison between usable query position and total usability execution time for 10M

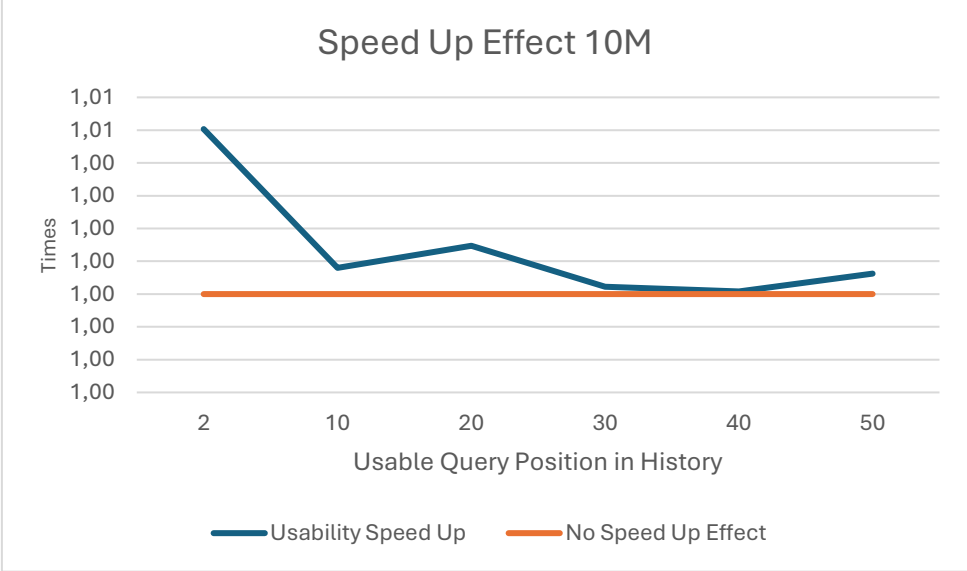


Figure 34: Speed-up effect by usable query position for 10M

At 10M rows, total execution time scales by roughly an order of magnitude relative to the 1M run (around 51.3–51.6 seconds per session instead of 4.5 seconds), as expected for a 10x larger fact table, but the relationship between query position and the usability speed-up is essentially unchanged: the speed-up little above 1 across every position tested, with no noticeable upward or downward trend as the distance between the new query and its matching base query grows

from 2 to 50. In other words, the negligible-distance-effect result observed at 1M scale reproduces at 10M scale as well.

Conclusion – observations

From this experiment we conclude that the usability execution time isn't affected by the distance of new query and its matching base. There is a slight delay after there is more than 20 queries between new and its base, but the delay is negligible (less than 0,4% extra time overhead). Repeating the same experiment on the pkdd99_star_10M dataset confirms that this holds independently of dataset size: even though absolute execution times grow by roughly an order of magnitude at 10M rows, the usability speed-up remains around 1.00x across all tested positions, with no meaningful trend as the distance between base and new query increases. This indicates that the position of the matching query within the history is not a practically significant factor in usability performance, regardless of the scale of the underlying dataset

6.2.6. Effect of Query History Growth on Usability Coverage and Performance

The sixth experiment investigates how the proportion of reusable queries in a workload evolves as the query history accumulates over time. Unlike the coverage experiment of Section 6.2.3, which fixes the size of a single session and varies the share of reusable queries within it, this experiment keeps the workload fixed and instead grows the history incrementally, batch by batch, to observe how usability coverage and total execution time behave as an analytical session matures.

The objective is to determine whether the benefit of query reuse is mostly appreciated early in a session, once a handful of base queries have been cached, or whether coverage continues to climb meaningfully as more and more queries are added to the history, and to quantify the corresponding effect on total execution time as the history grows.

Query workload:

The workload consists of a fixed pool of 100 queries over the loan cube, submitted in five sequential batches of 20 queries each. After every batch is submitted, the cumulative query history (i.e., all batches submitted so far) is used to determine, for the batch just submitted, how many of its 20 queries are usable with respect to that history, and both the direct-execution time and the usability-execution time for the batch are recorded. This is repeated for all five batches, so that the fifth and final measurement reflects coverage and timing behavior once the full history of 100 queries has been accumulated. The experiment was run against the pkdd99_star_1M dataset.

Table 9: Usability coverage and execution time per batch as query history grows

SessionID	Total Queries	Total Direct Time (ms)	Total Usability Time (ms)	Avg Direct Time (ms/query)	Avg Usability Time (ms/query)	Overall speed-up (direct / usability)	Usability Hits/Coverage Number	Usability Hits/Coverage Percent
1	20	1149,58	1140,56	57,48	57,03	1,01	3	15,00
2	40	1151,90	1173,24	28,80	29,33	0,98	7	17,50
3	60	2387,70	1689,07	39,80	28,15	1,41	13	21,67
4	80	5379,32	2958,66	67,24	36,98	1,82	22	27,50
5	100	6768,37	3454,52	67,68	34,55	1,96	30	30,00

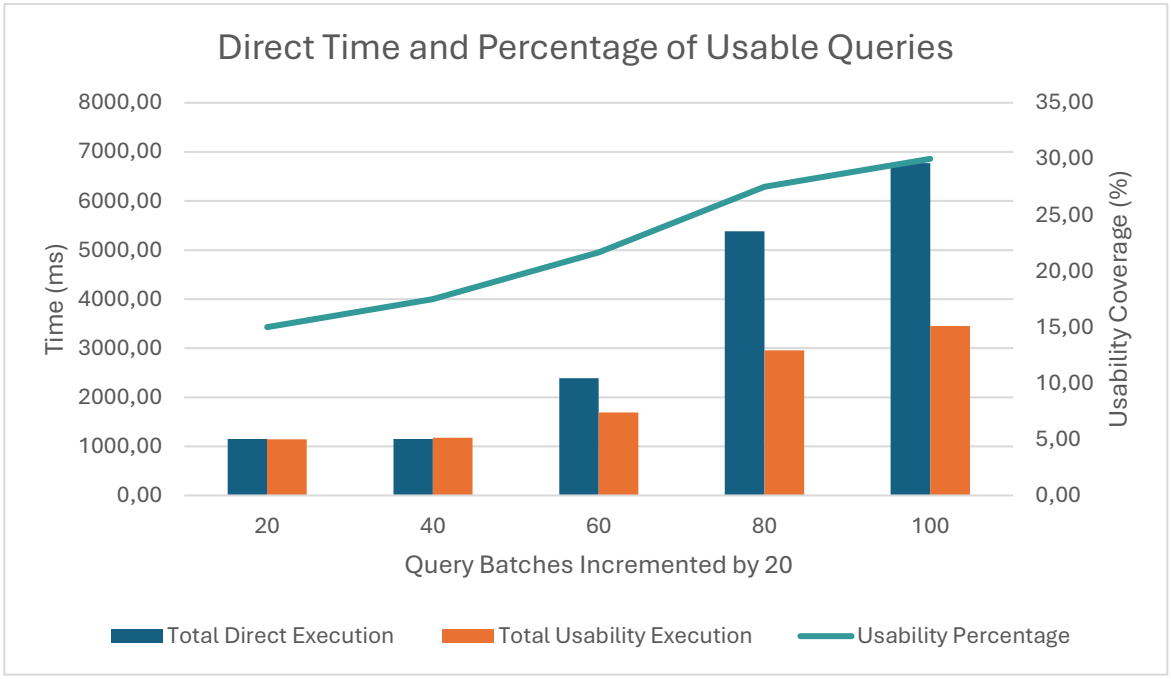


Figure 35: Usability coverage percentage and total time as a function of accumulated history size

Conclusion – observations

As additional batches of queries are introduced into the history, usability coverage rises consistently with each batch, from 15.0% at the first batch (20-query history) to 30.0% at the fifth and final batch (100-query history), exactly doubling over the course of the experiment. This confirms that, for this workload, the pool of candidate base queries available for reuse grows with the history, increasing the likelihood that an incoming query in a later batch finds a usable match, with no sign of the coverage growth saturating within the tested range.

In terms of total execution time, the usability-based approach starts close to parity with direct execution at the first two batches, with a speed-up of 1.01x at batch 1 and a marginal slowdown of 0.98x at batch 2, before becoming consistently and increasingly faster from the third batch onward, reaching 1.41x at batch 3, 1.82x at batch 4, and 1.96x at batch 5.

This pattern indicates that the performance benefit of query reuse may not fully be appreciated early in a session when the history and its coverage are still small, but strengthens steadily as both the history size and the proportion of reusable queries within it increase together.

6.2.7. Effect of Query Complexity (Gamma/Sigma Anatomy) on Usability

The seventh experiment studies how the structural complexity of a query, in terms of the number of grouper (gamma) dimensions and selection (sigma) atoms it involves, affects both the speed-up obtained from usability-based execution and the computational cost of the usability mechanism itself. While the preceding experiments treat all queries in a workload uniformly, this experiment isolates query complexity as an independent factor in order to determine whether more complex

base/new query pairs are systematically faster, slower, or unaffected when answered through usability, compared to simpler pairs.

Three complementary aspects of complexity are examined separately:

- Sigma-fixed, gamma-varying: the selection condition is held structurally equivalent across all tested pairs, while the number of grouper dimensions, and the number of those dimensions that change level between the base and the new query, is progressively increased.
- Gamma-fixed, sigma-varying: the symmetric case, in which the grouping structure is held fixed while the selection condition's complexity is varied.
- Gamma-and-sigma mutually varying: a third workload in which both the grouper dimensions and the selection atoms are changed simultaneously and respectively between the base and the new query at each complexity level, as the two single-factor workloads.

All the query pairs follow a same pattern of progressive difficulty ((1,1) through (3,3)) that is described below on "query workload".

The objective is threefold: first, to determine whether the usability speed-up (execution time using direct database access versus execution time using usability) remains stable, improves, or degrades as query complexity increases; second, to determine whether the computational cost of usability itself, i.e., the time required to check and derive a usable answer, is sensitive to how many gamma dimensions or sigma atoms are involved, independently of whether the overall speed-up is preserved; and third, to determine whether varying both gamma and sigma complexity simultaneously produces an effect that is simply the combination of the two single-factor effects, or whether a distinct, compounded effect emerges when both are varied together.

Query workload:

For the gamma-varying case, six query pair "types" were constructed, of progressively increasing complexity, denoted P1 through P6:

•P1 (1,1): one gamma dimension, one sigma atom.

•P2 (2,1): two gamma dimensions, one of which changes level between the base and the new query (a single-dimension roll-up).
•P3 (2,2): two gamma dimensions, both of which change level between the base and the new query.
•P4 (3,1): three gamma dimensions, one of which changes level.
•P5 (3,2): three gamma dimensions, two of which change level.
•P6 (3,3): three gamma dimensions, all three of which change level.

In each pair type, the base query's sigma is held structurally equivalent to the new query's sigma (same dimensions, same number of atoms), so that the only varying factor between query pairs of different types is the number of gamma dimensions and the number of those dimensions whose grouping level changes between base and new.

The symmetric workload, gamma-fixed and sigma-varying, was constructed following the same progressive (1,1) through (3,3) pattern, holding the grouper expression structurally equivalent across all tested pairs while varying the number of sigma atoms and the number of those atoms that change between the base query's and the new query's selection condition.

A third workload was constructed by applying the same progressive (1,1) through (3,3) complexity pattern to both the gamma and sigma sides of each query pair at once: at each pair type, the same number of gamma dimensions and the corresponding number of sigma atoms change level/value between base and new query simultaneously. All three workloads were executed against the pkdd99_star_1M dataset, with each pair type measured over five repeated rounds and the reported figures corresponding to the arithmetic mean of those five rounds, consistent with the measurement methodology of Section 6.1.4.

Table 10: Usability speed-up and usability execution time for the six gamma-complexity query types (P1–P6)

QueryID	Direct Time (ms)	Usability Time (ms)	Speed-up (direct / usability)
P1	2079,50	72,69	29,51
P2	1548,93	65,98	23,61
P3	678,39	62,37	11,52
P4	538,82	72,48	8,24
P5	591,84	76,34	8,04
P6	602,67	78,52	7,73

Table 11: Usability speed-up and usability execution time for the six sigma-complexity query types

QueryID	Direct Time (ms)	Usability Time (ms)	Speed-up (direct / usability)
P1	2046,47	46,11	44,92
P2	2172,51	43,18	51,73
P3	691,02	44,42	15,92
P4	2207,04	38,92	60,53
P5	1605,09	36,43	45,69
P6	517,90	34,42	15,77

Table 12: Usability speed-up and usability execution time for the six pair types with gamma and sigma complexity varied simultaneously

QueryID	Direct Time (ms)	Usability Time (ms)	Speed-up (direct / usability)
P1	2188,09	56,83	38,51
P2	2183,96	67,87	32,18
P3	564,71	26,03	21,69
P4	2243,92	98,03	22,89
P5	1690,63	78,80	21,45
P6	515,49	70,02	7,36

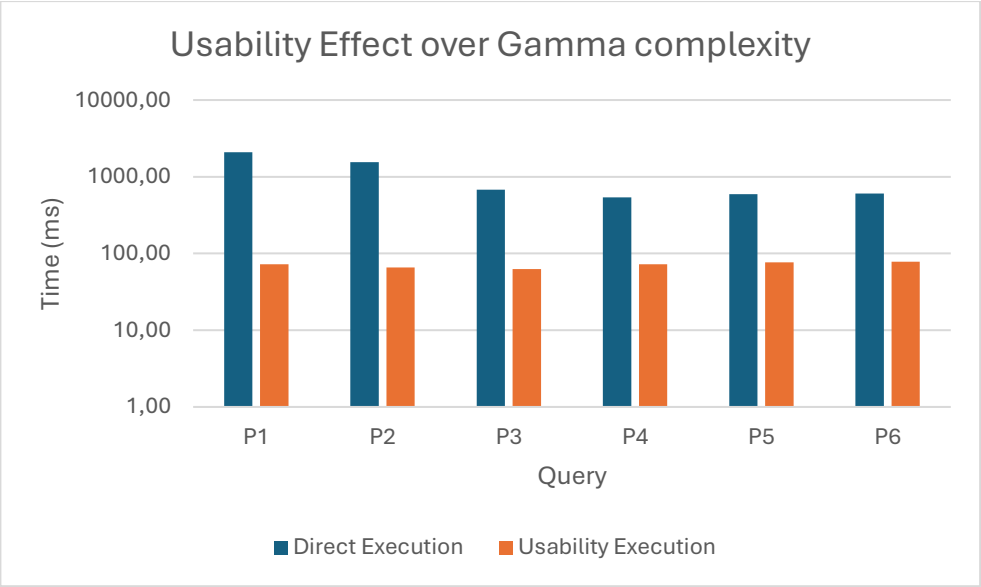


Figure 36: Usability execution time across increasing gamma complexity

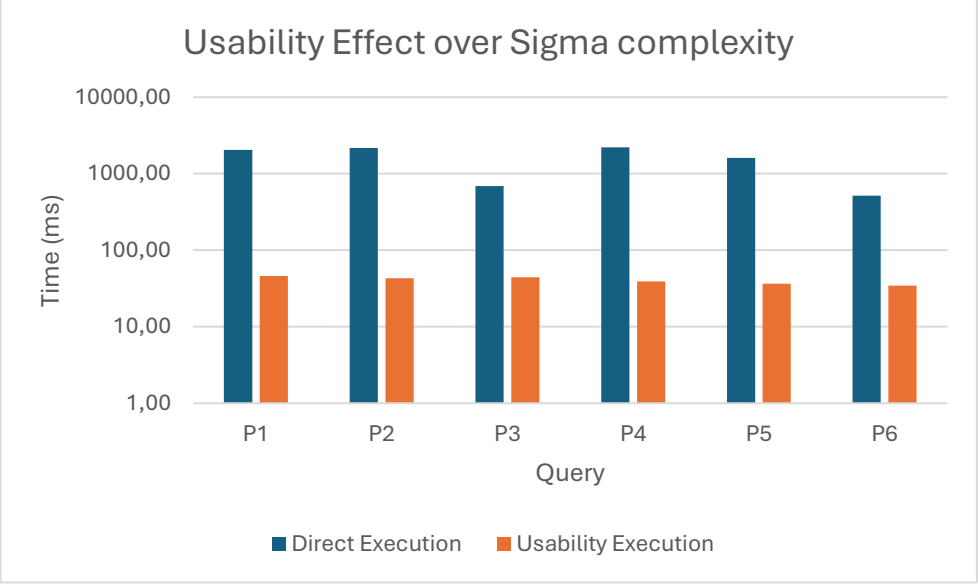


Figure 37: Usability execution time across increasing sigma complexity

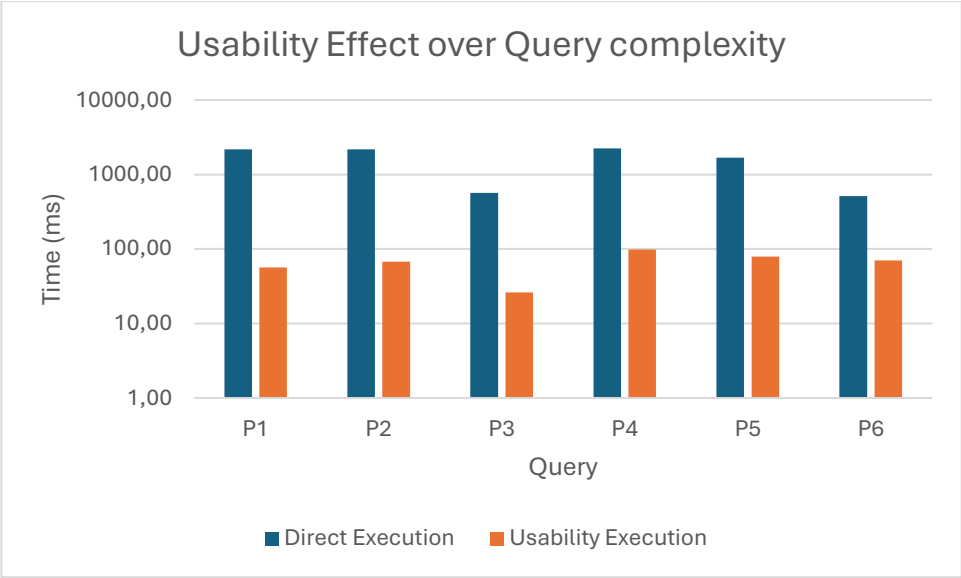


Figure 38: Usability execution time across increasing combined gamma & sigma complexity

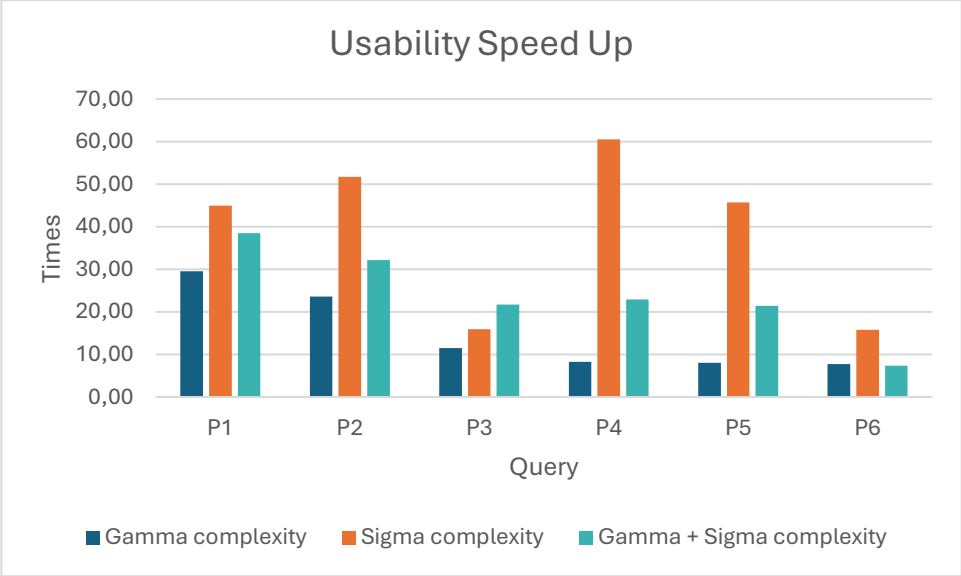


Figure 39: Usability speed-up (direct execution time / usability execution time) across the six query complexity types comparing gamma-only, sigma-only, and combined gamma and sigma complexity variation

Conclusion – observations

Across the six **gamma-complexity** query pair types, the usability speed-up declines steadily and monotonically as the number of rolled-up gamma dimensions grows, from 29.51x at P1 (a single dimension rolled up) down to 7.73x at P6 (all three

dimensions rolled up simultaneously), leading to a drop in relative speed-up from the simplest to the most complex pair type.

Usability execution time itself, however, remains within a comparatively narrow range, from 62.4 ms to 78.5 ms across all six pair types, which indicates that the decline in speed-up is driven mainly by the corresponding fall in direct execution time for these particular queries (from roughly 2080 ms at P1 down to 540–600 ms at P3–P6) rather than by usability becoming markedly more expensive to compute as more gamma dimensions are rolled up.

The **sigma-complexity** workload shows a less regular pattern: speed-up does not decrease monotonically across P1–P6, ranging from 15.77x to 60.53x, with the two highest values occurring at P2 and P4 rather than at the simplest pair type. Usability execution time, by contrast, decreases steadily as sigma complexity increases, from 46.1 ms at P1 down to 34.4 ms at P6, suggesting that, for this dataset and workload, an increasing number of changed sigma atoms does not introduce additional computational cost during derivation, and may in fact correspond to slightly cheaper roll-ups.

The **combined gamma-and-sigma** workload, in which both forms of complexity are increased simultaneously, demonstrates the steepest overall decline in speed-up of the three workloads, from 38.51x at P1 down to 7.36x at P6, a roughly 5 times drop. Usability execution time in this workload is also the most inconsistent of the three, ranging from 26.0 ms to 98.0 ms with no clearly monotonic trend across the six pair types, which suggests that varying gamma and sigma complexity together does not simply reproduce the additive effect of the two single-factor workloads, but introduces additional variability into the cost of result derivation.

Taken together, these results indicate that increasing gamma complexity has the most consistent, monotonic effect on usability speed-up, increasing sigma complexity has comparatively little effect on usability execution time on its own, and varying both simultaneously compounds into an overall drop in speed-up.

Usability execution time though, in absolute terms remains in every case tested at least an order of magnitude below the corresponding direct execution time.

6.2.8. Conclusions – Observations

Overall, these experiments provide a comprehensive evaluation of the usability framework by examining both its performance benefits and its potential costs associated with query analysis and history management.

The experimental results confirm that the proposed usability framework can effectively reduce query execution costs in multidimensional analytical environments. The evaluation shows that usability-based reuse provides increasing benefits as data size and query reuse opportunities grow, while introducing minimal computational overhead. Additionally, the mechanism remains relatively stable under different history configurations, indicating that it can support efficient query processing without requiring strict assumptions about workload characteristics.

In more detail, the seven experiments presented in this section provide a comprehensive picture of the performance behavior of the proposed usability framework along the dimensions of dataset size, query composition, history size, history position, overall usability effect in parallel to the history growth, and query structural complexity.

The data-scaling experiment (Section 6.2.1) shows that usability-based execution outperforms direct database execution across almost the entire range of tested dataset sizes, reaching over 15 times faster total execution time at the 1M and 10M row scales. The only exception is the smallest dataset, where usability is marginally slower than direct execution, by a negligible margin of a few milliseconds. At the 100M-row scale, the speed-up narrows, since as the cost of executing non-reusable queries grows unboundedly with dataset size while the number of reusable queries remains

fixed, the relative contribution of query reuse to the total execution time naturally diminishes.

The overhead-decomposition experiment (Section 6.2.2) shows that the cost of checking whether a query is usable is negligible, adding less than 1% to the total direct execution time even in the worst case where no usable query is ultimately found. Since usability-based answer generation can be up to three orders of magnitude faster than direct database execution, the checking cost is consistently outweighed by the benefit obtained whenever reuse succeeds.

The coverage experiment (Section 6.2.3) confirms that increasing the proportion of reusable queries within a workload consistently improves overall performance, with usability-based execution remaining more than 10 times faster than direct execution across all tested coverage levels, and improving further as coverage increases.

The history-size experiment (Section 6.2.4) reveals a measurable cost of the mechanism: as the query history grows, the time required to search it for a usable base query introduces a small performance penalty, with the usability speed-up dropping to as low as 0.85, meaning usability execution can become up to 15% slower than direct execution under these conditions. This effect, while consistent, remains modest in absolute terms.

The position experiment (Section 6.2.5) shows that the execution time is largely insensitive to how far a usable base query is located within the history, with only a slight, negligible delay (under 0.4% overhead) observed once the distance between the new query and its base exceeds 20 positions.

The history-growth experiment (Section 6.2.6) shows that usability coverage increases steadily as additional queries are introduced into the history session, doubling from 15.0% to 30.0% as the history grows from 20 to 100 accumulated queries, with the total execution time advantage of usability-based execution over

direct execution progressively increasing, indicating that the more an analyst moves further within a session, the more chances are that a usable base can be found and the more they can benefit from it.

The query complexity experiment (Section 6.2.7) shows that the speed-up obtained from usability-based execution declines as the structural complexity of the query increases, especially when complexity is expressed through the number of grouper dimensions rolled up. It also declines in general when expressed through the number of changed selection atoms alone, and last but not least, when both forms of complexity are increased together. This shows that gamma and sigma complexity compound when varied simultaneously, although usability execution time remains at least an order of magnitude below direct execution time across every complexity level tested.

Taken together, these results indicate that cube query usability provides substantial performance benefits that grow with dataset size and reuse opportunity, at a checking cost that is consistently negligible relative to the benefits obtained. This benefit is preserved both as a session's query history accumulates over time and as the structural complexity of the queries involved increases. The one parameter that introduces a measurable, if modest, cost is the size of the query history that must be searched, suggesting that history management and indexing, discussed further in Chapter 8, are the most promising avenue for further improving the framework's scalability.

CHAPTER 7

Conclusion and Future Work

This thesis starts from a simple observation about how analytical sessions behave: an analyst exploring a data cube rarely poses a single, isolated query, but instead issues a sequence of related questions over the same underlying dataset, drilling down for detail, rolling up for a summary, or narrowing and widening a selection condition. Conventional OLAP query engines treat every one of these questions as an independent request, executing each of them against the underlying database regardless of how closely it resembles a question asked earlier. The central contribution of this thesis is a theoretical and practical answer to the question of when, and how, a previously computed cube query result can be reused to answer a new one, without re-executing it against the database.

With respect to the first goal of this thesis, the theoretical conditions under which a cube query can be derived from a cached base result were formalized by extending the relational notions of query containment, view usability, and query rewriting to the hierarchical, multidimensional setting of OLAP. The resulting framework, cube query usability, rests on the notion of perfect rollability and on a set of six conditions, concerning the underlying data source, the shared dimensions and distributive aggregate measures, the structure of the selection and grouping expressions, and the hierarchical relationship between sigma and grouper levels, whose simultaneous satisfaction guarantees that a base query's cached cells contain exactly the information needed to compute a new query, with no omissions and no surplus.

With respect to the second goal, a new usability subsystem comprising three new classes, *UsabilityOptimizer*, *CubeQueryUsabilityChecker*, and *CubeQueryUsabilityExecutor*, was designed, implemented and integrated within the Delian Cube Engine. The subsystem searches the session's query history for a usable base query, verifies the six conditions of the usability theorem, and when a usable base is found, derives the new result in memory through filtering, roll-up, and re-aggregation of the base query's cached cells. The integration required no modification to the engine's pre-existing classes and falls back to standard database execution whenever no usable base query can be found, so correctness is preserved in every case regardless of session state.

With respect to the third goal, the six conditions of usability were expressed as a concrete, checkable set of structural and hierarchical tests, evaluated entirely on the syntax of the two queries being compared, without requiring either query to be executed or its detailed cells to be materialized. This syntactic nature of the checks is what keeps the cost of testing for usability low, a property that was subsequently confirmed experimentally.

With respect to the fourth goal, the experimental evaluation confirmed that usability-based execution generally reduces overall query latency relative to direct database execution under a wide range of conditions. Usability-based execution was found to outperform direct database execution by more than an order of magnitude on mid-sized datasets, with the speed-up narrowing, but never disappearing, as the underlying dataset grows very large. The overhead introduced by checking the six usability conditions was found to be negligible, amounting to less than one percent of direct execution time even in workloads with no reusable queries at all, which confirms that the usability check can be performed unconditionally on every incoming query without a meaningful performance penalty. Increasing the proportion of reusable queries within a session was shown to improve, rather than harm, the performance of usability-based execution, while the size of the query history and the position of a reusable query within that history were found to have only a marginal effect, with a small, bounded slowdown observed only for histories

large enough that a linear scan of past queries becomes noticeably more expensive. Two further experiments extended this evaluation along complementary directions: examining how usability coverage evolves as a query history accumulates over the course of a session, and examining whether the structural complexity of a query, expressed through the number of grouping dimensions or selection conditions involved, affects either the achieved speed-up or the computational cost of the usability mechanism itself. Coverage was found to double, from 15.0% to 30.0%, as the history grew from 20 to 100 queries, with the total-time advantage of usability strengthening correspondingly from near parity to an almost 2 times speed-up. The usability speed-up was found to decline as query complexity increased, most consistently along the gamma (grouping) dimension, less consistently along the sigma (selection) dimension alone, and most steeply when both were varied together, indicating that the two sources of structural complexity compound rather than simply combine, even though usability execution time remained at least an order of magnitude cheaper than direct execution throughout.

Taken together, these results indicate that cube query usability is not merely a theoretical curiosity but a practical optimization: it is inexpensive to check, it integrates into an existing query engine with no breaking changes, and it delivers substantial reductions in query execution time precisely in the interactive, session-based workloads that motivated this thesis in the first place. The framework developed here extends the existing relational notions of query containment, view usability, and query rewriting to hierarchical multidimensional spaces, filling a gap that, to the best of the knowledge gathered in this thesis, had not been comprehensively addressed by prior work, and provides both the theoretical grounding and the working implementation needed to exploit it in practice.

Although the experimental evaluation demonstrates the effectiveness of the usability-based query reuse mechanism, several directions remain open for further investigation.

One possible extension is the evaluation of the framework under more complex and dynamic workloads. The current evaluation focuses on controlled query sessions with predefined workloads in order to isolate the impact of individual parameters. Future experiments could examine real-world OLAP workloads, where queries may vary significantly in their structure, selection predicates, grouping levels, and aggregation requirements. Such an evaluation would provide a better understanding of the behavior of usability-based query reuse in practical analytical environments.

Another important direction concerns the management of the query history. The current approach stores previously executed queries and searches this history to identify reusable results. As the size of the history increases, more advanced indexing or organization techniques could be investigated to improve the discovery process. For example, queries could be clustered based on their signatures, dimensions, or similarity relationships, allowing the system to avoid unnecessary comparisons and efficiently locate potentially reusable queries.

Furthermore, the usability framework could be extended to support more complex query relationships. Future work could investigate partial usability scenarios, for example whenever a new query cannot be calculated just from a previous one but perhaps can be calculated from the combination of two or more already cached base queries. Investigating such cases may increase the applicability of the framework to a wider range of analytical queries.

An additional research direction involves exploring optimized query selection strategies. When multiple previously executed queries are usable for answering a new query, the current framework can determine whether reuse is possible, but selecting the most efficient candidate remains an important optimization problem. Future work could consider cost-based approaches that evaluate factors such as query size, result size, storage cost, and expected computation time to select the optimal reusable query.

REFERENCES

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *9th Annual ACM Symposium on Theory of Computing (STOC)*, pages 77–90, 1977.

[Del25] Delian Cubes. https://github.com/DAINTINESS-Group/DelianCubeEngine, 2025. DAINTINESS GROUP University of Ioannina.

[Hal01] Alon Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10:270–294, 2001.

[JPT10] Christian S. Jensen, Torben Bach Pedersen, and Christian Thomsen. Multidimensional Databases and Data Warehousing. *Synthesis Lectures on Data Management.* Morgan & Claypool Publishers, 2010.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART *Symposium on Principles of Database Systems (PODS)*, pages 95–104, 1995.

[Vas23] Panos Vassiliadis. A Cube algebra with Comparative Operations: Containment, Overlap, Distance and Usability. *arXiv:2203.09390*, 2023.

[Vas23b] Panos Vassiliadis. Cube Query Answering via the Results of Previous Cube Queries. In *25th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP 2023),* Ioannina, Greece, March 28, 2023.

[VS00] Panos Vassiliadis and Spiros Skiadopoulos. Modelling and optimisation issues for multidimensional databases. In *Advanced Information Systems Engineering, 12th International Conference CAiSE 2000, Stockholm, Sweden, June 5-9, 2000, Proceedings*, volume 1789 of Lecture Notes in Computer Science, pages 482–497. Springer, 2000.

SHORT BIOGRAPHICAL SKETCH

Iliana Filippou was born and raised in Ioannina, Greece. She graduated from Mathematics Department, University of Ioannina, Greece and worked as a teacher. She then pursued a M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, University of Ioannina, Greece, while working at the banking sector, and then, as a developer at Niki Digital Engineering on the automotive industry. She now works as a developer at EY. She hopes the environment survives humans and loves her parrot pet, Rio.