Query Optimization and Execution for Multi-Dimensional OLAP

Ahmad Taleb

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Doctor of Philosophy (Computer Science) at
Concordia University
Montreal, Quebec, Canada

April, 2011

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By:	Ahmad Taleb
Entitled:	Query Optimization and Execution for Multi-Dimensional OLAP

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

signed by the examining committee:

A. Agarwal	Chair
D. Lemire	External Examiner
L. Wang	External to Program
T. Fancott	Examiner
V. Haarslev	Examiner
T. Eavis	Thesis Supervisor

Approved By

Chair of Department or Graduate Program Director

Spring 2011

Dean of Faculty

ABSTRACT

Query Optimization and Execution for Multi-Dimensional OLAP

Ahmad Taleb

Online Analytical Processing (OLAP) is a database paradigm that supports the rich analysis of multi-dimensional data. While current OLAP tools are primarily constructed as extensions to conventional relational databases, the unique modeling and processing requirements of OLAP systems often make for a relatively awkward fit with RDBM systems in general, and their embedded string-based query languages in particular. In this thesis, we discuss the design, implementation, and evaluation of a robust multi-dimensional OLAP server. In fact, we focus on several distinct but related themes. To begin, we investigate the integration of an open source embedded storage engine with our own OLAP-specific indexing and access methods. We then present a comprehensive OLAP query algebra that ultimately allows developers to create expressive OLAP queries in native client languages such as Java. By utilizing a formal algebraic model, we are able to support an intuitive Object Oriented query API, as well as a powerful query optimization and execution engine. The thesis describes both the optimization methodology and the related algorithms for the efficient execution of the associated query plans. The end result of our research is a comprehensive OLAP DBMS prototype that clearly demonstrates new opportunities for improving the accessibility, functionality, and performance of current OLAP database management systems.

ACKNOWLEDGEMENTS

It is really an honour and a privilege to express my gratitude and indebtedness to my supervisor, Todd Eavis for his priceless support, encouragement and inspiration. His rich experience, wealth of knowledge, and critical and creative thinking has given me direction and insight in pursuing this research.

Special thanks are extended to my uncle Dr. Nasser Taleb for his kindness, support and encouragement. He does not realize how much I have learned from him. I am really glad that I have an uncle like him in my life.

Also, I would like to take this opportunity to personally thank all of my colleagues and friends who have been so supportive and giving of their time especially Yassine Chakir, Yasser Mahmood, Ahmad Al-Tamimi, Nour Eddine Caidi, Mohamad Solihat, Lami Abu Sharikh and Waqas.

Finally, I treasure the invaluable support and encouragement of my dear mother and father. I also owe my loving thanks to my wife and my children. They have endured a lot from my research and traveling. If not for their support, understanding and encouragement it would not be possible to complete this research.

Table of Contents

List of Tables **xii**

List of Figures **xiii**

1 Introduction **1**

1.1 Research overview 3

1.2 Thesis Structure 6

2 Basic Background Material **7**

2.1 Introduction 7

2.2 Defining OLAP 8

2.2.1 OLAP: A Functional Definition 8

2.3 The Data Warehouse and Data Cube 11

2.3.1 Data Warehouse Architecture 11

2.3.2 The Data Cube 13

2.4 Background 14

2.4.1 Sidera ROLAP Architecture 15

2.4.2 Sidera Backend Architecture 16

2.4.2.1 Cube Indexing 16

2.4.2.2 Hierarchical Attribute Representation 18

2.4.2.3 Caching 19

2.4.2.4 Backend Query Engine Model 22
2.5 Berkeley DB 23
2.5.1 Architecture of Berkeley DB 25
2.5.2 Berkeley DB databases and environments 26
2.6 XML DOM 28
2.7 Conclusions 29

3 Efficient OLAP Storage Engine 30
3.1 Introduction 30
3.2 Related Work 32
3.3 Motivation 38
3.4 Encoding Dimension Tables and the Fact Table 41
3.4.1 Encoding Dimension Tables 43
3.4.2 Encoding the Fact Table 47
3.4.3 Dimension Table Storage 50
3.4.3.1 Hierarchical Attributes 51
3.4.3.2 Non-Hierarchical Attributes 54
3.5 Cube Indexing Integration 58
3.5.1 Berkeley R-tree Model 60
3.6 Integration of Berkeley DB with other Sidera Components 67
3.7 Backend Processing Logic 70
3.7.1 The Query Resolution Algorithms 70
3.7.2 Query Transformations 72
3.7.3 Query Processing 74
3.7.4 Post Processing 77
3.8 Experimental Results 77
3.8.1 The Test Environment 78

3.8.2 Non-hierarchical Attributes: FastBit Bitmap versus Berkeley DB B-tree 80
3.8.3 Single Node Experimental Evaluation 81
3.8.3.1 Index Construction 83
3.8.3.2 OLAP Query Resolution 85
3.8.4 Parallel Experimental Evaluation 90
3.9 Review of Research Objectives 92
3.10 Final Thoughts 94
3.11 Conclusions 95

4 OLAP Query Language 97
4.1 Introduction 97
4.2 Related Work 100
4.3 Motivation 104
4.4 Preliminary Material 104
4.4.1 Sidera OLAP Model 105
4.4.2 Object Oriented OLAP Query 110
4.5 OLAP Algebra 113
4.5.1 OLAP Algebra Operations 115
4.5.1.1 SELECTION Operator 115
4.5.1.2 PROJECTION Operator 117
4.5.1.3 Set operations on Data Cubes 119
4.5.1.4 CHANGE_LEVEL Operator 121
4.5.1.5 CHANGE_BASE Operator 122
4.5.1.6 PIVOT Operation 124
4.5.1.7 DRILL_ACROSS Operator 124
4.5.2 Algebra Simplifications 127

4.6 OLAP Grammar 127

4.7 OLAP Algebraic laws for Improving OLAP Expression Trees 131

4.7.1 Laws involving SELECTION 136

4.7.1.1 Better Joins 137

4.7.1.2 Combining conditions 140

4.7.1.3 Pushing laws 141

4.7.1.4 Pulling laws 143

4.7.2 Laws involving PROJECTION 145

4.7.2.1 Decomposition Law 148

4.7.3 Laws for CHANGE_LEVEL and CHANGE_BASE 149

4.7.3.1 Removal Law 149

4.7.3.2 Pushing and Pulling Laws 150

4.7.3.3 Merging Law 151

4.7.4 Commutative, Associative and Trivial Laws 153

4.8 OLAP Metadata Storage 154

4.8.1 OLAP Metadata Grammar 155

4.8.1.1 Simple OLAP Schema 160

4.9 Review of Research Objectives 161

4.10 Conclusion 163

5 Multi-Dimensional OLAP Query Processor 165

5.1 Introduction 165

5.2 Related Work 170

5.3 Motivation 174

5.4 Parsing OLAP XML Queries 175

5.4.1 DOM Parsing 176

5.4.2 Sidera Parser: DOM graph to Parse Tree 177

5.5 The Pre-processor: Semantic checking 179
5.6 From Parse Trees to Logical Query Plans 182
5.6.1 Conversion to Initial OLAP Logical Query Plan 183
5.7 Improving the OLAP Logical Query Plan 186
5.8 OLAP Physical Query Plan Generation 193
5.8.1 Estimating Sizes of intermediate Cubes 195
5.8.1.1 Estimating the Size of a PROJECTION 196
5.8.1.2 Estimating the Size of a SELECTION 198
5.8.1.3 Estimating the Size of UNION, INTERSECTION, DIFFERENCE199
5.8.1.4 Estimating the Size of a CHANGE_LEVEL 200
5.8.1.5 Reducing the Cost of the Logical Query Plan 201
5.8.2 Choosing an order for binary operators 203
5.8.3 Implementations for OLAP algebraic operators 207
5.8.3.1 Choosing a SELECTION Method 207
5.8.3.2 Choosing a PROJECTION Method 210
5.8.3.3 Choosing Binary Operations Methods 211
5.8.3.4 Choosing a Method for CHANGE_LEVEL and CHANGE_BASE212
5.8.4 Pipelining OLAP operations 212
5.8.4.1 Pipelining SELECTION 213
5.8.4.2 Pipelining PROJECTION 213
5.8.4.3 Pipelining Binary Operations 215
5.8.4.4 Pipelining CHANGE_LEVEL 217
5.8.5 Physical Query Plan Notation 217
5.8.5.1 Access method 217
5.8.5.2 Physical Operators for SELECTION 218
5.8.5.3 Physical Operators for PROJECTION 219
5.8.5.4 Physical Operators for Binary Operations 220

5.8.5.5 Physical Operator for CHANGE_LEVEL 221
5.8.5.6 Additional Physical Operators (Get_realValues) . . . 221
5.8.5.7 Final Physical Query Plan 222
5.9 OLAP Query Execution 224
5.9.1 In-memory Hash Table Representation 226
5.9.2 Index Based SELECTION Algorithm 229
5.9.2.1 Cost of the SELECTION operation 236
5.9.3 Algorithms for the PROJECTION operator 238
5.9.4 Hash Table Based Algorithms for Binary Operations 243
5.9.4.1 UNION 244
5.9.4.2 INTERSECTION 245
5.9.4.3 DIFFERENCE 245
5.9.4.4 DRILL_ACROSS 245
5.9.4.5 Cost of Binary operations 246
5.9.5 Algorithms for CHANGE_LEVEL Operation 246
5.10 The Sidera Server 251
5.11 Result Sets 257
5.12 Review of Research Objectives 259
5.13 Conclusions 262

6 Experimental Results 264
6.1 Introduction 264
6.2 The Test Environment 265
6.3 Current Query Processor Implementation 269
6.4 Single Node Experimental Evaluation 271
6.4.1 Query Engine with and without **LAW_5** 271

6.4.2 Sidera Query Engine with and without **LAW_2**, **LAW_4** and **LAW_13** 273
6.4.3 Index scan Versus Sequential cube scan 277
6.4.4 Query Compilation Time versus Query execution Time 278
6.4.5 Validating the Cost Model 279
6.4.6 Scalability 281
6.5 The Sidera OLAP Query Engine versus Query Engines 285
6.5.1 Sidera Query Engine versus MySQL 286
6.5.1.1 Dimension Count 287
6.5.1.2 Fact Table Size 289
6.5.2 Sidera Engine versus Microsoft SQL Analysis Service 290
6.5.2.1 Dimension Count 291
6.5.2.2 Fact Table Size 292
6.5.2.3 Sidera Engine versus MOLAP SQL Analysis Services 294
6.6 Parallel Query Engine Resolution 296
6.7 Conclusions 299

7 Conclusions and Future Work **300**
7.1 Summary 300
7.2 Future Work 302
7.3 Final Thoughts 304

Bibliography **305**

A Twelve SQL Queries **316**

B Sixteen SQL Queries **319**

List of Tables

6.1 Dimension Tables. 267
6.2 Estimated cost (Block I/O and processing time) versus actual time. . 281

List of Figures

1.1 The Parallel ROLAP Server Architecture. 4

2.1 Pivot Operation 10

2.2 Slicing and Dicing a three-dimensional cube. 10

2.3 Roll-up and Drill-down on a simple three dimensional cube. 11

2.4 Data Warehouse Architecture 12

2.5 Two views of the base cuboid of a three dimensional cube. (a) MOLAP model and (b) ROLAP model. 14

2.6 The Parallel ROLAP Server Architecture. 16

2.7 (a) Hilbert packing and (b) physical file on disk. 18

2.8 The hMap data structure of mapGraph. 20

2.9 The xMap data structure. 20

2.10 Components on each of the local processing node 23

2.11 Berkeley DB Subsystems 27

3.1 A simple Star Schema. 42

3.2 User-supplied (native) data values for the data warehouse. 44

3.3 Encoding Employee dimension. 45

3.4 Product Mapping Table (Linear dimension). 46

3.5 Customer Mapping Table (Linear dimension). 47

3.6 Encoded Fact Table/base view (Sales/ABC). 48

3.7 Lattice of 3-dimensional cube(ABC). 49
3.8 Product-Employee (AB) data cuboid. 50
3.9 Recno Berkeley DB databases for (a) the Type attribute in dimension Product and (b) the Province attribute in dimension Customer. . . . 52
3.10 Enhanced mapGraph for Product dimension. 54
3.11 Employee Bitmap indexes (Age and FirstName). 55
3.12 Usage of Bitmap Indexes. 57
3.13 Three-dimensional indexed cube (Sales) on local disk. 59
3.14 Berkeley Btree index that has references to all indexed group-bys stored in one Berkeley DB physical file 61
3.15 Hilbert R-tree indexing of the three-dimensional cube (Sales) before and after the integration of the Berkeley DB with our server. 63
3.16 Blocks occupied in the Berkeley DB physical file for the R-tree index for group-by BC. 65
3.17 Hilbert R-tree index physical files. 65
3.18 The physical structure of the indexed cube for three-dimensional cube (Sales). 66
3.19 The indexed cube for the three-dimensional cube (Sales) without using Berkeley DB. 66
3.20 Block diagram of the software architecture on each local processing nodes. 69
3.21 Hilbert R-tree Search. 75
3.22 Sample SQL queries. 82
3.23 Comparison of the running time of three different sets of queries for Berkeley Btree versus FastBit bitmap. 82
3.24 Index Construction using Sidera DBMS supported by Berkeley versus Sidera Hilbert R-tree index of the three cube sizes. 84

3.25 Index Construction using Sidera DBMS supported by Berkeley versus Sidera Hilbert R-tree index, as a function of dimension count. 85
3.26 Comparison of Berkeley supported Sidera Query Engine versus Sidera Query Engine for three different query counts. 87
3.27 Berkeley Database Hit Rate for three query counts. 88
3.28 Comparison of Berkeley supported Sidera DBMS versus the old Sidera DBMS for the three different cube sizes. 89
3.29 Comparison of Berkeley supported Sidera DBMS versus the old Sidera DBMS, as a function of dimension count. 89
3.30 Comparison of answering OLAP hierarchical queries in Berkeley supported Sidera DBMS versus Sidera DBMS for three cube sizes. 91
3.31 Parallel wall clock time for index construction. 92
3.32 (a) MySQL Basic Architecture (b) MySQL Architecture after adding our OLAP storage engine (OLAPStorage). 95

4.1 (a) Sidera Conceptual Cube Model and (b) Simple Symmetric hierarchies for Customer and Time. 106
4.2 (a) Simple Object Oriented OLAP model grammar.(b) Simple Object Oriented OLAP model. 108
4.3 OLAP query written in SQL. 112
4.4 OLAP query written in MDX. 112
4.5 Basic OLAP query written in OOP Java. 114
4.6 (a) The three dimensional cube *SOLD.* (b) The result of the SELECTION(Time.Month = Jan)SOLD query. 117
4.7 The result of PROJECTION operator. 119
4.8 Two Cubes(*C1* and *C2*) share common feature and measure attributes. 120

4.9 (a) *C1* INTERSECTION *C2* (b) *C1* UNION *C2* (c) *C1* DIFFERENCE *C2*. 121
4.10 Result of a change level operation. 123
4.11 Result of the CHANGE_BASE operation. 123
4.12 Pivot Operation: (a) the original view. (b) the result of the PIVOT operation. 124
4.13 Two cubes (C1 and C2) with different measure attributes. 126
4.14 Two cubes (C1 and C2) with different measure attributes. 126
4.15 Core operations of the OLAP algebra. 128
4.16 PROJECTION elements. 129
4.17 SELECTION elements. 130
4.18 Set Operations. 131
4.19 Change Level, Change Base and Pivot operations. 132
4.20 Drill Across operation. 132
4.21 The XML format for the OLAP query illustrated in Figure 4.5. . . . 133
4.22 (a) Translation of SQL to an intial relational algebra expression. (b) The effect of applying some relational algebra laws. 134
4.23 Initial OLAP algebra tree. 136
4.24 Two-dimensional cube with the most detailed level values. 138
4.25 Initial SELECTION expression tree. 139
4.26 After applying LAW_1 the initial tree of Figure 4.25. 140
4.27 SELECTION *pushing laws*. 142
4.28 (a) Initial OLAP expression tree and (b) its equivalent after applying the SELECTION pushing laws. 144
4.29 (a) Initial OLAP expression tree. (b) Improving the initial expression by pulling SELECTION up and then pushing it down the tree. 145

4.30 (a) Initial OLAP expression tree.(b) Improving the initial expression by introducing new PROJECTION. 147
4.31 (a) Initial OLAP expression tree. (b) Improving the initial expression by decomposing the PROJECTION. 149
4.32 (a) Initial OLAP expression tree. (b) Result of pulling up CHANGE_LEVEL(K).151
4.33 (a) Initial OLAP expression tree. (b) Result of merging two CHANGE_LEVELs.153
4.34 Commutative and Associative rules. 154
4.35 DATABASE Element. 156
4.36 Cube Schema. 156
4.37 Dimension Schema. 157
4.38 Hierarchy Element. 158
4.39 Simple Hierarchy Schema. 159
4.40 Multiple and Parallel Hierarchies. 160
4.41 Sample OLAP Schema. 162

5.1 The major parts of the query processor. 167
5.2 Sidera Backend Query Processor. 168
5.3 Query Compilation and Execution Steps. 169
5.4 OLAP XML query to parse tree 176
5.5 XML DOM graph/tree 178
5.6 Parse Tree. 180
5.7 Initial OLAP Algebra. 185
5.8 Translation of the parse tree of Figure 5.6 to an initial OLAP algebra tree 186
5.9 The effect of query rewriting. 189
5.10 Cube Sale Schema. 190
5.11 Complex OLAP query in a simple string form. 191

5.12 Initial OLAP Logical Query Plan of Complex Query in Figure 5.11. . 192
5.13 A preferred OLAP logical query plan showing the effects of our OLAP query optimizer on Figure 5.12 194
5.14 Notation for the size of cubes and dimension tables. 196
5.15 Cube Sale statistics. 202
5.16 Logical query plan for an OLAP query defined by our OLAP grammar. 203
5.17 Two candidates (a) and (b) for the preferred Logical query plan in Figure 5.16. 204
5.18 Three ways to union four cubes. 205
5.19 (a) Initial plan tree with cost (5400). (b) Optimized plan tree with cost (2850). 206
5.20 Execution of a pipelined SELECTION. 214
5.21 (a) Execution of a pipelined PROJECTION. (b) Block PROJECTION result until the PROJECTION process terminates. 215
5.22 A physical plan for the logical query plan of Figure 5.9. 223
5.23 OLAP physical query plan for OLAP logical plan of Figure 5.10. . . . 225
5.24 Example of **is_block_intersect()** function. 235
5.25 Hash Table Array stores the result of the PROJECTION operation. . 244
5.26 Result of PROJECTION. (b) Result of CHANGE_LEVEL. 251
5.27 A block diagram of the software architecture on each Sidera Backend processing node. 254
5.28 The Sidera frontend. 254
5.29 The Sidera Backend Node. 258
5.30 Result Set DTD Grammar. 259
5.31 Result sent from the frontend to the client side as an XML message. . 260

6.1 Query running time with and without LAW_5 as the number of records in the data cube increases. 273
6.2 Query running time with and without LAW_5 as the number of dimensions increases in the data cube. 274
6.3 Query response time with and without LAW_2, LAW_4 and LAW_13 as the size of the fact table varies. 275
6.4 Query response time with and without LAW_2, LAW_4 and LAW_13 as the size of the number of dimension varies. 275
6.5 Comparison of number of blocks retrieved for the query engine before and after the use of LAW_2, LAW_4 and LAW_13 for the same queries. 276
6.6 Sequential Scan versus Hilbert R-Tree index Scan. 278
6.7 Query compilation time compared to Query execution time for different query groups (group i is answered from i-dimensional cuboids). . . . 280
6.8 Initial and preferred OLAP query plan and its equivalent in SQL. . . 282
6.9 Four Possible physical plans. 283
6.10 Execution time for sixteen OLAP queries as a function of the data cube size. 285
6.11 Comparison of resolving OLAP queries using our OLAP query engine against MySQL Query Engine. 288
6.12 Comparison of the Sidera query engine with MySQL server. 289
6.13 Sidera Query Engine against MySQL Engine as the number of records increases. 290
6.14 Comparison of our query engine versus Microsoft SQL Analysis Services.292
6.15 Comparison of the Sidera engine versus Microsoft Analysis Services for three different cube sizes. 293
6.16 Running time in the Sidera Server versus the MOLAP SQL server. . . 294
6.17 Sidera Engine versus Microsoft Analysis Services (MOLAP storage). . 295

6.18 Sidera Engine (fully materialized cube) versus Microsoft Analysis Services (MOLAP storage). 296
6.19 Parallel clock time for distributed query resolution. 298
6.20 Parallel speedup. 298

Chapter 1

Introduction

Data warehousing and Online Analytical Processing (OLAP) are two of the most important components of contemporary Decision Support Systems (DSS). Collectively, they allow organizations to make effective decisions regarding both their current and future state. That being said, current OLAP tools are primarily constructed as extensions to conventional relational database management systems. As a result, they can be constrained by both conceptual and architectural elements primarily designed for transaction processing systems. Our primary focus in the current research is therefore the design of a robust infrastructure that specifically targets OLAP storage, querying, and processing requirements.

Commercially, a number of vendors have offered more OLAP-oriented products in recent years. Both Microsoft Analysis Services [66, 55] and Oracle OLAP [89], for example, offer partial solutions in this context. While ultimately linked to their flagship relational database management systems, which store warehouse data as a series of standard tables, these products also provide options for the storage of OLAP *cubes*, a data model often associated with OLAP query processing. Still, indexing and access options are somewhat limited and are designed primarily for environments where resource requirements (particularly memory) are relatively modest. Given that data warehouses are at least partly defined by their sheer size, these “small scale” solutions are not ideally suited to the needs of contemporary — or future — decision support environments.

In addition, all current warehouse/OLAP systems utilize query mechanisms that were designed decades ago. Specifically, they rely upon a combination of string based query languages such as SQL and MDX, along with various proprietary extensions. These languages (and their APIs) have little in common with the safe, flexible Object Oriented languages commonly used in today’s development environments. Not only do these languages make client side programming less effective (e.g., no compile time type checking, no semantic verification, no ability to re-factor code, plus the requirement to interleave distinct programming models), but they also make it very difficult for the DBMS server to effectively exploit OLAP-specific constructs at query resolution time. In other words, the requirement to work with existing query languages and APIs largely prevents the backend server from effectively optimizing user queries to take full advantage of either OLAP conceptual structures (e.g., concept hierarchies and aggregation paths) or physical layer extensions (e.g., enhanced indexing or sorting opportunities).

For these reasons, we are currently investigating the design, implementation, and evaluation of an OLAP-aware multidimensional storage and query engine. In short, the server is designed to efficiently resolve OLAP queries written in native OOP languages such as Java. While the DBMS interface is intended to be clean and intuitive, the infrastructure required to provide this functionality is quite complex. To this end, we propose a comprehensive multi-dimensional OLAP algebra, as well as an associated language grammar, that can be used to support the language libraries visible to the client side programmer (We note that the client libraries themselves are the subject of a related PhD thesis). The DBMS backend natively supports the OLAP algebra and is able to optimize initial query plans and execute them efficiently.

Given the complexity of the query resolution process, we have chosen to ground our conceptual work by integrating the core research themes into a working OLAP prototype. Known as Sidera, the current system [38] was designed from the ground up as a high performance parallel OLAP server and consists of a set of (largely uncoordinated) components that provide various OLAP-specific services (e.g., data structure generation, indexing, querying). Figure 1.1 shows the parallel Sidera system model. While providing basic functionality, the existing Sidera server only supports trivial range queries that have been hard-coded in a proprietary syntax. In the remainder of this thesis we will discuss how Sidera has been extended in order to provide (i) a comprehensive OLAP algebra that will eventually support a full range of OLAP queries (ii) a robust OLAP query grammar that provides a concrete foundation for client side languages (iii) a reliable and efficient OLAP storage engine that can be exploited by the query resolution engine (iv) an execution environment that takes full advantage of the first three elements of this list.

1.1 Research overview

The current research is essentially divided into three stages. In the first stage, we explore the design of a robust OLAP storage engine that supports the processing workload typically found in multi-Terabyte OLAP environments. In particular, we seek to provide reliable and efficient disk storage, with all of the functionality one

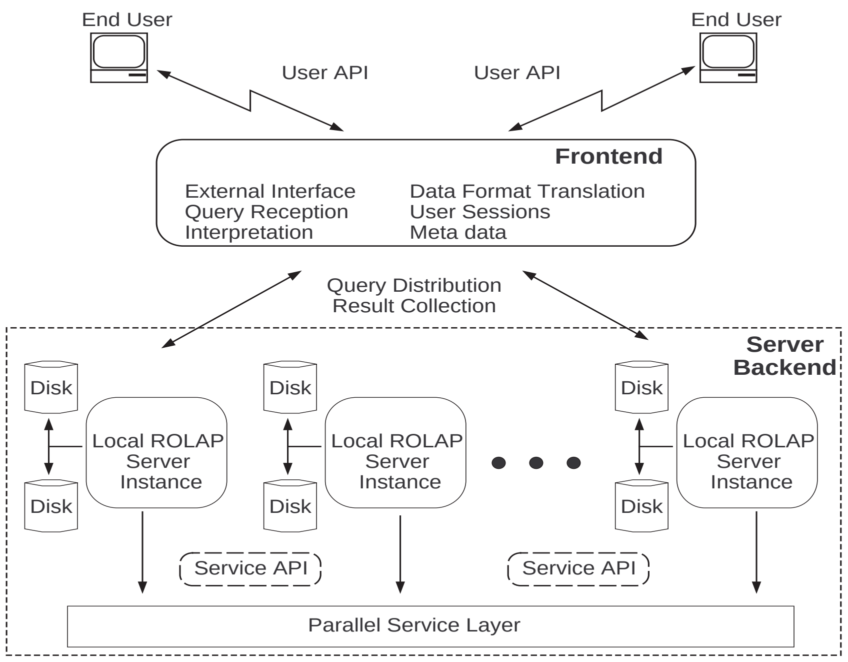


Figure 1.1: The Parallel ROLAP Server Architecture.

would expect in a contemporary DBMS (e.g., optimized block layout, caching, locking). To do so, we have integrated the open source Berkeley DB libraries into the existing Sidera code base. In addition, we have incorporated bitmap indexing facilities into Sidera in order to provide significant performance improvements for query processing that relies on arbitrary attribute restrictions. Experimental evaluation demonstrates that not only does the new storage engine provided a dramatically simplified representation of the OLAP data store, but that even view construction and access times have improved by 15 - 25%.

In the second stage, we have focused on a comprehensive multi-dimensional OLAP algebra and associated XML-based query grammar. Similar in scope to the relational algebra that has supported relational database management systems for the past 30+ years, the OLAP algebra consists of a small set of core operations that collectively define the processing logic found in OLAP environments. In addition, the query grammar is extensively supplemented with various meta data elements that allow for a very expressive representation of the basic operations. Taken together, the algebra and augmented grammar can support intuitive queries that more directly map to the object oriented conceptual model typically associated with the OLAP domain.

In the third stage, we extend the Sidera server to exploit both the storage and query facilities presented above. In particular, we discuss the primary properties of the OLAP operations and present an extensive series of laws for manipulating the initial user queries that arrive from the client side. Specifically, we borrow from existing relational database theory to identify parsing and optimization mechanisms that allow the server to transform initial — often naive — query plans into ones that utilize fewer computational and physical resources and, thus, are likely to execute much more quickly. In conjunction with the plan optimization strategies, we provide an extensive treatment of physical execution plans and algorithms that allow the server to actually carry out the logic of the derived plans. To support the choices that we have made, we provide an extensive experimental evaluation that addresses the value of the optimization and execution strategies. In addition, we compare the latest version of Sidera with a pair of enterprise database management systems often used in data warehouse settings (MySQL and Microsoft Analysis Services). The results suggest that even when the test scenarios are ideally suited to the commercial systems, the methods utilized by the Sidera server show great potential in practical warehouse settings.

1.2 Thesis Structure

The thesis is organized as follows. Chapter 2 provides basic background material needed to understand Online Analytical Processing (OLAP), the Sidera DBMS, Berkeley DB, and the XML parsing used in the storage engine. The succeeding chapters present the core contributions of the thesis. Chapter 3 describes the integration of the Berkeley DB and the FastBit components into our OLAP server. It also presents a series of experimental results that highlight the various performance advantages of our OLAP storage engine.

Chapter 4 describes the comprehensive multi-dimensional OLAP algebra and grammar, as well as the metadata components. An extensive treatment of OLAP operations and their logical manipulation is provided. Chapter 5 discusses the design of the DBMS query compilation framework. Specifically, we look at the two primary components: (i) the query compiler that parses and optimizes initial query plans using the laws defined in the previous chapter and (ii) the OLAP query execution engine that translates the query plan into a physical result set. A set of execution algorithms is also presented for each of the primary OLAP operations. In addition, we discuss the integration of the new storage and query components into Sidera's parallel execution environment.

Chapter 6 provides an extensive set of experimental results that assesses the viability of our storage, optimization and execution strategy. Finally, in Chapter 7, we offer final conclusions and briefly describe possible future work.

Chapter 2

Basic Background Material

2.1 Introduction

Enterprise systems are becoming increasingly more complex. Organizations today utilize a mixture of older, centralized systems and newer, distributed systems. A wide variety of technologies is provided by an even larger number of vendors. Faced with this environment, IT departments have started to develop new concepts and tools both for managing information technologies, and processing the wealth of data and information generated by them.

In this chapter, we examine the current trends, technologies, and terminology relevant to an understanding of Online Analytical Processing or OLAP[23, 25, 31]. Section 2.2 defines On-line Analytical Processing. In Section 2.3, we discuss the data warehouse and the data cube. In Section 2.4, we briefly describe the Sidera parallel ROLAP backend architecture, and its software framework. In Section 2.5, we provide an overview of the Berkeley DB model. We present the DOM parsing in Section 2.6, while Section 2.7 concludes the chapter with brief summary

2.2 Defining OLAP

The term OLAP was not coined until 1992. In that year E. F. Codd, who first introduced the relational data model in 1970, presented a report entitled "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate"[25]. In this paper Codd indicated the twelve elements required of any OLAP application. The following five points are perhaps the most important themes taken from his report:

1. **Multidimensional conceptual view.** The primary focus is the relationship between dimensions, rather than the presentation of transactional records.

2. **Transparency.** The end user should not have to be concerned about the details of data access or conversions.

3. **Accessibility.** OLAP should present the user with a single logical schema of the data, as opposed to a complex physical model.

4. **Flexible reporting.** The DBMS must be capable of presenting data to be synthesized, or information resulting from animation of the data model, according to any possible orientation.

5. **Unlimited dimensional and aggregation levels.** A serious tool should support at least 15 dimensions.

2.2.1 OLAP: A Functional Definition

While commercial OLAP systems may provide many functions, there is a minimal set that can and should be defined by any OLAP application. These functions are listed below, while graphical models are shown in Figures 2.1, 2.2, and 2.3.

- **Pivot.** This OLAP operation allows users to re-organize the axes of the cube. Pivot deals with presentation. Figure 2.1 provides a simple example of how the pivot operation works in practice.

- **Slice.** This is an operation whereby we select a subset of a multi-dimensional array (or cube) corresponding to a single value for one dimension member. This operation allows the user to focus in on values of interest. Figure 2.2 shows the process for a single value of the “color dimension”.

- **Dice.** The dice operation is a slice on more than two dimensions of a data cube (or more than two consecutive slices). The user can draw attention to meaningful blocks of aggregated data. In Figure 2.2, we show a multi-dimensional subcube of a larger cube space.

- **Roll-up.** This is a specific analytical technique whereby the user navigates among levels of data ranging from the most detailed (down) to the most summarized (up) along a concept hierarchy. Figure 2.3 illustrates how the “color dimension”, originally listed at the most detailed level, is aggregated in order to provide a break down by summer and winter colors.

- **Drill down.** This is a specific analytical technique whereby the user navigates among levels of data ranging from the most summarized (up) to the most detailed (down) along a concept hierarchy. Figure 2.3 shows how the “item dimension” is broken down to its item number.

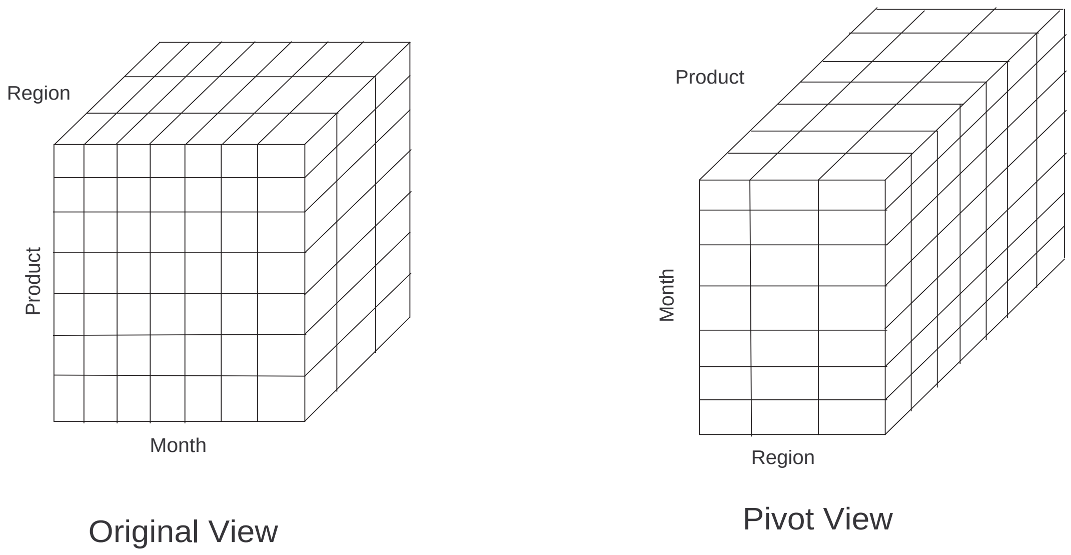


Figure 2.1: Pivot Operation

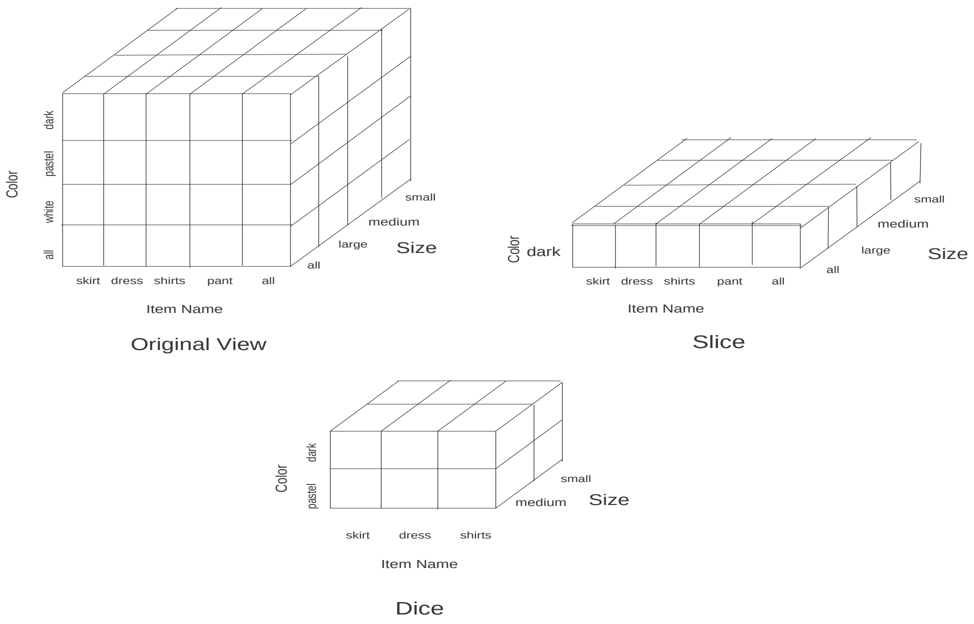


Figure 2.2: Slicing and Dicing a three-dimensional cube.

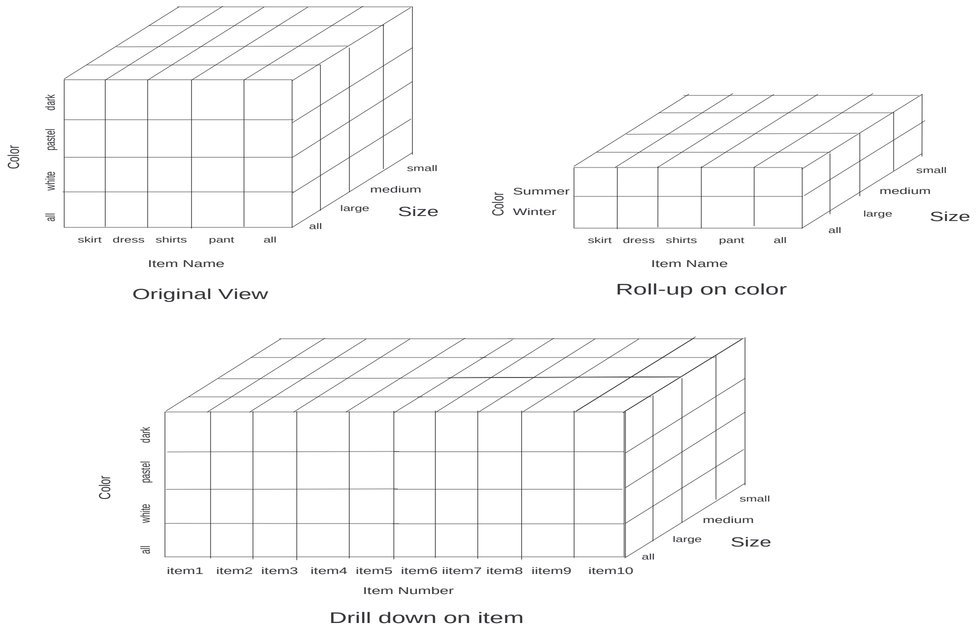


Figure 2.3: Roll-up and Drill-down on a simple three dimensional cube.

2.3 The Data Warehouse and Data Cube

We note that the concept of the data warehouse begins with the physical separation of a company's operational and decision support environments. In other words, a data warehouse is a distinct corporate database management system (DBMS) that is designed to facilitate super fast queries times, as well as the analysis of multidimensional data. The data warehouse is the central data repository for virtually all OLAP systems.

2.3.1 Data Warehouse Architecture

Data warehouses can be seen as a three-tier architecture [23, 53]. The canonical data warehouse architecture is shown in Figure 2.4. The possible data sources are shown on the left. Information is extracted from various legacy systems and operational

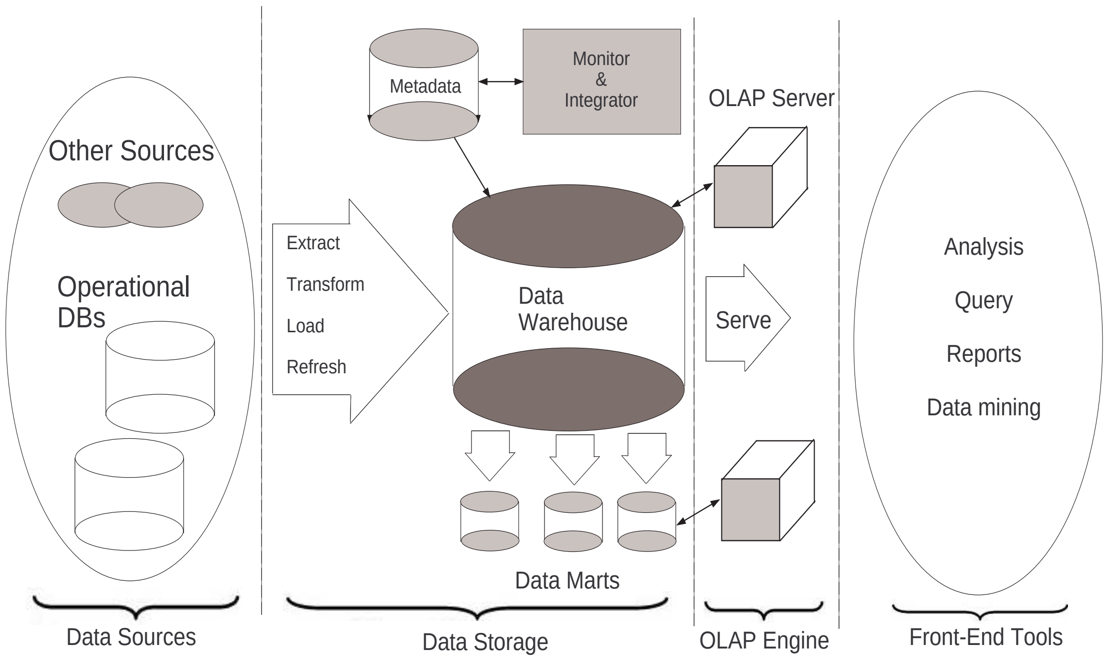


Figure 2.4: Data Warehouse Architecture

sources, consolidated, summarized, and loaded into the data warehouse. Strictly speaking, this first step is outside the scope of the warehouse proper. Several data marts are shown in the second stage; each one is a small warehouse designed for a specific department. At this stage, we have the actual data warehouse, which contains the "decision support" data and associated software. We can refer to this component as the first tier. The second tier contains the OLAP server/engine that allows the users to access and analyze data in the warehouse. In practice, there are many forms of OLAP servers; they are used for the same aims but they differ in their internal data representations. Finally, the third tier includes the front end tools that provide a graphical interface for the top managers and decision makers.

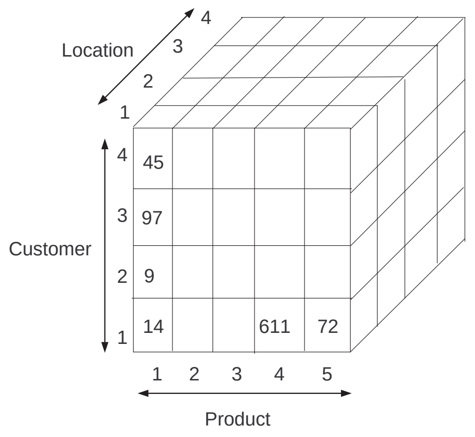
2.3.2 The Data Cube

The data cube is a multidimensional data model that supports OLAP processing. It can be described as a data abstraction that allows one to view aggregated data from a number of perspectives. A data cube consists of dimensions and measures. Dimensions are also known as attributes. Attributes can be of two types. *Feature* attributes represent entities, such as employee and product. *Measure* attributes refer to the items of interest. The measure attributes are aggregated according to the feature attributes.

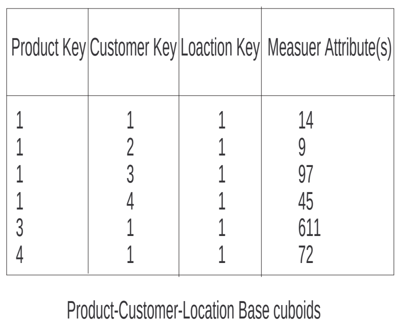
For a d-dimensional space, $\{A_1, A_2, \ldots, A_d\}$ we have $O(2^d)$ attribute combinations. We often refer to this collection of views as the *power set*. In OLAP, views are also known as *cuboids* or *group-bys*. Each view represents a distinct combination of feature attributes, and can be seen as presenting an aggregation of the measure attribute.

The data cube consists of the base cuboid plus (2^d)-1 cuboids. Since the base cuboid contains all the feature attributes, it can be used to compute all the coarser cuboids by aggregating across one or more of its dimensions. The data cube can be described as "full" if it contains all 2^d possible views, or "partial" if only a subset of views has been constructed.

The group-bys can be pre-computed and stored to disk to improve real time query performance. If the data is physically stored as a multi-dimensional array, we have what is called a MOLAP design. MOLAP provides implicit indexing along the axes of the multi-dimensional array but performance sometimes deteriorates as the space — and the associated cube array — becomes more sparse (high dimensionality/high cardinality). Relational OLAP, or ROLAP, stores group-bys (view/cuboids) as distinct tables and tends to scale well since only those records that actually exist are



(a)



Product Key	Customer Key	Loaction Key	Measuer Attribute(s)
1	1	1	14
1	2	1	9
1	3	1	97
1	4	1	45
3	1	1	611
4	1	1	72

Product-Customer-Location Base cuboids

(b)

Figure 2.5: Two views of the base cuboid of a three dimensional cube. (a) MOLAP model and (b) ROLAP model.

materialized and stored. However, it requires explicit multidimensional indexing in order to be used effectively.

Figure 2.5 shows the construction of the base cuboid of a three dimensional cube (Product, Customer, Location). Figure 2.5(a) presents the MOLAP model, while Figure 2.5(b) depicts the ROLAP case.

2.4 Background

In this section, we present a simple architectural overview of the current Sidera system. In particular, we look at the backend architecture that performs the query resolution.

2.4.1 Sidera ROLAP Architecture

Contemporary data warehouses have grown enormously in size, with the largest now pushing into the multi-terabyte range. For these massive data sets, multi-CPU systems offer great potential. The Sidera server was designed from the ground up as a high performance parallel OLAP server.

Figure 2.6 provides a simple illustration of the hardware/ software architecture for the query engine. Here, the Sidera frontend node represents the server's public interface. Its core function is to receive user requests and to pass them along to backend nodes for resolution. The frontend node does not participate in query resolution, other than to collect the final result from the backed instances and return it to the user.

Queries are distributed to each of the p nodes in parallel, allowing each of the processing nodes to participate equally in every query. Load balancing errors due to set partitioning are typically less than 2%. In effect, each backend node serves as an independent ROLAP server that is fully responsible for storage, indexing, query planning, I/O, buffering, and meta data management. Note that a Parallel Service API provides functionality (sorting, aggregation, communication, etc.) that allows local servers (backend nodes) to operate independently.

Sidera provides cube generation algorithms that are fully parallelized and are load balanced and communication efficient on both shared disk and shared nothing cluster architectures. Methods for both full cube (all 2^d views) and partial cube ($< 2^d$) materialization are supported [30, 80, 83]. After running the cube generation algorithm, each backend node houses a portion of each of the O (2^d) cuboids in the full or partial cube.

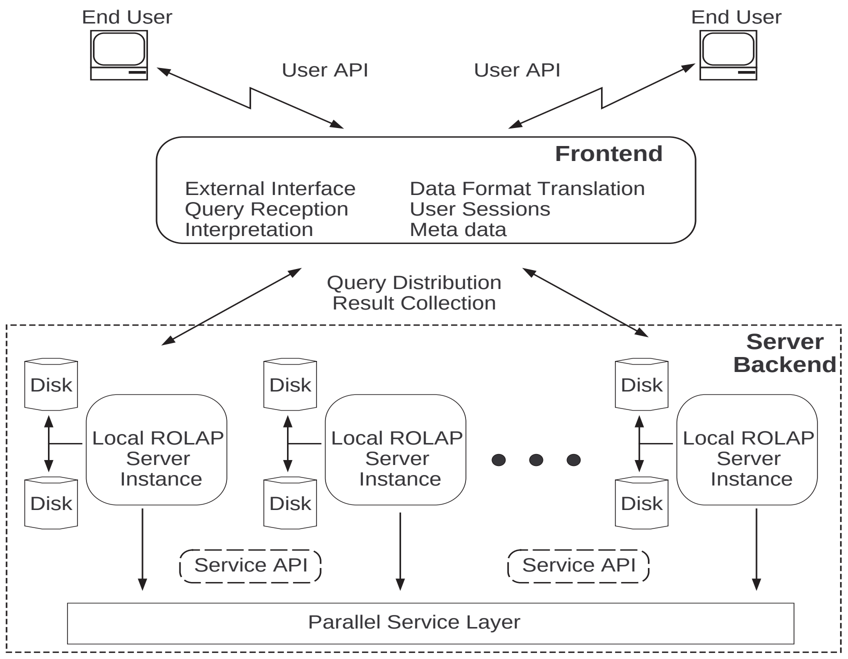


Figure 2.6: The Parallel ROLAP Server Architecture.

2.4.2 Sidera Backend Architecture

Recall that each backend node operates independently to answer a query from the cuboids that are housed in each of the nodes. In this section, we will describe the backend architecture in terms of the constituent components on the local nodes. In particular, we will discuss the indexing of the cube, as well as the hierarchical data structures and caching framework used to support hierarchical queries. Finally, we describe the software architecture utilized on each processing node.

2.4.2.1 Cube Indexing

As mentioned in the previous section, each node contains a fragment of each of the $O(2^d)$ cuboids in order to improve the query response time. However, it is important to remember that high dimension group-bys (cuboids) may still be very large, As such, indexing is critical. In Sidera's case, explicit multi-dimensional indexing is provided by a forest of parallelized R-trees. The R-tree [50] indexes are packed using a Hilbert space filling curve [99] so that arbitrary k-attribute range queries more closely map to the physical ordering of records on disk. For each cuboid fragment on a node, the basic process is as follows:

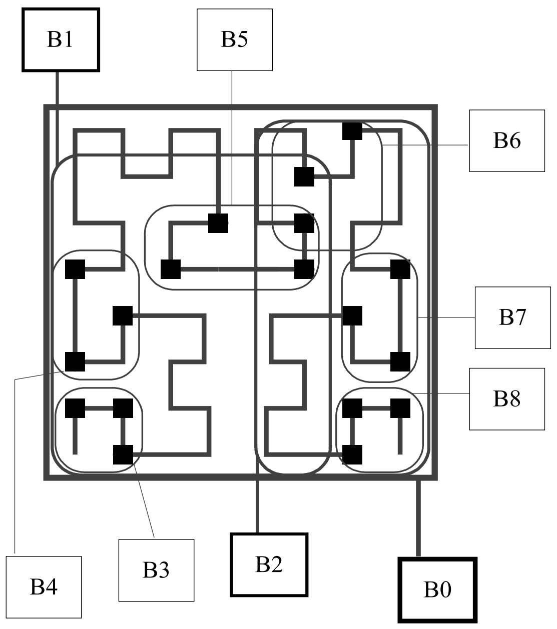
1. Sort the data based upon the Hilbert sort order [99]. Associate each of the n points (records) with m pages of size n/M. Write the base level to disk as (cuboid_name.hil).

2. Associate each of the m leaf node pages with an ID that will be used as a file offset by parent bounding boxes.

3. Construct the remainder of the R-tree index by recursively packing the bounding boxes of lower levels until the root node is reached.

The end result is a Hilbert-packed R-tree for each cuboid fragment in the system. The Hilbert packed R-tree is stored on disk as two physical files: a .hil file that houses the data in Hilbert sort order, and a .ind file that houses the R-tree metadata and the bounding boxes that represent the index tree.

Figure 2.7 illustrates the structure of a small Hibert packed R-tree, with n = 18 and M = 3. In Figure 2.7(a), we see the data points (B3-B8) organized via the Hilbert curve. Boxes that represent the index (B0-B3) are then constructed as described above. In Figure 2.7(b) we see the physical files (.hil and .ind) that are used to store the Hilbert packed R-tree for a given group-by (cuboid). As you can see in Figure 2.7(b), each index block contains the IDs of the blocks that are accessed from it. B0 contains the IDs of B1 and B2 that, in turn contain the IDs of the data



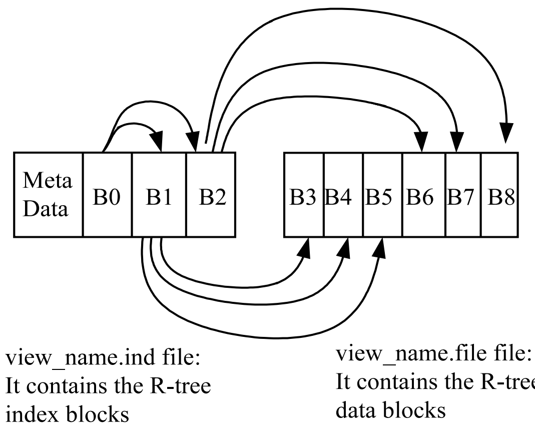

Meta Data
B0
B1
B2
B3
B4
B5
B6
B7
B8
view_name.ind file: It contains the R-tree index blocks
view_name.file file: It contains the R-tree data blocks
(b)

Figure 2.7: (a) Hilbert packing and (b) physical file on disk.

blocks B3, B4, B5, B6, B7, and B8. However, Blocks (B3 . . . B8) hold the Hilbert ordered records.

2.4.2.2 Hierarchical Attribute Representation

One of the most important research problems in the area of Decision Support Systems is the efficient manipulation of hierarchical dimensions stored in the data warehouse, thereby improving the efficiency of querying multidimensional data [105, 107, 74, 34, 84, 69, 60, 90, 76, 64, 63, 21, 57, 101]. However, little research effort has focused upon the manipulation of simple and complex hierarchies in ROLAP at query run-time.

Sidera supports the hierarchal queries by building mapGraph [39], a suite of algorithms and data structures for the manipulation of attribute hierarchies in “real time”. mapGraph builds upon the notion of hierarchy *linearity* [115, 75]. We say that a hierarchy is linear if for all *direct descendants* $A_{(j)}$ of $A_{(i)}$ there are $|A_{(j)}|+1$ values, $x_1 < x_2 \ldots < x_{|A_{(j)}|}$ in the range $1 \ldots |A_{(i)}|$ such that

$$A_{(j)}[k] = \sum_{l=x_k}^{x_{k+1}} A_{(i)}[l]$$

where the array index notation [ ] indicates a specific value within a given hierarchy level. Informally, we can say that if a hierarchy is linear, there is a contiguous range of values $R_{(j)}$ on $A_{(j)}$ that may be aggregated into a contiguous range $R_{(i)}$ on $A_{(i)}$.

Sidera uses a sorting technique to establish linearity for each dimension hierarchy, with data subsequently being stored at the finest level of granularity. It then uses a compact, in-memory data structure called mapGraph to support efficient real time transformations between arbitrary levels of the dimension hierarchy [39]. While a number of commercial products and several research papers do support hierarchical processing for *simple* hierarchies — those that can be represented as a balanced tree — mapGraph is unique in that it can enforce linearity on unbalanced hierarchies (optional nodes), as well as hierarchies defined by many-to-many parent/child relationships. The end result is that users may intuitively manipulate complex cubes at arbitrary granularity levels and can invoke drill down and roll up operations at will. Figure 2.8 provides an illustration of the hMap structure that is used for the simplest hierarchy forms, while Figure 2.9 provides the structure that is used for the unbalanced hierarchy forms.

2.4.2.3 Caching

While the parallel indexing facilities provided by the Sidera server support effective disk-to-memory transfer characteristics, optimal query response time relies to a great

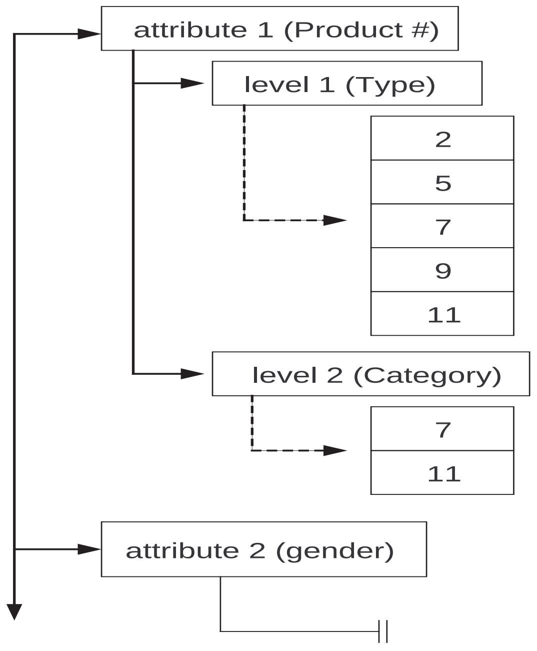


Figure 2.8: The hMap data structure of mapGraph.

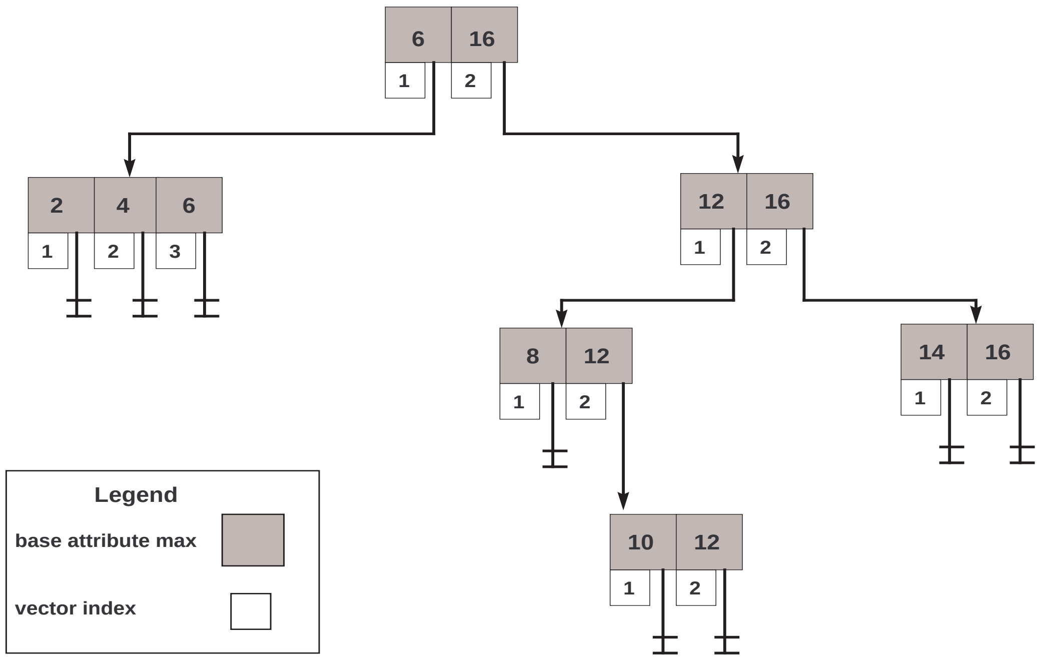


Figure 2.9: The xMap data structure.

extent on an effective caching framework. Given the sizable memory capacity of our parallel ROLAP server, it is expected that a significant proportion of user queries will be answered in whole or in part from a *hot cache*. Sidera provides a natively multi-dimensional, hierarchy-aware caching model. Specifically, resolved partial queries are cached on each node of the parallel machine. For a new k-attribute range query, with ranges $\{R_1, R_2, \ldots, R_k\}$, the cache mechanism must determine if, for each attribute A_i, the range R_i of the user query is a subset of the range on A_i of the cached query. If, for all k attributes, subset ranges are found, the cached query is used in place of disk retrieval. At present, the Cache Manager does not process partial matches. That is, it does not answer queries partly from the cache and partly from disk. This, however, is the subject of ongoing research. Specifically, the logic required for this form of decision making will eventually be integrated into the query optimizer.

The Cache Manager is used in conjunction with the mapGraph to perform translations between hierarchy levels. For a k-attribute user query, an arbitrary number of attributes can be re-mapped simultaneously. Note that queries are cached in their preliminary state — that is, they are cached in their base attribute form before final transformations have been applied. This permits hierarchies to be mapped to arbitrary levels — caching at levels above the base would prevent the cache from answering queries at finer levels of granularity. It is important to note that the cache forms the basis of the core Five Form query model. Specifically, all OLAP servers should be able to support at least five basic OLAP-specific queries: roll-up, drill-down, slice, dice, and pivot. The query engine transparently manipulates the cache contents to further refine previous user queries. A drill down, for example, is produced merely by translating hierarchy levels within the current cache.

2.4.2.4 Backend Query Engine Model

The software architecture on each processing node forms a clean modular design. Figure 2.10 illustrates how the cube indexing, hierarchy, caching, and view components fit into the larger framework. The first three have already been discussed. The View Manager maintains meta-data about the format and sort orders of views physically stored on disk. It is used when queries cannot be resolved from the Cache.

Algorithm 1 provides a high level description of Sidera's query resolution logic. At startup, each local node initializes the mapGraph hierarchy manager with meta data that describes the dimension hierarchies and then initializes the View Manager by scanning the local cube fragment. Because partial data cubes (subset of 2^d) are often constructed in practice, the View Manager can then be used to identify the cheapest surrogate view that can resolve the user query. The frontend broadcasts the queries to each processing node (backend) so that a partial result can be computed. The main backend thread will then invoke the query engine module to process the received query from the frontend.

The query interface is designed to be transparent, so that the user need not be aware of physical storage properties. For this reason, before query resolution takes place, the user's query must be transformed, taking into account the dimensions' hierarchical specifications, the existence of the cheapest surrogate group-by, and the attribute sort orders. An initial result is obtained either from the cache or, if necessary, from disk. If obtained from the cache, relevant records will be retrieved via the Cache Manager. Otherwise, disk access is required. In this case, raw data is received from the Hilbert R-tree indexes and eventually inserted into the Cache.

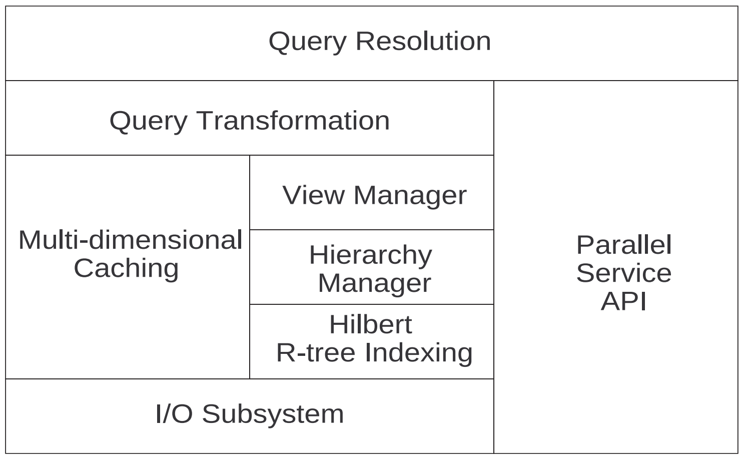


Figure 2.10: Components on each of the local processing node

Once the initial result set has been resolved against the base level data, post processing must be performed in order to produce the final result. The initial results must be translated back into the level of detail required by the user. This function is again performed in conjunction with the mapGraph Hierarchy Manager. A Parallel Sample Sort [104] is performed to order records as per the user request and to permit efficient merging and aggregation. Note that the sorting subsystem is heavily optimized to minimize the movement of multi-value records. If surrogates or hierarchies have been specified, some form of additional aggregation will also be required. At this point, each processing node has part of the final result as per the user request. Finally, if results are required on the front end, then we collect partial result from each node with an MPI Gather operation.

2.5 Berkeley DB

Some applications require only the simple file system read/write services. Others need all the power and flexibility that relational, hierarchical, and/or object databases

Algorithm 1 Backend node server

```
1: Initialize the main backend server
2: Initialize the mapGraph Hierarchy Manager (hM) with the dimension hierarchy meta data
3: Initialize the View Manager ($vM$) with meta data about the physically stored cube
4: While server is running do
      1. receive a set M of user-defined query (uQ) parameters
      2. transform query using the mapGraph Hierarchy and View Managers
      3. check the Cache Manager (cM) for a match on uQ
      4. **if** a valid match is found in the local Cache Manager **then**
   6:      get initial result $I$ from the Cache Manager
      5. **else** {otherwise, go to disk to answer the query}
   8:      get initial result $I$ by accessing the disk to answer the query(uQ)
   9:      add $I$ to the Cache Manager
      6. **end if**
      7. perform OLAP post processing on the initial result $I$.
      8. **if** results required on front end **then**
  12:      collect $R$ with MPI_Allgather
      9. **end if**
```

offer. Berekely DB falls between the low level file system and the high-level relational, hierarchical, and object-oriented engines. Berkeley has an advantage over full DBMS systems in that it it does not add the complexity and processing overhead required to support full database query languages.

Berkeley DB is a general-purpose library written in the C programming language that can be used as a fast, cost-effective data management layer for application developers. Because it is an embedded library, it can be compiled and linked directly into the target application. Berkeley DB is available on all main commercial or open source operating systems. Berkeley DB provides a simple function-call API for data access and management in C, C++, and Java, as well as a wide variety of popular scripting languages.

2.5.1 Architecture of Berkeley DB

It is important to remember that Berkeley DB is not a full relational, hierarchical, or object-oriented database management system. However, there are five major subsystems in Berkeley DB that can be used to implement high-level database functionality without the need for significant query processing overhead or memory resources.

The five main subsystems that are provided by Berkeley DB are, as follows:

1. **Access Methods.** The access subsystem provides several different ways of organizing data. This includes methods for inserting, deleting, and updating entries in the database. Berkeley DB offers four access methods: Btree, Hash, Queue, and Recno. The Btree stores data in a balanced tree structure [14], while the Hash access method implements the extended linear hashing algorithm [71]. The Queue access method stores fixed length records sequentially, with logical

record numbers as keys. Finally, the Recno access method includes both fixed and variable length records with logical record numbers as keys. When the database is created, the developer can select the appropriate access method for the application. Berkeley DB uses a default access method if none is selected.

2. **Buffer Pool.** Berkeley DB offers a shared memory cache used to store the most frequently accessed data. The cached data can then accessed and shared among the processes using the database files.

3. **Transactions.** Berkeley's transaction subsystem ensures that the data doesn't get corrupted if a group of database recordes changes from one state to another. We can think of transaction in terms of the ACIDity property [94].

4. **Locking.** the locking subsystem uses the two phase locking [91] mechanism to provide concurrent access and isolation in the database.

5. **Logging.** The ACIDity property is supported by a write-ahead logging implementation [91].

Figure 2.11 shows the relationships between the subsystems in Berkeley DB. As depicted in Figure 2.11, an application invokes the access methods and the transaction APIs to perform Berkeley DB operations. The access methods and transaction subsystems in turn make calls into the Memory Pool, Locking and Logging subsystems on behalf of the application.

2.5.2 Berkeley DB databases and environments

In this section, we briefly review some of the important concepts that one must understand when building a Berkeley DB application. A *Berkeley database* contains

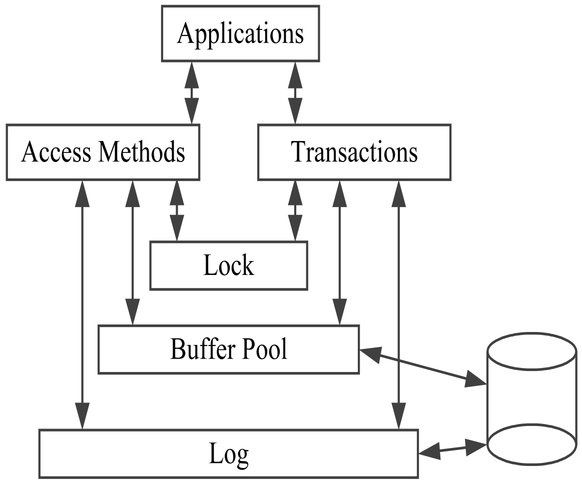


Figure 2.11: Berkeley DB Subsystems

records which consist of key/value pairs. Because of this pairing of keys and data, we may think of a Berkeley database as a two column relation. Keys and data may be arbitrarily complex. A Berkeley database contains a single collection of data organized according to a chosen access method at database creation time (Hash, Btree, Recno, or Queue). A database in Berkeley would normally be referred to as a table in most database systems. Consequently, a standard Berkeley database application uses multiple Berkeley database files. However, Berkeley DB also offers a mechanism called an *environment* that can efficiently manage and coordinate multiple databases. A Berkeley database environment is essentially equivalent to a database in relational database systems. Finally, a Berkeley *handle* is a reference to the physical Berkeley implementation. For example, to use a Berkeley database, you have to obtain a database handle.

A Berkeley DB environment is important when dealing with multiple, related databases. It can be used to efficiently pack hundred of databases into a single file, thereby significantly simplifying the design of the application. Moreover, since

Berkeley is just a library that can be linked into the application space, there is no external server to control threads and manage tables. Instead, Berkeley environments serve as a common place where various control and management functions can be staged.

2.6 XML DOM

As mentioned in Chapter 1, OLAP queries will be received by the Sidera server in XML-based format. In this short section, we briefly review the key concepts relevant to the process of parsing the XML OLAP query. XML is an extensible markup language and can essentially be described as a text based notational mechanism for representing arbitrary, self-describing data elements. Because it is text based, it may be shared across a wide variety of computing platforms. The XML DOM (Document Object Model) defines a standard way for accessing and manipulating XML documents [32]. According to DOM, everything in the XML document is a node. DOM nodes include the document node, element node, attribute node, text node, and comment node. The XML DOM presents the XML data as a tree structure often referred to as a node tree. The terms "parent", "children", and "siblings" are used to describe the relationships between the nodes in the node-tree. The tree itself starts at the root node and moves downward towards the text nodes at the leaf level.

A DOM parser is used to create the DOM node tree (this will be discussed in Chapter 5). In other words, the parser can be used to read the XML document and convert it into nodes that can be accessed and manipulated with programming languages such as Java, C++, etc. By using an XML DTD (Document Type Definition) — essentially an XML schema — we can formally define what elements, attributes, and entities are legal in the XML document. Therefore, the DOM parser not only converts the XML to a DOM node tree, but it also verifies if the received XML matches the XML DTD.

2.7 Conclusions

Both the size and complexity of data analysis has grown enormously in recent years. Typically a data warehouse is used to support the process of data analysis since the warehouse represents a large, consistent repository for enterprise-wide corporate information. In turn, OLAP systems allow users to manipulate the data contained in the data warehouse. In this chapter, we have examined the concepts of Online Analytical Processing and the data cube. OLAP operations were illustrated along with explanations on how these operations are performed so as to provide meaningful measures of summarized multi-dimensional data. We also examined the data warehouse architecture as a three tiered model. We then presented the architectural model for the Sidera platform, a robust parallel OLAP server designed for cluster applications. The server consists of a publicly accessible frontend and a collection of identical backend servers.

We then discussed the architecture of the open source Berkeley embedded database. Specifically, we showed the five major subsystems in Berkeley DB that can be used to implement high-level database functionality. Finally, we presented the XML DOM parser that is used to parse the XML OLAP query on the frontend server. The DOM parser will be further discussed in Chapter 5.

Chapter 3

Efficient OLAP Storage Engine

3.1 Introduction

One of the more important problems in the area of Decision Support Systems is the efficient access and querying of multi-dimensional data stored in the data warehouse [105, 107, 74, 34, 84, 69, 60, 90, 76, 64, 63, 21, 57, 101]. Somewhat surprisingly, perhaps, relatively little research has been published on this topic. It is true that a number of data structures and indexing methods have been developed for OLAP-oriented multi-dimensional data. However, to our knowledge, no attempt has been made to integrate such concepts into viable DBMS systems.

We begin by noting that the Sidera server[38] is intended to be an OLAP DBMS. In fact, the first subject that one must focus on in the implementation of an OLAP DBMS is the OLAP storage engine that is responsible for how the data of the data warehouse is stored effectively on disk, as well as quickly accessed. The existing storage engine of the Sidera server employs several components (indexing, hierarchy manager, caching, etc.) that are used to answer multi-dimensional OLAP queries in a very efficient manner. Specifically, for a d-dimensional fact table that is associated with d dimensions, the current Sidera storage engine creates the R-tree indexed data cube as a ($2*2^d$) separate standard disk files. As was illustrated in Section 2.4, two separate files (with .ind and .hil suffixes) represent the R-tree indexing for a group-by in a d-dimensional cube. These simple files are not databases in any sense and cannot efficiently support DBMS/OLAP queries. Moreover, they are underpowered because of a lack of reliability (transactions, concurrency and recovery) and are at a performance disadvantage relative to a true database system. However, as was discussed in Section 2.4, Berkeley DB combines the power of a relational storage engine (scalability, reliability, transactions, etc.) with the simplicity of a file system-based data storage. Therefore, Berkeley DB has the potential to improve the existing storage engine in the Sidera server.

In this chapter, we describe in detail the integration of Berkeley DB functionality into the Sidera server [38]. In addition, we discuss the storage of dimension tables. Recall that while the Sidera storage engine stores a cube by means of the R-tree indexing method, it does not currently maintain the data for the dimension tables. In general, therefore, our approach is to merge our existing code base that is used to implement the backend Sidera server with the embedded Berkeley DB discussed in Section 2.5. Dimensions tables are then encoded into a compact integer format. Specifically, non-hierarchical attributes are stored as a set of FastBit bitmap indexes, while hierarchical attributes are stored as Berkeley DB databases. Ultimately, the integrated architecture is a very efficient OLAP storage engine that will allow us to explore query resolution and optimization research that would be difficult, if not impossible, to evaluate in a simulated DBMS environment.

This chapter is organized as follows. In Section 3.2, we will discuss previous work. Section 3.3 will present the primary motivation of our work. In Section 3.4, we discuss how the native data of the data warehouse is represented in our server. A detailed description on how the data of the dimension tables are stored on disk and accessed is provided in Section 3.5. A detailed description on how we integrate the existing Sidera Cube Indexing and Access components with Berkeley DB is provided in Section 3.5. In Section 3.6, we describe the integration of the Berkeley code base into other Sidera server components. In Section 3.7, we illustrate the structure of the query engine following the integration process. Extensive experimental results are provided in Section 3.8. Section 3.9 is a review of chapter objectives with final thought about our storage engine provided in Section 3.10. The chapter's conclusions are provided in Section 3.11.

3.2 Related Work

The multidimensional data from data warehouses or data marts are presented to business users by OLAP servers. In practice, there are many forms of OLAP servers (e.g. MOLAP, ROLAP and HOLAP). They are used for the same aims; however, they differ in their internal data representations (Relational, Multidimensional, etc.). As was illustrated in Section 2.3, central to OLAP is the data cube that consists of the base group-by plus (2^d)-1 group-bys. While the cube can be defined as a logical data model, it often forms the basis of a physical model as well. Specifically the group-bys (views) can be pre-computed and stored to disk to improve real time query performance. Therefore, the first issue that one must address with the design and implementation of an OLAP server is how to deal with the very large amounts of historical data found in fact tables, cuboids and dimension tables. Specifically, we must look at the best indexes, representations and data structures that support the efficient manipulation of the data in the data warehouse. In this section, we will review the academic literature that has been developed for the storage, representation, indexing and accessing of data in the data warehouse (fact tables, dimension tables, cubes, hierarchies, etc.). Moreover, we will also discuss some of the existing OLAP storage engines used in commercial OLAP servers [3, 66, 55, 102, 78, 2, 1] and identify the benefits provided by them, as well as their weaknesses.

A number of data structures and indexing methods [10, 18, 19, 42, 54, 67, 65, 72, 106, 111] have been developed for OLAP-oriented multi-dimensional data. A number of papers [65, 111, 67, 72, 106] have studied the reduction of cube size in ROLAP by exploiting the fact that a large amount of the data in the cube is redundant. For example, Key [65], BU-BST [111] and QC-Tables [67] eliminate redundancy from the cube by removing redundant tuples. They don't focus on how the data is finally stored on disk, however, but simply on what tuples are to be removed.

Sismanis et al. [106] propose a non-relational tree-based cube structure (called DWARF) that eliminates prefix and suffix redundancies to create a cube data structure that is both compressed and searchable along attribute hierarchies. In other words, the DWARF cube supports the representation of dimension hierarchies while removing redundancy from cube data. QC-Trees [72] accomplishes much of the same redundancy-reduction of DWARF. However, DWARF [106] and QC-trees exploit complex tree-based models that are not straight-forward and not supported by any widely used product. Moreover, they suggest storing the entire cube as a monolithic relation of fixed sized tuples, which is far from compact. In particular, a large number of NULL values for records that are associated with cuboids with a low number of dimensions would be introduced since they store the non-redundant data in a single relation. The CURE cube [79] accomplishes much the same objectives as [106, 72] but with a more compact table storage and simpler structures. It supports the representation of dimension hierarchies and relatively compact table storage. Specifically, instead of storing the non-redundant data in one single relation, they exploit appropriate data representations that store tuples, according to the cuboid (group-by) they belong to. That being said, the ROLAP storage techniques in [106, 72, 10] lack the multi-dimensional indexing schemes that are essential in ROLAP servers in order to accelerate selective queries.

Apart from the structures that were developed for the compact representation of the data cube in the ROLAP environment[106, 72, 10], a significant number of publications [49, 98, 97, 44] focused on indexing methods that speed access to data in the data warehouse (fact tables, cuboids and dimension tables). In [49], the authors use a group of B-trees to index the multi-dimensional data of the data cube. Roussopoulos et al. [97] propose a data cube indexing scheme called cube tree. The cube tree is based upon the concept of a packed R-tree [98]. A complete study of general purpose multi-dimensional indexing techniques is provided in [44]. In essence, although many of these methods make use of relatively intuitive designs, in practice only a couple of these methods have found their way into practical/commercial systems. A significant amount of academic research has been focused on the R-tree and its variants [98, 50, 15] (there is some commercial support for R-tree indexing in DBMS systems such as Oracle). Guttman is the first to have described the structure of the R-tree as a true multi-dimensional extension of the B-tree [50]. In [103], the authors propose the R+-tree, a variation of the R-tree, that does not allow for overlapping between bounding boxes at the same level of the tree index. Another variation of the R-tree described in [15] is called the R*-tree reduces overlapping during node splitting by using object re-insertion technique.

Space filling curves are another popular approach that supports multi-dimensional indexing. In short, these curves convert the multi-dimensional space into a single dimension. The single dimension can then be indexed by a “regular” index like a B-tree. The analysis of the most common space filling curves — Hilbert, Peano, row-wise and Gray, described by Jagadish [59] — identified the Hilbert curve as the most effective for multi-dimensional queries. Roussopoulos and Leifker [98] described how the R-tree can be created based on the lowX packing mechanism. Kamel and Faloutsos [62] presented an improved technique that utilized the Hilbert curve as a pre-packing mechanism for the R-tree. In practice, the Hilbert curve has generally been shown to be the most effective curve [62] because it ensures that points that are close to one another in the original space are close to one another on the final curve.

Commercially, various OLAP servers [3, 89, 102, 78, 2, 1] offer their own proprietary OLAP storage engine. These OLAP products work specifically with cubes. In other words, they pre-aggregate group-bys for performance reasons. Most of the current OLAP servers offer two different storage models — Relational-based storage (ROLAP) and array-based multidimensional storage(MOLAP).

Mondrian is an open source OLAP server, written in Java[78]. Mondrian is a pure ROLAP server that works with an external RDBMS. It uses a RDBMS instead of developing a new optimized storage engine to store and manage the multi-dimensional OLAP data. In other words, Mondarian does not require a storage engine of its own. Instead, it accesses and reads the data in the RDBMS and stores that data in its own cache.

SAS (Statistical Analysis System) OLAP server is a multidimensional data store designed to provide quick access to pre-aggregated data [102]. It supports ROLAP, MOLAP and HOLAP data storage modes. SAS OLAP server has its own OLAP storage engine that manages the storage of the data on disk and optimizes the access to that data. Moreover, SAS OLAP storage engine provides many optimization techniques such as compression, indexing, sorting and buffering for Base SAS Tables.

Microsoft SQL server, a complete database management system, provides the Microsoft Analysis Services and includes a number of OLAP and data warehousing capabilities [1, 66, 55]. Microsoft Analysis Services (MAS) supports MOLAP, ROLAP and HOLAP storage modes. In the MOLAP storage mode, a copy of the data source (fact table and aggregations) is stored in an optimized multi-dimensional structure to speed up the query process. Moreover, queries can be answered without accessing the source data because a copy of it resides in the multidimensional structure. In ROLAP storage mode, the fact data and aggregations are stored as a set of indexed tables/views in the relational database. Because no copy of the source data is created, queries are resolved from the indexed source data. In ROLAP, query processing is slower than MOLAP; however, it can save storage space and can scale to very large data warehouses. In the HOLAP storage mode, it tries to combine the best of the two storage models, ROLAP and MOLAP, and uses the MOLAP storage mode for those queries that can be resolved from the data of the aggregations that are stored in multidimensional structures. It uses the ROLAP storage when data is needed from the source data to perform an operation such as drill-down.

Microsoft Analysis Services provides many types of indexes (Clustered, Non-clustered, Unique, Index with included columns, Full-text, Spatial, Filtered and XML) in order to optimize performance. All these indexes (except the spatial index) are not designed to natively index a multi-dimensional space. In other words, they do not take a multidimensional space and divide it into index-able regions or partitions. However, with the spatial indexing, the multi-dimensional space is converted into a single dimension that is then indexed using the B-tree index structure. Note that MAS supports the MDX and SQL query languages for querying OLAP cubes and dimensions.

SAP Business Intelligence (SAP BI) is another OLAP server that provides data warehousing functionalities [3]. SAP BI supports ROLAP and MOLAP storage. In the case of the ROLAP storage mode, the data of cuboids is stored in a Datastore or an InfoCube. A DataStore stores the data in flat database tables, while an InfoCube stores the data as a set of relational tables according to the star schema (fact table surrounded by dimension tables). In a MOLAP storage case, however, the multi-dimensional data are physically stored in arrays in the form of compressed files on the Microsoft Analysis server. In MOLAP mode, SAP BI provides the SAP MOLAP Bridge that is an interface between a SAP BI application and the MOLAP data stored in Microsoft Analysis services.

Oracle database includes a multi-dimensional OLAP analytical server called Oracle OLAP [89]. OLAP data can be stored in either MOLAP multidimensional structures known as "Analytic Workspaces" or in relational tables (ROLAP). In the ROLAP storage mode, cubes and dimensions do not contain any data themselves. Instead, they refer to existing oracle relational tables by only adding additional meta-data (hierarchies, feature attributes, measures, etc.) to existing relational tables. In the MOLAP storage mode, cubes and dimensions are fully loaded. In other words, the data for cubes and dimensions are computed and stored at load time.

Finally, Oracle Essbase is a MOLAP server that stores aggregated cubes using two proprietary storage structures called Block Storage Option (Essbase BSO or later referred to as Essbase Analytics) and Aggregate Storage Option (Essbase ASO or more recently, Enterprise Analytics) to physically model the cube [2]. First, Essbase BSO minimizes storage requirement by representing the data of the hypercube as a set of blocks, where each block contains a multi-dimensional array composed of "dense" dimensions, while blocks are created for sparse dimensions when required. While BSO is very compact for aggregate data in small applications, it does not scale well for large applications. Therefore, Essbase ASO is provided as a storage option for large applications.

3.3 Motivation

Though the academic research efforts described in the previous section have attempted to model OLAP-oriented multi-dimensional data, they have been only a partial success because no attempt has been made to integrate such methods and concepts into viable OLAP DBMS systems. Commercially, the existing OLAP storage engines described in the previous section store OLAP databases as a set of related tables and generally offer limited indexing options with respect to the optimization required to support complex OLAP queries. In addition, they also have their own OLAP string-based query languages (MDX and OLAP DML) that add complexity and processing overhead to the DBMS.

Recall that the existing Sidera server described in Section 2.4 has attempted to support very large fact tables by the use of several components (cube indexing, caching, hierarchy, etc) that support OLAP multidimensional queries. The end result is an R-tree indexed cube that actually consists of $2 * 2^d$ separate file objects (i.e., standard disk files). This means that two separate files (with .ind and .hil suffixes) represent the R-tree indexing for a group-by in a d-dimensional cube. These simple files are not databases in any sense and cannot efficiently support DBMS/OLAP queries. In other words, the indexed cube is stored in a set of low-level simple files — for example, a ten-dimensional indexed cube is represented in 2048 files — that do not offer the integrity, concurrency and performance advantages of database systems. However, as discussed in Section 2.4, Berkeley DB provides various storage services of relational engines. It can be used to store any kind of data in any format in one single physical Berkeley DB file (Berkeley environment). Moreover, Berkeley DB combines the benefits of relational database systems and file system data storage. Therefore, Berkeley DB allows us to improve the existing storage engine of our Sidera server. For example, the indexed cube can be stored in one physical Berkeley DB file that can be easily managed, accessed and maintained.

With respect to dimension data, we note that such tables might have hierarchical (e.g., the Province attribute in dimension Location) and non-hierarchical attributes (e.g., Age attribute in dimension Customer). The existing Sidera server does not describe how the data of dimension tables are stored and accessed efficiently on disk. However, Sidera provides an efficient hierarchy manager (mapGraph) that is built upon the notion of linearity to support hierarchical attribute levels. Therefore, we need to provide a mechanism to store and access the data of dimension tables in order to support mapGraph, as well as non-hierarchical attributes.

For the above reasons there is clearly an opportunity for further improvement of the existing backend Sidera OLAP storage engine. Given the above, the following objectives have been identified with respect to the design and implementation of a reliable and efficient OLAP storage engine for the current Sidera environment.

1. Provide simpler, more intuitive representation of a materialized data cube.

2. Support fast indexed cube creation. Currently, a large number of files ($2 * 2^d$) need to be opened to store the R-tree indexes for a d-dimensional cube. This introduces significant disk thrashing that affects the cube creation time.

3. Provide an efficient encoded form of the data warehouse. The encoded form of the data warehouse not only dramatically decreases storage requirements (for example, customer 64-character/byte string might be mapped to a byte integer) but it significantly improves performance for key operations such as conditional checks.

4. Support fast hierarchy manager (mapGraph) creation. When queries are being processed, the dimension hierarchies are required to be read in order to create an in-memory structure in the fastest way. In the current server, the dimension hierarchies are stored as a set of disk files, one for each hierarchical attributes.

5. Ensure an efficient and fast storage for non-hierarchical attributes. Recall the current server answers only very simple queries that do not require data for non-hierarchical attributes (e.g. Age in dimension Customer). However, it is necessary to provide an efficient way to store and access non-hierarchical attributes in order to boost the run-time performance for real OLAP queries and reports.

6. Support locking/concurrency, file system caching, data redundancy and other core DBMS functionality. While not directly utilized at the present time, these database functionalities are very important if Sidera is going to function as a realistic OLAP DBMS.

7. Provide efficient query execution for complex hierarchical OLAP queries.

8. Provide an infrastructure that permits the exploration of new research themes (e.g., OLAP query optimization). The OLAP storage engine is the first component in the implementation of an OLAP DBMS. Our goal in this chapter is to provide an efficient storage engine for the current Sidera server so that we can focus on additional research issues for our server (e.g., query optimization and execution).

3.4 Encoding Dimension Tables and the Fact Table

In this section, we shall describe how the native data for dimension tables and fact table(s) are encoded in our Sidera server. Specifically, our approach is to convert the native data warehouse into a more compact integer format. Figure 3.1 shows a simple de-normalized Star Schema that forms the basis of the relational data warehouse that is supported by our ROLAP Sidera server. It consists of a single, very large fact table (Sales) housing the measurement records (Total_Sales) associated with a given organizational process. Sales consists of three feature attributes (ProductNumber, EmployeeNumber and CustomerNumber) and one measure attribute (Total_Sales). Each foreign key (FK) in the fact table (Sales) has a corresponding record in a dimension table. For example, ProductNumber in the fact table is a primary key in

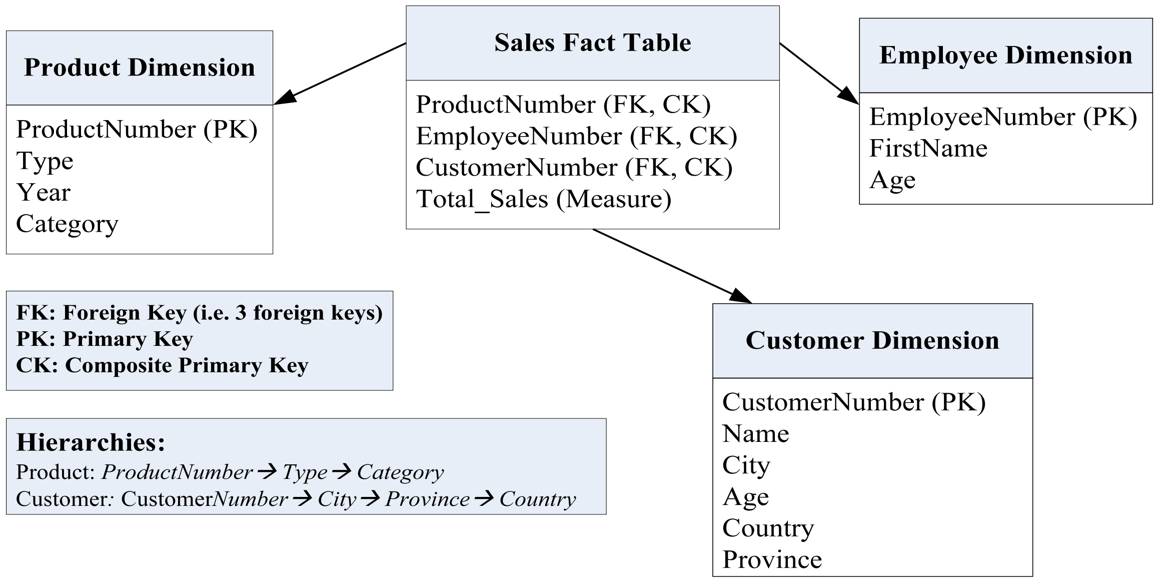


Figure 3.1: A simple Star Schema.

the dimension Product. The primary key of the fact table is a composite primary key that consists of three foreign keys (ProductNumber, EmployeeNumber and CustomerNumber). Note that the fact table that is supported with our server contains only one measure attribute (e.g. Total_Sales in Figure 3.1) that can be aggregated with the sum aggregation function.

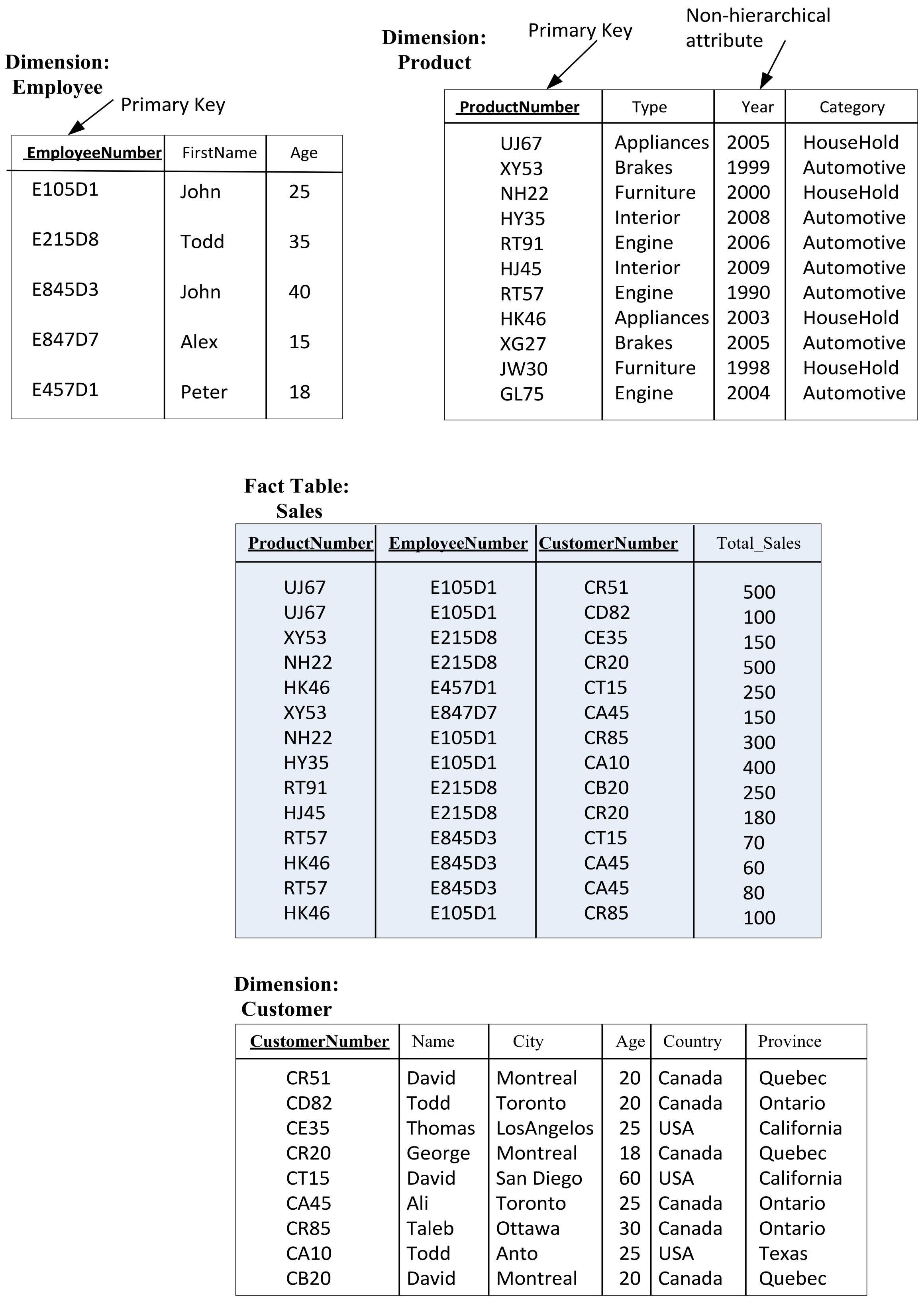
Each dimension simply contains a list of descriptive fields. In addition to this, two dimensions, Product and Customer, have hierarchies that are distinctly defined on them. Product has a three level hierarchy (ProductNumber → Type → Cateogry). The base level is ProductNumber attribute, which can be interpreted as the finest level of granularity on that dimension. The secondary attribute is Product Type, while the tertiary attribute is Product Category. Also, Customer has a four level hierarchy (CustomerNumber → City → Province → Country), where CustomerNumber is the base level, City is the secondary level attribute, Province is the tertiary level attribute, and Country is the attribute at level four. In this case, the primary is the most detail specific level while the fourth level is the least detailed. Dimension Employee does not have any hierarchy.

Figure 3.2 illustrates the native form of the data for the schema of Figure 3.1. We can read the first record of the fact table and derive that the total number of sales of product number (UJ67) that employee number (E105D1) sold to customer number (CR51) is 500. Our objective in this section is to show how to convert the data of the data warehouse from its native form (manifested in the data of Figure 3.2) to a more useful integer form that enables us to more effectively optimize and execute complex OLAP queries. Note that the fact table and dimension tables of Figure 3.2 have few records; however, in a real system the fact tables could have 100 million records or more. Note as well that the OLAP system requires metadata that describes the structure of the available dimensions, hierarchies, measures, facts, cubes, etc. Metadata storage concerning OLAP environment will be covered in Section 4.8.

3.4.1 Encoding Dimension Tables

Before generating the cube of a star schema (fact table surrounded with dimension tables) that is supported by our server, all dimension tables must be encoded. Encoding of a non-hierarchal dimension — a dimension that doesn't contain any hierarchy — is accomplished by a linear pass through the native data set, while encoding of hierarchical dimension is achieved by enforcing hierarchy linearity [115, 75] on the dimension. The result of this encoding process is a set of mapping tables, one for each dimension table, which are physically stored on local disk.

Encoding a non-hierarchical dimension (called dimensionName) is a very simple process. We change the schema of dimensionName by adding a new column called


Primary Key
Primary Key
Non-hierarchical attribute

Dimension: Employee

EmployeeNumber	FirstName	Age
E105D1	John	25
E215D8	Todd	35
E845D3	John	40
E847D7	Alex	15
E457D1	Peter	18

Dimension: Product

ProductNumber	Type	Year	Category
UJ67	Appliances	2005	HouseHold
XY53	Brakes	1999	Automotive
NH22	Furniture	2000	HouseHold
HY35	Interior	2008	Automotive
RT91	Engine	2006	Automotive
HJ45	Interior	2009	Automotive
RT57	Engine	1990	Automotive
HK46	Appliances	2003	HouseHold
XG27	Brakes	2005	Automotive
JW30	Furniture	1998	HouseHold
GL75	Engine	2004	Automotive

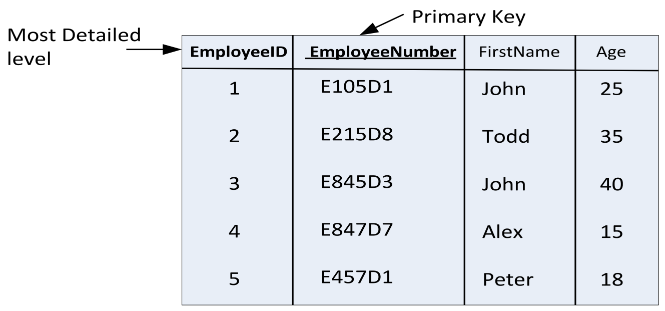
Fact Table: Sales

ProductNumber	EmployeeNumber	CustomerNumber	Total_Sales
UJ67	E105D1	CR51	500
UJ67	E105D1	CD82	100
XY53	E215D8	CE35	150
NH22	E215D8	CR20	500
HK46	E457D1	CT15	250
XY53	E847D7	CA45	150
NH22	E105D1	CR85	300
HY35	E105D1	CA10	400
RT91	E215D8	CB20	250
HJ45	E215D8	CR20	180
RT57	E845D3	CT15	70
HK46	E845D3	CA45	60
RT57	E845D3	CA45	80
HK46	E105D1	CR85	100

Dimension: Customer

CustomerNumber	Name	City	Age	Country	Province
CR51	David	Montreal	20	Canada	Quebec
CD82	Todd	Toronto	20	Canada	Ontario
CE35	Thomas	LosAngelos	25	USA	California
CR20	George	Montreal	18	Canada	Quebec
CT15	David	San Diego	60	USA	California
CA45	Ali	Toronto	25	Canada	Ontario
CR85	Taleb	Ottawa	30	Canada	Ontario
CA10	Todd	Anto	25	USA	Texas
CB20	David	Montreal	20	Canada	Quebec

Figure 3.2: User-supplied (native) data values for the data warehouse.

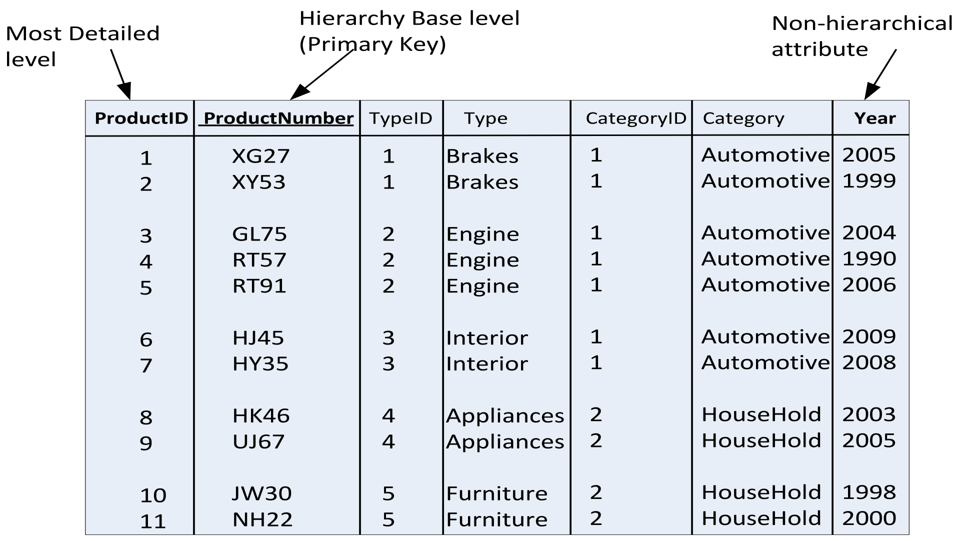

Most Detailed level
Primary Key

EmployeeID	EmployeeNumber	FirstName	Age
1	E105D1	John	25
2	E215D8	Todd	35
3	E845D3	John	40
4	E847D7	Alex	15
5	E457D1	Peter	18

Figure 3.3: Encoding Employee dimension.

dimensionNameID, and then we make a linear pass through the native data set and give a consecutive integer value to dimensionNameID from *1* to n, with n equivalent to the cardinality C of the dimension (dimensionName) primary key. Figure 3.3 shows the mapping dimension table for a non-hierarchical dimension (Employee); the schema of the encoded Employee dimension is (EmployeeID, EmployeeNumber, FirstName, Age). Note that the maximum value of EmployeeID (5) equals the cardinality of the primary key in dimension Employee (EmployeeNumber).

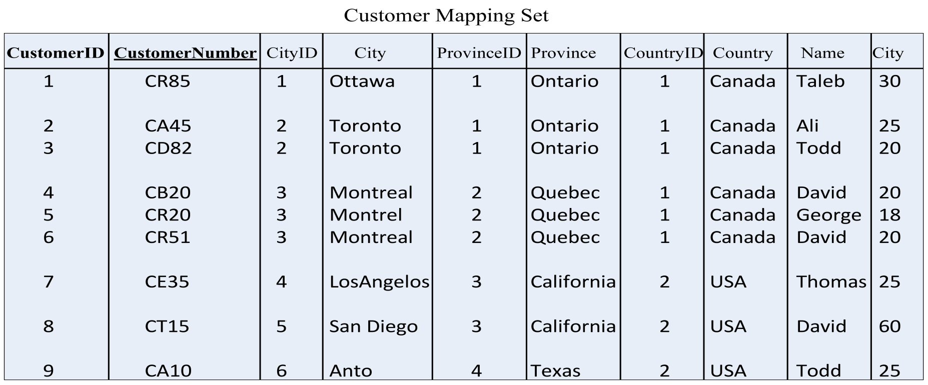
Recall that the Sidera server enforces hierarchy linearity in order to encode the hierarchical dimension tables in the data warehouse. In Section 2.4, we described in detail how the Sidera server hierarchy manager component (mapGraph) builds upon the notion of dimension hierarchy linearity [115, 75]. We encode a hierarchical dimension table (for example called dim) by building a mapping table that is sorted by $A_k, A_k - 1, \ldots, A_1$, where A_1 is the base attribute in the hierarchical dimension. For each hierarchical attribute level Att in dim, we change the schema of dim by adding a new column called AttID. Finally, the values of AttID are created as consecutive integer IDs. Specifically, we associate the consecutive distinct values for each column Att with consecutive integer IDs. Figure 3.4 illustrates the mapping table for the product dimension with the three-level hierarchy (ProductNumber $\rightarrow$ Type


Most Detailed level
Hierarchy Base level (Primary Key)
Non-hierarchical attribute

ProductID	**ProductNumber**	TypeID	Type	CategoryID	Category	**Year**
1	XG27	1	Brakes	1	Automotive	2005
2	XY53	1	Brakes	1	Automotive	1999
3	GL75	2	Engine	1	Automotive	2004
4	RT57	2	Engine	1	Automotive	1990
5	RT91	2	Engine	1	Automotive	2006
6	HJ45	3	Interior	1	Automotive	2009
7	HY35	3	Interior	1	Automotive	2008
8	HK46	4	Appliances	2	HouseHold	2003
9	UJ67	4	Appliances	2	HouseHold	2005
10	JW30	5	Furniture	2	HouseHold	1998
11	NH22	5	Furniture	2	HouseHold	2000

Figure 3.4: Product Mapping Table (Linear dimension).

$\rightarrow$ Category). The schema of the linear hierarchy dimension Product is (ProductID, ProductNumber, TypeID, Type, CategoryID, Category, Year). The Product mapping table consists of 11 records, with 11 equivalents to the cardinality of the primary attribute (ProductNumber). We can see in Figure 3.4 that the Product dimension contains only one non-hierarchical attribute called Year. Figure 3.5 shows the result of encoding the Customer's dimension. We can see how the distinct values of each hierarchical attribute are associated with integer values. For example, 1 is country Canada and 2 is province Quebec. The new schema is (CustomerID, CustomerNumber, CityID, City, ProvinceID, Province, CountryID, Country, Name, Age). Age in dimension Customer is known as the non-hierarchical attribute. In this chapter, we mention only the encoding of the simple symmetric hierarchy (e.g. dimension Customer); however, other types of hierarchies such as ragged and multiple hierarchies can be easily supported and are discussed in the Sidera mapGraph paper [39].



Customer Mapping Set

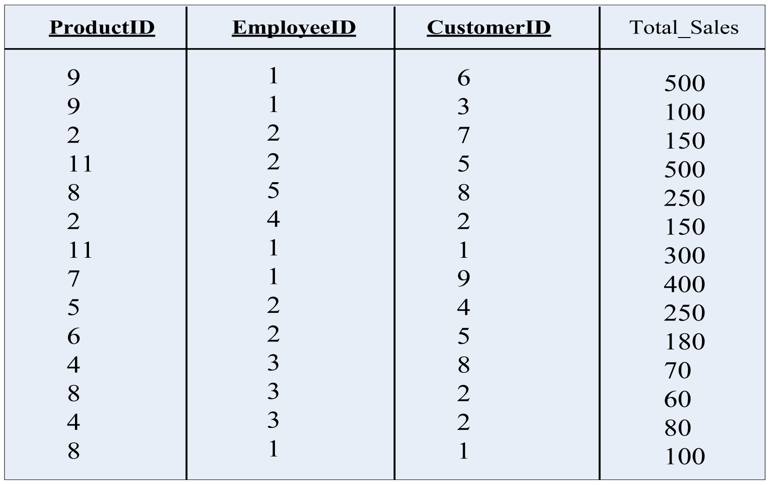
CustomerID	CustomerNumber	CityID	City	ProvinceID	Province	CountryID	Country	Name	City
1	CR85	1	Ottawa	1	Ontario	1	Canada	Taleb	30
2	CA45	2	Toronto	1	Ontario	1	Canada	Ali	25
3	CD82	2	Toronto	1	Ontario	1	Canada	Todd	20
4	CB20	3	Montreal	2	Quebec	1	Canada	David	20
5	CR20	3	Montrel	2	Quebec	1	Canada	George	18
6	CR51	3	Montreal	2	Quebec	1	Canada	David	20
7	CE35	4	LosAngelos	3	California	2	USA	Thomas	25
8	CT15	5	San Diego	3	California	2	USA	David	60
9	CA10	6	Anto	4	Texas	2	USA	Todd	25

Figure 3.5: Customer Mapping Table (Linear dimension).

3.4.2 Encoding the Fact Table

Fact table encoding is accomplished by encoding its feature attributes. Encoding a feature attribute is to convert it from its native form to an integer form. Each feature attribute (FK) in the fact table is associated with the primary key in a dimension table. Also, the records of the mapping dimension table have consecutive integer numbers, 1, 2, . . ., n, one for each primary key value. We thus encode each feature attribute Att, which relates the fact with a dimension dim, by replacing its native value with the encoded attribute dimID value taken from dimension dim. In addition, each feature attribute name Att that relates dimension dim becomes dimID.

Figure 3.6 illustrates the encoded fact table (Sales) of Figure 3.2. The encoded fact table now has three feature attributes (ProductID, EmployeeID, CustomerID) and one measure attribute (Total_Sales). The values of feature attributes are integer values taken from the mapping dimension tables of Figure 3.3, 3.4 and 3.5. We can see for example that the first record of the encoded fact table is (9, 1, 6, 500) which corresponds to ProductNumber CR51, EmployeeNumber E105D1, and CustomerNumber CR51. The encoded fact table is physically stored on disk. Data in the



ProductID	EmployeeID	CustomerID	Total_Sales
9	1	6	500
9	1	3	100
2	2	7	150
11	2	5	500
8	5	8	250
2	4	2	150
11	1	1	300
7	1	9	400
5	2	4	250
6	2	5	180
4	3	8	70
8	3	2	60
4	3	2	80
8	1	1	100

Figure 3.6: Encoded Fact Table/base view (Sales/ABC).

fact table is stored only for the most detailed encoded values (Primary key encoded values).

As was illustrated in Chapter 2, Sidera server stores a large number of pre-computed aggregate cuboids (2^d cuboids for d-dimensional cube). The data in the cuboids is physically stored only for the most detailed level. The basic encoded fact table can be considered to be the base cuboid. Since the base cuboid contains all dimensions, it can be used to compute all cuboids in the cube. For example, we consider the encoded fact table of Figure 3.6 to be the base cuboid of the 3-dimesnional cube of Figure 3.2 . Sidera uses the base cuboid to compute other cuboids that can be queried more efficiently at run time. The data in the cube is physically stored for the most detailed encoded values (e.g. ProductID, CustomerID, etc.). For convenience, we typically refer to the dimensions with letter names. For example, Product = A, Employee = B and Customer = C for the base cuboid of Figure 3.6. So a Product-Employee-Customer data cube might be labelled as ABC. Using ABC as the base cuboid, then, the space in the Sidera server contains $2^3 = 8$ cuboids/views: ABC,

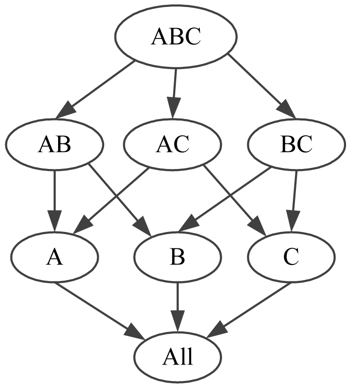


Figure 3.7: Lattice of 3-dimensional cube(ABC).

AB, AC, BC, A, B, C and All.

The relationship between views is captured by the data cube lattice that defines the parent child relationship between views in the cube space. Figure 3.7 provides the lattice of 3-dimensional cube (ABC). Note that the lattice begins with the base cuboid/view that contains the full complement of three dimensions (ABC). ABC has higher granularity than its children (AB, BC and AC). In other words, ABC has more detailed information than its children. Note that in our server, all nodes of the lattice (full cube) are physically materialized and stored to disk for the most detailed encoded integer values. However, partial cubes (i.e., not all cuboids) can be materialized and stored.

Note that the top of the lattice (ABC) corresponds to the encoded fact table of Figure 3.6. Sidera uses the base cuboid (e.g. ABC in our example) to compute all other cuboids in the lattice. For example, the base cuboid of Figure 3.6 (ABC or Product-Employee-Customer) can be used to compute the child view AB (Product-Employee cuboid). Figure 3.8 illustrates the data of view AB. It shows the same

ProductID	**EmployeeID**	Total_Sales
9	1	600
2	2	150
11	2	500
8	5	250
2	4	150
11	1	300
7	1	400
5	2	250
6	2	180
4	3	150
8	3	60
8	1	100

Figure 3.8: Product-Employee (AB) data cuboid.

fact measurement as its parent view (ABC) but this time aggregated on just A (ProductID) and B(EmployeeID) (i.e., lower granularity). For example, consider the first two records of view ABC. Here, the measures of these two records are aggregated (Total_Sales = 500+100 = 600) in view AB because of the duplication in A and B.

3.4.3 Dimension Table Storage

Having seen the representation of dimension tables, we must now consider how these dimension tables are physically stored and represented on disk. As was mentioned, the data cube (views in the lattice) is generated by our server by creating all views in the lattice rooted at the base cuboid (For example, the encoded fact table in Figure 3.6). Therefore, the basic cuboid (encoded fact table) and other views have no descriptive dimension information available; this makes the report of an OLAP query executed against the cube alone very hard to read. Also, OLAP query constraints are often specified on the attributes of the dimensions (e.g., All customers who are older than 30). Thus, we require joins between a very large fact table and dimension tables in cases involving OLAP reports or query constraints. For these reasons, it is not sufficient just to spread the records of the dimension tables among different disk blocks. Instead, we need better organization of dimension tables so that we can minimize join operations between the fact table (cuboids) and dimension tables in order to improve performance.

3.4.3.1 Hierarchical Attributes

For each sub-attribute in the hierarchy, we now create a Berkeley DB database with the Recno access method. The Recno access method is backed with a flat-text file that provides fast sequential read access. Therefore, this access method supports very fast creation of the mapGraph at run time. As was illustrated in Chapter 2, the entry of a Berkeley DB database is of the form (key, data). Therefore, for all hierarchical attributes (i.e., Type in dimension Product) other than the base attribute (e.g. ProductNumber in dimension Product), we create a Berkeley DB database that associates the key with the data. The key is the encoded attribute value (i.e. values of attribute TypeID in dimension Product), while the data have two values: a native attribute representation (e.g. values of attribute Type from dimension Product) and an integer value that represents the corresponding maximum encoded value in the primary attribute. Figure 3.9 illustrates how the hierarchical attributes "Type" of dimension Product and "Province" of dimension Customer are stored as two Berkeley DB databases. Moreover, it is possible to encapsulate multiple databases, one for each sub-attribute level, in a single Berkeley DB file on disk. In other words, we use one Berkeley DB physical file to encapsulate the hierarchy data in a dimension.

Recall that the Sidera hierarchy manager (mapGraph) exploits the notion of hierarchy linearity. Figure 2.8 provides an illustration of the hierarchy manager that

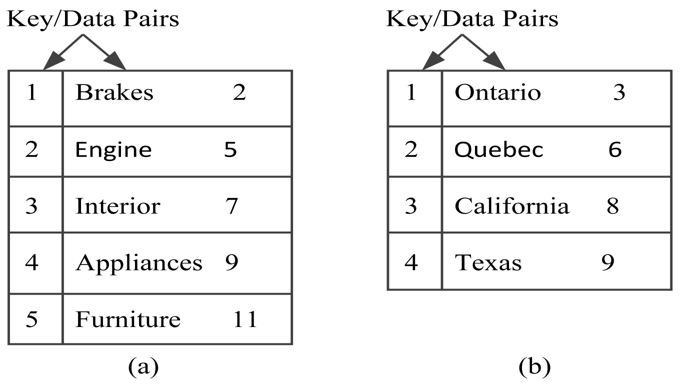


Figure 3.9: Recno Berkeley DB databases for (a) the Type attribute in dimension Product and (b) the Province attribute in dimension Customer.

allows us to map between arbitrary levels of encoded data (integer) values of a dimension hierarchy. The existing Sidera in-memory data structure mapGraph supports the following range translations: (i) mapping from an encoded base level attribute value (integer value) $A_i(1)$ to the corresponding encoded sub-attribute value $A_i(\text{j})$, j ≥ 2; (e.g. In the Product dimension ProductID = 6 corresponds TypeID = 3) (ii) mapping from a sub-attribute $A_i(\text{j})$ encoded value (integer) to the corresponding range(s) on the base attribute $A_i(1)$ (e.g. In the Customer Dimension ProvinceID =3 corresponds to CustomerID = 6 and 7). However, OLAP query constraints are often specified on the native values of the attributes. Also, it is difficult or even impossible for the user to read an OLAP report with encoded integer values. Therefore, we have to enhance the current mapGraph to provide O(1) conversion of native, user-supplied values to encoded system-specific values (integer values) and vice versa. For each sub-attribute Att, we create an in-memory hash table that associates each native representation Att(nat) with encoded value(s) Att(en). We also add the native values for all encoded values. This allows us to convert from encoded value to native value in O(1). In practice, the size of the mapGraph would likely be no more than a few dozen kilobytes for large data cube problems because the collective size of a d-attribute mapGraph depends on the cardinalities of non-base levels exclusively, which are very small compared to the base level (e.g. the Product hierarchy might have 10,000 different products for the base level, however only two categories (HouseHold and Automotive) for the category level).

Figure 3.10 illustrates the enhanced mapGraph structure for the dimension hierarchy Product of Figure 2.8. Now in Figure 3.10, we add two hash tables, one for each hierarchical attribute level other than the base level (Type, category). For a sub-attribute (A_j), j $\geq$ 2, the associated map is made up of the maximum encoded value from the range on (A_1), corresponding to the current encoded value of (A_j). We add the native values of (A_j) in order to convert any encoded value of (A_j) to its native value. For example, we can use the Category Hash Table to retrieve the encoded value (2) of Automative category in O(1). Then, we can use the map associated with Category to find all ProductIDs ($1 \rightarrow 7$) that are Automative. Finally, the native value of the encoded value 2 of Type attribute is Engine and it corresponds to the base level (ProductID) range $3 \rightarrow 5$. In the next chapter, we will describe how our Sidera server uses the enhanced mapGraph to efficiently answer hierarchical OLAP queries.

At run time, when hierarchies are processed, they are read from the Berkeley DB databases into main memory and used to create the enhanced mapGraph (e.g. Figure 3.10 contains the hierarchy of dimension Product). Specifically, for each sub-attribute in the hierarchy, we read all records from its corresponding Berkeley DB sequentially and insert them into the mapGraph. Note that the enhanced mapGraph supports the translation from encoded sub-attributes and native values to encoded

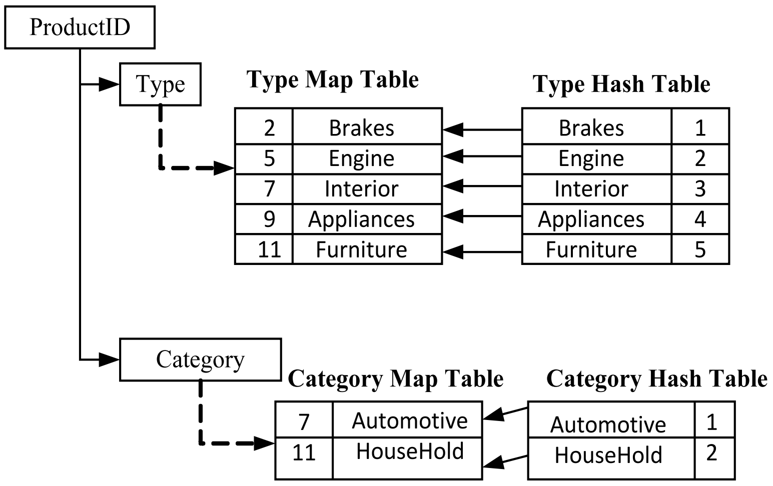


Figure 3.10: Enhanced mapGraph for Product dimension.

base level values and vice versa. However, it supports the translation of encoded and native values between non-base hierarchical levels. Finally, the enhanced mapGraph can be used to minimize the join between the cube and dimension tables when hierarchical attributes are involved in the OLAP analysis. In other words, hierarchical OLAP queries can be resolved by accessing the data of the appropriate cuboid/view in the cube space and the data of the enhanced mapGraph.

3.4.3.2 Non-Hierarchical Attributes

As noted, the enhanced mapGraph is very useful when hierarchical attribute levels are involved in the OLAP analysis. However, we need also to represent and organize non-hierarchical attributes because they are also very useful in the OLAP analysis (e.g. what are the total sales for all customers who are older than 25?). Recall that the data in the cube in our Sidera server is physically stored for the most detailed level of encoded integer values (e.g. ProductID in Figure 3.8). If a non-hierarchical attribute is involved in the restriction of an OLAP query or in an OLAP report, then

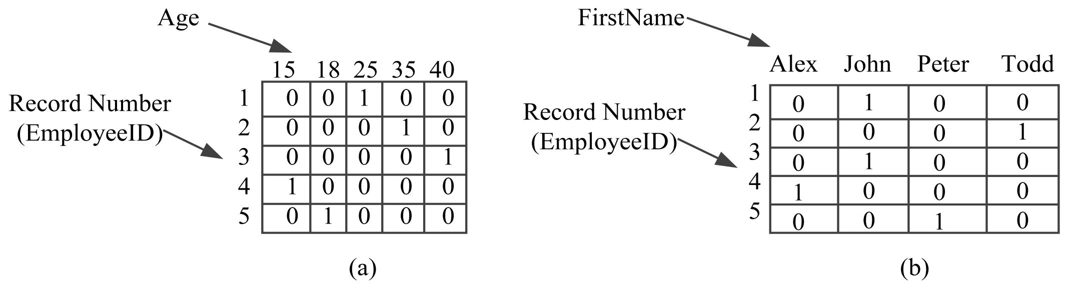


Figure 3.11: Employee Bitmap indexes (Age and FirstName).

joins between the appropriate cuboid/view and dimension tables are required. Since records of an encoded dimension table have integer numbers, then a bitmap index data structure for each non-hierarchical attribute allows us to easily find those records (record numbers) containing specific values on a given attribute in the dimension. In practice, for each non-hierarchical attribute we provide one bit string for each distinct value on the dimension (i.e., the cardinality). For k-non-hierarchical attributes, with each attribute having m distinct values, we would therefore have *(k*m)* bit strings. In practice, a compression technique (typically Run Length Encoding) is used to make the bitmap indexes reasonably space efficient.

The encoded dimension of Figure 3.3 (Employee) consists of two non-hierarchical attributes, Age and FirstName. The current dimension has five records (i.e. five employees), numbered 1 through 5. Figure 3.11(a) illustrates the bitmap index for the first attribute Age. The bitmap consists of five bit strings, each of length 5. Also, because there are four different values for the FirstName attribute, the bitmap index for this attribute has four bit strings as shown in Figure 3.11(b). In each string, the 1's indicates the encoded values for the primary key.

The advantage of bitmap indexes for non-hierarchical attributes is that they allow us to identify the most detailed encoded integer values (e.g. EmployeeID) with specified values in several non-hierarchical attributes without retrieving any records from the specified dimension. To identify the records holding a random subset of the values from a given non-hierarchical attribute, we can do a binary OR on the bit-strings from that attribute. Also, to identify the partial matches on a group of non-hierarchical attributes in a given dimension, we can simply perform a binary AND on the OR maps from each attribute. Figure 3.11 illustrates how to identify those employees (EmployeeID) with Age $>$ 25 and FirstName = John. These binary operations (AND and OR) can be done quite quickly since binary operations are natively supported by the CPU.

Recall that the cube in our server contains the most detailed values from dimension tables as feature attributes (EmployeeID, ProductID, etc.). As such, bitmap indexes allow us to eliminate joins between the cube and dimension tables whenever non-hierarchical attributes are involved within the OLAP analysis. For example, consider the encoded fact table shown in Figure 3.6 and the bitmap indexes of Figure 3.11. Assume that the user asks the following query: what are the total sales of all employees who are older than 25 and their first name is John? Since Age and FirstName are non-hierarchical attributes, then we use their bitmap indexes to answer the query's condition (Age $>$ 30 and firstName = john). EmployeeID 3 satisfies the condition (See Figure 3.11). Subsequently, we access the fact table and aggregate the total sales where EmployeeID = 3. The final result of this query is (Total_Sales = 210).

FastBit is used to create a very efficient compressed bitmap index. Fastbit uses the Word-Aligned Hybrid compression mechanism to compress the bitmap indexes [41]. This compression scheme produces a FastBit compressed index that is up to 10 times faster than the compressed bitmap index (run-length encoding) implementation

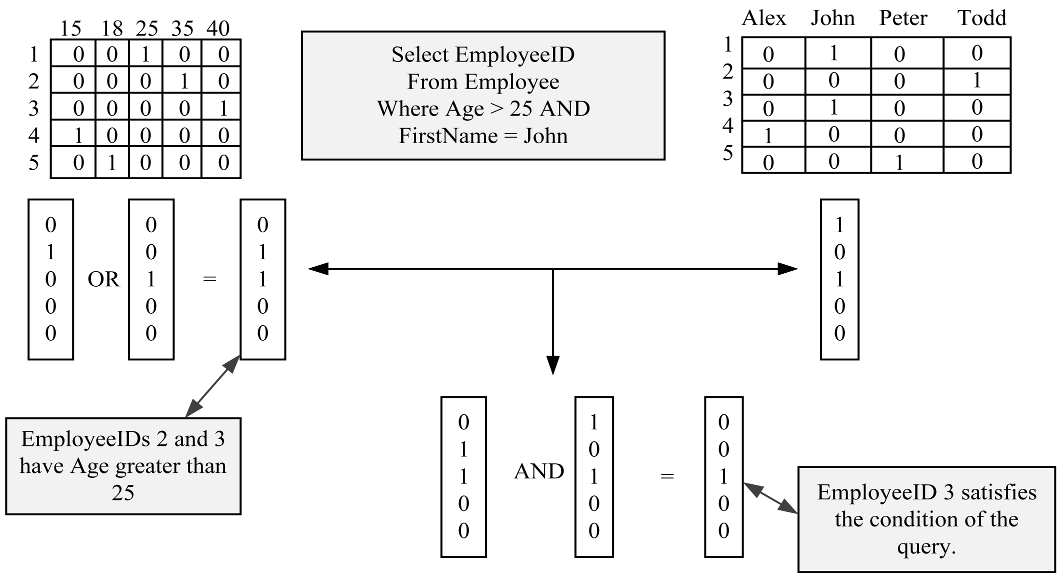

15 18 25 35 40
1 0 0 1 0 0
2 0 0 0 1 0
3 0 0 0 0 1
4 1 0 0 0 0
5 0 1 0 0 0
Select EmployeeID From Employee Where Age > 25 AND FirstName = John
Alex John Peter Todd
1 0 1 0 0
2 0 0 0 1
3 0 1 0 0
4 1 0 0 0
5 0 0 1 0
0 1 0 0 0 OR 0 0 1 0 0 = 0 1 1 0 0
1 0 1 0 0
EmployeeIDs 2 and 3 have Age greater than 25
0 1 1 0 0 AND 1 0 1 0 0 = 0 0 1 0 0
EmployeeID 3 satisfies the condition of the query.

Figure 3.12: Usage of Bitmap Indexes.

from popular commercial database management (DBMS) product (e.g., Microsoft and Oracle). FastBit compressed bitmap indexes for each non-hierarchical attribute also provide very efficient searching and retrieval operations compared with B+ tree and B* tree. Given FastBit's performance and its open source licence, we chose to integrate the FastBit library with our ROLAP Sidera Server. This integration allows us to create a set of compressed bitmap indexes for non-hierarchical attributes (e.g., Age from dimension Employee, Year from dimension Product). In other words, after the integration of the FastBit component with our server, all non-hierarchical attributes are represented as FastBit compressed bitmap indexes that are created and stored on the local disk.

3.5 Cube Indexing Integration

As was discussed in Subsection 3.4.2, Sidera uses the encoded fact table (i.e. see Figure 3.6) to generate the full cube as 2^d views in a d-dimensional space. Recall that the data in the cube is physically stored only for the most detailed levels (e.g., CustomerID, ProductID). Moreover, the Sidera indexing component described in Section 2.4 illustrates how the full cube can be materialized for a d-dimensional space as $2*(2^d)$ physical files. In other words, the indexed R-tree for each cuboid is stored on disk as two physical files: .hil file that houses the data in Hilbert sort order, and .ind file that houses the R-tree index metadata and the bounding boxes that represent the index tree. For example, consider the 3-dimensional cube called (Sales) that was depicted in the cube lattice of Figure 3.7. The indexed cube is stored on disk as 16 physical files that are illustrated in Figure 3.13. Recall that A refers to the dimension Product, B to the dimension Employee and C to the dimension Customer. ABC represents the view Product-Employee-Customer (e.g. the encoded fact table of Figure 3.6), while BC represents the view Employee-Customer, etc. As previously noted, the number of physical files that represent the indexed cube in our server grows exponentially as the number of dimensions increase. In addition, these files (.hil and .ind file in Figure 3.13) are not databases and are not particularly efficient for OLAP queries. With the benefits Berkeley DB can provide we can have a simpler, easy-to-manage and more intuitive representation of an indexed materialized cube.

We will now look at the integration of the embedded Berkeley DB into the Sidera indexing component (discussed in Section 2.4). While Berkeley provides four access methods (BTree, Hash, Recno and Queue) that perform very well in the context of the primary index, it is not sufficient in its current form to efficiently support the

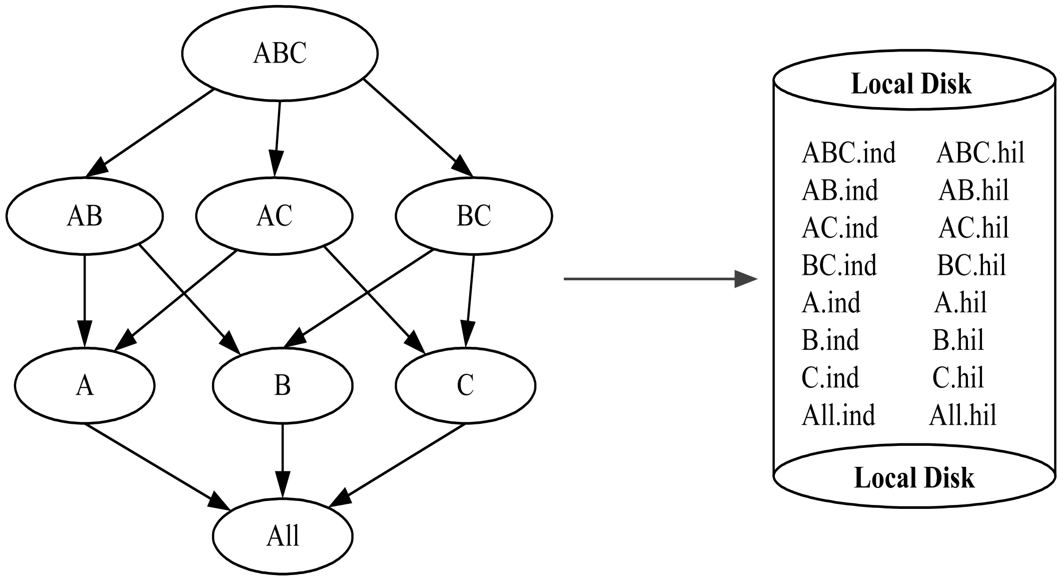


Figure 3.13: Three-dimensional indexed cube (Sales) on local disk.

multi-dimensional queries that are executed by the Sidera sever. Specifically, while Berkeley "understands" the notion of index/data combinations, it has no mechanism to directly support multi-dimensional R-trees.

We have described the Sidera cube indexing component and the concepts of Berkeley DB and environments in Section 2.4 and 2.5 respectively. The integration process consists of combining the source code written in C++ for the cube indexing with the code of Berkeley DB. Then we compile the source code with support for the C++ API. Finally, we install the compiled code that contains the compiled Berkeley DB and the compiled code that supports the building of R-tree cube. After this integration, Berkeley DB can be used to create a Berkeley DB database with the Hilbert R-tree access method. Moreover, the data in the R-tree Berkeley DB database is stored and retrieved by using the C++ APIs that are provided by the

Berkeley DB. In other words, one can write a C++ application that creates and manipulates the Berkeley DB database R-tree by using the same set of Berkeley DB C++ APIs. However, we add one database type called DB_RTREE as would be used for a standard B-tree to the existing database types in the Berkeley DB (DB_BTREE, DB_HASH, DB_QUEUE, DB_RECNO, or DB_UNKNOWN). Finally, after the integration of Berkeley DB with our code, the developer must be aware that there is a new database type called DB_RTREE that can be used with the original Berkeley C++ APIs in order to create and manipulate a Berkeley DB database with the R-tree access method.

3.5.1 Berkeley R-tree Model

As described in Section 2.5, Berkeley supports the storage of many databases (i.e.Berkeley Database Objects) in one physical file. Internally, this physical file contains one master database supported by the BTree access method that has references to all the databases that are stored in the same file. Keys in this primary BTree are the database names that are stored in the physical file, and the data of the primary BTree consists of the meta-data page number for each database name.

After the integration of the source code (Berkeley and Sidera), instead of building the indexed cube as $2*2^d$ physical files, we build the indexed cube as $O(2^d)$ Berkeley database objects for a d-dimensional fact table, one Berkeley database with the R-tree access method for each materialized group-by. In addition, these Berkeley databases are stored in one physical file and are associated with the same Berkeley environment. Specifically, the Berkeley DB physical file contains a master database that has references to all Hilbert R-tree indexed group-bys (stored as Berkeley databases) in the same file. Keys in the master database are the indexed group-by name, and the data

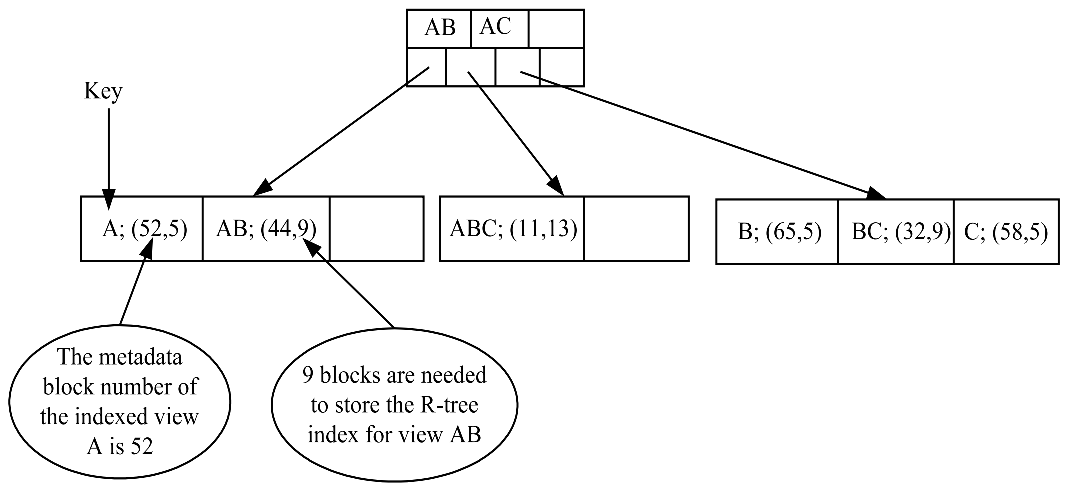

AB
AC
Key
A; (52,5)
AB; (44,9)
ABC; (11,13)
B; (65,5)
BC; (32,9)
C; (58,5)
The metadata block number of the indexed view A is 52
9 blocks are needed to store the R-tree index for view AB

Figure 3.14: Berkeley Btree index that has references to all indexed group-bys stored in one Berkeley DB physical file

contains two values: the indexed cuboid page meta-data number and the size in terms of the number of blocks occupied by this indexed cuboid. Consider the 3-dimensional cube (Sales) represented in the lattice cube of Figure 3.6. The names of the group-bys in the cube are: ABC, AB, AC, BC, A, B and C. Recall that Product = A, Employee = B and Customer = C. So for example, AB is the Product-Employee view. For simplicity, we ignore the “All” view. Figure 3.14 illustrates the master BTree that has references to all indexed group-bys that will be stored in one Berkeley DB physical file. For example, we can see in Figure 3.14 that the metadata of the indexed view ABC is stored at block number 11 and this indexed view needs 13 blocks to be stored.

Algorithm 2 describes how the Hilbert R-tree indexed cube is stored as Berkeley databases in one physical Berkeley DB file. To begin, we create and open a Berkeley DB environment handle (i.e. dbenv) that encapsulates one or more Berkeley database objects, one for each indexed cuboid in the cube. Then, for each group-by X in the cube, we create the Hilbert R-tree indexed view as a Berkeley DB database with the

Algorithm 2 Hilbert R-tree Indexed Cube

```
Input: A set S of group-bys and a cube name called C.
Output: Hilbert R-tree indexed group-bys stored as Berkeley database objects in
    one physical file.
1: Create and initialize a Berkeley environment dbenv.
2: For each group-by named X in S
       • Create a database handle db.
       • Associate the database handle db with the Berkeley environment dbenv.
       • Open the database using the handle db. This database handle db represents
         the Hilbert R-tree index for X.
         db.open(NULL, C, X, DB-RTREE, DB-CREATE, 644);
       • Close the database handle db.
3: Close the Berkeley environment handle dbenv.
```

database type DB_RTREE in the open method (i.e. db.open()). Note that the original open method in the Berkeley DB opens the database associated with a Berkeley DB physical file that supports the following access methods: Btree, Hash, Queue and Recno. Also, after open is called, we can read or write from or to the Berkeley DB database by using the putting and the getting methods (get(), put(), etc.). However, using the Berkeley DB open method with the database type (DB_RTREE) and a view X means that if the Berkeley DB database (X) doesn't exist then we first create it and store in it the Hilbert R-tree index corresponding to group-by X. In the open method, the name of the physical file that will be used to back one or more Berkeley DB R-tree databases will be the name of the cube (e.g., C).

In Section 2.4, we explained how the Hilbert R-tree indexed group-by is created and stored as two physical files: the .ind file houses the R-tree index of the group-by and the .hil file stores the data of the group-by in Hilbert sorted order (See Figure 2.7).

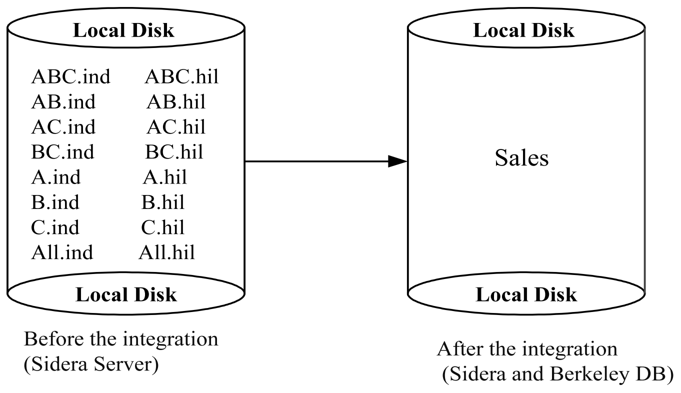


Figure 3.15: Hilbert R-tree indexing of the three-dimensional cube (Sales) before and after the integration of the Berkeley DB with our server.

When we open a Berkeley DB database handle with a database type equivalent to DB_RTREE, a Hilbert R-tree indexed group-by is created and stored in one Berkeley DB physical file that has the same name as the cube. Figure 3.15 illustrates that the index of the three-dimensional cube (Sales) — that is represented as eight cuboids in the data cube lattice of Figure 3.6 — is stored on disk in one single Berkeley database file called Sales, instead of the sixteen standard non-database files shown in Figure 3.13. The physical file Sales in Figure 3.13 encapsulates a Berkeley DB internal Btree and eight R-tree indexed cuboids (ABC, AB, etc.). Note that Figure 3.14 illustrates the structure of this internal Btree that has references to all indexed group-bys that are stored in the same file. Recall that letter names refer to dimensions (e.g. A refers to dimension Product).

As was discussed above, the internal master database (BTree) is used to store references to all indexed group-bys that are stored in one physical file. For example in Figure 3.14, the metadata block of the group-by BC is stored at block number 32, while its size is 9 blocks. The metadata block of a given group-by contains the structure of its R-tree index. In other words, it contains the group-by name, the number of index blocks, the number of data blocks, the block number of the root, the block number of the first data block and the size of each block. Moreover, the construction mechanism of the Hilbert R-tree index itself (after the integration of Berkeley DB with Sidera) follows the same mechanism that is used with the Sidera server. However, many R-tree indexes are stored in the same file. Therefore, we use the Hilbert R-tree index for a group-by implies that for an m-block index, the index block ID values will run from block i, …, Block *(i + m - 1)*, where i is the block number of the root in the R-tree index (e.g. block 32 stores the root of the indexed group-by BC), with the IDs strictly increasing in value in a top-to-bottom/left-to-right fashion. Furthermore, the blocks of the data set are also stored in consecutive blocks in the physical file. Figure 3.16 illustrates the structure of the Hilbert R-tree for the group-by BC. We can see in the metadata that the root block is at block 33 (B33) and there are 3 indexes (B33, B34 and B35). Also, the data blocks are stored in consecutive blocks (26, 27, 28, 29, 30 and 31). Figure 3.17 shows how this indexed group-by is stored in two physical files before the integration. Note that it is not necessary to have in the metadata of Figure 3.17 the first data block number and the root number.

Finally, in Figure 3.18 we see how we store the Hilbert R-tree index in one physical Berkeley DB file for the three-dimensional cube (Sales in Figure 3.6) that has eight group-bys (ABC, AB, AC, etc.) represented in the cube lattice. Specifically, this figure illustrates the index cube as a set of blocks. We have some block numbers (0, 1, 36, 37 and 60) that represent the internal Berkeley Btree master database. Note that this Btree is shown in Figure 3.14. Moreover, for each indexed group-by, the following blocks are required: one block to store the metadata, consecutive blocks to

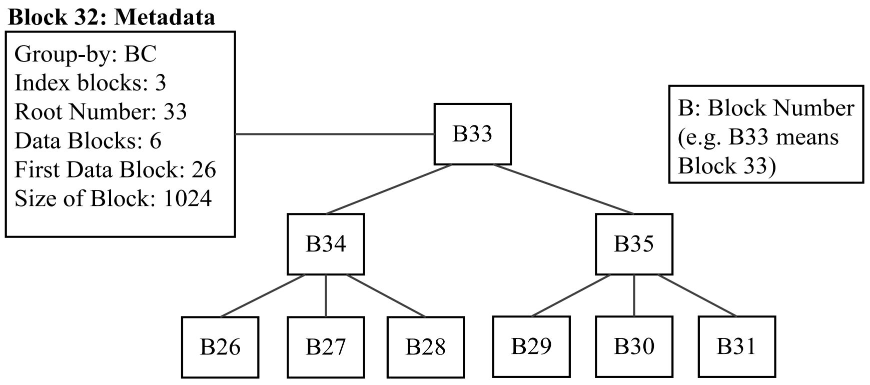


Figure 3.16: Blocks occupied in the Berkeley DB physical file for the R-tree index for group-by BC.

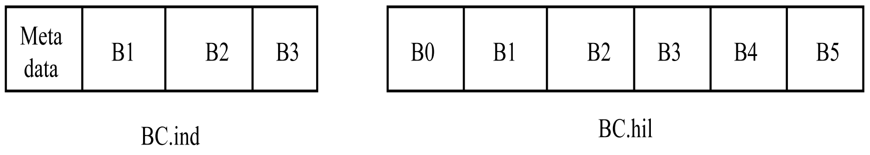


Figure 3.17: Hilbert R-tree index physical files.

store the data blocks in their Hilbert ordered form, and consecutive blocks to store the Hilbert R-tree index. For example, the metadata for the indexed group-by (ABC) is at block number 11, blocks (2, 3, 4, …, 10) are used to house the data set, and block 12 to block 15 store the index blocks. Finally, Figure 3.18 illustrates that the indexed cube of the 3-dimensional cube (Sales) is stored in one physical Berkeley DB file called (Sales) that consist of 66 blocks/pages. Moreover, without the integration of the Berkeley DB into our server, Figure 3.19 shows how this indexed cube is stored on disk as 14 non-database, standard disk files (we ignore the "All" group-by). For example, the R-tree index for group-by AC is stored in two files: AC.ind and AC.hil. AC.ind consists of the metadata block (block 0 in the file) and three index blocks (block 1, 2 and 3 in the file). AC.hil consists of six blocks (block 0 to 5) that house the data set for the group-by AC in Hilbert order form.

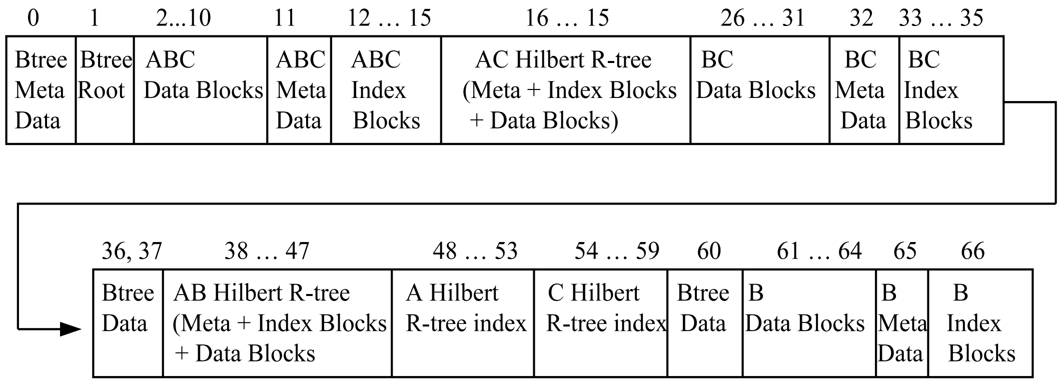


Figure 3.18: The physical structure of the indexed cube for three-dimensional cube (Sales).

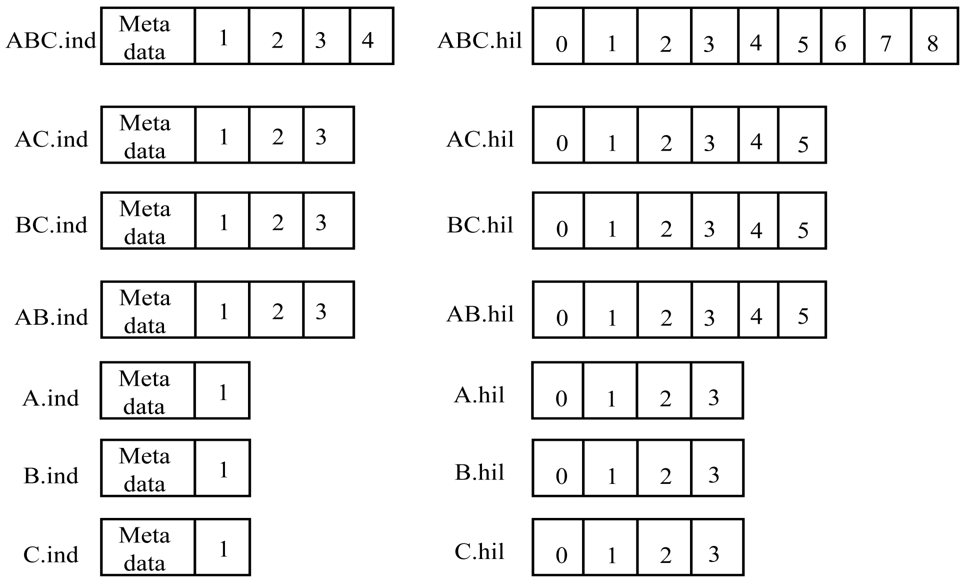


Figure 3.19: The indexed cube for the three-dimensional cube (Sales) without using Berkeley DB.

3.6 Integration of Berkeley DB with other Sidera Components

Taken collectively, the software architecture on each backend node forms a clean modular design. In Section 2.4, we discussed how the various components of the Sidera backend node fit together into larger framework. In addition, we showed the main subsystems of Berkeley DB and the relationships between them. In Section 3.4, we explained how hierarchical and non-hierarchical attributes are stored as Berkeley DB Recno databases and as a set of FastBit compressed bitmap indexes respectively. Moreover, in Section 3.5, we explained how the cube indexing is integrated with Berkeley DB. Our purpose in this section is therefore to describe the integration of the other components in the Sidera backend node (hierarchy manager, cache, etc.).

mapGraph in the Sidera OLAP server is used to provide advanced functionality that would not be available within a standard database system. In particular, it is used to support efficient real-time transformation between arbitrary levels of the dimension hierarchy [39]. While Berkeley DB is used to store the indexed cube as Berkeley databases in one Berkeley physical file, supporting drill down and roll up operations rely to a great extent on the integration of the mapGraph with Berkeley DB. Therefore, we have integrated the code implementation of the hierarchy manager with Berkeley DB to support the hierarchical OLAP queries. After the integration, the mapGraph is initialized with hierarchy meta data that describes the multi-level dimension hierarchies. As was discussed in section 3.4, the data of each hierarchical attribute is stored in a Berkeley DB database with the Recno access method. When hierarchies are processed, the mapGraph is built in the main memory by reading all

Berkeley DB databases that contain the data of the dimension hierarchies. The integration of the cube indexing and hierarchy manager components with Berkeley DB provides an effective indexing mechanism for the cube model, and supports the cube operations (Roll-up, Drill-down) that are particularly common in practical applications.

In addition to the hierarchy manager, the OLAP caching mechanism must be integrated with Berkeley DB. Note that in this context, the term “caching” has nothing to do with the Berkeley caching component that stores recently accessed disk blocks. In the current case, we are referring to Sidera’s own multi-dimensional cache that stores recently accessed cube queries. After the integration, our OLAP server has two caches — Berekeley DB caching and Sidera multidimensional caching — that store recently accessed disk blocks and recently resolved OLAP queries.

Finally, we have integrated the Sidera’s View Manager component with Berkeley DB. The View Manager maintains meta-data of the indexed group-bys that are stored in the Berkeley physical file. In short, the ViewManager is used to select the database object that will most efficiently answer the pending query. The View Manager is initialized by scanning the primary master Btree database that contains references to all indexed group-bys.

Figure 3.20(a) illustrates how the Hierarchy, Caching, View, and Cube Indexing components sit on top of FastBit and Berkeley DB. Figure 3.20(b) provides a slightly more detailed representation of the same information. In Figure 3.20, we see the software on each processing node of the Sidera server (Backend server). We note that we have not integrated all of the Berkeley subsystems at this time. In particular, given the primarily read only nature of OLAP DBMSs, the transaction and locking

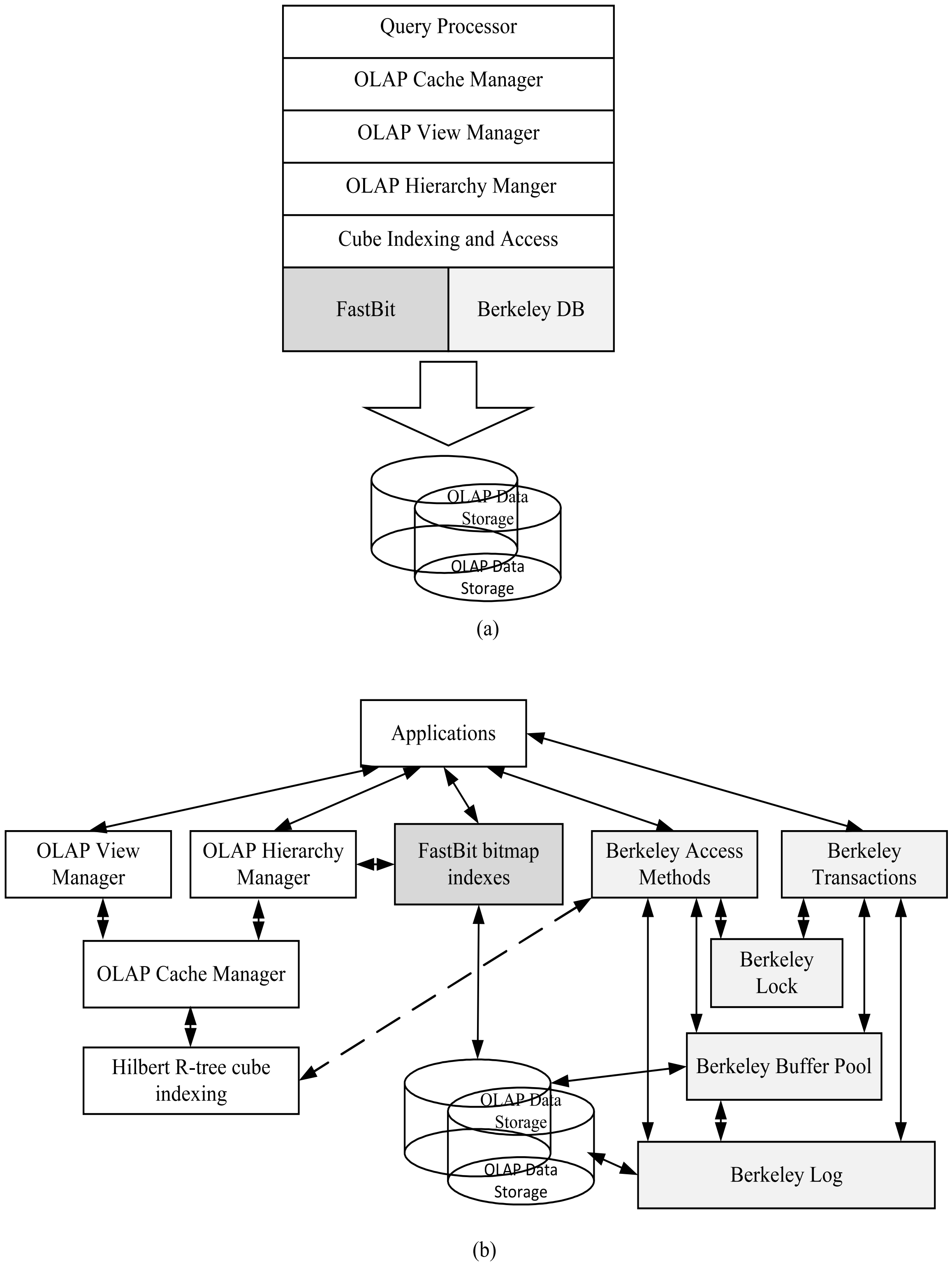


Figure 3.20: Block diagram of the software architecture on each local processing nodes.

mechanisms are not as crucial (though they may be added at a later date).

3.7 Backend Processing Logic

Figure 3.20 shows the software model on each of the p nodes of the backend in the Sidera server. As previously noted, each backend node executes exactly the same application code. In this section, we discuss the processing logic on the backend server instances.

3.7.1 The Query Resolution Algorithms

The initial query engine in the Sidera backend server, described in Section 2.4, was designed for querying multi-dimensional data in the presence of hierarchies. It wasn't particularly robust, however, since it was little more than a standalone simulation. Algorithm 3 provides the updated algorithm of the backend server instances after the integration of Sidera backend codes with Berkeley DB. We stress that the initial query engine in Sidera pre-dates the work in this proposal. Our purpose in this section is therefore to describe the query engine after the integration of our backend server components with Berkeley DB. The original Sidera query engine answers very simple OLAP range queries that have been hard-coded. Specifically, non-hierarchical attributes are not supported as well as hierarchical native values. Therefore, Algorithm 3 doesn't need to use the FastBit components that are used to store the non-hierarchical attributes in a set of compressed bitmap indexes (They are used by the query engine discussed in the following chapters).

To begin, the local instance first initializes the mapGraph hierarchy manager with meta data describing the multi-level dimension hierarchies (identical on each node),

Algorithm 3 Sidera Backend Query resolution after the integration

```
1: Initialize the main backend server
2: Initialize the mapGraph Hierarchy Manager (hM) with the dimension hierarchy
   meta data
3: Initialize the View Manager ($vM$) with meta data about the physically stored
   cube
4: while server is running do
5:    receive a set M of user-defined query (uQ) parameters
6:    invoke tranformQuery(uQ, hM, vM)
7:    check the Cache Manager (cM) for a match on uQ
8:    if a valid match is found in the local Cache Manager (cM) then
9:       get initial result $I$ from the Cache Manager
10:   else {otherwise, go to disk to answer the query}
11:      Check the cache Manager (cM) for a match on the surrogate view $V$ that
         was selected to resolve the user query.
12:      if a valid match is found in the local Cache Manager (cM) then
13:         get initial result $I$ by invoking processQuery(uQ, db)
14:      else
15:         Create a Berkeley database (db) and associate it to dbenv.
16:         Open the Berkeley database (V) that contains the Hilbert R-tree index for
            group-by V.
17:         get initial result I form disk, $I$ = processQuery(uQ, db)
18:         Add db and V to the cache Manager cM
19:      end if
20:   end if
21:   final result $R$ = perform OLAP post processing on the initial result $I$.
22:   if results required on front end then
23:      collect $R$ with MPI_Allgather
24:   end if
25: end while
```

initializes the View Manager by reading the master Berkeley database to get the format and the sort orders of the indexed views stored physically on disk, and then initializes the Berkeley environment that contains references to all the Berkeley databases sharing the environment and shared memory segments for caching databases and caching control information.

Before query resolution actually takes place, the raw query is transformed, taking into account the hierarchical specifications. An initial result is then obtained either from the OLAP Sidera cache or, if necessary, from disk. If obtained from the cache, the cached attributes are re-ordered to match the order of the user query. If disk access is required, we first have to check if the surrogate target — the view V that can satisfy the actual user request — is found in the local cache. If V is found, this means that a Berkeley database that represents the Hilbert R-tree index for group-by V was opened and cached in the Berkeley Cache Manager, in which case the Berkeley database will be utilized. Otherwise, we open a Berkeley database that represents the Hilbert R-tree index for group-by V to answer the user query. Then, initial data is retrieved via the Hilbert R-tree Berkeley database and is added to the Sidera OLAP cache. Query-specific post processing is then performed. Finally, because Sidera is a fully parallelized query engine, the partial results may need to be collected and merged before presentation to the end user. We use a standard Gather operation from the MPI libraries (Message Passing Interface).

3.7.2 Query Transformations

Algorithm 3 utilizes a function called **transformQuery** to convert the user query into a hierarchy-aware form that can be utilized by the query engine. This algorithm is described in Algorithm 4. The primary function is to create new range and hierarchy

Algorithm 4 Query Transformation Algorithm

```
Input: A user-defined query $uQ$ containing dimension set $M$, a hierarchy manager
    $hM$, and a view manager $vM$.
Output: Optimized query format.
 1: retrieve the actual view $V$ from the view Manager, where $V$ contains dimension
    set $T$, $M \subseteq T$.
 2: create a new attribute range array $newR$ of size $|T|$.
 3: create a new hierarchy range array $newH$ of size $|T|$.
    {populate $newR$ and $newH$}
 4: for each attribute $i$ in $V$ do
 5:   if $uQ$ contains $V[i]$ then
 6:     $low$ = the range minimum for $V[i]$
 7:     $high$ = the range maximum for $V[i]$
 8:     $l$ = hierarchy level for $V[i]$
 9:     if $l$ does not represent the base level then
10:       set newR.low = hM.getBaseLow($V[i]$, l, low)
11:       set newR.high = hM.getBaseHigh($V[i]$, l, high)
12:     end if
13:   else
14:     set high/low wildcards
15:   end if
16: end for
17: update the current user query $uQ$ with $newR$, $newH$, and $V$.
```

arrays. The range array provides the new high/low values for each of the A_i attributes in the user query. These are specified in terms of the base attribute. The hierarchy array will continue to reflect the hierarchy level requested by the user but will be updated with wildcards to indicate full range matching on peripheral attributes. By peripheral, we mean dimensions that were not part of the user query but that may be part of the surrogate view selected to actually resolve the user query.

3.7.3 Query Processing

The Berkeley database object that represents the Hilbert R-tree index for the surrogate has been opened after the query transformation. Note that the Berkeley database object is initialized by reading the database metadata that gives the full structure of the R-tree (e.g. number of index blocks, number of leaf blocks, number of levels, etc). If cached results are not available, the **processQuery()** function is used to retrieve the initial data that comes from the disk via the Berkeley database Hilbert R-tree index. This process is described in Algorithm 5. Note that the core algorithm for query processing is the Linear Breadth First Search strategy [96] that is used in the Sidera Server. Our purpose here is therefore to describe how the query is processed after storing the indexed cube (indexed group-bys) in one Berkeley physical file.

Algorithm 5 describes the search strategy that is used to manipulate a Hilbert R-tree index. The algorithm traverses the node in the R-tree based on the BFS technique that visits the node in the tree level by level in a left to right fashion. Queries are answered as follows. For a level i of the tree, the algorithm identifies at level i-1 the j nodes(block numbers) that intersect the user query. It places these block numbers into a list W. Using the block numbers in W, the algorithm traverses the blocks at level i-1 and replace W with a new list W. This procedure is repeated until

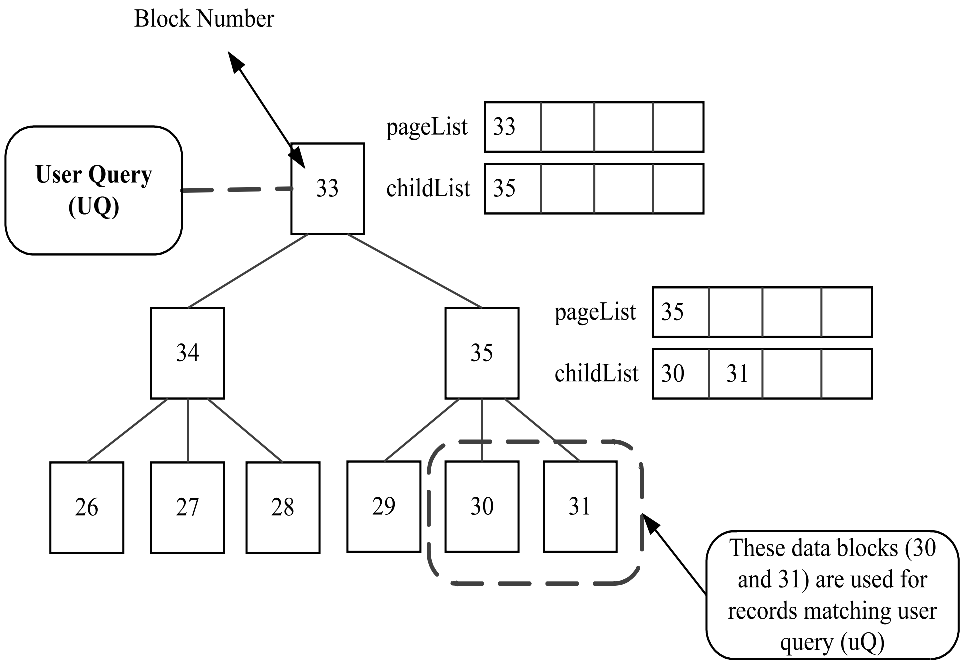


Figure 3.21: Hilbert R-tree Search.

the leaf level has been reached. At this point, the algorithm identifies and returns the d-dimensional records encapsulated by the user query uQ.

Figure 3.21 illustrates the corresponding graphical depiction for the query processing algorithm. Suppose that the current OLAP query is resolved from group-by BC, one of the group-bys of the three-dimensional cube (Sales) mentioned in Figure 3.18. Therefore, Figure 3.21 shows how we search the Hilbert R-tree index for the group-by BC discussed in Figure 3.14. Note how the selected data blocks consist of a strictly increasing set of block numbers (30, 31). These blocks are used to identify records matching the current user query.

Algorithm 5 Linear Breadth First Search Query Processing

```
Input: A Berkeley database metadata and a user query uQ.
Output: Fully resolved query.
 1: Initialize pageList with page/block number of the root index block.
 2: while not at the leaf level do
 3:   childList = new empty list
 4:   for each page/block number in the page list do
 5:         for each block number $b$ at level $i$ do
 6:               if $b$ is found in the pageList then
 7:                     using $b$ as a block offset in the physical file that contains all
 8:                     the indexed group-bys, read the relevant disk block $B$ into
 9:                     memory.
10:               end if
11:               for each child block $j$ of $B$ that intersects $uQ$ do
12:                     add $j$ to the childList
13:               end for
14:         end for
15:   pageList = childList
16:   end for
17: end while
18: for each block number $i$ in the current page list do
19:   using $i$ as an offset, read the relevant disk block $B$ into memory.
20:   Process $B$ for records matching $uQ$.
21: end for
```

3.7.4 Post Processing

Once the initial result set has been constructed in Algorithm 3, post processing must be performed in order to produce the final result. This process is described in Algorithm 6. Note that the post processing routines are completely oblivious to the source of the initial result (cache or disk).

The translateHierarchyValues() function is used to map base level values in the initial result set into their appropriate counterpart at the destination level of the hierarchy (as defined by the user query). The system uses the Hierarchy Manager (mapGraph), and hierarchy array (constructed in Algorithm 4 for this purpose). A Parallel Sample Sort is performed to order records as per the user request and to permit efficient merging and aggregation. Note that the sorting subsystem is heavily optimized to minimize the movement of multi-value records. If surrogates or hierarchies have been specified, some form of additional aggregation will also be required. This is the purpose of the orderAndAggregate() function. At this point, the result is ready for its return to the user.

3.8 Experimental Results

In the previous sections, we described the integration of the backend Sidera components with Berkeley DB. In this section, our focus turns to the effectiveness of the integration. Specifically, we provide experimental testing that has shown very good results, with building of the indexed cube and the query processing superior to the old Sidera server. Recall that the existing query engine doesn't support non-hierarchical attributes; however, we provide some tests to demonstrate the advantage of using FastBit versus the standard Btree index. In the first section, we discuss the test

Algorithm 6 ROLAP Post Processing Algorithm

```
Input: user query $uQ$, initial result $I$, hierarchy manager $hM$
Output: final result $R$
 1: set the user-specified view $U$ from $uQ$
 2: set the actual view $V$ from $uQ$
 3: if $uQ$ contains hierarchies then
 4:     invoke translateHierarchyValues(uQ, hM, I)
 5: end if
 6: do parallel sample sort
    {permute intermediate results as per user request}
 7: if a surrogate was used or a hierarchy is required (or both) then
 8:     $R$ = orderAndAggregate(I);
 9: else
10:     $R$ = arrangeSortedRecords(I);
11: end if
12: return $R$;
```

environment as it relates to the hardware, software, and data that we use in our evaluation. We will then look at a sequence of tests in order to highlight the importance of our integration with respect to the old Sidera server.

3.8.1 The Test Environment

To begin, we note that all evaluations are conducted with the Sidera engine running on (1) a single Linux workstation and (2) a 17-node Linux cluster (frontend + 16 backend servers). Though Sidera is a fully parallelized system, we have used a single node test environment for some experiments because the components on each node of the parallel machine are designed to work independently. In terms of the primary test platform, it is a Linux-based workstation running a standard copy of the 2.6.x kernel, with 1 GB of main memory and a pair of 3.2 GHz dual CPU boards. Disks are 160 GB drives at 7200 RPM. All software components of the backend Sidera architecture have been implemented using C++, STL (the Standard Template Library), and the MPI communication functions (even though we run some of the tests on a single node, MPI libraries are utilized for the parallel testing). The Berkeley DB components, which are implemented in the C languages, are integrated into Sidera components. We compiled and installed the source code of the Berkeley DB core (db4.7.25) in each one of the nodes in the parallel machine.

Data sets are generated using a custom data generator developed specifically for the Sidera environment. We note that while our data generator provides a mechanism for generating skewed data distributions, we have not used them in this case since our primary goal is to measure the efficiency of the new architecture versus the original components.

Instead, values are randomly generated and uniformly distributed. With respect to the data sets, we first generate a multi-dimensional Fact Table (the dimension count varies with the particular test), with cardinalities arbitrarily chosen in the range 2-10000. Depending on the test involved, row counts typically vary from 100,000 to 10,000,000 records. The primary fact tables are then used to compute fully materialized data cubes containing hundreds of additional views or cuboids. For example, a 10-dimensional input set of 1,000,000 records produced a data cube of 1024 views and approximately 120 million total records. Once the cubes are materialized, we index the tables using the R-tree mechanism provided by the Sidera engine. The indexed cube is created in Berkeley DB as well in order to provide a comparative system.

Because individual millisecond-scale queries cannot be accurately timed, we use the standard approach of timing queries in batch mode (Note that only 15 queries are used to check the viability of the FastBit bitmap index versus Btree index). In our case, an automated query generator constructs batches of 1000 range queries against several kinds of hierarchies that are supported by Sidera server (e.g. Symmetric Strict Hierarchies, Ragged Strict Hierarchies, etc.), in which high/low ranges are randomly generated for each of k dimensions, randomly selected from the d-dimensional space, $k < d$. Sort orders are also randomly determined. We note that this form of query generation actually overestimates query response time since users typically query low-dimensional views that can be easily visualized. In the succeeding tests, five batches of queries are generated and the average run-time is computed for each plotted point. Finally, when necessary, we use the "drop_caches" option available in the newer Linux kernels to delete the OS page cache between runs.

3.8.2 Non-hierarchical Attributes: FastBit Bitmap versus Berkeley DB B-tree

To demonstrate the superiority of the FastBit Bitmap index implementation for non hierarchical attributes versus the popular Btree indexing technique implemented in Berkeley DB, we constructed and queried non-hierarchical attributes using both implementations. Here, in this test, we create a dimension (called Customer) with five non-hierarchical attributes (called Age, FirstName, LastName, Balance and Nationality) and 1,000,000 records (the cardinality of the primary key called CustomerID in the dimension Customer). We note that the primary key (CustomerID) dimension has specific integer numbers, 1, 2, ..., 1,000,000. Moreover, the cardinalities of non-hierarchical attributes are arbitrarily chosen in the range 100 - 1000.

We constructed 3 sets of queries against the Customer dimension, each set containing five queries. These queries are executed using the two indexing methods (FastBit bitmap and Berkeley Btree) that are used for non-hierarchical attributes (e.g. Age, Salary, etc.) in our server. Before discussing the results of this test, we present in Figure 3.22 the SQL format of two sample queries from each set. In Figure 3.22, we can see that the first set (Set_1) contains only look-up queries on a single non-hierarchical attribute. The second set (Set_2) includes partial queries, while the third set (Set_3) consists of range queries with one or more attributes.

Figure 3.23 shows a comparison of the running time using the two indexing implementations for three different sets of queries. For the first set (Set_1) of queries (look-up on one attribute), the running time is very close to the optimal indexing scheme (Btree). However, when we move to the more complex queries such as queries in Set_2 and Set_3 (partial and range queries on more than on attribute), there is a factor of two to three increase in running time for the Berkeley DB Btree indexing method. This is due to the efficient bitwise logical (AND and OR) operations directly supported on the compressed FastBit bitmap indexes. For a high number of non-hierarchical attributes, the performance of Btrees is poor.

Finally, in this test, we note that the size of the Berkeley DB Btree indexes is four times greater than the size of the compressed FastBit bitmap indexes. Specifically, the size of the Berkeley DB Btree indexes is 13.8 MB, while the size of the FastBit indexes is 3.5 MB for the five non-hierarchical attributes mentioned above in dimension Customer that has 1,000,000 records.

3.8.3 Single Node Experimental Evaluation

In this section, we have used a single node test environment. We will look at a sequence of tests that highlight the importance of the integration of the Sidera codes with Berkeley DB in terms of the index cube construction and query resolution. In the following two sections, we compare the index construction for the cube in a single

Set_1 (Look-up queries):

```
SELECT CustomerID FROM Customer WHERE Age = 40;

SELECT CustomerID FROM Customer WHERE Nationality = “Sudan”;
```

Set_2 (Partial-match queries):

```
SELECT CustomerID FROM Customer
WHERE (Age = 40 or Age =21) AND Balance= 1000;

SELECT CustomerID FROM Customer
WHERE Nationality = “Canada” AND FirstName = “Salas” Or Age = 40;
```

Set_3 (Range queries):

```
SELECT CustomerID FROM Customer
WHERE Age > 40 AND Age < 80 AND Balance> 1000

SELECT CustomerID FROM Customer
WHERE Balance>500 AND Balance< 1000 AND Nationality = “USA” OR Age>40;
```

Figure 3.22: Sample SQL queries.

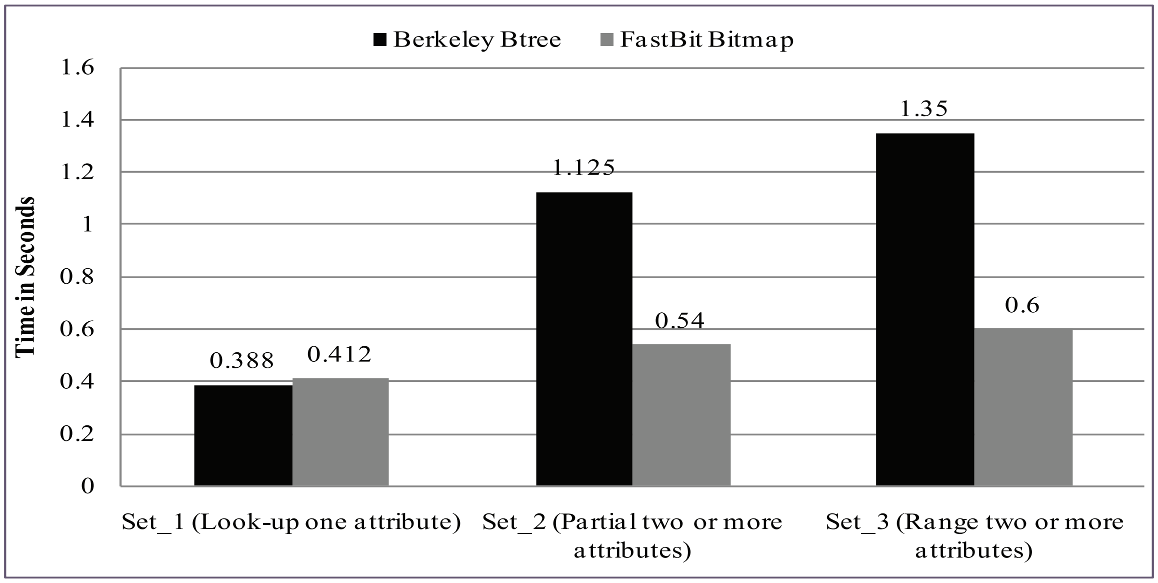


Figure 3.23: Comparison of the running time of three different sets of queries for Berkeley Btree versus FastBit bitmap.

backend node before and after the integration of the codes in Berkeley DB. Then, we highlight the execution of batches of 1000 queries in the indexed cube that is stored in Berkeley DB.

3.8.3.1 Index Construction

In this section, we compare the index construction of the cube after the integration of the Sidera codes with Berkeley DB against the current index construction that is used in the Sidera server. We look at the impact of both Fact table size and dimension count in creating the Hilbert indexed cube in Berkeley DB.

3.8.3.1.1 Fact Table Size In this section, we compare our implementation (integrating Berkeley libraries into our Sidera DBMS) against the current index construction used in the current Sidera server. The full cube (2^d) was generated from input sets (Fact Table) ranging in size from 10,000 records to 1,000,000 records, and includes 9 dimensions. The comparison will evaluate the construction of the Hilbert R-tree in Sidera DBMS after the integration against the construction of the Hilbert R-tree in the current Sidera server [38] on the same views/group-bys generated from the fact table. Figure 3.24 shows the running time for index cube construction using Berkeley DB after the integration of the Sidera access method (Hilbert R-tree) versus the Hilbert R-tree index used in Sidera server. As noted, the indexed cube in Berkeley is represented only in one single physical file; however, it is represented in 2 * the number of views in the current Sidera Sever. The running time for index construction in Sidera DBMS supported by Berkeley is much better than the current technique in Sidera Server. On average, the integration of Berkeley into our server reduces the index cube construction time by 40% - 60%. The primary reason for this reduction is

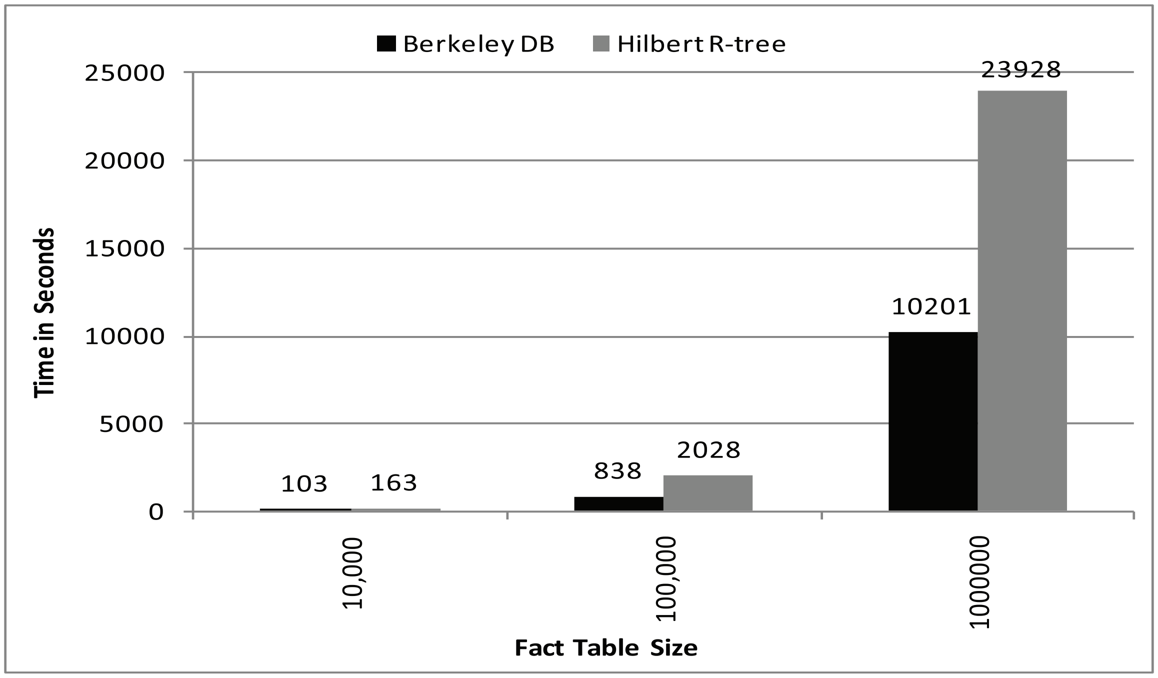


Figure 3.24: Index Construction using Sidera DBMS supported by Berkeley versus Sidera Hilbert R-tree index of the three cube sizes.

the need for only one physical file to store the index cube in the Berkeley supported Sidera DBMS. Consequently, the Berkeley DB file uses much more sequential storage for the indexed cube. In other words, everything is written in one stream on the disk. However, storing the indexed cube as $2*2^d$ physical files produces a significant amount of disk thrashing.

3.8.3.1.2 Dimension Count In addition to the impact of the data set size, we also look at the impact of an increase in dimension count on index cube construction. Recall that with d dimensions, the full cube generates (2^d) views/cuboids. For the current test, the data set size is constant at one million records. Figure 3.25 illustrates the running time for index cube construction of both Sidera DBMS after the integration of Berkeley DB and our current R-tree index in the Sidera server as the

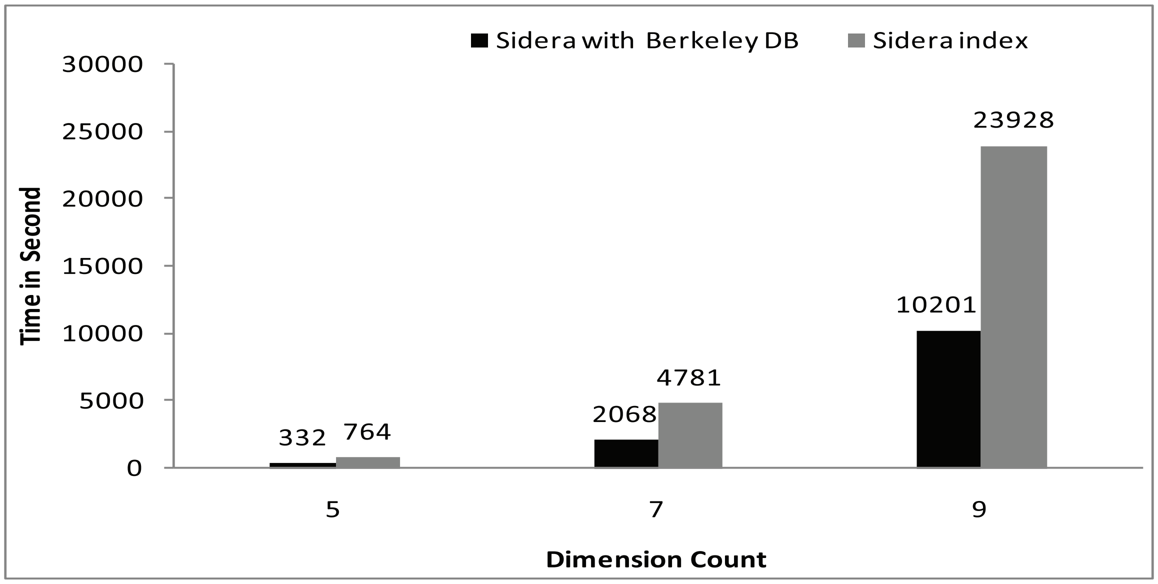


Figure 3.25: Index Construction using Sidera DBMS supported by Berkeley versus Sidera Hilbert R-tree index, as a function of dimension count.

dimension count increases to a maximum of nine, a number indicative of the maximum number one might expect to see in many practical environments. Perhaps not surprisingly, we observe that the running time when using Berkeley DB to create the indexed cube reduces by 40% to 60% due to the fact that we are storing the index cube in one physical file. Again, the indexed cube is written in one stream to one Berkeley physical file; however, the number of physical files that are used to store the indexed cube in Sidera increased exponentially as the number of dimensions is increased.

3.8.3.2 OLAP Query Resolution

Throughout this section we will look at a series of OLAP query tests, each intended to highlight the importance of the integration of the Berkeley components into our OLAP Sidera sever. Specifically, we will present the comparison between the existing

Sidera query engine and the Sidera query engine that is supported by the Berkeley DB project. Note that both engines answer only very simple OLAP range queries that have been hard-coded specifically to test our engine. Moreover, these queries are against the data in the encoded integer form. We will present some experimental evaluations that highlight the importance of the integration of the Berkeley DB into our OLAP server.

3.8.3.2.1 Query Count In this case, we create a cube from an input set of 1 million records, 9 non-hierarchical dimensions, and mixed cardinalities 100-10000, with the full cube representing over 200 million records and 12 Gigabytes of total data. We generate the Hilbert R-tree indexed cube in the Berkeley supported Sidera DBMS in one physical file and in the old Sidera server in 1024 files (2 files per view). We then use our query generator to generate batches of non-hierarchical queries. By non-hierarchical queries, we mean those queries whose ranges have been restricted to the base attribute.

Figure 3.26 shows the total response time for non-hierarchical queries in the Berkeley supported Sidera query engine versus the Sidera query engine. Results are shown for 100, 500, and 1000 non-hierarchical OLAP queries. The graph shows the improvement that we gain from the integration of the Berkeley DB into our Sidera server. In all three cases, the integration of the Berkeley code into our Sidera DBMS reduces the OLAP query resolution time by 15% - 20%.

The reduction in response time is due to a number of factors. First, in the Berkeley supported Sidera DBMS, we cache each opened Berkeley Database object, so we don't have to re-open it again if a query needs to be answered from the cached Berkeley database object. Figure 3.27 shows the average Berkeley database hit rate as the

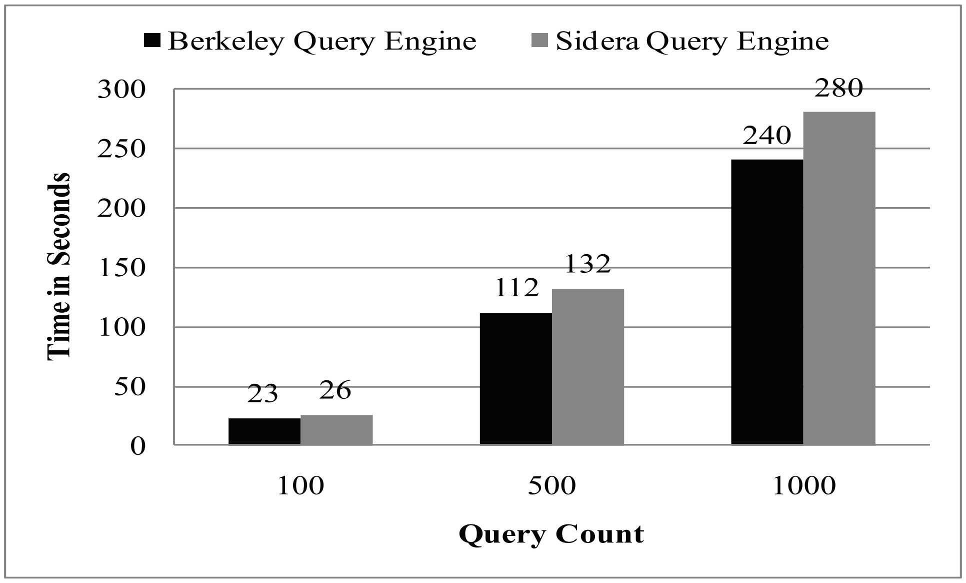


Figure 3.26: Comparison of Berkeley supported Sidera Query Engine versus Sidera Query Engine for three different query counts.

number of queries increase from 100 to 1000. Second, in order to answer a query in the Sidera server we have to open two files that represent an indexed view; however in the integrated Sidera server we begin by opening just one file that contains the indexed cube and then simply seek to the appropriate position.

3.8.3.2.2 Fact Table Size In this section, we compare our integration of the Berkeley DB libraries into Sidera OLAP server against the existing Sidera server. In this case, we create a fact table with 9 non-hierarchical dimensions. As always, we employ batches of 1000 queries. The direct comparison will, of course, evaluate the execution of the same query batches in Berkeley supported Sidera DBMS against the old Sidera DBMS. Figure 3.28 shows the running time for data sets ranging in size from 10,000 records to 1,000,000 records. Our integration scheme improves the Sidera OLAP query resolution by 15%-20% as the number of records increases in the fact

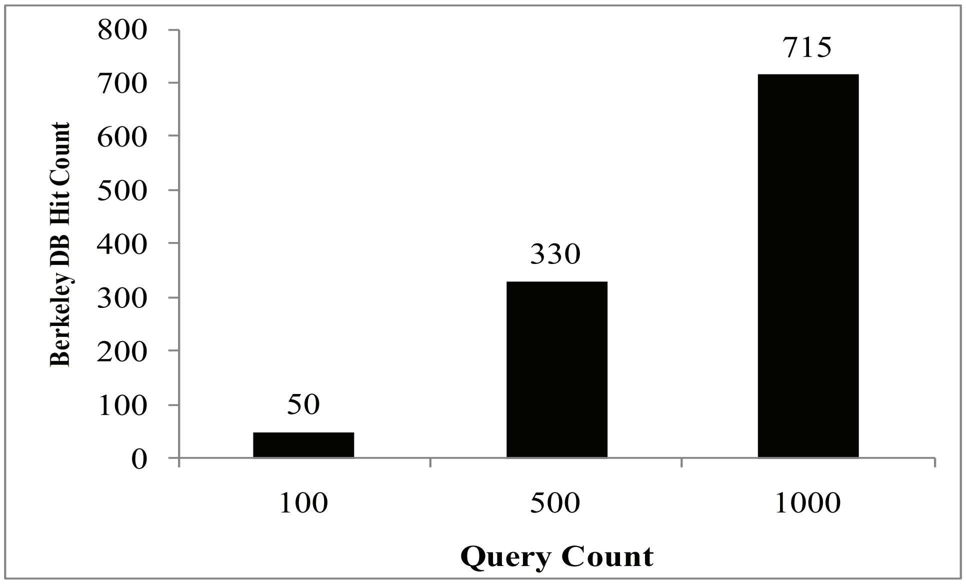


Figure 3.27: Berkeley Database Hit Rate for three query counts.

table.

3.8.3.2.3 Dimension Count In addition to the impact of the data set size, we also look at the impact of an increase in dimension count on query resolution. Recall that with the Sidera indexing, for each group-by in the cube we need two physical files. However we store all the indexed views in one Berkeley physical file. For the current test, we again generate 1000 non-hierarchical queries, this time holding the data set size constant at one million records. Figure 3.28 illustrates the performance of both our approach (integration Berkeley DB codes into our Sidera server) and the current Sidera engine as the dimension count increases to a maximum of nine. Figure 3.29 shows that our new approach again decreases the running time by 15%-25%.

3.8.3.2.4 Hierarchy Manager Recall that symmetric strict hierarchies, ragged strict hierarchies, and non-strict hierarchies are supported by the mapGraph [39]. It

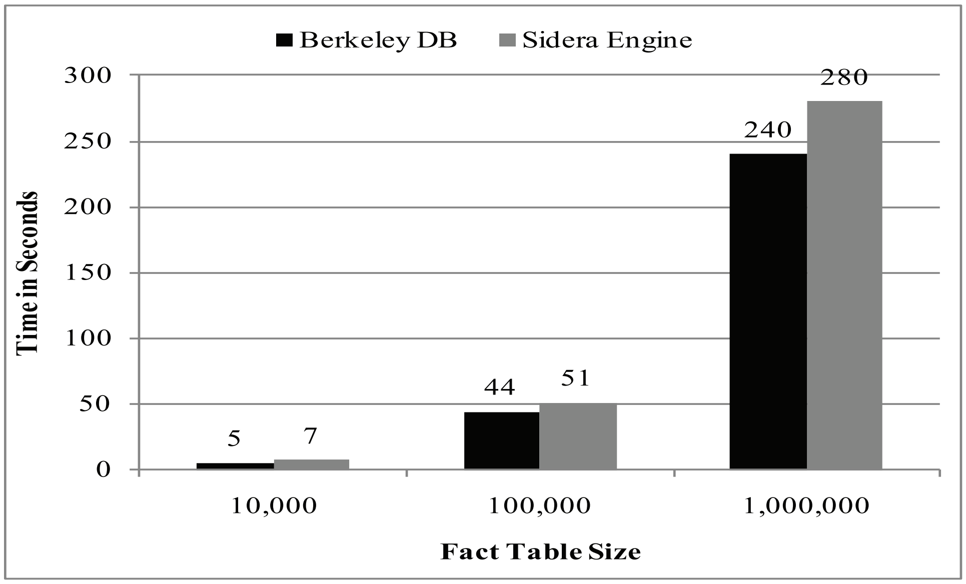


Figure 3.28: Comparison of Berkeley supported Sidera DBMS versus the old Sidera DBMS for the three different cube sizes.

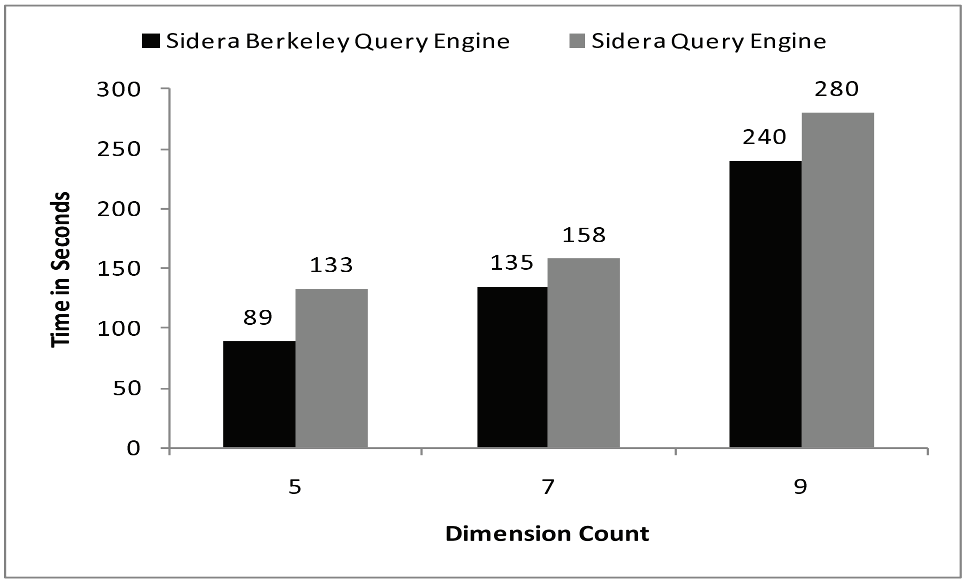


Figure 3.29: Comparison of Berkeley supported Sidera DBMS versus the old Sidera DBMS, as a function of dimension count.

is important to compare the performance of the Berkeley supported Sidera DBMS against the current Sidera engine in resolving hierarchical OLAP queries. In this case, we create 9 dimension hierarchies made up of a mixture of symmetric strict, ragged strict and non-strict forms. As always, we employ batches of 1000 OLAP queries, this time in hierarchical form only. The direct comparison will evaluate the resolution of OLAP hierarchical queries in Sidera Query engine against the Berkeley supported Sidera server on the same query batches.

Figure 3.30 shows the running time for data sets ranging in size from 10,000 records to 1,000,000 records. There are two points that must be made with respect to a direct interpretation of the results. First, answering OLAP hierarchical queries in the Berkeley supported Sidera query engine is faster than the old Sidera query engine by an average of 15%. Second, the total overhead is less than 25% relative to the non-hierarchical case described in the previous section. It is important to place these results into context. The integration of the graph manager with Berkeley DB allows it to answer hierarchical OLAP queries with a modest overhead compared to the non hierarchical case. We note that this overhead is more than acceptable given the power and flexibility that the graph manager provides.

3.8.4 Parallel Experimental Evaluation

In this section, we have used a 1, 4, 8, and 16 node test environment. We will look at a test that highlights the importance of our integration in terms of the index cube construction. The full cube (2^d) was generated from an input set (Fact Table) of 1,000,000 records, and 10 dimensions. The comparison will evaluate the construction of the Hilbert R-tree in Sidera DBMS after the integration against the construction of the Hilbert R-tree in the current Sidera server [38].

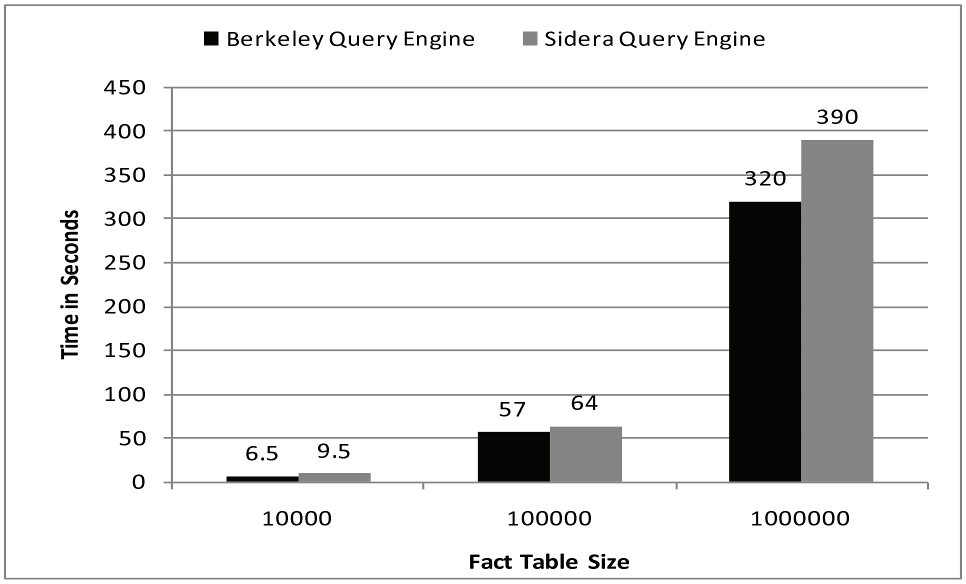


Figure 3.30: Comparison of answering OLAP hierarchical queries in Berkeley supported Sidera DBMS versus Sidera DBMS for three cube sizes.

Figure 3.31 shows the parallel wall clock time observed for index construction in the Berkeley supported Sidera server and the old Sidera server as a function of the number of processors used. The running time for index construction in Sidera DBMS supported by Berkeley is much better than the current technique in Sidera Server. On average, the integration of Berkeley into our server reduces the index cube construction time by 40% - 60%. Note that this ratio is the same as that seen on one node. This underscores the fact that each Sidera backend DBMS can be viewed as a mostly independent DBMS process. From one to 16 processors, our Berkeley supported Sidera OLAP server achieves close to optimal speedup of 14.9 on 16 processors.

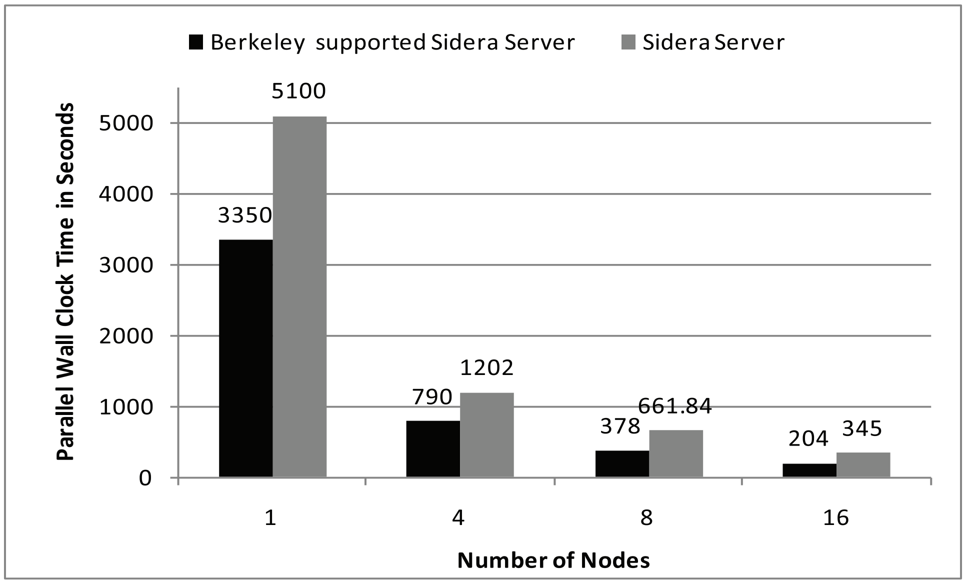


Figure 3.31: Parallel wall clock time for index construction.

3.9 Review of Research Objectives

In section 3.3, we identified a number of objectives of our research. We now review those goals to confirm that they have in fact been accomplished.

1. **Provide simpler, more intuitive representation of a materialized data cube.** Specifically, we require a single, cohesive data repository. We described the integration of the Berkeley DB components into Sidera sever. After the integration, hundreds of files that represent the Hilbert R-tree indexed cube in Sidera Server are efficiently packed into one single Berkeley DB database file. Not only is this far easier for the DBMS to manage, it also simplifies supplementary operations such as archiving and replication.

2. **Support fast indexed cube creation.** By storing the Hilbert indexed cube (O(2^d) indexed group-bys) in one Berkeley physical file, and using the Berkeley

environment that serves as a shared repository of control information, we are able to reduce the time to create the Hilbert R-tree indexed cube in the Berkeley-supported Sidera server by a factor of 15%.

3. **Provide an efficient encoded form of the data warehouse.** Specifically, we maintain the notion of attribute linearity in the hierarchy. This ensures that the base level has consecutive integer values. In addition, for each non-hierarchical attribute in a dimension, we ensure that the primary key of the dimension is an integer with a consecutive integer form. Finally, the fact table contains the integer-values of the primary keys for all dimensions that surround it. This compact integer format of the data decrease storage requirements of the materialized cube and also improves performance for key operations such as conditional checks.

4. **Support fast hierarchy manager (mapGraph) creation.** We mentioned in Section 3.4 how hierarchical attributes are stored in their linear form in Berkeley DB databases by using the simplest and the fastest access method available in Berkeley DB (Recno access method). This access method allows very fast creation of the hierarchy manager at run-time, as it is ideally suited to sequential read environments.

5. **Ensure efficient and fast storage for non-hierarchical attributes.** By storing non-hierarchical attributes as a set of compressed FastBit bitmap indexes, we are able to transform queries with restrictions on these attributes to values of the base attribute level (record number). This result is improved performance relative to the most popular indexing method (Btree).

6. **Support locking/concurrency, file system caching, data redundancy and other core DBMS functionality.** Though we are not directly untilizing all of these features at the present time, they will greatly increase the functionality of the system in the future. For example, we can provide a far more flexible storage architecture that includes concurrency, thread management, caching, durability, etc.

7. **Provide efficient query execution for complex hierarchical OLAP queries.** The Berkeley supported Sidera server reduces the running time for hierarchical OLAP queries relative than the old Sidera server. It does this by the integration of Berkeley functionalities with the Sidera server.

8. **Provide an infrastructure that permits the exploration of new research themes (e.g., OLAP query optimization).** We have chosen to merge our existing Sidera code base with the open source Berkeley project and the FastBit bitmap indexes in order to provide a very efficient OLAP storage engine. This OLAP storage engine allows us to focus on new research ideas such as OLAP query languages (subject of Chapter 4) and OLAP query optimization and execution (will be discussed in Chapter 5).

3.10 Final Thoughts

Recall that the Sidera server is designed to be an OLAP DBMS. In this chapter we have described an efficient and reliable storage engine that is used specifically for our Sidera target platform. However, the same principles and methods could be applied to integrate our storage engine into other DBMS systems like PostgreSQL or MySQL. Specifically, we could use our storage engine model — with or without Berkeley

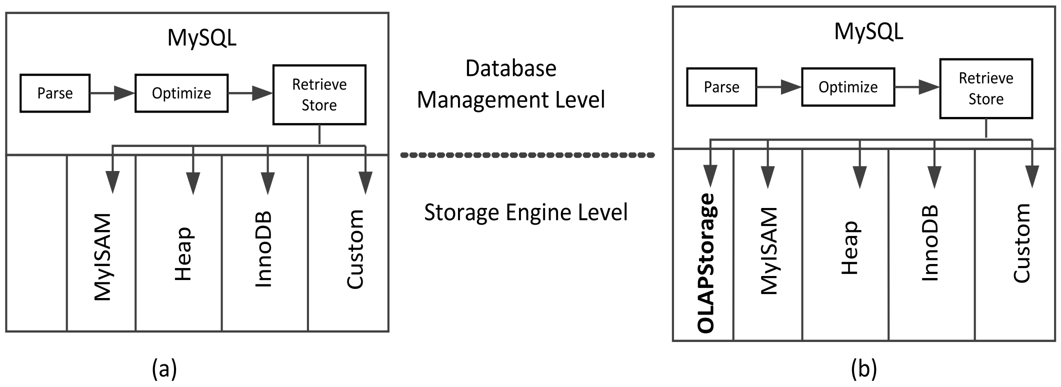


Figure 3.32: (a) MySQL Basic Architecture (b) MySQL Architecture after adding our OLAP storage engine (OLAPStorage).

DB components — to add R-tree and bitmap data warehouse storage (cube and dimensions) to these other DBMSs. For example, MySQL supports numerous storage engines such as MyISAM, InnoDB, Archive, MEMORY, etc. [81]. Figure 3.32(a) illustrates the basic architecture of MySQL. Our storage engine can be added to MySQL server as a pluggable engine. Figure 3.32(b) illustrates the architecture of MySQL after adding our storage engine, which we refer to as OLAPStorage. We note, of course, that in order to take full advantage of the new storage subsytem, the DBMS query engine might also have to be extended.

3.11 Conclusions

In this chapter, we have described the integration of Berkeley DB components into the Sidera DBMS. This integration significantly enhances the existing Sidera storage engine. Specifically, Sidera now stores the Hilbert R-tree index in one Berkeley DB physical file. We have discussed in detail how this integration would be performed in a practical environment. In addition, we have described how dimension tables and fact tables are encoded into a more compact integer format. Furthermore, we have explained how the storage of non-hierarchical attributes as a set of compressed FastBit bitmap indexes can be used to enhance OLAP analysis. We also described how the data of hierarchical attributes are stored and accessed in order to allow the efficient creation of the Sidera mapGraph at run time.

Experimentally, our results support the integration approach that we have taken. Test results demonstrate that the running time to build the indexed cube in the Berkeley supported Sidera server is better than the old Sidera server. Also, we provide evidence that our Berkeley integrated Sidera server has the potential to significantly boost run-time query performance. Finally, we demonstrate that the FastBit bitmap indexing method offers tangible performance advantages for non-hierarchical attributes relative to the widely used Btree indexing technique. In summary, our work in this chapter demonstrates how one would construct a reliable and efficient storage engine for a contemporary OLAP server.

Chapter 4

OLAP Query Language

4.1 Introduction

In the previous chapter, we focused on the construction of a reliable and efficient OLAP storage engine. Specifically, we integrated the Berkeley DB functionality into the current Sidera OLAP environment [38] in order to enable and enhance its efficiency when resolving basic OLAP queries. Our research elaborated on an existing OLAP architecture that resolves only simple OLAP queries that have been hard-coded in a proprietary syntax. The general theme of this doctoral research is to enhance the Sidera server to efficiently handle sophisticated and arbitrarily complex OLAP queries. In this chapter, we describe a comprehensive OLAP algebra and grammar that will be used to efficiently express, and subsequently optimize, OLAP client query requests.

We begin by noting that the new Sidera server provides an Object-Oriented model for the OLAP environment. This means that the user can logically assume that everything in the OLAP storage is simply an object (e.g., cube, hierarchy, dimension). Moreover, one can assume that the entire data repository — which might be TeraBytes in size — exists entirely in the memory of the local machine as a series of one or more such cube objects (e.g., cube, dimensions, hierarchy). Not surprisingly, a query language that supports this genre of transparent model must be relatively sophisticated. In fact, Sidera provides "native language" query facilities. In other words, native OOP languages (e.g., Java) will directly support interaction with the backend OLAP database server without the need to embed an intermediate, non-OOP language such as SQL or MDX. The new Sidera server provides a pre-processor that essentially translates standard OOP source code into an OLAP query grammar that has been developed specifically for OLAP analysis. Our work in this context is listed below:

1. A comprehensive multi-dimensional OLAP algebra supporting fundamental query operations for cube-specific operations.
2. An OLAP query grammar that presents the developer with an Object-Oriented representation of the primary OLAP structural elements.
3. OLAP Algebraic Laws that allow rewriting of OLAP queries using our multi-dimensional OLAP algebra.
4. OLAP Metadata that provides the necessary information to answer an OLAP query.
5. An OLAP query compiler that parses and optimizes an OLAP query.
6. An OLAP query execution engine that executes the optimized OLAP query.
7. A new Result Set format that specifies how the results of OLAP queries are returned to the end-user.

The first four themes mentioned above are the focus of this chapter while the last three topics will be explained in the following chapter. In terms of the former, we note at this point that we have developed a comprehensive multi-dimensional OLAP algebra. The comprehensive OLAP algebra reduces the complexity of using existing relational algebras to write OLAP queries (via SQL or MDX) and also subsequently allows for the optimization (OLAP algebraic laws) of OLAP queries written in native OOP languages such as Java. Closely associated with the algebra, we have developed robust Document Type Definition (DTD) encoded OLAP query grammar that provides a concrete foundation for client language queries. The grammar, in turn, is the basis of a native language query interface that eliminates the reliance on an intermediate, string-based embedded language.

This chapter is organized as follows. In Section 4.2, we present an overview of related work, while Section 4.3 discusses the primary motivation of our research. In Section 4.4, we briefly explain some preliminary materials including the Sidera OLAP conceptual model and the different types of OLAP queries that an end user might write. In Section 4.5, we define fundamental query operations against a cube-specific OLAP Algebra. In Section 4.6, we explain the Document Type Definition OLAP query grammar, which defines the proper format of the OLAP queries that can be handled by our server. Section 4.7 discusses the algebraic laws for our OLAP algebra. In Section 4.8, we describe the XML DTD that defines the proper format of the schema (Metadata) of the OLAP environment (cube, dimensions, hierarchies, etc.). Section 4.9 reviews this chapter's objectives and a final conclusion is provided in Section 4.10.

4.2 Related Work

For more than 30 years, Structured Query Language (SQL) has been the de-facto standard for data access within the relational DBMS world. Because of its *relative age*, however, numerous attempts have been made to modernize database access mechanisms. Two themes in particular are noteworthy in the current context. In the first case, Object Relational Mapping (ORM) frameworks have been used to define type-safe mappings between the DBMS and the native objects of the client applications. With respect to the Java language, industry standards such as JDO (Java Data Objects) [61], as well as the open source Hibernate framework [13] have emerged. In all cases, however, it is important to note that while the ORM frameworks do provide *transparent persistence* for individual objects, additional string-based query languages such as JDOQL (JDO), or HQL (Hibernate) are required in order to execute joins, complex selections, sub-queries, etc. The result is a development environment that often seems as complex as the model it was meant to replace.

More recently, Safe Query Objects (SQO) [28] have been introduced. Rather than explicit mappings, safe queries are defined by a class containing, in its simplest form, a *filter* and *execute* method. Within the filter method, the developer encodes query logic (e.g., selection criteria) using the syntax of the native language. The compiler checks the validity of query types, relative to the objects defined in the filter. The *execute* method is then rewritten as a JDO call to the remote database. The approach is quite elegant, though it can be difficult to accurately model completely arbitrary SQL statements.

In contrast to the ORM models, a second approach extends the development languages themselves. The Ruby language [8], for example, employs *ActiveRecords* to dynamically examine method invocations against the database schema. HaskellDB [7], on the other hand, "decomposes" queries into a series of distinct algebraic operations (e.g., restrict, project). Microsoft's LINQ extensions (C# and VisualBasic) [20] are also quite interesting in that they essentially integrate the mapping facilities of the ORM frameworks into the language itself (via the ubiquitous SELECT-FROM-WHERE format). It should be noted that none of previous languages are in any way OLAP-aware. Therefore, concepts such as cubes, dimensions, aggregation hierarchies, granularity levels, and drill down relationships map poorly at best to those languages that are based upon the standard logical model of relational systems.

In terms of OLAP and BI specific design themes, most contemporary research builds in some way upon the OLAP *data cube* operator [47]. In addition to various algorithms for cube construction, including those with direct support for dimension hierarchies [107, 79], researchers have identified a number of new OLAP *operators* [12, 29], each designed to minimize in some way the relative difficulty of implementing core operations in "raw SQL". There has also been considerable interest in the design of supporting algebras [93, 51, 95]. The primary focus of this work has been to define an API that would ultimately lead to transparent, intuitive support for the underlying data cube. In a more general sense, these algebras have identified the core elements of the OLAP conceptual data model.

Commercially, a somewhat orthogonal pursuit in the OLAP context has been the design of domain- specific query languages and/or extensions. SQL, for example, has been updated to include the CUBE, ROLLUP, and WINDOW clauses [77], though vendor support for these operations in DBMS platforms is inconsistent at best [58]. In addition to SQL, many commercial applications support Microsoft's MDX query language [73]. While syntactically reminiscent of SQL, MDX provides direct support for both multi-level dimension hierarchies and a crosstab data model.

Oracle has traditionally used something called OLAP DML (Data Manipulation Language)[86] to improve aggregation performance in the data warehouse. It provides several extensions to the standard SQL GROUP BY clause to make query reporting faster and easier. Two new keywords have been defined (i) CUBE and (ii) ROLLUP. CUBE can be used as an efficient replacement for a collection of individual group-by statements, each specifying a subset of attributes. ROLLUP calculates aggregate functions such as SUM, COUNT, MAX, MIN and AVG at increasing levels of aggregation, from the most detailed up to a simple total.

MDX and Oracle's DML are the most widely used OLAP query languages. However, they are essentially embedded string based languages with irregular structures. The integration of these string-based languages into application level source code is typically associated with one or more of the following limitations:

- Comprehensive compile-time type checking is often impossible. All parsing is performed at run-time by a possibly remote, often overloaded server.
- Developers must merge two fundamentally incompatible programming models (i.e., procedural OOP versus a non-procedural DBMS query language).
- There are few possibilities for the kind of code re-use afforded by OOP concepts (e.g., inheritance and polymorphism).
- The use of embedded query strings (i.e., JDBC/SQL) severely limits the developer's ability to efficiently re-factor source code in response to changes in schema design.

Finally, we note that query languages such as SQL and MDX are typically encapsulated within a programmatic API that exposes methods for connection configuration, query transfer, and result handling. While relational systems utilize mature standards (e.g., JDBC, ODBC), a little work has focused on API for OLAP (e.g., OLAP4J). A recent attempt to do so was the ill-fated JOLAP specification, JSR-69 [6], an industry-backed initiative to define an enterprize-ready, Java-oriented meta data and query framework based upon the Common Warehouse Meta-model [5]. JOLAP proved to be exceedingly complex and, consequently, no viable JOLAP-aware applications were ever developed. At present, the most widely supported API is arguably XML for Analysis (XMLA) [4], a low-level XML/SOAP mechanism running across HTTP. In practice, XMLA is effectively just a wrapper for MDX, though XMLA result sets are structured in an OLAP-aware format. Recently, OLAP4J (OLAP for Java) was proposed as an open Java API for OLAP server [85]. It is like JDBC in relational databased, but for accessing multi-dimensional data. OLAP4J allows one to get MDX support for free. OLAP4J-compliant OLAP server includes SQL Server Analysis Service and SAP Business Information Warehouse. Moreover, it can associated with Mondrian in order to get olap4j compliance out of any common relational database management system such as MySQL. However, OLAP4J is a fairly simple OOP wrapper and does not support any kind of optimizations (or perhaps even complex queries). In other words, its performance will be tied to performance of the XMLA servlet.

4.3 Motivation

As per the previous section, one may note that past research efforts have attempted to either represent OLAP queries in non-OLAP aware languages (e.g., SQL) and/or as embedded string-based OLAP query languages (such as Oracle DML). In the remainder of this chapter, we propose a comprehensive OLAP algebra and grammar, a set of OLAP algebraic laws, and OLAP metadata storage elements designed to achieve the following goals:

1. Support the creation of a native client-side OOP OLAP query language.
2. Reduce the complexity of directly utilizing the relational algebra in the OLAP context (i.e. SQL or MDX) via the application of a comprehensive multidimensional OLAP algebra.
3. Provide the developer with an Object Oriented representation of the primary OLAP structural elements, as well as providing the foundation for a concrete OLAP client query language.
4. Support query optimization and execution by means of applying new formally defined multi-dimensional OLAP algebraic laws.
5. Provide the format of the OLAP metadata.

4.4 Preliminary Material

In this section, we provide a brief description of the Object Oriented OLAP schema that is used to model the OLAP environment. We also show a simple object-oriented query that a client might write in Java.

4.4.1 Sidera OLAP Model

One of the great burdens associated with enterprize ORM projects is the design of accurate data models. Even when a model can be formally identified, it is often the case that the conceptual view of the data divers widely even between departments of the same organization. In the OLAP context, however, the conceptual view of the data has reached a level of maturity whereby virtually all analytical applications essentially support the same high level view of the data.

Briefly, we consider analytical environments to consist of one or more *data cubes*. Each cube is composed of a series of d dimensions (sometimes called $feature$ attributes) and one or more $measures$. The dimensions can be visualized as delimiting a d-dimensional hyper-cube, with each axis identifying the members of the parent dimension (e.g., the days of the year). Cell values, in turn, represent the aggregated measure (e.g., sum) of the associated members. Figure 4.1(a) provides an illustration of a very simple three dimensional cube named Order. This cube has three feature attributes (Number, Month and Type) which relate the cube itself with three distinct dimensions: Customer, Time and Product. Furthermore, it also possesses one measure attribute (Quantity_Ordered). We can see, for example, that 55 units of Interior Product were ordered by customer number C1 during the month of June (assuming a Count measure).

Beyond the basic cube, however, the conceptual OLAP model relies extensively on aggregation hierarchies provided by the dimensions themselves. In fact, hierarchy traversal is one of the more common and important elements of analytical queries. In practice, there are many variations on the form of OLAP hierarchies [74, 34] (e.g.,

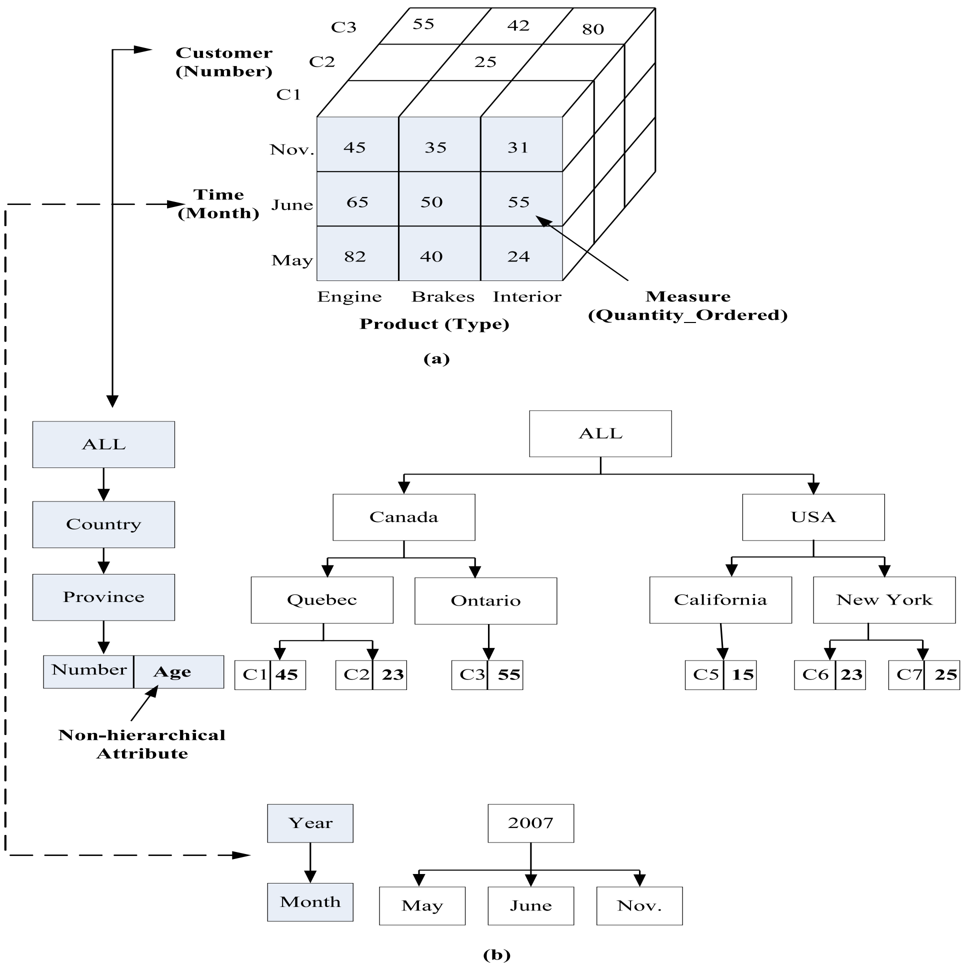
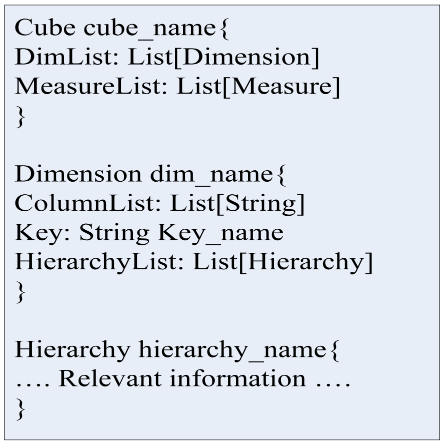


Figure 4.1: (a) Sidera Conceptual Cube Model and (b) Simple Symmetric hierarchies for Customer and Time.

symmetric, ragged, non-strict). Figure 4.1(b) illustrates two simple dimension hierarchies (Customer and Time). These hierarchies might be used to identify intuitive groupings. For example, customer C1 and C2 reside in the province of Quebec, while C3 resides in Ontario. Moreover, we note in Figure 4.1(b) that each customer number (e.g., C1) has a specific age. Age is a descriptive attribute in dimension Customer which is a non-hierarchical attribute that is not part of any given hierarchy. In other words, attribute Age depends on the primary key of dimension Customer (we assume here that customer number is the primary key). For example, in Figure 4.1(b) we can see that for a customer number there is only one value for age (e.g., customer C5 is 15 years old).

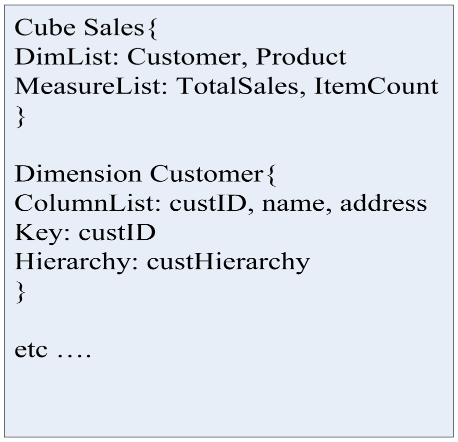
In our case, we exploit the fact that there is a single, fairly well understood conceptual model that we can expose to users — this can be applied in virtually all OLAP cases. In this sense, our OLAP server (Sidera) provides the programmers with an OLAP conceptual model for the data model. The idea is that programmers will only have to understand the environment at this conceptual level of abstraction. In other words, the programmers do not have to know the details of the physical or even logical schema is required since we are building a system that is fully optimized for the conceptual cube model. That being said, the new Sidera server provides an Object Oriented Model for the OLAP conceptual environment. From a developer's perspective, everything in the server is considered as an object (e.g., cube, dimension, measure, hierarchy, group-by, etc.) that is housed in the local memory.

Figure 4.2(a) provides a simple example of a grammar for the OLAP environment. We can see how one can recursively construct a high level cube from constituent objects. We note that there are many more classes in the full object oriented model. In



```
Cube cube_name{
DimList: List[Dimension]
MeasureList: List[Measure]
}

Dimension dim_name{
ColumnList: List[String]
Key: String Key_name
HierarchyList: List[Hierarchy]
}

Hierarchy hierarchy_name{
.... Relevant information ....
}
```

(a)



```
Cube Sales{
DimList: Customer, Product
MeasureList: TotalSales, ItemCount
}

Dimension Customer{
ColumnList: custID, name, address
Key: custID
Hierarchy: custHierarchy
}

etc ....
```

(b)

Figure 4.2: (a) Simple Object Oriented OLAP model grammar.(b) Simple Object Oriented OLAP model.

Figure 4.2(b), we see a concrete instantiation of the model. In an object oriented way, we create a cube called sales that contains two dimensions (Customer and Product) and two measures (TotalSales and ItemCount). We can also see the schema for dimension Customer.

We now give a formal definition of our OLAP conceptual model as follows:

Cube: An N-dimensional cube C is constructed as <*D, F, M, BasicCube*> where:

- D is a set of dimension tables D_i of C, where $D = \{D_1, D_2, ..., D_N\}$, where $1 \leq i \leq N$.
- F is a set of feature attributes F_i of C, $F = \{F_1, F_2, \ldots, F_N\}$, where $1 \leq i \leq N$.
- M is a list of measure attributes M_j of C, $M = \{M_1, M_2, \ldots, M_k\}$, where $j \leq k$.
- *BasicCube* is a set of cells that describes the *facts* (measure attributes) at the particular level of detail which is specified by F.

Dimension Tables: A dimension table (D_i) is a relation table. The schema of D_i is written as *schema(D_i) = <ColumnList, Key, Hierarchy>* where:

- *ColumnList* is a set of dimension attributes $D_i.A_j$ of D_i,*ColumnList* = $\{D_1.A_1, \ldots, D_1.A_n\}$, where n is the number of attributes in dimension D_i.

- *Key* is an attribute $D_i.A_k$ of *ColumnList*, where $D_i.A_k$ is the deepest level of detail for dimension D_i, where $1 \leq k \leq n$.

- *Hierarchy* is a set of hierarchies $D_i.H_j$ of D_i, *Hierarchy* $=\{D_i.H_1, D_i.H_2, \ldots, D_i.H_z\}$, where $j \leq z$ and z is the number of hierarchies associated with dimension D_i. Each hierarchy $D_i.H_j$ is of the form $D_i.H_j = \{H_j,\ D_i.A_r \rightarrow \ldots \rightarrow D_i.A_l\}$, where $D_i.A_r$ is the root hierarchal attribute level while $D_i.A_l$ is the leaf level in hierarchy H_j of dimension D_i.

Feature Attributes: A feature attribute F_i refers to a specific attribute A_j in dimension D_k, where i,$k \subseteq [1,N]$. It is of the form $F_i = \{D_k.A_j\}$, F_i is an attribute in the *ColumnList* of dimension D_k.

Basic Cube: A basic cube is a multidimensional representation for the end user with a schema of the form schema*(BasicCube)* $= \{F, M\}$. An instance of a *BasicCube* is the set of *cells/facts/records/tuples* that are described by the values of measure attributes M at the level defined by F. Through the course of this thesis, we will use the terms $cells, facts, records$, and *tuples* interchangeably.

For example, we consider the three-dimensional cube *Order* of Figure 4.1. In our model, it can be written as *Order=<DOrder, FOrder, MOrder, BasicCubeOrder>*.

Below, we list the details of dimension tables *DOrder*={*Customer, Time, Product*} with the following schema as an example:

- schema*(Customer)* = <{Customer.Province, Customer.Country, Customer.Number, Customer.Age}, Customer.Number, Customer.Hier>. The dimension Customer consists of four attributes, one key (Customer.Number), and one hierarchy called $Customer.Hier = \{Hier,\ Customer.Country \rightarrow Customer.Province \rightarrow Customer.Number\}$. The details of other dimensions can be defined in the same manner.

FOrder can be written as *FOrder* = { *Time.Month, Product.Type, Customer.Number*}, while the list of measure attributes can be expressed as *MOrder* = {*Quantity_Ordered*}. The schema of the *BasicCubeOrder* is written as *schema(BasicCubeOrder)* = {*Time.Month, Product.Type, Customer.Number, Quantity_Ordered*}. An example of a cell/fact in the *BasicCubeOrder* is the cell {May, Engine, C1, 82} from Figure 4.1.

4.4.2 Object Oriented OLAP Query

Given the relational model of the underlying DBMS, BI querying typically relies upon non-procedural SQL or one of its proprietary derivatives. Unlike transactional databases, however, which are often cleanly modeled by a set-based representation, the nature of BI/OLAP environments argues against the use of such languages. In particular, concepts such as cubes, dimensions, aggregation hierarchies, granularity levels, and drill down relationships map poorly at best to the standard logical model of relational systems. Moreover, the difficulty of integrating non-procedural queries languages into application level source code can be significant. Larger development projects typically encounter any of several associated limitations (as was discussed in Section 4.2).

Given the above, Sidera provides a clean integration between the server's Object-Oriented OLAP (OOP) conceptual model (discussed in the previous subsection) and the OLAP client query language. In other words, the OLAP client queries can be specified in any Object Oriented Language such as Java or C#, with the programmer assuming that all OLAP data (cubes, dimensions, etc.) is stored in the local memory as a series of one or more cube objects.

The new Sidera server provides a source code re-writing mechanism that interprets the client's OOP OLAP query specification and decomposes it into the core operations of our comprehensive OLAP algebra. These operations are given concrete form within our OLAP query grammar and are then transparently delivered at run-time to the backend analytics server for processing. Note that the pre-processing work conducted on the client-side of the Sidera server — For example, defining Java libraries, parsing source code and generating XML versions of users' queries — is being performed by another grad student in our research group. The processing described in this thesis begins once the user's query is received on the backend Sidera server.

In terms of the client side compilation process, the pre-processor of the Sidera server takes as input the original source file and then, using the parse tree constructed from this source, converts the relevant source elements into an XML decomposition of the OlapQuery. Throughout this process, various DOM utilities and services are exploited in order to generate and verify the XML. Finally, once the source has been transformed, it is run through a standard Java compiler and converted into an executable class file. We note that, in practice, this translation step would be integrated into a build task (ANT, makefile, IDE script, etc.) and would be completely transparent to the programmer. At run-time, the Object-Oriented OLAP query will

```
SELECT Product.Type, Customer.Province, SUM(Quantity_Ordered)

FROM Order, Product, Customer, Time

WHERE Customer.Number = Order.Number AND
             Product.Type = Order.Type AND
             Time.Month = Order.Month AND
              Customer.Age > 40 AND
              Time.Year = 2007 AND Time.Month between 5 AND 10

GROUP BY Product.Type, Customer.Province
```

Figure 4.3: OLAP query written in SQL.

```
SELECT
            { [Product].[Type].ALLMEMBERS } ON COLUMNS,
            {  [Customer].[Province].ALLMEMBERS  }  ON ROWS

FROM  [Order]

WHERE (
            [Measures].[Quantity_Ordered],
            [Time].[Year].[2007], [Time].[Month].[May], [Time].[Month].[June],
            [Customer].[Age].[45],[Customer].[Age].[55]
            )
```

Figure 4.4: OLAP query written in MDX.

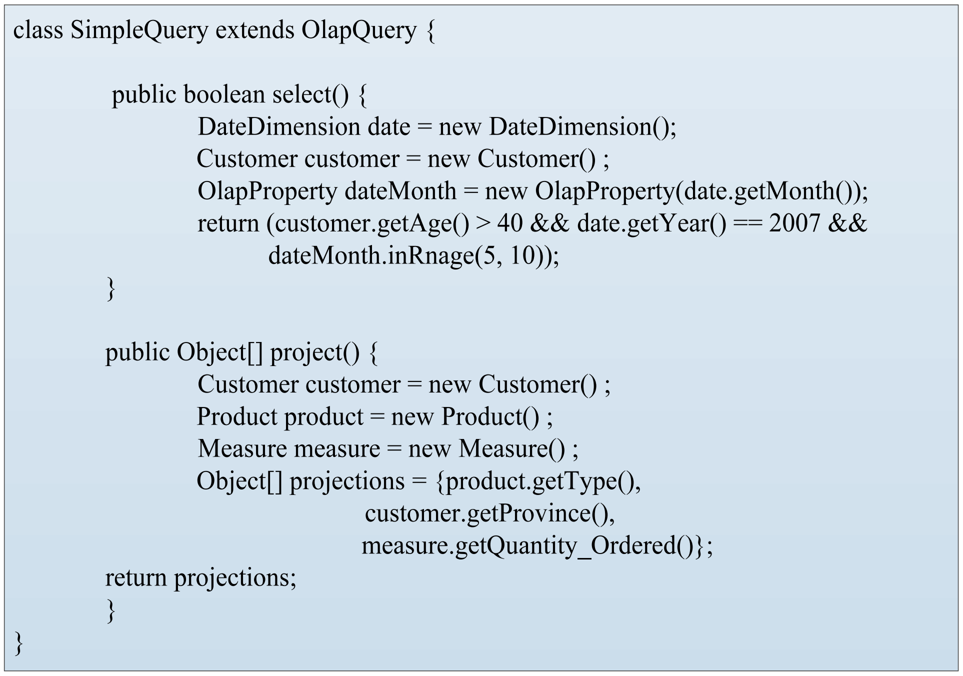
be sent to the Sidera server in an XML format where it must be parsed, optimized and processed. The result is returned to the end-user in an XML format.

Figure 4.3 and Figure 4.4 illustrate equivalent OLAP queries that are written in the string-based query languages SQL and MDX respectively. These queries are written against the data warehouse that corresponds to the conceptual model illustrated in Figure 4.1. For example, the data warehouse consists of a fact table (Order) that represents the view of Figure 4.1(a) and three dimension tables (Product, Customer and Time) outlined in Figure 4.1(b). In Figure 4.3 and Figure 4.4, both OLAP queries compute the total quantity ordered by customers whose age is greater than 40 (e.g., Customer C1 and C3 from Figure 4.1) where the year is 2007 and the month is between May and October. The result is then grouped by product type and customer province. While functional, the insertion of these queries into application level source code has one or more limitations, as was discussed in Section 4.2. Moreover, the joins between the fact table (e.g., Order) and dimension tables (Product, Customer and Time) must be specified.

Figure 4.5 illustrates how the string-based queries of Figure 4.3 and Figure 4.4 can be encoded and written in a very intuitive Object-Oriented manner (e.g., Java) by the client/programmer. Specifically, the user assumes that the basic cube Order, and its associated dimensions (Customer, Product and Time) of Figure 4.1, are housed in the local memory. Moreover, all OOP features such as inheritance, polymorphism, class encapsulation, etc. can be directly applied to this query. Once defined, the Sidera client side pre-processor converts the query of Figure 4.5 into a new query that corresponds to our OLAP query grammar (to be discussed in Section 4.6). It then re-writes the client's query in an XML format. At run-time, it is this XML-based query that will be sent to the new Sidera server for processing. Note that Figure 4.21 illustrates the XML format of the OO query of Figure 4.5. One important point to understand is that while the Java version may be slightly more verbose in this very simple example, it extends to complex queries very cleanly. In contrast, MDX can become almost unreadable as the complexity of the query grows.

4.5 OLAP Algebra

Given the complexity of directly utilizing the relational algebra in the OLAP context (via SQL or MDX), we define fundamental query operations against a cube-specific



```
class SimpleQuery extends OlapQuery {

        public boolean select() {
                DateDimension date = new DateDimension();
                Customer customer = new Customer() ;
                OlapProperty dateMonth = new OlapProperty(date.getMonth());
                return (customer.getAge() > 40 && date.getYear() == 2007 &&
                        dateMonth.inRnage(5, 10));
        }

        public Object[] project() {
                Customer customer = new Customer() ;
                Product product = new Product() ;
                Measure measure = new Measure() ;
                Object[] projections = {product.getType(),
                                customer.getProvince(),
                                measure.getQuantity_Ordered()};
        return projections;
        }
}
```

Figure 4.5: Basic OLAP query written in OOP Java.

OLAP algebra. In other words, we describe simple semantics representing a comprehensive multi-dimensional OLAP algebra that can directly exploit the clean Object-Oriented conceptual model discussed in the previous section. Moreover, since our Object-Oriented OLAP queries are written at a very high level against the conceptual model, our OLAP query processor (to be explained in Chapter 5) must do a lot of additional processing to supply missing details. Thus, an OLAP query is translated internally into an OLAP algebra expression that ultimately makes alternative forms of an OLAP query easier to create, explore, manipulate and optimize (e.g., push and pull operations, replace operations). Specifically, when an OLAP query is submitted to our OLAP DBMS (Sidera server), its query optimizer tries to find the most efficient equivalent OLAP algebra expression before evaluating it. This process can be quite effective, in part because the inner, lower-level operations of our OLAP Sidera DBMS are very similar to the OLAP algebra operations defined by the algebra. In the pages that follow, we will introduce and describe the operations of this algebra. We note that while our algebraic model is the most comprehensive such model of which we are aware, we draw extensively upon previous research in the area [93, 12, 29, 51, 95]. Briefly, our OLAP algebra is not newly defined algebra; however, the idea that we are providing a comprehensive algebra that represents all common OLAP operations (not just one or two) and we are providing optimization laws and execution algorithms that show how and why an OLAP algebra is a good idea in practice.

4.5.1 OLAP Algebra Operations

Our OLAP algebra consists of OLAP operators and atomic operands. The OLAP algebra allows the building of OLAP expressions (referred to as an OLAP query) by applying OLAP operators to atomic operands and/or other OLAP expressions. All operands and the results of OLAP expressions are themselves cubes. Below, we list and describe the OLAP operators of our comprehensive OLAP algebra. We note at the outset that the language of OLAP algebras has yet to be standardized, it is nevertheless the case that a core set of operations has been consistently identified in the literature [95].

4.5.1.1 SELECTION Operator

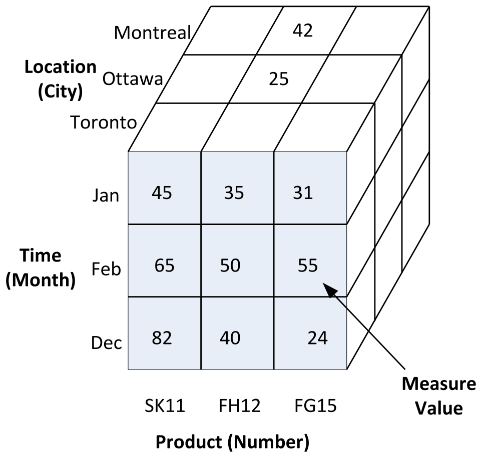
The selection operator identifies one or more cells from within the full d-dimensional search space and is a core OLAP operation. Its application produces what is commonly referred to as “slicing and dicing”. This operator is applied to a data cube and produces a subset of the same data cube. The dimension members in the resulting cube are those that satisfy some conditions C that involve the dimension members of the cube. We denote this operation as: SELECTION(C)(Cube). The schema of the resulting cube is the same as that of the original schema, and C is a conditional expression including operands that are either constants or feature attributes of the input cube. From the user's perspective, the query is executed against the physical data cube such that the selection criteria will be iteratively evaluated against each and every cell. If the selection test evaluates to true, the cell is included in the result; if not, then it is ignored. We note that this operator (selection) is the most important operation that has been defined in the multidimensional algebras presented in the literature [95, 93, 51, 68, 52, 43]. However, our SELECTION operator represents a comprehensive definition among those founds in the literature [95, 93, 51, 68, 52, 43].

Let $C =< D, F, M, BasicCube >$ be an N-dimensional cube with N dimensions D $=\{D_1, \ldots, D_N\}$, F is the set of feature attributes, M is the set of measure attributes and *BasicCube* is the core basic data cube of cube C. Note that we will use the symbol C in the formal definitions of other OLAP algebraic operators.

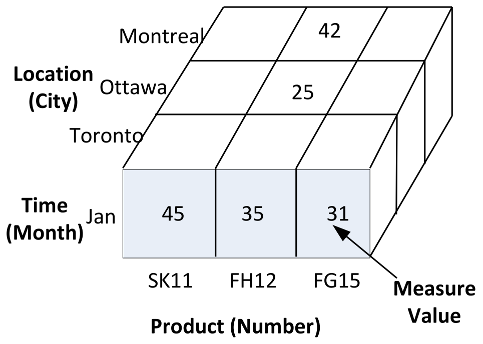
More formally, we can define the SELECTION operator as **SELECTION(**D_i**.**A_j **Op Cte)C**, where:

- $D_i.A_j$ is an attribute in dimension D_i (domain($D_i.A_j$) is the domain of attribute A_j in dimension D_i),
- **Op** is a conditional operator such as $\{<, >, =, \ldots$, etc.$\}$, and
- **Cte** is one or more values in domain($D_i.A_j$).

The result of **SELECTION (**D_i.A_j **Op Cte)C** is a cube *C1<D, F, M, BasicCube1>*, where sets *D, F,* and *M* are equivalent to those in the input cube *C* and schema(*BasicCube1*) = schema(*BasicCube*), but cells of *BasicCube1* are only those cells that satisfy the



(a)



(b)

Figure 4.6: (a) The three dimensional cube *SOLD*. (b) The result of the SELECTION(Time.Month = Jan)SOLD query.

restriction ($D_i.A_j$ Op Cte). Note that the SELECTION operator can have one or more conditions that are connected via AND and OR logical operators.

Let the three dimensional cube SOLD be as illustrated in Figure 4.6(a). SOLD is composed of three dimensions (Product, Time and Location), three feature attributes (e.g., Location.City, Time.Month, etc.) and one measure attribute (Units_Sold)). We can see, for example, that 77 units of Product FH12 were sold in the Ottawa location during the month of January (assuming a Count measure). Given this, the value of the OLAP expression **SELECTION(Time.Month = Jan)SOLD** is depicted in Figure 4.6(b).

4.5.1.2 PROJECTION Operator

This operator is used for the identification of presentation attributes, including both the measure attribute(s) and dimension members. It is used to extract, from a source cube, a new cube that has only some of the original dimension members and measure attribute(s). The schema of the output cube is the set of dimension members and measure attribute(s) specified with the PROJECTION operator. This operation has been drawn extensivenly upon previous research in the area (multidimensional algebras) [95, 93, 51, 68, 52, 43]. However, in our case, it is defined at a very high OLAP conceptual level that allows us to combine the power of all projections found in the literature [95, 93, 51, 68, 52, 43]. Formally, the PROJECTION operation can be written as:

$$\textbf{PROJECTION } (D_i.A_j, \textbf{ y}) \textbf{ C},$$

where $D_i.A_j$ is a list of dimension attributes, and y $\subset$ M. The resulting cube is *C1<D1, F1, M1, BasicCube1>*, where:

- D1 is a set of dimensions that are mentioned within the PROJECTION.
- F1 = list of dimension attributes $D_i.A_j$.
- M1 = y.
- Schema(BasciCube1) = <F1, M1>. Note that the measure(s) (M1) values of BasicCube1 are aggregated at the level of the attribute(s) in F1.

Again consider the three dimensional cube *SOLD* of Figure 4.6(a). We can project this cube onto a new cube with two dimension attributes (Product.Number and Time.Month) and one measure Units_Sold with the following OLAP algebra expression: **PROJECTION(Time.Month, Product.Number, Units_Sold)(SOLD)**. The resulting cube is depicted in Figure 4.7. Note how the measure values *(Units_Sold)* of product FH12 are re-calculated accordingly. For example, the total sales for Jan/FH12 = 35 + 25 + 42 = 102.

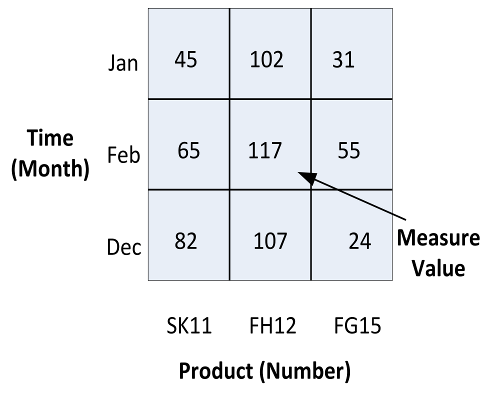


Figure 4.7: The result of PROJECTION operator.

4.5.1.3 Set operations on Data Cubes

OLAP set operations (UNION, INTERSECTION and DIFFERENCE) can be applied to data cubes. We consider only Union, Difference and Intersection because they are the most relevant ones in the literature [95, 93, 51, 68, 52, 43]. They are defined as follows on arbitrary data cubes C1 and C2:

- C1 UNION C2 is the union of two cubes sharing common dimensional axes. If two cells from C1 and C2 have the same feature attribute values, then we can add their measure attribute values (measure attributes are assumed to be numeric).

- C1 INTERSECTION C2 is the intersection of two cubes sharing common dimensional axes. If two cells are intersected, then we subtract the larger measure attribute values from the smaller.

- C1 DIFFERENCE C2 is the difference of two cubes sharing common dimensional axes. When two cells have the same feature attribute values, then the cell value of C1 will be included in the output if its measure value is greater

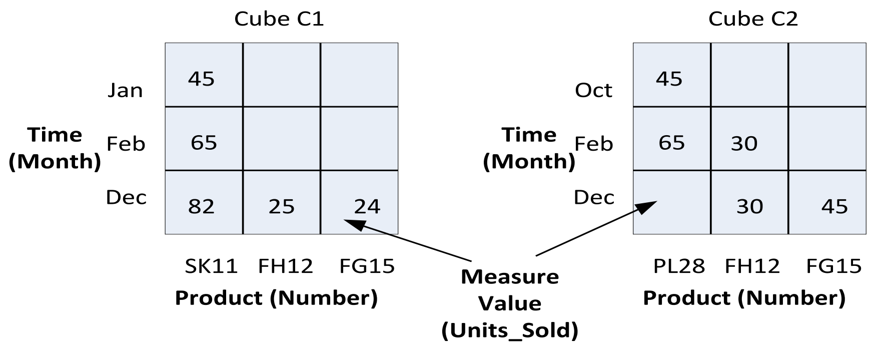


Figure 4.8: Two Cubes(*C1* and *C2*) share common feature and measure attributes.

than that of C2.

Let C1<D, F, M, BasicCube1> and C2<D, F, M, BasicCube2> be two cubes sharing common dimensions, feature attributes and measure attributes. In other words, they have common schemas but they might have different cell values. More formally, we can define set operators as:

- C = C1 UNION C2
- C = C1 INTERSECTION C2
- C = C1 DIFFERENCE C2

In the above formal definitions, the resulting cubes have the same schema (<D, F, M>) as any one of the source cubes but the value of the cells are calculated according to the operation being performed(UNION, INTERSTCION or DIFFRENCE). Note that C1 and C2 can sometimes be the results of other OLAP algebra expressions.

Suppose we have two cubes *C1* and *C2* as in Figure 4.8. Note that both cubes have the same schema (same dimensions, feature attributes, and measure attribute). The intersection, union, and difference of C1 and C2 are denoted as C1 INTERSECTION

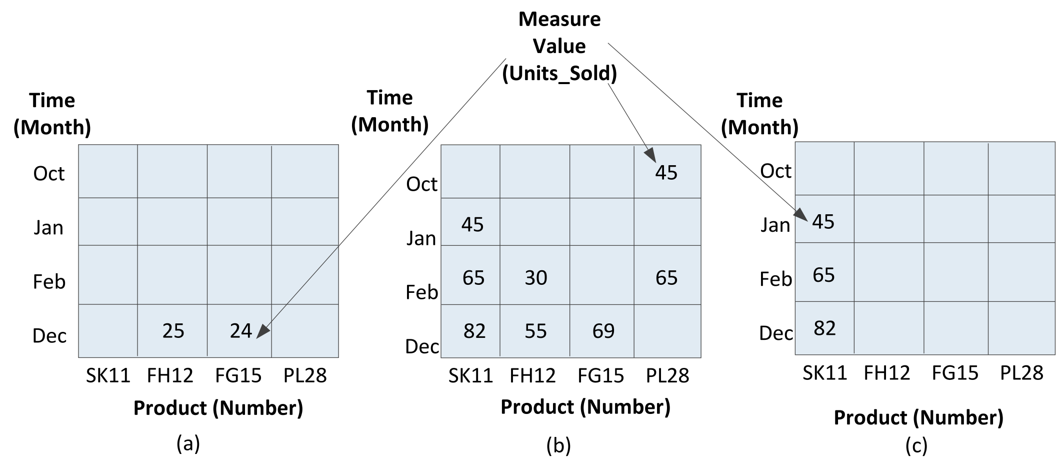


Figure 4.9: (a) *C1* INTERSECTION *C2* (b) *C1* UNION *C2* (c) *C1* DIFFERENCE *C2*.

C2, C1 UNION C2, and C1 DIFFERENCE C2 respectively. The result cubes are shown in Figure 4.9(a), 4.9(b), 4.9(c) respectively.

4.5.1.4 CHANGE_LEVEL Operator

This operator is an analytical operator whereby the user navigates among levels of data ranging from the most detailed (down) to the most summarized (up) or vice-versa amongst a concept hierarchy [95, 93, 51, 68, 52, 43]. It is considered as a modification of the granularity of aggregation. We typically refer to these processes as “roll-up” and “drill down.” We are provinding a comprehensive definition of the roll-up and drill-down operations since in the literature these two operations are defined as two distinct operations. However, we provide only one operation (CHANGE_LEVEL) in a very high conceptual level.

Consider the N-dimensional cube C = <*D, F, M, BasicCube*> outlined in Section 4.4. Formally, we denote the change level operator as:

$$\textbf{CHANGE\_LEVEL}(D_i.A_j \rightarrow D_i.A_k)\ \mathbf{C},$$

such as $D_i \in D$, $D_i.A_j$ is a feature attribute that relates cube C with dimension D_i and $D_i.A_k$ is a hierarchical attribute level in dimension D_i. The resulting cube of this operation is another cube *C1 = <D, F1, M, BasicCube1>*. Note that the result cube C1 has the same set of dimensions and measure attributes as in the source cube C. However, they have different (i) feature attributes (C1 has a new set of feature attributes) F1 = F - $D_i.A_j$ + $D_i.A_k$, and (ii) cells in the basic cube (BasicCube is not equal BasicCube1). Moreover, this operator can be used to change the levels of more than one feature attributes at the same time. It can be expressed as CHANGE_LEVEL($D_i.A_j \rightarrow D_i.A_k$, $D_r.A_s \rightarrow D_r.A_t$, ...) C, where i,r =[1 ... N].

Consider the three dimensional cube *SOLD* of Figure 4.6(a). Figure 4.10 illustrates how the "Product" dimension, originally listed at a more detailed level number, is aggregated in order to provide a break down by Brake and Engine product types. Figure 4.10 is the result of the following OLAP expression: *CHANGE_LEVEL (Product.Number → Product.Type)(SOLD).* In this expression, Product.Number is a feature attribute in the source cube SOLD and Product.Type is a hierarchical attribute level in dimension Product. Note how attribute Product.Type in Figure 4.10 becomes a feature attribute instead of the attribute Product.Number.

4.5.1.5 CHANGE_BASE Operator

This operator represents the addition or deletion of one or more dimensions from the current result cube (C)[95, 93, 51, 68, 52, 43]. Aggregated cell values must be re-calculated accordingly. Using cube *C = <D, F, M, BasicCube>* of the previous section, this operated is formally denoted as follow:

$$\text{CHANGE\_BASE}(\ D_i.A_j \rightarrow \textit{Action})\text{C},$$

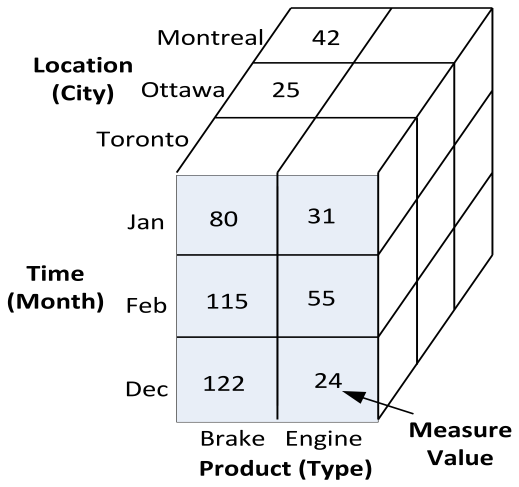


Figure 4.10: Result of a change level operation.

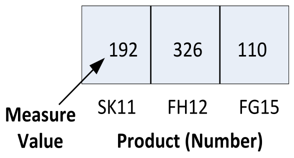


Figure 4.11: Result of the CHANGE_BASE operation.

where *Action* can be *Remove* (to remove a dimension from cube C) or *Add* (to add a dimension to cube C) and $D_i.A_j$ is a feature attribute that relates cube C with dimension D_i in the case of *Remove* and is an attribute in dimension D_i in the case of *Add*. In other words, a dimension D_i is deleted by removing the feature attribute that relates the cube with D_i; however, the addition of a dimension D_i occurs by adding an attribute from D_i to the cube. The resulting cube is another cube C1 = <D1, F1, M, BasicCube1> that has different dimensions, feature attributes and basic cube relative to that of the source cube C.

Again, we consider the *SOLD* cube of figure 4.6(a). The result of the following expression *CHANGE_BASE(Time.Month→Remove, Location.City→Remove)(SOLD)* is depicted in Figure 4.11.

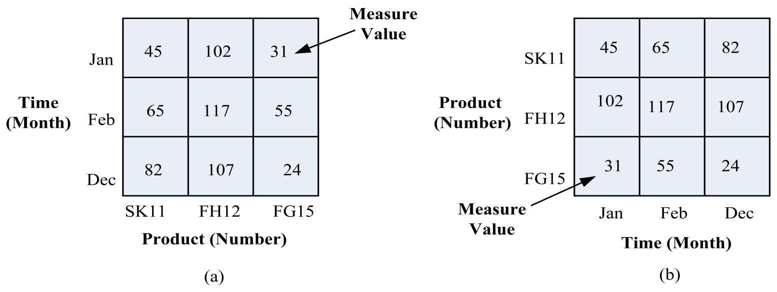


Figure 4.12: Pivot Operation: (a) the original view. (b) the result of the PIVOT operation.

4.5.1.6 PIVOT Operation

This OLAP operation allows users to re-organize the axes of the cube. In other words, Pivot deals with presentation only. No recalculation of cell values is required. most of previous publications in the multidimensional algebras have not discussed this operation [95, 93, 51, 68, 52, 43]. Utilizing the cube C of the previous section, a formal definition of this operator is defined as follow:

$$\text{PIVOT } (D_i.A_j \rightarrow D_k.A_l) \text{ C},$$

where $D_i.A_j$ and $D_k.A_l$ are feature attributes in cube C. This operator re-organizes the axes of cube C so that $D_k.A_l$ is viewed instead of $D_i.A_j$ and vice versa. The result cube is equivalent to the source cube. Figure 4.12(b) provides a simple example of how the pivot operation works on the original two-dimensional view called Purchase in Figure 4.12(a). The expression is written as PIVOT(Product.Number $\rightarrow$ Time.Month)Purchase.

4.5.1.7 DRILL_ACROSS Operator

The DRILL ACROSS is the integration of two independent cubes in order to compare their measure attributes, each possessing common dimensional axes [95, 93, 52, 43].

In effect, this is a cube "join" (possibly a self join) that changes or extends the subject of analysis. This operation has received little attention in the literature. We are providing a very comprehensive defintion of this operation (DRILL_ACROSS) with respect to those definitions discussed in in the publications [95, 93, 52, 43]. In other words, our operator combines the power of all drill across operations discussed previously. Consider two cubes C1 = <D1, F1, M1, BasicCube1> and C2 = <D1, F1, M2, BasicCube2> having the same set of dimensions and feature attributes but with different sets of measure attributes (M1 and M2). The formal definition of the drill across operation is denoted as:

C1(M1) DRILL_ACROSS C2(M2).

The result of this operation is another cube C = <D1, F1, M, BasicCube>, where M is the union of sets M1 and M2 and BasicCube contains the union of BasicCube1 and BasicCube2 with the new set of measure attributes M.

Consider the two cubes (*C1* and *C2*) of Figure 4.13, with both cubes having the same feature attributes (Product Number and Time Month). The measure attribute in *C1* is the total number of units sold during each month for each product number. However, the measure attribute in *C2* is the ordered quantity during each month for each product. The drill across result is found with the following OLAP operation:

C1(Units_Sold) DRILL_ACROSS C2(Quantity_Ordered)

The above expression produces the result shown in Figure 4.14. Note that the result of this operation is another cube that has the same feature attributes, and includes a direct comparison of the measure attribute(s).

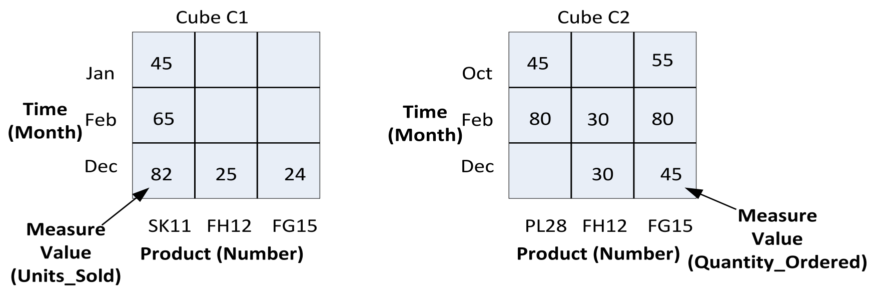
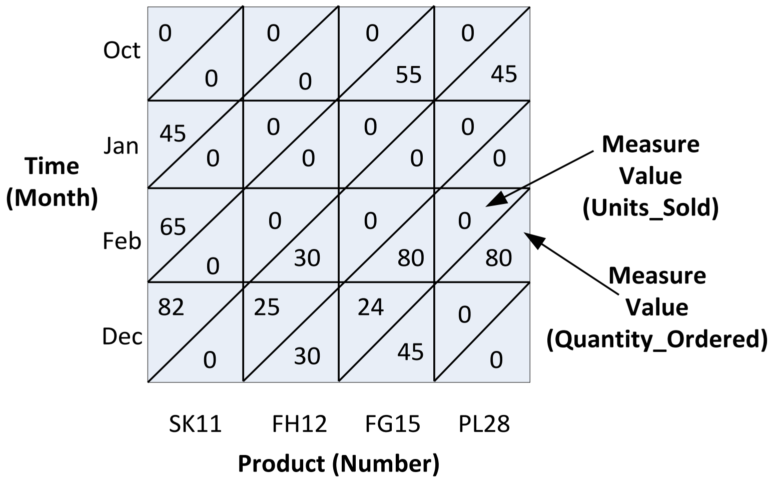


Figure 4.13: Two cubes (C1 and C2) with different measure attributes.



	SK11	FH12	FG15	PL28
Oct	0 / 0	0 / 0	0 / 55	0 / 45
Jan	45 / 0	0 / 0	0 / 0	0 / 0
Feb	65 / 0	0 / 30	0 / 80	0 / 80
Dec	82 / 0	25 / 30	24 / 45	0 / 0

Figure 4.14: Two cubes (C1 and C2) with different measure attributes.

4.5.2 Algebra Simplifications

It is important at this stage to point out that while logical data warehouse models typically require explicit joins between fact (measure) and dimension tables to provide OLAP reports with descriptive dimension information and to resolve query constraints that are often specified on the attributes of the dimensions — there is no such requirement with our algebra since our OLAP server implicitly knows when and how to do the joins. Finally, and perhaps most importantly, we note that the OLAP algebra is primarily read only, in that database updates are performed via distinct ETL processes. Therefore, there is no requirement for database services such as concurrency and locking.

4.6 OLAP Grammar

We encapsulate the operations of the OLAP algebra defined in Section 4.5 in a formal OLAP schema encoded in Document Type Definition (DTD). In turn, the DTD makes it easy to code, control and validate the associated XML document, particularly in a collaborative academic setting. We note that it would also be possible to utilize the more powerful XML schema mechanism [114]. Specifically, XML schemas allows control over the content of a document as well as its structure. They support a set of built-in data types (integer, date, Boolean, etc.), the ability to create new data types, and provide control over elements and attributes (e.g., the exact number of times an element appears), inheritance, etc.

The DTD is defined recursively and can handle very complex OLAP queries. In short, it indicates the proper format of the OLAP queries that are handled by our backend server. The DTD is made up of elements of the form <!ELEMENT

```
<!-- Data queries-->
<!ELEMENT DATA_QUERY (
          CUBE_NAME,
          OPERATION_LIST,
          FUNCTION_LIST?)>
<!ELEMENT CUBE_NAME (#PCDATA)>
<!ELEMENT OPERATION_LIST (
  PROJECTION,
  SELECTION?,
  CHANGE_LEVEL?,
  CHANGE_BASE?,
  PIVOT?,
  DRILL_ACROSS?,
  UNION?,
  INTERSECTION?,
  DIFFERENCE?)>
```

Figure 4.15: Core operations of the OLAP algebra.

name(components)>. The components can be other elements that may in turn be augmented with cardinality expressions indicating the number of occurrences of the element (e.g., * element may occur 0 or more times)[113].

Figure 4.15 defines the core structure of an OLAP query received by the back-end server. Each query is associated with a single cube (though references to other cubes are possible), as well as one Operation List and zero or one Function Lists. We do not consider cube functions extensively in our current research. However, for the sake of completeness, we may informally define a cube function as one that is logically associated with a result set, rather than a specific cell or dimension member. The common *top10* function would be a simple example. The Operations List contains the algebraic elements of the query, and each operation may occur exactly zero or one times in a single query. The one exception is PROJECTION, which must exist in every OLAP query (we assume that no defaults are available for the display attributes). Many OLAP queries can in fact, be expressed with nothing more than a

```
<!-- Projection -->
<!ELEMENT PROJECTION (
            MEASURE_LIST,
            ATTRIBUTE_LIST?)>
<!ELEMENT MEASURE_LIST (MEASURE+)>
<!ELEMENT ATTRIBUTE_LIST (PROJECTION_DIMENSION+)>
<!ELEMENT PROJECTION_DIMENSION (
            DIMENSION_NAME,
            ATTRIBUTE)>
<!ELEMENT MEASURE (#PCDATA)>
<!ELEMENT DIMENSION_NAME (#PCDATA)>
```

Figure 4.16: PROJECTION elements.

projection. Figure 4.16 illustrates that a PROJECTION is defined as a listing of one or more measures and zero or more dimension attributes.

Figure 4.17 demonstrates that a SELECTION is defined as a listing of one or more dimensions, each associated with an expression. The expressions of the dimensions are connected via logical operators (AND and OR). In effect, the expression represents a query restriction on the associated dimension. Simple expressions may be combined to form compound expressions (via logical AND and OR) and can be recursively defined. In other words, as with any meaningful programming language, conditional restrictions can be almost arbitrarily complex.

In Figure 4.18, we illustrate the simplicity of the set operation specifications. Set operations are simply represented as nested data queries, defined relative to the current query. Figure 4.19 illustrates three operations. First, the CHANGE_LEVEL is defined as a listing of one more dimensions, each associated with a target level. In effect, the target level represents a hierarchy level (attribute) on the associated dimension. Second, the CHANGE_BASE operation is expressed as a list of one or more dimension attributes. A specification is necessary in this operation for each

```
<!-- Selection -->
<!ELEMENT SELECTION (DIMENSION_LIST)>
<!ELEMENT DIMENSION_LIST (DIMENSION | COMPOUND_DIMENSION)>
<!ELEMENT COMPOUND_DIMENSION (DIMENSION_LIST, LOGICAL_OP, DIMENSION_LIST)>
<!ELEMENT DIMENSION (DIMENSION_NAME, EXPRESSION)>
<!ELEMENT DIMENSION_NAME (#PCDATA)>

<!-- Dimension Expressions -->
<!ELEMENT EXPRESSION (RELATIONAL_EXP | COMPOUND_EXP)>
<!ELEMENT COMPOUND_EXP (EXPRESSION, LOGICAL_OP, EXPRESSION)>
<!ELEMENT RELATIONAL_EXP (BASIC_EXP | OLAP_EXP)>
<!ELEMENT BASIC_EXP (SIMPLE_EXP, COND_OP, SIMPLE_EXP)>
<!ELEMENT OLAP_EXP (SIMPLE_EXP, OLAP_OP, OLAP_LIST)>
<!ELEMENT SIMPLE_EXP (EXP_VALUE | ARITHMETIC_EXP)>
<!ELEMENT ARITHMETIC_EXP (SIMPLE_EXP, ARITHMETIC_OP, SIMPLE_EXP)>

<!ELEMENT EXP_VALUE (
        CONSTANT |
        ATTRIBUTE |
        FUNCTION_LIST)>

<!ELEMENT CONSTANT (#PCDATA)>
<!ELEMENT ATTRIBUTE (#PCDATA)>

<!-- Dimension Operators -->
<!ELEMENT LOGICAL_OP (#PCDATA)>
<!-- AND | OR -->

<!ELEMENT COND_OP (
        RELATIONAL_OP |
        EQUALITY_OP )>

<!ELEMENT RELATIONAL_OP (#PCDATA)>
<!-- GT | GTE | LT | LTE) -->

<!ELEMENT EQUALITY_OP (#PCDATA)>
<!-- EQUALS | NOT_EQUAL -->

<!ELEMENT OLAP_OP (#PCDATA)>
<!-- IN_RANGE | IN_LIST)> -->

<!ELEMENT OLAP_LIST (VALUE+)>
<!ELEMENT VALUE (#PCDATA)>

<!ELEMENT ARITHMETIC_OP (#PCDATA)>
<!-- ADD | SUBTRACT | MULTIPLY | DIVIDE) -->
```

Figure 4.17: SELECTION elements.

```
<!-- Union -->
<!ELEMENT UNION (DATA_QUERY)>

<!-- Intersection -->
<!ELEMENT INTERSECTION (DATA_QUERY)>

<!-- Difference -->
<!ELEMENT DIFFERENCE (DATA_QUERY)>
```

Figure 4.18: Set Operations.

dimension attribute either to be added or removed to/from the current data query result. Third, the PIVOT is defined as a listing of pairs of dimensions. Each pair of dimensions represents both the old dimension in the current data query result and the new dimension that will replace the old dimension in the new presentation.

Finally, in Figure 4.20, we illustrate the DRILL_ACROSS operation. DRILL_ACROSS is defined as a nested data query with zero or one comparison facts, defined relative to the current data query.

As previously noted, at run time the Object Oriented OLAP user query will be sent to the server in an XML format. Figure 4.21 illustrates the XML format for the OOP OLAP query in Figure 4.5. Note that this XML OLAP query corresponds direcly to our OLAP query grammar.

4.7 OLAP Algebraic laws for Improving OLAP Expression Trees

In this section, we describe a number of laws for our comprehensive OLAP algebra. To illustrate the motivation for this process, first recall that a query in traditional relational databases, written in SQL, is translated internally into an initial relational algebra expression that can be then transformed into equivalent, but more efficient

```
<!-- Rollup/Drill down -->
<!ELEMENT CHANGE_LEVEL (CHANGE_LEVEL_LIST+)>
<!ELEMENT CHANGE_LEVEL_LIST (DIMENSION_NAME, TARGET_LEVEL)>
<!ATTLIST CHANGE_LEVEL_LIST
                                    direction (UP | DOWN) #REQUIRED>
<!ELEMENT TARGET_LEVEL (#PCDATA)>

<!-- Changing the base -->
<!ELEMENT CHANGE_BASE (CHANGE_BASE_LIST+)>
<!ELEMENT CHANGE_BASE_LIST (PROJECTION_DIMENSION)>
<!ATTLIST CHANGE_BASE_LIST
                                     modification (ADD | REMOVE) #REQUIRED>

<!-- Pivot -->
<!ELEMENT PIVOT (PIVOT_LIST)>
<!ELEMENT PIVOT_LIST (PIVOT_PAIR+)>
<!ELEMENT PIVOT_PAIR (OLD_DIMENSION, NEW_DIMENSION)>
<!ELEMENT OLD_DIMENSION (#PCDATA)>
<!ELEMENT NEW_DIMENSION (#PCDATA)>
```

Figure 4.19: Change Level, Change Base and Pivot operations.

```
<!-- Drill across -->
<!ELEMENT DRILL_ACROSS (
                        DATA_QUERY,
                        COMPARE_FACT?)>
<!ELEMENT COMPARE_FACT (#PCDATA)>
```

Figure 4.20: Drill Across operation.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE QUERY SYSTEM "ClientQuery.dtd" []>
<QUERY><DATA_QUERY> <CUBE_NAME>ORDER</CUBE_NAME><OPERATION_LIST>

<!-- Display Attributes: Product.Type, Customer.Province, Measure(Quantity Ordered)-->
<PROJECTION>
    <MEASURE_LIST> <MEASURE>Quantity_ordered</MEASURE></MEASURE_LIST>
    <ATTRIBUTE_LIST>
     <PROJECTION_DIMENSION><DIMENSION_NAME>Product</DIMENSION_NAME>
        <ATTRIBUTE>Type</ATTRIBUTE>
     </PROJECTION_DIMENSION>
     <PROJECTION_DIMENSION><DIMENSION_NAME>Customer</DIMENSION_NAME>
        <ATTRIBUTE>Province</ATTRIBUTE>
     </PROJECTION_DIMENSION>
    </ATTRIBUTE_LIST>
</PROJECTION>

<!--Example of compund dimension: Customer(condition) AND Time (Condition)-->
<!-- Restriction: Customer.Age > 40 AND (Time.year =2007 AND Time.Month between (May
and October))-->
<SELECTION>
     <DIMENSION_LIST><COMPOUND_DIMENSION>
        <DIMENSION_LIST>
        <DIMENSION><DIMENSION_NAME>Customer</DIMENSION_NAME>
        <EXPRESSION><RELATIONAL_EXP>
        <BASIC_EXP>
           <SIMPLE_EXP><EXP_VALUE><ATTRIBUTE>Age</ATTRIBUTE></EXP_VALUE></SIMPLE_EXP>
           <COND_OP><RELATIONAL_OP>GT</RELATIONAL_OP></COND_OP>
           <SIMPLE_EXP><EXP_VALUE><CONSTANT>40</CONSTANT></EXP_VALUE></SIMPLE_EXP>
        </BASIC_EXP>
        </RELATIONAL_EXP></EXPRESSION></DIMENSION>
        </DIMENSION_LIST>

<LOGICAL_OP>AND</LOGICAL_OP>

      <DIMENSION_LIST><DIMENSION><DIMENSION_NAME>Time</DIMENSION_NAME>
       <EXPRESSION><COMPOUND_EXP>
        <EXPRESSION><RELATIONAL_EXP><BASIC_EXP>
          <SIMPLE_EXP><EXP_VALUE><ATTRIBUTE>Year</ATTRIBUTE></EXP_VALUE></SIMPLE_EXP>
          <COND_OP><RELATIONAL_OP>EQUALS</RELATIONAL_OP></COND_OP>
          <SIMPLE_EXP><EXP_VALUE><CONSTANT>2007</CONSTANT></EXP_VALUE></SIMPLE_EXP>
        </BASIC_EXP></RELATIONAL_EXP></EXPRESSION>
        <LOGICAL_OP>AND</LOGICAL_OP>
          <EXPRESSION><RELATIONAL_EXP>
            <OLAP_EXP>
            <SIMPLE_EXP><EXP_VALUE><ATTRIBUTE>Month</ATTRIBUTE></EXP_VALUE></SIMPLE_EXP>
            <OLAP_OP>IN_RANGE</OLAP_OP>
            <OLAP_LIST><VALUE>May</VALUE><VALUE>October</VALUE></OLAP_LIST>
            </OLAP_EXP>
          </RELATIONAL_EXP></EXPRESSION>
        </COMPOUND_EXP></EXPRESSION>
       </DIMENSION></DIMENSION_LIST>
    </COMPOUND_DIMENSION></DIMENSION_LIST>
</SELECTION>
</OPERATION_LIST>
</DATA_QUERY>
</QUERY>
```

Figure 4.21: The XML format for the OLAP query illustrated in Figure 4.5.

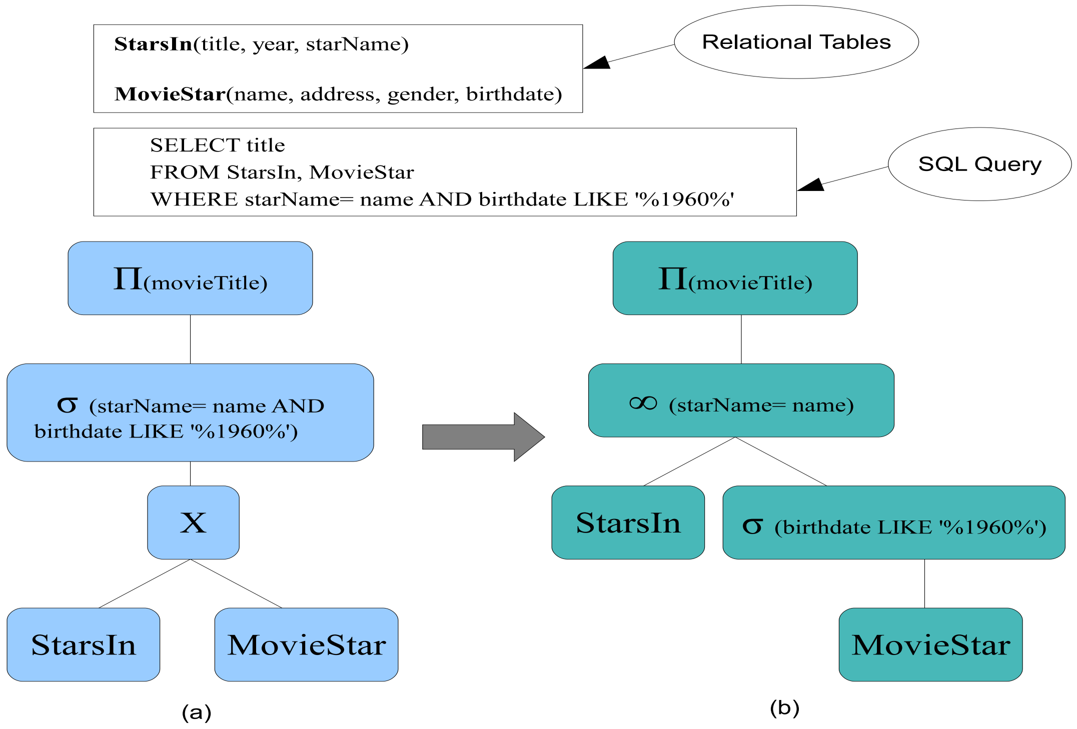

StarsIn(title, year, starName)
MovieStar(name, address, gender, birthdate)
Relational Tables
SELECT title
FROM StarsIn, MovieStar
WHERE starName= name AND birthdate LIKE '%1960%'
SQL Query
Π(movieTitle)
σ (starName= name AND birthdate LIKE '%1960%')
X
StarsIn
MovieStar
(a)
Π(movieTitle)
∞ (starName= name)
StarsIn
σ (birthdate LIKE '%1960%')
MovieStar
(b)

Figure 4.22: (a) Translation of SQL to an intial relational algebra expression. (b) The effect of applying some relational algebra laws.

ones by applying various relational algebraic rules. For example, the most common relational algebraic laws are (1) pushing the selection (σ) as far as possible, (2) combining selection (σ) with Cartesian product (X) to produce joins (∞), (3) introducing new projections (Π) when necessary, etc. In Figure 4.22(a), we can see how the SQL is transformed into an initial tree of relational algebra operations. Figure 4.22(b) improves the initial expression by applying common relational algebraic rules in some meaningful way. Specifically, we split the two parts of the selection (starname = name) and (birthdate LIKE '%1960'). The first condition involves attributes from both sides of the product, but they are equated, so the product and selection can be combined to produce an equijoin. The latter condition is pushed down the tree.

As noted, however, we are working in an OLAP environment, so we will be trying to apply similar logic to operations that are part of our OLAP algebra. Specifically, a number of OLAP-specific laws will be discussed. These laws can be used to turn an OLAP algebra expression tree into a more efficient, but logically equivalent, expression tree.

Figure 4.23 illustrates an initial OLAP algebra tree equivalent to the user's query outlined in Figure 4.21. The initial tree consists of operators from our OLAP algebra (i.e., SELECTION and PROJECTION). As was discussed in Chapter 3, the Sidera server stores cube data only for the most detailed encoded integer value (e.g., ProductID, CustomerID). Therefore, common OLAP analysis, such as the application of OLAP query constraints (e.g., condition Customer.Age>40 in Figure 4.23) or descriptive OLAP reports (e.g., Customer.Province mentioned within the PROJECTION operator in Figure 4.23), could not be performed from the fact table alone since it is the dimension tables that store descriptive attributes. In other words, we generally require joins between the cube and the dimension tables because (i) the query constraints are often specified on the attributes of the dimensions and (ii) descriptive attributes make OLAP reports easier to read. Moreover, we note that whenever descriptive dimension attributes are utilized by OLAP algebra operators, *inner joins* are required between the fact table and dimension tables. For example, the fact table of cube Order must be joined with the Customer and Time dimension tables to resolve the query constraints in Figure 4.23 and with Product and Customer for the PROJECTION operator.

Before digging in to the details, we briefly note the following objectives of our OLAP algebra laws:

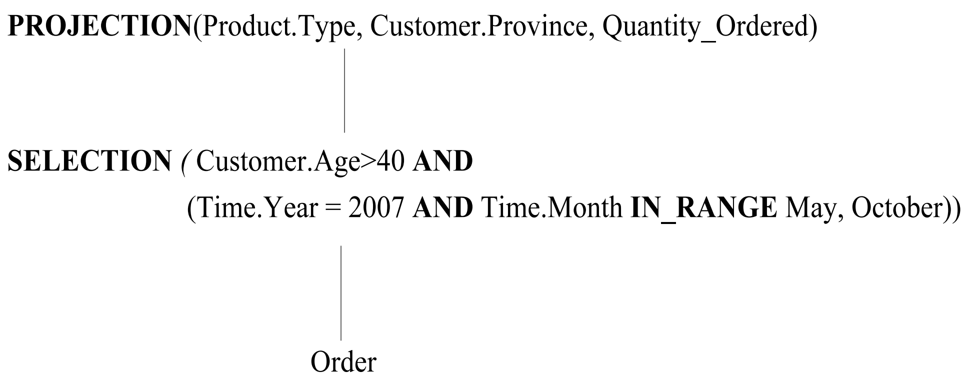

PROJECTION(Product.Type, Customer.Province, Quantity_Ordered)
SELECTION (Customer.Age>40 AND
(Time.Year = 2007 AND Time.Month IN_RANGE May, October))
Order

Figure 4.23: Initial OLAP algebra tree.

- Re-write OLAP operations in the expression tree against the schema of the cube and dimension tables that are stored in the OLAP server.
- Re-order OLAP operations in order to improve the expression tree.
- Eliminate (or reduce) joins between the cube and dimension tables. In other words, our server does not perform traditional relational sort or hash-based joins. Instead, it uses the structures (mapGraph) and indexes (FastBit bitmap) to perform better joins between the dimension tables and the cube.

4.7.1 Laws involving SELECTION

As noted, SELECTION is the core OLAP operation and is commonly referred to as "slicing and dicing". For convenience, we will use Cond1 and Cond2 to represent query restrictions. Moreover, we use **C** to refer to a view/cuboid.

4.7.1.1 Better Joins

A selection result depends on inner/natural joins between the cube and dimension tables in order to exclude cube rows that don't satisfy the query restriction specified on the descriptive attributes of the dimension. For example, in Figure 4.23, the Order fact table/cuboid must be joined with dimensions (Customer and Time) to get those rows in the fact table satisfying the conditions (Customer.Age > 40 AND Time.Year=2007 AND Time.Month IN_RANGE (May, October)). In order to perform better joins between the cube and dimension tables, we change the restriction of the selection operation so that it can be performed on the relevant cuboid/view alone.

Let the 2-dimensional cube C = <D, F, M, BasicCube>, where D={Dim1, Dim2}, F={Dim1.Dim1ID, Dim2.Dim2ID}. Note that Dim1ID and Dim2ID are the most detailed encoded values of dimensions Dim1 and Dim2 (e.g., ProductID in dimension Product). Therefore, in order to have better joins between the dimension tables (mentioned within the SELECTION operation) and the cube, we apply the following law:

LAW_1:

$$\textbf{\textit{SELECTION}}\ \textit{(Dim1(C1)}\ \textbf{\textit{AND/OR}}\ \textit{Dim2(C2)) (C)} =$$

$$\textbf{\textit{SELECTION}}\ \textit{(Dim1ID} = x\ \textbf{\textit{AND/OR}}\ \textit{Dim2ID} = y\textit{)(C)}$$

Where x and y are two sets of *Dim1ID*s and *Dim2ID*s that satisfy the conditions *C1* and *C2* associated with dimensions *Dim1* and *Dim2*.

Justification: Suppose a cell c is in the result of the left expression. Then there must be a record r that satisfies the restriction on dimension *Dim1* and a record

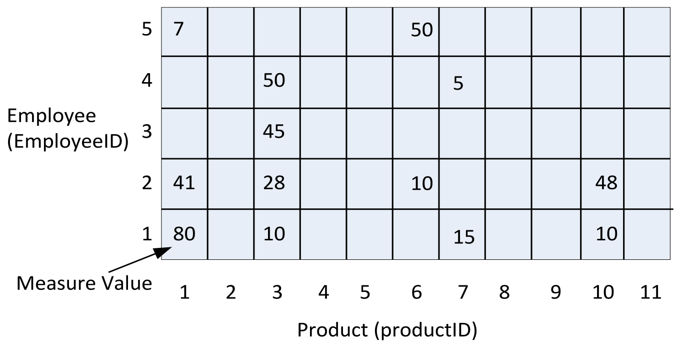
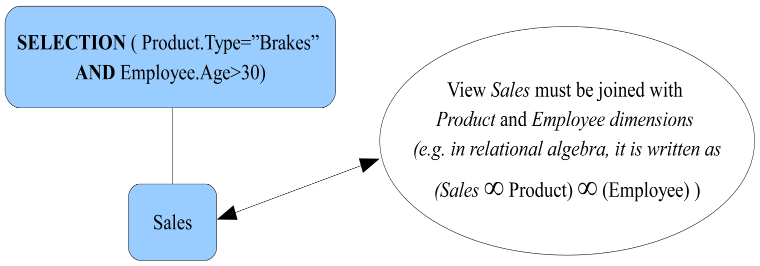


Figure 4.24: Two-dimensional cube with the most detailed level values.

s that satisfies the restriction on dimension *Dim2*. Moreover, *r* and *s* must agree with *c* on every attribute that each record shares with cell *c* (*Dim1ID* and *Dim2ID*). When we evaluate the expression on the right, x is a set of Dim1IDs satisfying the restriction associated with Dim1, while y is a set of Dim2IDs satisfying the restriction on dimension Dim2. Dim1ID of record r must be in set x and Dim2ID of record s must be in set y. Thus a cell c1 is in the result of the right expression. Consequently, Dim1ID and Dim2ID of cell c1 must agree with one value from set x (Dim1ID of record r) and one value from set y (Dim2ID of record s). Therefore, we can say that c and c1 is the same cell.

We use the same logic if the logical operator between dimension conditions is an OR operator. Figure 4.24 provides an illustration of a very simple two dimensional view (called Sales) with the most detailed value stored for each dimension in the cube. Suppose that the schema of this view is (ProductID, EmployeeID, Measure(s)). Feature attributes (EmployeeID and ProductID) allow connections to dimension tables Employee and Product in Figures 3.3 and 3.4 respectively. Consider an initial selection expression depicted in Figure 4.25 and specified as:



SELECTION (Product.Type="Brakes" AND Employee.Age>30)
Sales
View *Sales* must be joined with *Product* and *Employee dimensions* (e.g. in relational algebra, it is written as (Sales ∞ Product) ∞ (Employee))

Figure 4.25: Initial SELECTION expression tree.

$$\boldsymbol{SELECTION}\ (Product.Type = Brakes\ \boldsymbol{AND}\ Employee.Age > 30)\ Sales$$

Using LAW_1, it can be written as:

$$\boldsymbol{SELECTION}\ (ProductID = x\ \boldsymbol{AND}\ EmployeeID = y)\ Sales$$

such that x = {1,2} (ProductID 1 and 2 have attribute Type equals Brakes) and y={2,3}(EmployeeID 2 and 3 have age greater than 30). Figure 4.26 shows the resulting expression tree after applying LAW_1 to the initial expression tree of Figure 4.25. In Figure 4.25, the view (Sales) must be joined with the Product and Employee dimension tables. However, in Figure 4.26, we access each dimension (Product and Employee) to get the most detailed level values that satisfy the condition associated with it, then the view Sales can be accessed alone to answer the query. In other words, the relevant view is accessed to return those rows that intersect with the sets of the dimension's detailed values satisfying the query restriction. Therefore, using this law, the R-tree index of the appropriate view can be used efficiently to answer the query. Without this law, we can't gain any benefit from the existence of the R-tree index. Finally, this law (LAW_1) is supported direcly by the in-memory hierarchy manager and FastBit bitmap indexes. However, it can be used by any OLAP server that supports the star schema storage.

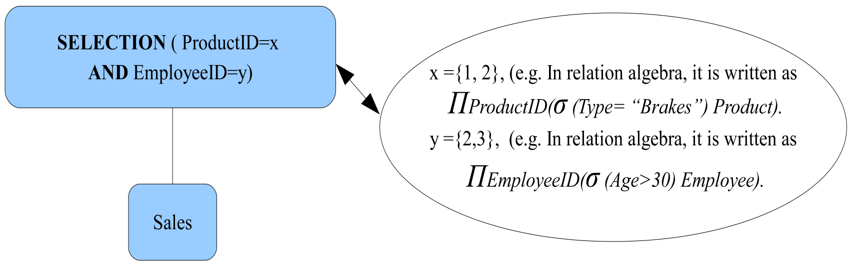

SELECTION (ProductID=x
AND EmployeeID=y)
Sales
x ={1, 2}, (e.g. In relation algebra, it is written as
ΠProductID(σ (Type= "Brakes") Product).
y ={2,3}, (e.g. In relation algebra, it is written as
ΠEmployeeID(σ (Age>30) Employee).

Figure 4.26: After applying LAW_1 the initial tree of Figure 4.25.

4.7.1.2 Combining conditions

When we have two or more consecutive SELECTION operators, we can replace them by only one SELECTION operator and connect their conditions with the AND operator(s). Thus, our second law for SELECTION is the combining law:

LAW_2:

SELECTION *(Cond1) (**SELECTION**(Cond2) C) =*

SELECTION *(Cond1 AND Cond2)(C)*

Justification: Suppose that a cell c is in the result of the left expression. Then the result of SELECTION(Cond2)C is a sub-cube C1 that contains cell c that satisfies cond2. We apply SELECTION(Cond1) to C1. The result is a sub-cube of the left expression that contains c that satisfies also Cond2. When we evaluate the right condition, cell c will again be in the result since c satisfies Cond1 and Cond2.

Since our OLAP server provides a very efficient multi-dimensional indexing scheme (i.e., the Hilbert R-tree index), this rule allows the SELECTION operation to benefit from this multi-dimensional indexing. Instead of accessing the appropriate R-tree index view to answer the first condition and then using the result cube to answer the second condition, the multi-dimensional index view can be efficiently used to answer both conditions simultaneously.

In addition to the above law, two SELECTION operators can be combined into only one SELECTION if there is a UNION operator between them. Moreover, the conditions of both SELECTIONs are connected with the OR operator. This law is written as follow:

LAW_3:

*(**SELECTION**(Cond1)C)* ***UNION*** *(**SELECTION**(Cond2)C) =*

SELECTION *(Cond1 OR Cond2)(C)*

Finally, LAW_2 and LAW_3 are very useful in any OLAP server that provides multidimensional indexing schemes (e.g., R-tree).

4.7.1.3 Pushing laws

Selection is a very important operation from the point of view of OLAP query optimization. In particular, Selection tends to reduce the size of the cubes. One of the most important objectives is to move the selection down the tree as far as it will go without changing what the OLAP expression tree actually does. In addition, pushing SELECTION down the tree makes it possible to be efficiently resolve the query from the apropriate multi-dimensional index view. The next family of laws allows us to push the SELECTION through other OLAP operators. Thus, we refer to this set of laws as the *pushing laws*. Figure 4.27 illustrates how the SELECTION can be pushed below other OLAP operators.

LAW_4:

p1	***SELECTION***(Cond)(C1 ***UNION*** C2) = ***SELECTION***(Cond)C1 ***UNION*** ***SELECTION***(Cond)C2
p2	***SELECTION***(Cond)(C1 ***DIFFERENCE*** C2) = ***SELECTION***(Cond)C1 ***DIFFERENCE*** ***SELECTION***(Cond)(C2)
p3	***SELECTION***(Cond)(C1 ***INTERSECTION*** C2) = ***SELECTION***(Cond)C1 ***INTERSECTION*** C2
p4	***SELECTION***(Cond) (***CHANGE_BASE*** (Ai->action) C1) = ***CHANGE_BASE***(Ai->action) ***(SELECTION***(Cond) C1)
p5	***SELECTION***(Cond) (C1(M1) ***DRILL_ACROSS*** C2(M2)) = ***SELECTION***(Cond) C1)(M1) ***DRILL_ACROSS*** ***SELECTION***(Cond) C2)(M2)

Figure 4.27: SELECTION *pushing laws.*

1. For a UNION, SELECTION must be pushed to both arguments of the UNION. p1 in Figure 4.27 illustrates this rule.

2. For a DIFFERENCE, SELECTION must be pushed to the first argument of the operator or to both arguments. For example, p2 in Figure 4.27 shows how we might push SELECTION to both arguments.

3. For an INTERECTION, SELECTION can be pushed to one of the arguments or both. p3 is an example of how we might push SELECTION to the first argument.

4. For CHANGE_LEVEL and CHANGE_BASE, SELECTION is pushed down to the argument. p4 provides an example of pushing SELECTION under CHANGE_BASE.

5. For a DRILL_ACROSS, SELECTION must be pushed to both arguments. p5 in Figure 4.27 shows this rule.

We shall provide a justification of one of the above variations as an example (p1, pushing the SELECTION under UNION) . Justifications for the remaining cases are straightforward.

Justification: Suppose that a cell c is in the result of SELECTION(Cond) (C1 UNION C2). Then the result of (C1 UNION C2) has a cell c that satisfies the condition parameter of the SELECTION operator. In addition, c can be a cell found only in C1, C2, or the result of two cells from both cubes C1 and C2. When we evaluate the right expression, SELECTION(Cond)C1 UNION SELECTION(Cond)C2, c will again be in the result of the right expression, because c matches the condition and can be found from C1, C2, or the result of the union.

For example, consider the two cuboids/views C1(Product.Number, Time.Month, Units_Sold) and C2(Product.Number, Time.Month, Units_Sold) of Figure 4.8. Figure 4.28(a) illustrates an intial OLAP expression tree. Using LAW_4, we can push the SELECTION operator to both arguments as depicted in Figure 4.28(b). It is an improvement to push the SELECTION to both arguments since we reduce the size of both C1 and C2 before the intersection. Moreover, if C1 and C2 are stored on disk in our server, then we can efficiently retrieve those cells from C1 and C2 satisfying the condition (e.g., Time.Month = Jan) by utilizing the R-tree index of C1 and C2.

4.7.1.4 Pulling laws

Pushing a selection down an OLAP expression tree is one of the most important steps performed by the query optimizer. However, we have found that in some situations it is essential to pull the SELECTION up the expression tree as far as it will go, and then push it down all possible branches. Consider two views/cuboids (C1 and C2) of Figure 4.8. We assume that we have the following OLAP algebra expression:

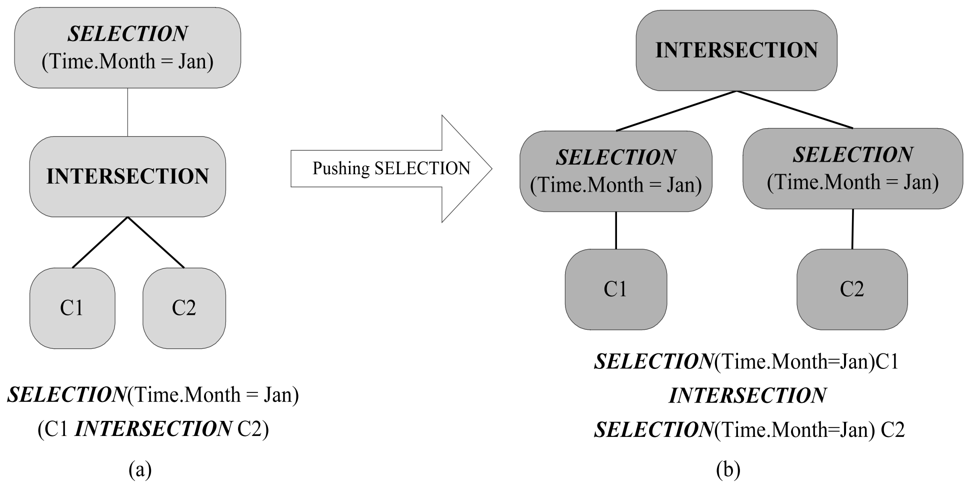


Figure 4.28: (a) Initial OLAP expression tree and (b) its equivalent after applying the SELECTION pushing laws.

$$
\begin{gathered}
(\textbf{PROJECTION}(\text{Product.Number, Time.Month, Units\_Sold}) \\
(\textbf{SELECTION}(\text{Time.Month} = \text{Dec})\text{C2}))\textbf{INTERSECTION} \\
\textbf{PROJECTION}(\text{Product.Number, Time.Month, Units\_Sold})\text{C1})
\end{gathered}
$$

The OLAP expression tree of the above OLAP algebra expression is shown in Figure 4.29(a). In this OLAP algebra tree, there is no way to push the SELECTION down the tree because it is already as far as it would go. However rule p3 in Figure 4.27 can be applied from right to left, to bring the SELECTION(Time.Month = Dec) above the INTERSECTION. Since C1 and C2 have the same schemas, then we may push the SELECTION to both arguments (C1 and C2). We can pull the SELECTION above the INTERSECTION and then push down if, and only if, the output of the INTERSECTION contains all attributes that are mentioned within the SELECTION. For example, because Time.Month is in the output schema of Figure 4.29(a), then we can pull the SELECTION operator up and then down. Figure 4.29(b) illustrates the expression tree resulting from pulling up and then down the SELECTION operator.

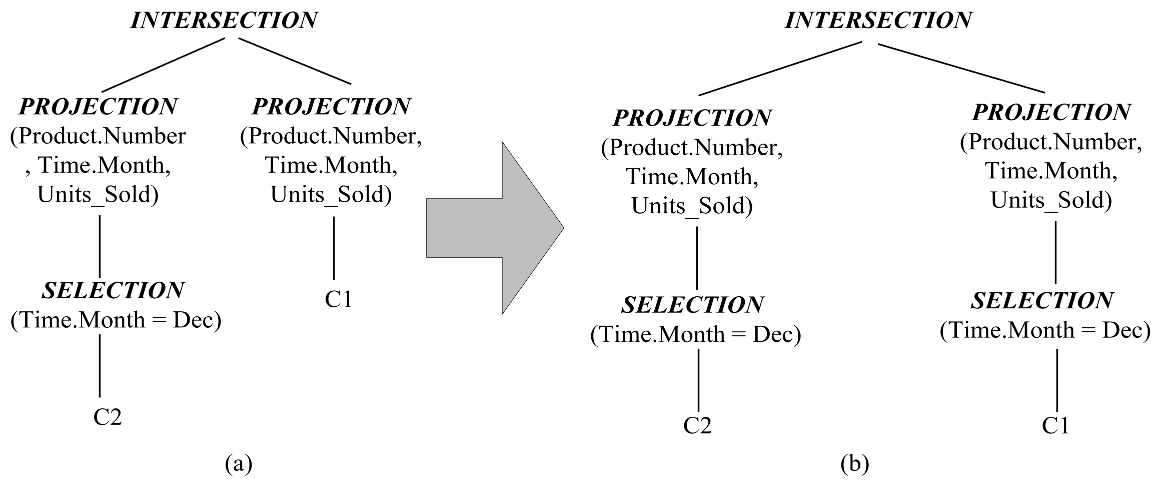

INTERSECTION
PROJECTION (Product.Number , Time.Month, Units_Sold)
SELECTION (Time.Month = Dec)
C2
PROJECTION (Product.Number, Time.Month, Units_Sold)
C1
(a)
INTERSECTION
PROJECTION (Product.Number, Time.Month, Units_Sold)
SELECTION (Time.Month = Dec)
C2
PROJECTION (Product.Number, Time.Month, Units_Sold)
SELECTION (Time.Month = Dec)
C1
(b)

Figure 4.29: (a) Initial OLAP expression tree. (b) Improving the initial expression by pulling SELECTION up and then pushing it down the tree.

This mechanism of pulling up and then pushing down the SELECTION operator is advantageous because the size of the view C1 is reduced in the intersection. Moreover, if C1 is stored in our server, then its R-tree index can be efficiently used to find those rows satisfying the condition (Time.Month=Dec). However, without this condition all cells in C1 must be accessed and read into the main memory.

In addition to the INTERSECTION operator, we can use the first rule *(p1)* in Figure 4.27 to pull up the SELECTION(s) that can be combined with the OR operator(s)(LAW_3) and then push them down to both arguments. Again, it is important to mention here that this technique — pull up and then down the SELECTION — can be used only if all attributes mentioned within the SELECTION(s) are in the output schema of the binary operation.

4.7.2 Laws involving PROJECTION

Recall that an OLAP query must always contain a projection because we assume that no default display attributes (output attributes) are available for the projection.

In other words, the PROJECTION operation determines the schema of the resulting cube query. Therefore, the guiding principle for PROJECTION laws is that we may introduce a new PROJECTION in the expression tree somewhere below an existing PROJECTION. If we do so, the new PROJECTION only eliminates attributes from the cube that are never used by any of the OLAP operators above. Therefore, we can introduce a new PROJECTION below a SELECTION in the expression tree. This is illustrated in the following law:

LAW_5:

PROJECTION(L,M)(***SELECTION***(Cond) C) =

PROJECTION(L,M)(***SELECTION***(Cond)***PROJECTION***(L1,M)C)

where L1 is the list of dimension attributes of cube C that are either used within the condition (Cond) of the SELECTION operator or are input attributes of L, L is the list of output dimension attributes and M is the list of output measure attributes.

To illustrate the importance of this law, consider the three-dimensional view of Figure 4.6(a). Figure 4.30 illustrates the application of the law. Assume that the user wants to see the total Units_Sold in all cities during the month of January (Jan). The initial expression tree is depicted in Figure 4.30(a) to answer the user's OLAP query. By applying this law, a new PROJECTION is introduced below the SELECTION to eliminate attribute (Product.Number), since the only required attributes/axes are Time.Month and Location.City. Figure 4.30(b) illustrates the resulting expression tree.

Since the new PROJECTION contains all attributes required to answer the current query, therefore it allows us to select the appropriate view from the input cube

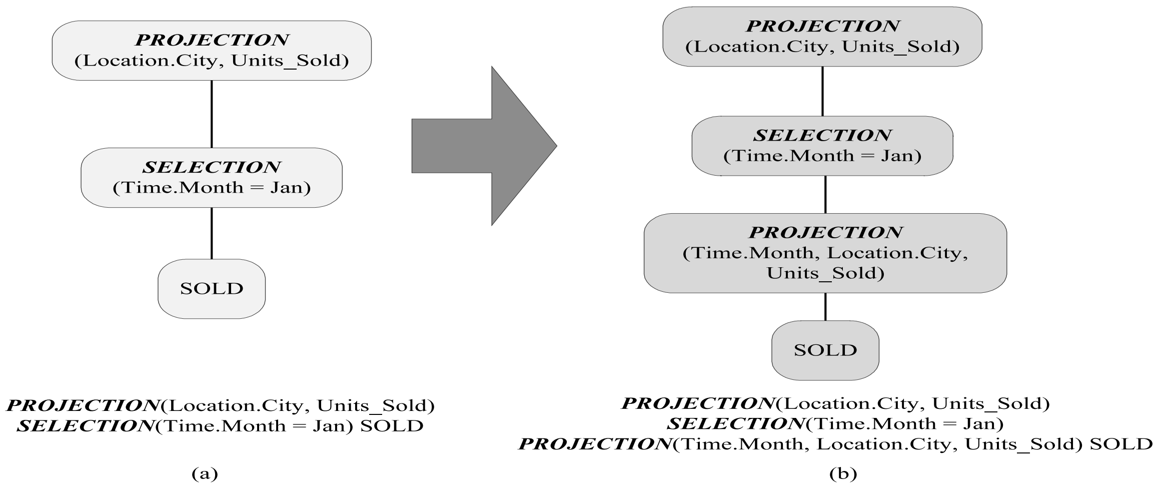

PROJECTION (Location.City, Units_Sold)
SELECTION (Time.Month = Jan)
SOLD
PROJECTION(Location.City, Units_Sold) SELECTION(Time.Month = Jan) SOLD
(a)
PROJECTION (Location.City, Units_Sold)
SELECTION (Time.Month = Jan)
PROJECTION (Time.Month, Location.City, Units_Sold)
SOLD
PROJECTION(Location.City, Units_Sold) SELECTION(Time.Month = Jan) PROJECTION(Time.Month, Location.City, Units_Sold) SOLD
(b)

Figure 4.30: (a) Initial OLAP expression tree.(b) Improving the initial expression by introducing new PROJECTION.

(i.e., materialized cube), or it reduces the size of the input cube (i.e., in-memory input view). In other words, this law is an improvement because it allows the query engine to select the best view (smaller than the base view) from the input cube or it reduces the size of the input view. This law is very useful in OLAP servers that provide cube materialization (pre-computed views are stored on disk). Instead of re-writing the whole query, we can just utilize this simple law to pick the best view to answer the query.

Because CHANGE_LEVEL and CHANGE_BASE are relevant to an existing result set, PROJECTION must be pushed below these operators. Thus, below are the pushing rules for the PROJECTION operator.

LAW_6:

1. ***PROJECTION***(L)(***CHANGE_LEVEL***(M) C) =
   CHANGE_LEVEL(M)(***PROJECTION***(L) C).

2. ***PROJECTION***(L)(***CHANGE_BASE***(M) C) =

CHANGE_BASE(M)(***PROJECTION***(L) C).

We note that a new PROJECTION cannot be introduced below the binary operations (UNION, INTERSECTION, DIFFERENCE, and DRILL_ACROSS) of our OLAP algebra.

4.7.2.1 Decomposition Law

Recall that Sidera stores the cube only for the most detailed dimension values (e.g., ProductID, EmployeeID). As was noted above, PROJECTION results depend on inner/natural joins between the cube and dimension tables to produce descriptive OLAP reports, since in our server it is the dimension tables that store descriptive attributes. Below is the decomposition law for PROJECTION:

LAW_7:

PROJECTION(L, M) C =

CHANGE_LEVEL(L1→L) ***PROJECTION***(L1, M)C

where L is a list of hierarchical attributes, L1 is the list of feature attributes in C that link the cube C with its corresponding dimensions mentioned in L, and M is one or more measure attributes in C. This law should produce an improved query plan since it defers the joins between the cube and the dimension tables to a later step of the query execution. Using this law, the PROJECTION can be answered from the appropriate view (e.g., C) without joining with dimension tables, while the joins will be required for the CHANGE_LEVEL operation that can be defered to the later step of the query. In our server (Sidera), the mapGraph structure can be used to translate between the base level data (i.e., the most detailed) in the cube and the hierarchy level listed in the initial query.

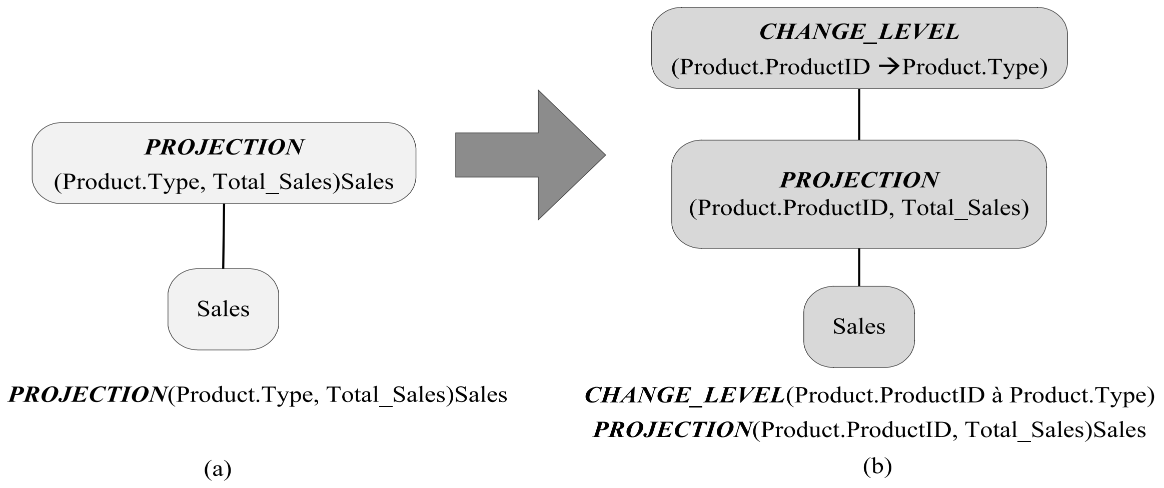

PROJECTION
(Product.Type, Total_Sales)Sales
Sales
PROJECTION(Product.Type, Total_Sales)Sales
(a)
CHANGE_LEVEL
(Product.ProductID →Product.Type)
PROJECTION
(Product.ProductID, Total_Sales)
Sales
CHANGE_LEVEL(Product.ProductID à Product.Type)
PROJECTION(Product.ProductID, Total_Sales)Sales
(b)

Figure 4.31: (a) Initial OLAP expression tree. (b) Improving the initial expression by decomposing the PROJECTION.

Consider the two dimensional cuboids in cube Sales of Figure 4.24 and its surrounding dimension tables (Employee and Product) of Figure 3.3 and 3.4. Figure 4.31(a) illustrates the intial OLAP expression tree with only one PROJECTION operation that needs to be joined with dimenion tables. Using LAW_7, the resulting expression tree is depicted in Figure 4.31(b).

4.7.3 Laws for CHANGE_LEVEL and CHANGE_BASE

As noted, the CHANGE_LEVEL and CHANGE_BASE operators are relevant to an existing result set. Therefore, they must be above the PROJECTION operator that defines the schema of the output cube result (LAW_6).

4.7.3.1 Removal Law

The CHANGE_BASE operator can be removed from an OLAP expression tree by the following *"removal"* law:

LAW_8:

CHANGE_BASE(L1→Add, L2→Remove) ***PROJECTION***(L, M)C = ***PROJECTION***(K, M)C.

where L, L1 and L2 are lists of attributes such that all attributes that are removed by the CHANGE_BASE operator (L2) must be in the input cube that is itself the result of the PROJECTION operator ($L2 \subset L$). K is a list of attributes that are in L, L1 and not in L2 ($K = L \cup L1 \notin L2$). This law is considered to be an improvement because only one PROJECTION will be executed instead of one PROJECTION and one CHANGE_BASE.

4.7.3.2 Pushing and Pulling Laws

LAW_9: If the CHANGE_LEVEL operator changes the result data from the most summarized (up) to the most detailed (down) along a concept hierarchy (Drill Down), then we pull the operator (CHANGE_LEVEL) up the tree until it reaches another CHANGE_LEVEL. In general, this law reduces the size of intermediate results because it pulls up the drill down operation that increases the size of the result cube. In addition, application of this law means that the root of the resulting expression tree will be a CHANGE_LEVEL operator. As a concrete example, consider the OLAP expression tree of Figure 4.32(a) (e.g., CHANGE_LEVEL(K) corresponds to a drill down operation). The resulting expression tree, after applying LAW_9, is depicted in Figure 4.32(b). The second tree is an improvement over the first one because after pulling the CHANGE_LEVEL operation up, we reduce the size of intermediate results. For example, in Figure 4.32(a) the size of the intermediate results (e.g., R) before the UNION is bigger than that of Figure 4.32(b) (e.g., R1) because CHANGE_LEVEL in the former changes the result data from the most summarized level to the most detailed level.

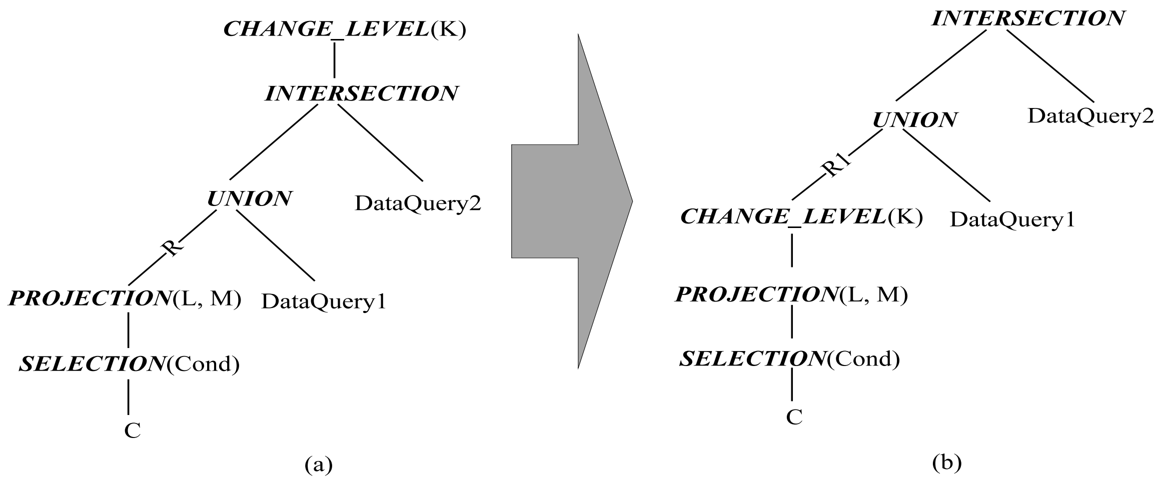


Figure 4.32: (a) Initial OLAP expression tree. (b) Result of pulling up CHANGE_LEVEL(K).

LAW_10: If the CHANGE_LEVEL operator navigates among levels of data ranging from the most detailed (down) to the most summarized (up) (Roll UP), then we push it down the tree until it reaches a PROJECTION operator. In general, pushing this type of CHANGE_LEVEL (Roll-up) operator reduces the size of intermediate results because the result of the roll up operation changes from the most detailed level to the most summarized (e.g., 12 month values is one year value). Note that the CHANGE_LEVEL can be pushed below the UNION, INTERSECTION and DRILL_ACROSS but not below the DIFFERENCE operator.

Finally, if a CHANGE_LEVEL involves both operations (a drill down and roll up) of an existing result set, then it is important to estimate the size of intermediate results before deciding whether to pull it up or push it down.

4.7.3.3 Merging Law

We refer to the next law concerning the CHANGE_LEVEL operator as the *merging law*, as it merges two or more consecutive CHANGE_LEVELs into one.

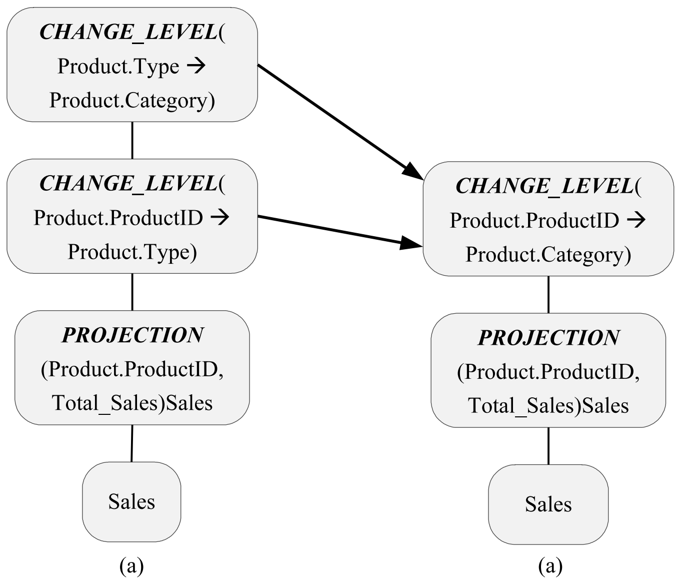
LAW_11:

$$\boldsymbol{CHANGE\_LEVEL}(\text{LI}\rightarrow\text{LO})\ \boldsymbol{CHANGE\_LEVEL}(\text{MI}\rightarrow\text{MO})\text{C} = \boldsymbol{CHANGE\_LEVEL}(\text{KI}\rightarrow\text{KO})\text{C}$$

where LI, LO, MI, MO, KI and KO are lists of hierarchical attributes. Recall that CHANGE_LEVEL is applied to an existing result cube to change one or more attribute level values. Therefore, all attributes mentioned in MI must be in the result view C. The parameters of the CHANGE_LEVELs are merged as follows:

- KI = LI ∪ MI, all attributes in MI and LI. however if some attributes in MI and LI belong to the same hierarchy, then we select only attributes from LI to be in KI.
- KO = LO ∪ MO, all attributes in LO and MO. However, if they have some attributes that belong to the same hierarchy, then we select those attributes from LO.

For example, consider the two-dimensional view (Sales) of Figure 4.24 and its surrounding dimensions(Employee and Product). Figure 4.33(a) illustrates the initial expression tree with two consecutive CHANGE_LEVELs. For example, LI = Product.Type, LO = Product.Category, MI = Product.ProductID and MO = Product.Type. Figure 4.33 illustrates the result expression tree produced by using LAW_11. In this example, KI = MI and KO = LO because (LI and MI) and (LO and MO) have attributes that belong to the same hierarchy (Product). The above law is likely to improve the performance if the CHANGE_LEVELs involve attributes from the same hierarchy because it reduces the number of translations between hierarchical levels.



CHANGE_LEVEL(Product.Type → Product.Category)
CHANGE_LEVEL(Product.ProductID → Product.Type)
PROJECTION (Product.ProductID, Total_Sales)Sales
Sales
(a)
CHANGE_LEVEL(Product.ProductID → Product.Category)
PROJECTION (Product.ProductID, Total_Sales)Sales
Sales
(a)

Figure 4.33: (a) Initial OLAP expression tree. (b) Result of merging two CHANGE_LEVELs.

4.7.4 Commutative, Associative and Trivial Laws

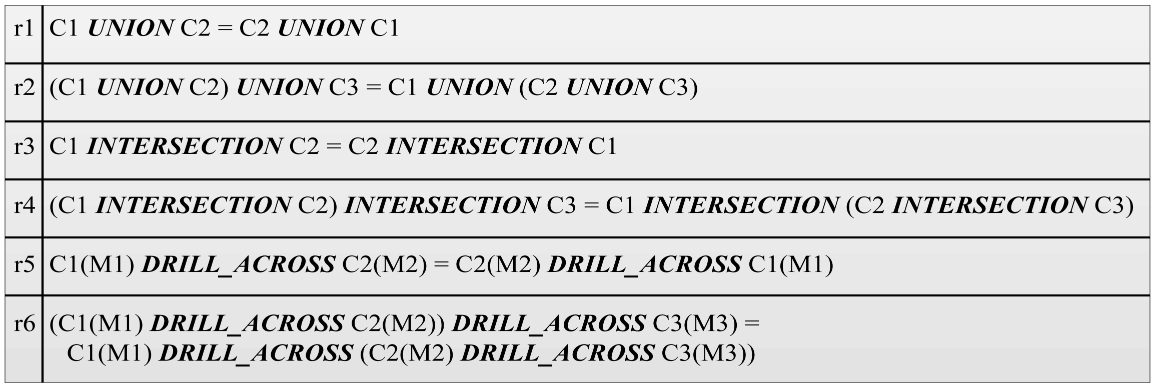
Several of our OLAP algebraic operators are both associative and commutative [27]. Our associative and commutative laws are illustrated in Figure 4.34 (***LAW_12***). Note that, strictly speaking, this is not a new law; it is merely a property of existing laws. There are many trivial laws for our OLAP algebra. However, we will specifically mention two of them that will be used by our query compiler (discussed in the next chapter).

LAW_13:

C1 ***INTERSECTION*** C1 = C1

C1 ***INTERSECTION*** C1 = C2

such that C2 has the same dimension members as C1. However the values of its measure attributes are double the values of measure attributes in C1.



r1	C1 ***UNION*** C2 = C2 ***UNION*** C1
r2	(C1 ***UNION*** C2) ***UNION*** C3 = C1 ***UNION*** (C2 ***UNION*** C3)
r3	C1 ***INTERSECTION*** C2 = C2 ***INTERSECTION*** C1
r4	(C1 ***INTERSECTION*** C2) ***INTERSECTION*** C3 = C1 ***INTERSECTION*** (C2 ***INTERSECTION*** C3)
r5	C1(M1) ***DRILL_ACROSS*** C2(M2) = C2(M2) ***DRILL_ACROSS*** C1(M1)
r6	(C1(M1) ***DRILL_ACROSS*** C2(M2)) ***DRILL_ACROSS*** C3(M3) = C1(M1) ***DRILL_ACROSS*** (C2(M2) ***DRILL_ACROSS*** C3(M3))

Figure 4.34: Commutative and Associative rules.

Finally, it should be possible for OLAP servers that store the data warehouse in the standard star schema format and provide multi-dimensional indexing schems and cube materialization to utilize some of our algebraic laws mentioned in this section. For example, LAW_2 is used to gain benefits from multi-dimensional indexing. LAW_11 can be used to minimize the translations between hierarchical levels. If the result is cached in the main memory, then the change base (with the remove action) and the change level can be answered from the cache direcly. However, LAW_8 is important in case of the add action. LAW_8 and LAW_9 are very important since the minimize the size of intermediate cube results.

4.8 OLAP Metadata Storage

Metadata describes the structure and constraints of the OLAP environment. It should include the structure of the available cubes, dimensions, hierarchies, measure attributes, etc. Moreover, it is important to emphasize that OLAP queries rely extensively on metadata (even more than relational databases) about the OLAP environment. For example, hierarchy information is crucial for answering even the simplest queries. Metadata must also be available to the query processor (Query Compiler and Query Execution Engine) that optimizes and executes the query. Therefore, it may also include information about the size of each cuboid/relation, existence of indexes, attributes cardinalities, etc.

In this section, we shall describe an XML DTD that defines the format of the schema used in our OLAP environment. A description of the native XML metadata storage mechanism is also provided. At the end of this section, we will give a concrete example that shows how an actual data cube is represented natively in XML.

4.8.1 OLAP Metadata Grammar

Our OLAP schema grammar (encoded as an XML DTD) defines the proper format of the OLAP metadata. It is defined recursively and made up of standard DTD elements. Figure 4.35 shows that a database houses one or more cubes. Typically, a database would represent all the cubes of a given organization or department. The dimensions specified at this level are shared by all cubes in the database. These *"global"* dimensions are commonly referred to as conformed dimensions.

A cube is the basic schema element. There may be many cubes in the database. Figure 4.36 demonstrates that each cube is made up of a cube name, one fact and one or more dimensions. A fact is a set of feature and measure attributes. Each feature attribute must be linked to a specific dimension. The kind of measurement operation must also be specified. Here, we define just three (sum, average, and count), but others can be added in the future.

Figure 4.37 depicts the schema of a dimension. Dimensions are made up of a set of descriptive attributes and one or more hierarchies. They must also provide a key that the fact table can reference. In some cases, dimensions simply hold a reference

```
<!ELEMENT DATABASE (DATABASE_NAME,
                                        CUBE+,
                                         DIMENSION_DEFINITION*)>

<!ELEMENT DATABASE_NAME (#PCDATA)>
```

Figure 4.35: DATABASE Element.

```
<!-- Cube Schema-->
<!ELEMENT CUBE (CUBE_NAME, FACT, CUBE_DIMENSION+)>

<!-- CUBE Name-->
<!ELEMENT CUBE_NAME (#PCDATA)>

<!-- Fact Table Schema-->
<!ELEMENT FACT (FEATURE+, MEASURE+)>

<!-- Feature Attributes-->
<!ELEMENT FEATURE (#PCDATA)>
<!ATTLIST FEATURE dimLink IDREF #REQUIRED>

<!-- Measure Attributes-->
<!ELEMENT MEASURE (#PCDATA)>
<!ATTLIST MEASURE type (sum | average | count) #REQUIRED>
```

Figure 4.36: Cube Schema.

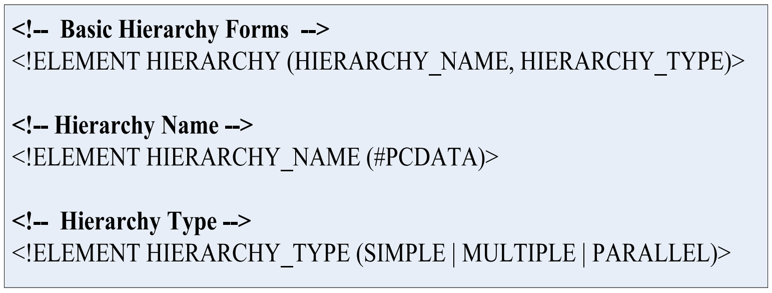
```
<!ELEMENT CUBE_DIMENSION (EXTERNAL_REFERENCE | DIMENSION_DEFINITION)>

<!-- Dimension Schema -->
<!ELEMENT DIMENSION_DEFINITION (DIMENSION_NAME,
                                ATTRIBUTE+,
                                HIERARCHY*)>

<!-- Dimension Key -->
<!ATTLIST DIMENSION_DEFINITION dimID ID #REQUIRED>

<!-- External Dimension -->
<!ELEMENT EXTERNAL_REFERENCE (DIMENSION_NAME)>
<!ATTLIST EXTERNAL_REFERENCE dimLink IDREF #REQUIRED>
<!ATTLIST EXTERNAL_REFERENCE scope (database|system) #REQUIRED>

<!ELEMENT DIMENSION_NAME (#PCDATA)>

<!-- Attribute Name and Type -->
<!ELEMENT ATTRIBUTE (#PCDATA)>
<!ATTLIST ATTRIBUTE type (int|float|string|date) #REQUIRED>
```

Figure 4.37: Dimension Schema.

to a shared dimension defined elsewhere. Attributes in a dimension require a storage type (e.g., int, float, string, or date).

In Figure 4.37, we can also see that a dimension can have one or more hierarchies. Hierarchies are the most complex part of the schema, and consist of both simple and composite forms (we note that although the entire schema file can be written by hand, we expect that UML modeling tools will typically be used to make the design process easier). All hierarchies are essentially graphs. Each has the same basic structure: a series of parent/child relationships. The distinctions between hierarchy forms[74] are actually established by the attributes of each hierarchy level. Figure 4.38 demonstrates the basic hierarchy form. Here, a dimension hierarchy is defined by its name and type. A hierarchy type can be simple, multiple, or parallel.

Figure 4.39 shows that a simple hierarchy is defined as a DRILL_DOWN element.



```
<!-- Basic Hierarchy Forms -->
<!ELEMENT HIERARCHY (HIERARCHY_NAME, HIERARCHY_TYPE)>

<!-- Hierarchy Name -->
<!ELEMENT HIERARCHY_NAME (#PCDATA)>

<!-- Hierarchy Type -->
<!ELEMENT HIERARCHY_TYPE (SIMPLE | MULTIPLE | PARALLEL)>
```

Figure 4.38: Hierarchy Element.

Also, simple hierarchies can be further sub-divided into three basic types: Symmetric, Asymmetric, or Generalized. Note that it is possible for simple hierarchies to be either strict (one-to-many relationships between parent and child nodes) or non-strict (many-to-many relationships between parent and child nodes).

A DRILL_DOWN pathway consists of a possibly nested series of parent/child relationships. Drill down relationships can be strict or non strict[74, 34](i.e., one-to-many versus many-to-many). A node has many characteristics: Node Name, Type, and Mandatory. Each node must have at least one name; however, multiple node names can be used in generalized hierarchies. The node type can be a root (top of hierarchy), base (detailed data level), or intermediate level. Finally, by default a node is mandatory, implying that the hierarchy graph is complete. If a node is not mandatory, this indicates that a level is in (i) a asymmetric/unbalanced hierarchy or (2) a ragged generalized hierarchy. Figure 4.39 illustrates the schema of a simple hierarchy as described above.

Figure 4.40 defines the structure of multiple hierarchies and parallel hierarchies[74, 34]. Multiple hierarchies are defined if we have two or more SIMPLE hierarchies that share a common criterion of analysis. In an exclusive multiple hierarchy there are no shared intermediate nodes. Finally, we define the parallel hierarchies that have two

```
<!--  Simple Hierarchies  -->
<!ELEMENT SIMPLE (DRILL_DOWN)>

<!--  Simple Hierarchy Type  -->
<!ATTLIST SIMPLE type (symmetric | asymmetric | generalized) #REQUIRED>

<!--  Simple Strict or not  -->
<!ATTLIST SIMPLE strict (yes | no) #IMPLIED>

<!--  A DRILL_DOWN Element  -->
<!ELEMENT DRILL_DOWN (NODE, CHILD)>
<!ELEMENT CHILD (NODE | DRILL_DOWN)>

<!-- DRILL_DOWN  strict or non strict  -->
<!ATTLIST DRILL_DOWN strict (yes|no) #IMPLIED>

<!--  Node Name  -->
<!ELEMENT NODE (NODE_NAME+)>
<!ELEMENT NODE_NAME (#PCDATA)>

<!--  Node Type  -->
<!ATTLIST NODE type (root | intermediate | base) #REQUIRED>

<!--  Node Mandotory or not -->
<!ATTLIST NODE mandatory (yes | no) #IMPLIED>
```

Figure 4.39: Simple Hierarchy Schema.

```
<!-- Multiple Hierarchies -->
<!ELEMENT MULTIPLE (SIMPLE, SIMPLE+)>
<!ATTLIST MULTIPLE exclusive (yes | no) #REQUIRED>

<!-- Parallel Hierarchies -->
<!ELEMENT PARALLEL (SIMPLE, SIMPLE+)>
<!ATTLIST PARALLEL independent (yes | no) #REQUIRED>
```

Figure 4.40: Multiple and Parallel Hierarchies.

or more SIMPLE hierarchies that have distinct criteria of analysis. This generally means that each hierarchy has a unique root. In an independent hierarchy, all nodes other than the base level are unique.

4.8.1.1 Simple OLAP Schema

Because data will be sent to/from the server in XML format, our meta-data storage is done natively in XML. Figure 4.41 illustrates the XML format of a simple OLAP environment that corresponds to our OLAP Metadata Grammar defined above. We can see in Figure 4.41 that we have one database (*Concordia University*) that has only one cube called *Sales* and a "global conformed" dimension called *Time*. Cube *Sales* consists of a measure attribute (*Total Sales*) of type sum and two features attributes (*CustomerID* and *DateID*) that relate cube *Sales* with two dimensions (*Customer* and *Time*).

The schema of each dimension is also included. For example, Dimension *Customer* has five attributes of mixed types (e.g., Age and City are attributes of type Int and String respectively) and a generalized hierarchy called *distribution*. The *Time* dimension has five attributes of type Int (*Day, Month, Year, Week,* and *Quarter*) and a multiple hierarchy called (*Date*). Specifically, The *Date* hierarchy includes two simple symmetric hierarchies which are: (i) Year (root) → Quarter (intermediate) → Month (intermediate) → Day(base) and (i) Year (root) → Week (intermediate) → Day (base). Note that the *Time* dimension is a global conformed dimension and would not actually be defined as part of any cube, it is defined as a part of the database. The *Customer* dimension, however, is a local dimension for cube *Sales*.

Finally, since we are already using Berkeley DB tools, we employ the Berkeley DB XML product engine to store the metadata. This allows us to use standard XML tools such as XPath and XQuery to easily manipulate the data of the OLAP schema.

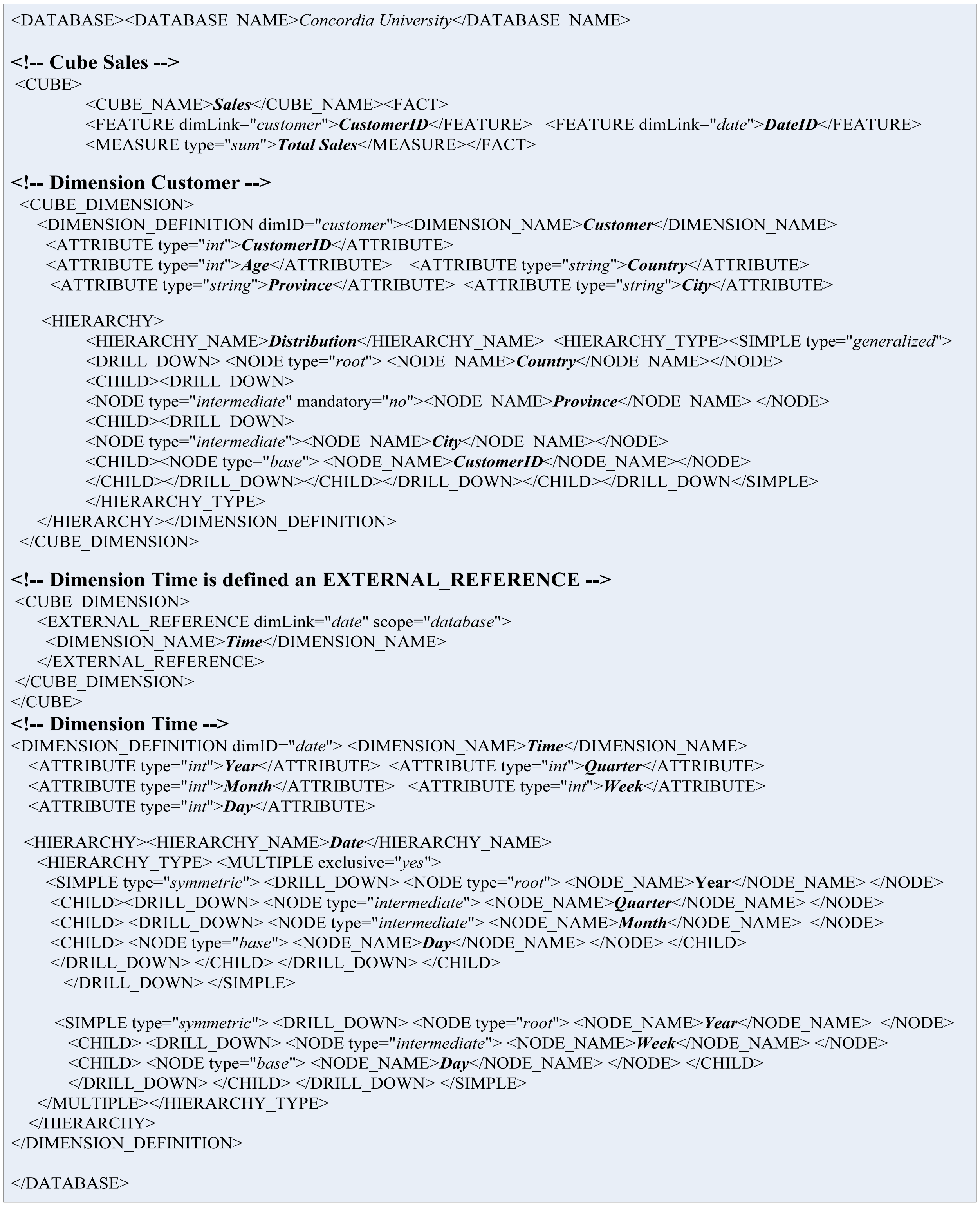
4.9 Review of Research Objectives

In Section 4.4, we identified a number of research objectives for this chapter. We now review those goals to confirm that they have in fact been accomplished.

1. **Support the creation of a native client-side OOP OLAP query language.** This is accomplished by means of the definition of a new multidimensional OLAP query algebra and grammar. The grammar, in turn, is the foundation of a native language query interface that eliminates the reliance on an intermediate, string based embedded language.

2. **Reduce the complexity caused by directly utilizing the relational algebra in the OLAP context (via SQL or MDX).** We propose a comprehensive multidimensional OLAP algebra that contains a set of cube-specific OLAP operations. In other words, it is a pure OLAP-aware algebra that directly exploits Sidera's conceptual model.

3. **Provide the developer with an Object Oriented representation of the**



```
<DATABASE><DATABASE_NAME>Concordia University</DATABASE_NAME>

<!-- Cube Sales -->
 <CUBE>
          <CUBE_NAME>Sales</CUBE_NAME><FACT>
          <FEATURE dimLink="customer">CustomerID</FEATURE>   <FEATURE dimLink="date">DateID</FEATURE>
          <MEASURE type="sum">Total Sales</MEASURE></FACT>

<!-- Dimension Customer -->
  <CUBE_DIMENSION>
    <DIMENSION_DEFINITION dimID="customer"><DIMENSION_NAME>Customer</DIMENSION_NAME>
      <ATTRIBUTE type="int">CustomerID</ATTRIBUTE>
      <ATTRIBUTE type="int">Age</ATTRIBUTE>    <ATTRIBUTE type="string">Country</ATTRIBUTE>
       <ATTRIBUTE type="string">Province</ATTRIBUTE>  <ATTRIBUTE type="string">City</ATTRIBUTE>

     <HIERARCHY>
          <HIERARCHY_NAME>Distribution</HIERARCHY_NAME>  <HIERARCHY_TYPE><SIMPLE type="generalized">
          <DRILL_DOWN> <NODE type="root"> <NODE_NAME>Country</NODE_NAME></NODE>
          <CHILD><DRILL_DOWN>
          <NODE type="intermediate" mandatory="no"><NODE_NAME>Province</NODE_NAME> </NODE>
          <CHILD><DRILL_DOWN>
          <NODE type="intermediate"><NODE_NAME>City</NODE_NAME></NODE>
          <CHILD><NODE type="base"> <NODE_NAME>CustomerID</NODE_NAME></NODE>
          </CHILD></DRILL_DOWN></CHILD></DRILL_DOWN></CHILD></DRILL_DOWN</SIMPLE>
          </HIERARCHY_TYPE>
    </HIERARCHY></DIMENSION_DEFINITION>
  </CUBE_DIMENSION>

<!-- Dimension Time is defined an EXTERNAL_REFERENCE -->
 <CUBE_DIMENSION>
    <EXTERNAL_REFERENCE dimLink="date" scope="database">
      <DIMENSION_NAME>Time</DIMENSION_NAME>
    </EXTERNAL_REFERENCE>
 </CUBE_DIMENSION>
</CUBE>
<!-- Dimension Time -->
<DIMENSION_DEFINITION dimID="date"> <DIMENSION_NAME>Time</DIMENSION_NAME>
   <ATTRIBUTE type="int">Year</ATTRIBUTE>  <ATTRIBUTE type="int">Quarter</ATTRIBUTE>
   <ATTRIBUTE type="int">Month</ATTRIBUTE>   <ATTRIBUTE type="int">Week</ATTRIBUTE>
   <ATTRIBUTE type="int">Day</ATTRIBUTE>

  <HIERARCHY><HIERARCHY_NAME>Date</HIERARCHY_NAME>
    <HIERARCHY_TYPE> <MULTIPLE exclusive="yes">
     <SIMPLE type="symmetric"> <DRILL_DOWN> <NODE type="root"> <NODE_NAME>Year</NODE_NAME> </NODE>
      <CHILD><DRILL_DOWN> <NODE type="intermediate"> <NODE_NAME>Quarter</NODE_NAME> </NODE>
      <CHILD> <DRILL_DOWN> <NODE type="intermediate"> <NODE_NAME>Month</NODE_NAME>  </NODE>
      <CHILD> <NODE type="base"> <NODE_NAME>Day</NODE_NAME> </NODE> </CHILD>
      </DRILL_DOWN> </CHILD> </DRILL_DOWN> </CHILD>
        </DRILL_DOWN> </SIMPLE>

      <SIMPLE type="symmetric"> <DRILL_DOWN> <NODE type="root"> <NODE_NAME>Year</NODE_NAME>  </NODE>
        <CHILD> <DRILL_DOWN> <NODE type="intermediate"> <NODE_NAME>Week</NODE_NAME> </NODE>
        <CHILD> <NODE type="base"> <NODE_NAME>Day</NODE_NAME> </NODE> </CHILD>
        </DRILL_DOWN> </CHILD> </DRILL_DOWN> </SIMPLE>
    </MULTIPLE></HIERARCHY_TYPE>
   </HIERARCHY>
</DIMENSION_DEFINITION>

</DATABASE>
```

Figure 4.41: Sample OLAP Schema.

primary OLAP structural elements as well as providing the foundation for a concrete OLAP client query language. We define a DTD-encoded multidimensional OLAP grammar developed specifically for BI analysis. Specifically, Sidera provides a pre-processor mechanism that translates standard OOP source code representing the user's OLAP query into an XML-based OLAP query that matches the format of our OLAP query grammar.

4. **Support query optimization and execution by means of applying new multi-dimensional OLAP algebraic laws.** By providing various OLAP laws for our OLAP algebra, we allow the query engine to turn an initial expression tree representing a user-defined query into a more efficient, but equivalent, OLAP expression tree. We will see in the next chapter how these laws will be used to support query optimization and execution.

5. **Provide the format of the OLAP metadata.** We discuss DTD-encoded OLAP metadata that defines the format of the schema for our OLAP environment. Moreover, the metadata is stored in Berkeley DB XML where we can easily use standard XML tools such as XQuery and XPath to manipulate that storage.

4.10 Conclusion

In this chapter, we have described a comprehensive multi-dimensional OLAP algebra. Our algebra — represents all common OLAP operations — reduces the complexity of using existing relational algebras to write OLAP queries (via SQL or MDX) and also subsequently allows for the optimization of OLAP queries written in native OOP languages such as Java. Moreover, we are providing optimization laws and execution algorithms that show how and why an OLAP algebra is a good idea in practice. In association with the algebra, we have developed a robust DTD-encoded OLAP query grammar that provides a concrete foundation for client language queries. The grammar, in turn, is the basis of a native language query interface that eliminates the reliance on an intermediate, string-based embedded language. Finally, the storage of the schema is done natively in XML.

In summary, our comprehensive OLAP query algebra (operations and laws), grammar and metadata storage are essential components in the process of resolving OLAP queries written in native OOP languages. In the next chapter, we will see how these components, as well as the storage engine discussed in Chapter 3, are integrated with the query compiler and execution engine to form a pure OLAP DBMS.

Chapter 5

Multi-Dimensional OLAP Query Processor

5.1 Introduction

In this chapter, we describe an OLAP query processor that efficiently parses and executes the OLAP queries discussed in the previous chapter. The parsing and execution of these OLAP query is of course contingent upon the data storage engine — composed of Sidera, the Berkeley DB and FastBit components — presented in Chapter 3. Recall that the new Sidera server provides a "native language" query facility that enables one to support native, client-side OOP querying without the need to embed an intermediate, non-OOP language such as SQL or MDX. Sidera provides persistent transparency via a source code re-writing mechanism that interprets the developer's OOP query specification and decomposes it into the core operations of our OLAP algebra (as previously mentioned in Chapter 4). These operations are given a concrete form within the OLAP grammar and then transparently delivered at run-time to the backend analytics server for processing. In this chapter, we focus on an OLAP query processor that includes a set of components for the efficient resolution of user-specified OLAP queries.

The basic process functions as follows. The user defines their query in a completely object-oriented manner. From here, the query is then compiled on the client's side, while the native compiler verifies its syntax. Then, the query is parsed and converted into a second query that corresponds to our robust OLAP query grammar (discussed in Chapter 4). After that, the query is written in an XML format and sent to the backend server. There, we must be able to interpret and execute the query efficiently. Figure 5.1 illustrates the major steps that must be taken in order to resolve the OLAP queries effectively. We note at the outset that the work conducted on the client-side of the Sidera server — for example, the Java Library API and source code parsing — is being performed by another student. Our focus in this chapter, therefore, is related to the components of the OLAP query processor (i.e. the query compiler and execution) which are both found in the backend server. Moreover, in this chapter, we discuss how the aforementioned XML OLAP query is interpreted and executed on the backend server in an efficient manner. Figure 5.2 describes our OLAP query processor in terms of the basic steps that must be taken inside the backend server in order to parse, optimize and execute a query. The components of the query processor are:

1. **OLAP Query Parser:** builds a tree structure from the received XML OLAP query.

2. **OLAP Query Translator:** turns the parse tree into an OLAP expression tree composed of our OLAP algebraic operators.

3. **OLAP Query Optimizer:** transforms the OLAP expression tree of step 2 into the best physical query plan to be executed against the actual data.

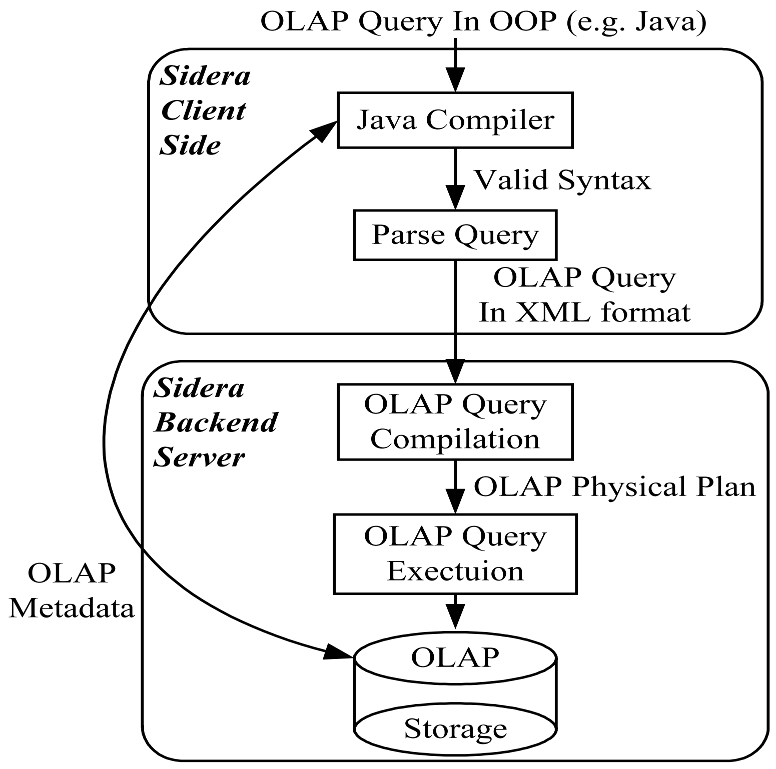


Figure 5.1: The major parts of the query processor.

4. **OLAP Query-Execution Engine:** takes a query-evaluation plan, executes the given plan and finally returns the answers to the user.

Note that the OLAP query compilation component in Figure 5.1 corresponds to the first three components of Figure 5.2 (Parser, Translator and Optimizer).

Figure 5.3 illustrates the major steps that must be taken on the backend sever in order to compile and execute the received XML OLAP queries. Essentially, the figure itself provides a direct mapping to the steps performed by the server. The following list gives the main ideas that we shall cover in this chapter.

1. **Parsing XML OLAP query:** A parse tree, representing the users OLAP query and its structure, is constructed.

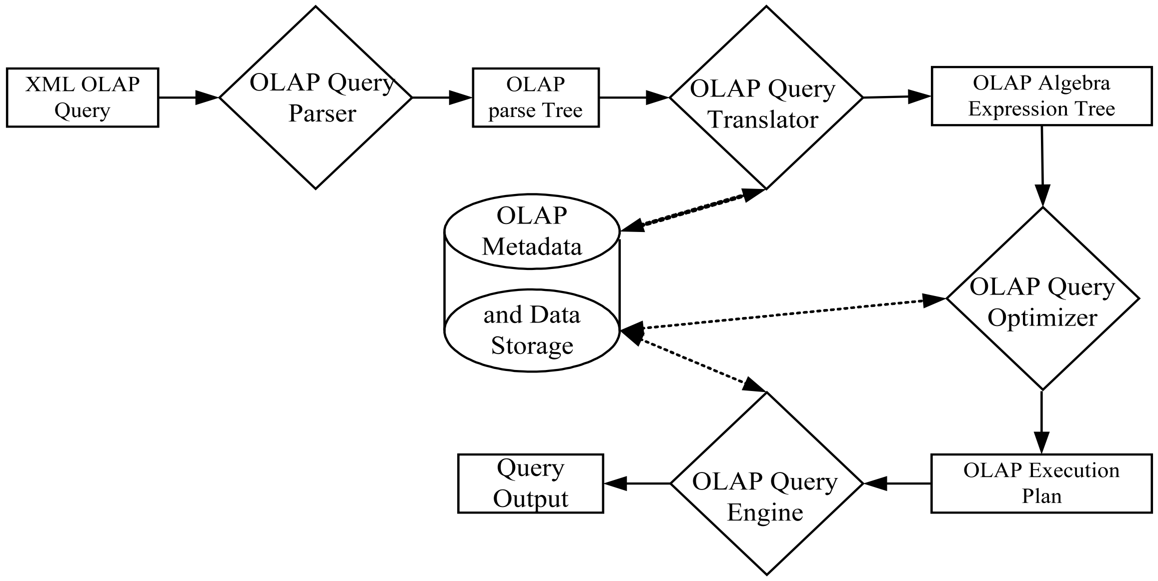


Figure 5.2: Sidera Backend Query Processor.

2. **Pre-processor:** The query is checked to make sure it is semantically valid.

3. **Logical Query Plan.** The valid parse tree is transformed into an initial logical query plan (OLAP algebra expression tree).

4. **Query rewriter:** The initial logical plan is transformed into an equivalent plan by applying OLAP algebraic laws (discussed in Section 4.7) to it. The resulting logical query plan is expected to take less time to execute.

5. **Physical plan generation:** The preferred logical query plan in step 4 is turned into a physical query plan by selecting an implementation for each OLAP algebraic operator and deciding how results are passed from one operation to another. It also includes information about how the required dimensions and cubes are accessed. In addition to this, it indicates the order in which OLAP operations may be performed.

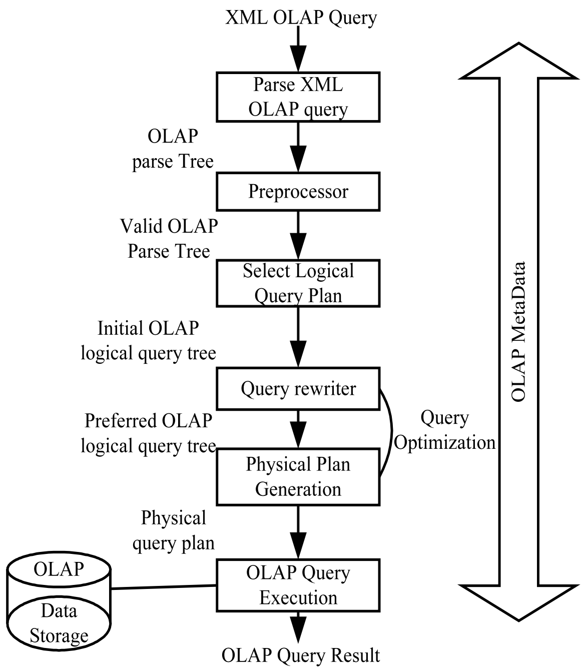


Figure 5.3: Query Compilation and Execution Steps.

6. **OLAP Query Execution:** The algorithms that retrieve the data from the Sidera OLAP data storage are executed.

7. **Result Set:** OLAP queries essentially extract a sub-cube from the original space. The result of the query will be combined into a XML based package and returned to the client side.

The chapter is organized in detailed sections. Within Section 5.2, we present related techniques in the area of OLAP query optimization and execution. Section 5.3 discusses the motivation for our work. A description on how we parse the XML OLAP query is provided in Section 5.4. In Section 5.5, we discuss the pre-processing component that checks the semantics of the received OLAP query. We will consider how the parse tree is converted into an initial logical query plan of OLAP algebra operations in Section 5.6. Then, Section 5.7 will illustrate how the various laws of Section 4.7 can be applied in order to improve the initial logical query plan. In Section 5.8, we will discuss how the preferred logical query plan is turned into a physical query plan. Section 5.9 describes the Sidera query engine, which has the responsibility for executing each step in the physical query plan. In Section 5.10, we discuss the structure of the Sidera server. In Section 5.11, we will describe the structure for the result of the OLAP query. Finally, Section 5.12 is a review of the chapter's objectives, with final conclusions provided in Section 5.13.

5.2 Related Work

A query processor (query compilation and execution) is an essential component in any database management system (DBMS). Specifically, query compilation transforms user queries into a sequence of database operations, while query execution executes those given operations. In other words, query compilation itself uses a query optimizer that can be used to transform a user query into a "query plan" that takes as little time as possible, while query execution refers to the algorithms that manipulate the data of the database. In traditional databases, the query optimizer optimizes a user query using two approaches; (1) rule-based or syntax-based enumeration and; (2) cost-based enumeration [40].

In the first approach, the query optimizer transforms the user query into an initial logical plan of relational algebra where one can apply many different relational algebraic re-writing rules with the objective of producing the optimized relational algebra expression plan. For example, since a selection reduces the size of the intermediate relation results, an important rule is to push the selection down as much as possible in a logical query plan. In the second approach, the query optimizer must turn the optimized logical relational algebra plan into a physical query plan. This is accomplished by considering many different physical plans from the preferred logical query and estimating the cost of each physical plan. Here, the traditional optimizer must maintain statistics about the database. The estimated cost of a physical plan depends on many factors such as the number and composition of records in a given table or index, the estimated size of intermediate results sizes, access methods, pipelining and materialization of intermediate results, etc. The physical plan with the least estimated cost is then passed to the query-engine execution, where it must be executed as a sequence of operations. The traditional query-engine defines the principal methods for execution of these operations (based on relational algebra operations). These methods are based on various strategies such as scanning, hashing, sorting and indexing. All of the widely used commercial database management systems (such as Oracle, Microsoft SQL, etc.) offer this form of traditional query optimization and execution for their database engines.

In the OLAP context, most of the existing research on the OLAP query processor focuses on the optimization of complex OLAP queries [9, 45, 16, 22, 17, 24, 70, 26, 46, 92, 109, 82, 100]. The existing publications in this area are divided as follows:

1. **Materialized views:** OLAP query optimization in most of the existing publications is based on the materialized views of the fact table [109, 92, 46, 26, 70, 24, 9]. Specifically, the OLAP query optimizer determines the best views to answer the current OLAP query and then re-write the query against the selected views.

2. **Index schemes** : Indexes (such as bitmap, join indexes, etc.) are used to

optimize complex OLAP queries [22, 87, 88].

3. **Parallel algorithms and Data partitioning:** Parallel processing and horizontal and vertical data partitions are also applied to optimize OLAP queries [16, 22, 100].

4. **OLAP-aware query optimization**: This technique rewrites OLAP queries using a multidimensional algebra [17, 45].

Note that the majority of the commercial database products such as Oracle and Microsoft focus on the first three techniques (1, 2, and 3) in order to optimize OLAP queries.

In [24], Chaudhuri et al. consider only a limited form of OLAP queries (Select-Project-Join SPJ queries). They enhance the traditional query optimizer used in commercial database systems by re-writing the query against the most appropriate materialized views. Their approach works as follows: a) determine the appropriate views for a given query; b) utilize these given views in order to rewrite many alternatives and; c) chose the preferred alternative with the smallest estimated cost.

Levy et al. in [70] proves that the complexity of determining the minimal rewriting query of SPJ queries is a NP-complete problem. They focus on how to substitute the selected views in order to obtain this given query. The algorithms presented in [2,4] focus only on the syntax of the query.

Srivastava et al. [109] discussed the rewriting of SPJG (Select-Project-Join-Group-by) queries into corresponding queries aided by the use of materialized views. Their algorithms were aware of both syntax and semantics when it came to rewriting queries. They did not provide an approach to selecting one rewrite amongst multiple rewrites, nor did they provide experimental results. In [9], the authors improve efficiency for query optimization processes. In short, their algorithm minimizes the number of alternative rewrites considered.

Finally, in [82], the authors improve the syntactic query rewrites by making use of metadata in order to determine the best view to answer a query. By the end of this process, the query can be rewritten by means of using the selected view and then finally applying the traditional query optimization. It is important to note that the studies in [109, 92, 46, 26, 70, 24, 9, 82] are not OLAP-aware query optimizations, as they are built and erected on top of traditional relational query optimizations.

In contrast to the above non-OLAP aware query optimization techniques, Bellatreche et al. in [17, 45] propose a pure OLAP optimization technique that rewrites OLAP queries by means of using a multidimensional OLAP algebra. Specifically, they define a multidimensional algebra that represents the core of their optimization. They provide a set of re-writing rules (similar to those on relational algebra) for each OLAP operator in their algebra. Also, they define a cost based model to measure the cost of a given plan. Optimization is achieved by re-writing the OLAP query using a set of multi-dimensional re-writing rules (similar to the rule base enumeration in relational database) in order to produce the best logical plan. In addition, their optimization algorithm takes into consideration the cubes on screen output format. That being said, their technique does not utilize materialized views, nor do the authors report any experimental results. Moreover, no algorithms were provided for OLAP operations. Finally, they did not take the physical definition of the algebraic operators into account.

The query optimization discussed in this chapter is based upon the same concepts in [17, 45]. However, algorithms for each OLAP algebraic operators in our OLAP algebra are provided. Also, a final OLAP physical plan of the optimized OLAP logical query plan is constructed. Experimental results are reported to demonstrate the efficiency of our work. To the best of our knowledge, in an OLAP environment, no pure OLAP query processor — compiler and execution — has yet to be proposed to answer native OOP OLAP queries. When we say "pure," we are implying that all server components are directly related to the OLAP environment (OLAP query optimization, OLAP execution, cubes, multi-dimensional indexes, etc.).

5.3 Motivation

In fact, the Sidera server is intended to be a pure OLAP DBMS. Having discussed the Sidera OLAP storage manager and query language, it is now necessary to present the associated OLAP query processor. Construction of an OLAP query processor is a complex task, involving not only OLAP query execution — the algorithms that manipulate the data in the OLAP data storage — but also the OLAP query compilation process that parses and optimizes a given OLAP query . In the remainder of this chapter, we present an OLAP query processor component for the Sidera server designed to achieve the following objectives:

1. Parse the received OOP OLAP query written in XML format such that an internal parse tree is created if the query is syntactically and semantically valid.
2. Provide an initial OLAP logical query plan that can be easily optimize later. Specifically, we convert the parse tree into an initial logical query plan composed of our OLAP algebraic operators.

3. Produce an improved OLAP logical query plan (i.e., one that requires less time to execute).

4. Pick the OLAP physical query plan with the smallest estimated cost. We consider many OLAP physical plans constructed from the OLAP logical plan. We propose a multidimensional cost model that allows us to evaluate and estimate the cost of each OLAP physical plan. Using our multidimensional cost model, we choose the plan with the least estimated cost and pass it to the query execution component.

5. Provide efficient algorithms for implementation of the operations of our OLAP algebra.

6. Allow for simple manipulation of the cube results. Our OLAP server exposes the result in a logical, read-only multi-dimensional array.

In the remainder of this chapter, we present the details of our new OLAP query processor that supports the efficient execution of native OOP OLAP queries. At the conclusion of this chapter, we will review this list of objectives and check the degree to which these objectives have been reached.

5.4 Parsing OLAP XML Queries

In Section 4.3, we saw an example of what the user's Java OOP OLAP query might look like. Our research actually commences once the OLAP query is translated to the XML format and sent to the Sidera server. Specifically, the received XML OLAP query must first be parsed to produce the initial tree. In this section, we discuss parsing of Object Oriented OLAP queries written in XML format. In fact, we present

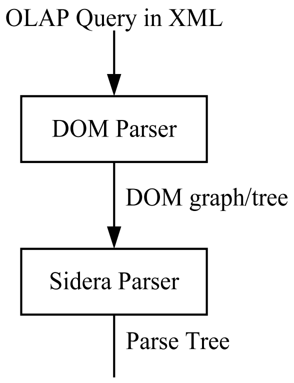


Figure 5.4: OLAP XML query to parse tree

two levels of parsing, as sketched in Figure 5.4. First, a DOM parser is used to produce the DOM tree [32]. Second, the Sidera parser is used to traverse the DOM tree and produce a parse tree.

5.4.1 DOM Parsing

The DOM parser parses the XML OLAP query into tags and verifies that it matches our newly developed OLAP query grammar. It inserts tags into a DOM graph/tree. The XML DOM module actually parses the OLAP XML query for us. It verifies that the XML query is syntactically correct and subsequently creates an internal tree representation. Because our server (Sidera) is written in C/C++, we use the Xerces-C++ XML DOM parser to create the DOM graph/tree. Xerces-C++ is a validating XML parser, written in a portable subset of C++ [112].

Recall that Figure 4.5 provides a simple Object Oriented OLAP query that can be written by the client. Also, Figure 4.21 illustrates the XML format that is received on the Sidera backend server for the OOP OLAP query in Figure 4.5. Note that this XML OLAP query corresponds to the OLAP query grammar described in Chapter 4. Specifically, one can see the projection attributes (product type, customer province and Quantity_Ordered) and the user's query restriction (i.e., Selection).

The DOM parser verifies that the syntax of the OLAP query corresponds to our DTD-encoded OLAP query grammar. If the syntax of the OLAP query does in fact match the query grammar, then a DOM graph/tree that represents the received OLAP query is created by the DOM parser. An example of the tree corresponding to Figure 4.21 is shown in Figure 5.5.

Figure 5.5 illustrates the DOM graph/tree that is produced by the DOM parser. The root node <QUERY> holds one <DATA_QUERY> node. The <DATA_QUERY> node holds two nodes, <CUBE_NAME> and <OPERATION_LIST>. The element node <CUBE_NAME> holds a text node with the value Order. The <OPERATION_LIST> node holds two nodes which are <SELECTION> and <PROJECTION>. We can see the display attributes and the measure(s) that are defined under the <PROJECTION> element. The <SELECTION> node has one <DIMENSION_LIST> that holds one <COMPOUND_DIMENSION>. The <COMPOUND_DIMENSION> has two <DIMENSION_LIST> nodes that are connected with an AND operator. A dimension restriction is specified under each <DIMENSION_LIST>. Due to the space limitation, Figure 5.5 shows only the condition on dimension Customer from the XML query in Figure 4.21. The restriction on dimension Time should be added under <DIMENSION_LIST>.

5.4.2 Sidera Parser: DOM graph to Parse Tree

After the DOM tree has been built, the DOM parser verifies that the received OLAP XML query has valid syntax corresponding to our OLAP query grammar. The Sidera parser is used to traverse the DOM graph/tree and to determine the type and the meaning of each node. In other words, it is used to convert the XML DOM graph/tree

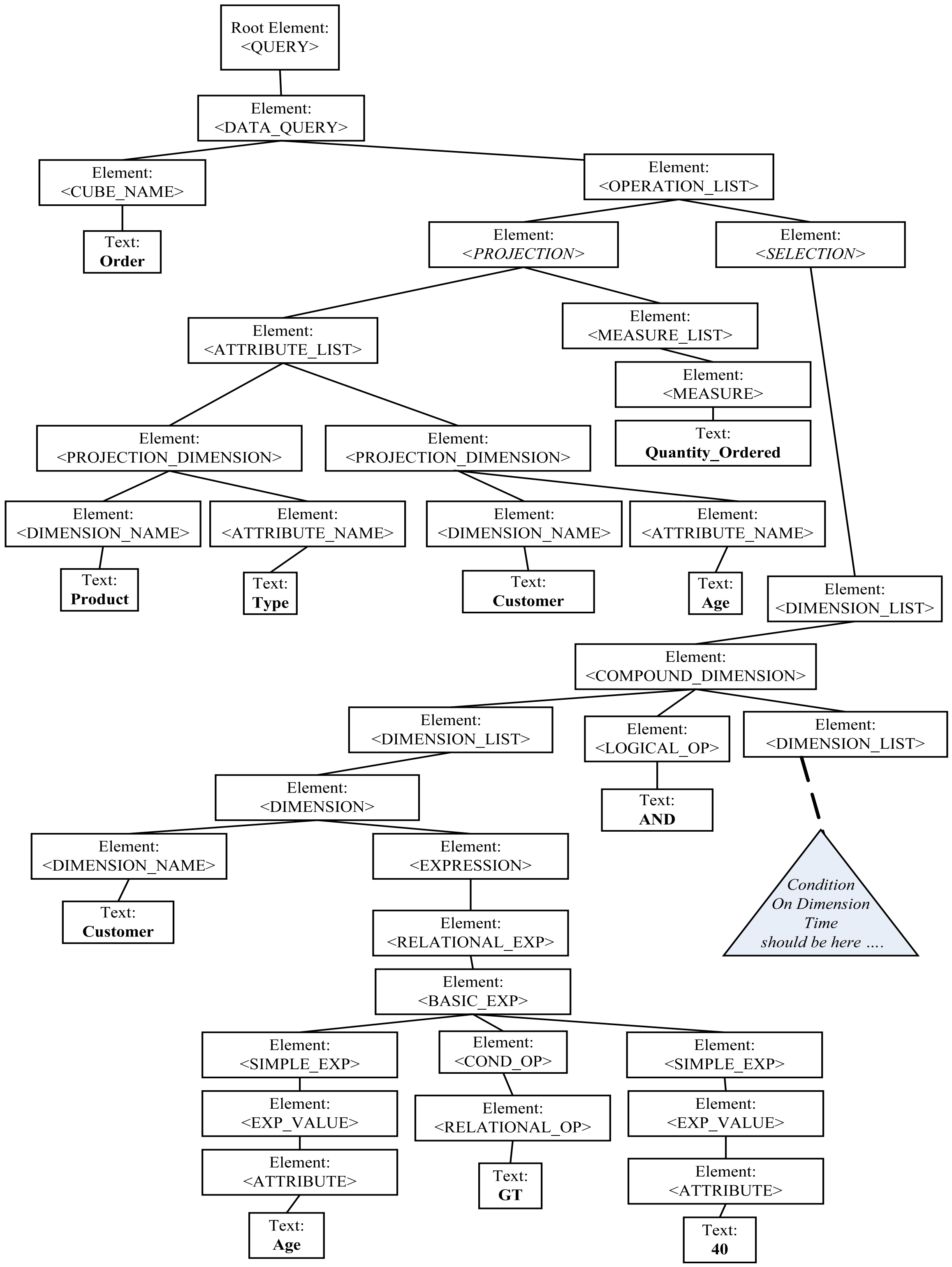


Figure 5.5: XML DOM graph/tree

to an internal parse tree representing the structure of the query. The purpose of this process is to convert the XML OLAP query into a very simple and minimized parse tree that represents the structure of the query in a compact but expressive form.

Our parser converts the DOM graph into an internal parse tree by removing all extraneous DOM nodes that are not needed to execute and optimize the query. For example, the DOM graph of Figure 5.5 is converted into the internal parse tree of Figure 5.6. Working down the tree in Figure 5.6, we can see that this parse tree is equivalent to the OLAP query represented in XML format. Specifically, it is executed against cube Order and consists of two OLAP operations (selection and projection). The projection operation consists of dimension attributes: **Product.Type**, and **Customer.Province** as well as one measure attribute: **Quantity_Ordered**. The selection operation within this parse tree consists of conditions on dimensions Customer (i.e., Age > 40) and Time (Year = 2007 AND Month IN_RANGE May October). Note that this parse tree is produced from top to bottom and essentially consists of the operation list and the details of each operation.

5.5 The Pre-processor: Semantic checking

The tasks of the parser in the previous section are to take an XML OLAP query, to convert it to a parse tree and to check if it is syntactically valid. Even if the query is syntactically valid, however, it may violate one or more semantic rules on the use of names, expressions, etc. The pre-processor is responsible for semantic checking. In short, it must check the OLAP query against the OLAP schema definition as follows:

1. **Cube name.** Every cube mentioned under the <cube_name> element must be a cube in the OLAP schema against which the query is executed. For example,

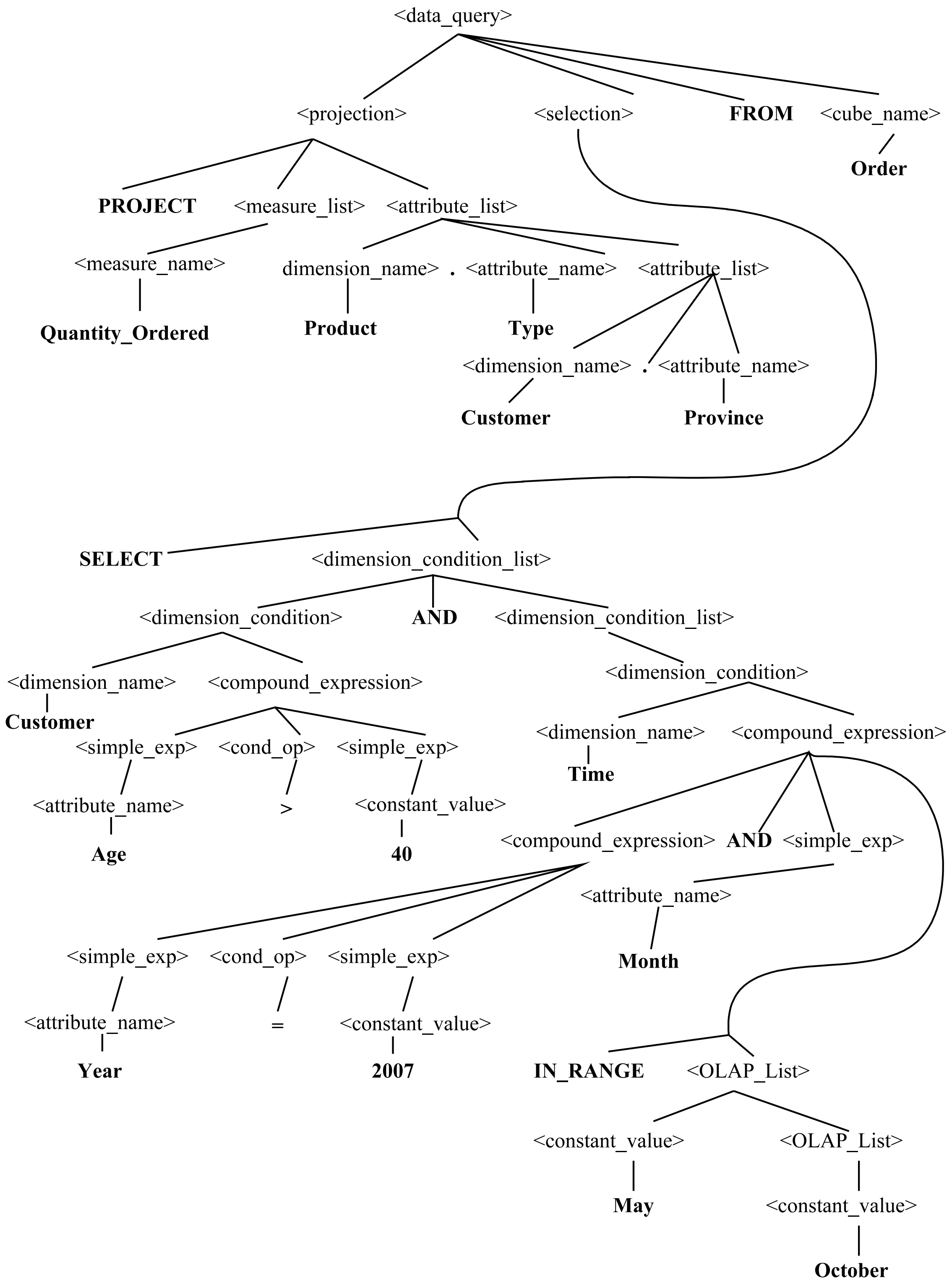


Figure 5.6: Parse Tree.

there would be a semantic error within the parse tree of Figure 5.6 if the cube name Order does not exist in the OLAP schema.

2. **Dimension uses.** Every dimension name mentioned under any OLAP operations such as selection, projection, etc. must be a dimension in the schema of the cube mentioned under the cube name. For instance in Figure 5.6, the Order cube must have Product, Customer, and Time dimensions; otherwise a semantic error would be produced.

3. **Attribute uses.** Every attribute that is mentioned under the dimension must be an attribute of that dimension. Likewise every attribute under the <measure_name> must be defined as a measure attribute in the cube specified under <cube_name>. For example, in Figure 5.6, Type must be an attribute of dimension Product in cube Order and Quantity_Ordered must be defined as a measure attribute in cube Order.

4. **Compatibility.** UNION, INTERSECT, and DIFFERENCE operations are applied between two OLAP queries. The results of two OLAP queries generated by any of the three above mentioned operations must be compatible. They must have the same number of attributes (features and measures), while each corresponding pair of attributes should have the same domain.

5. **Hierarchy uses.** Every dimension name that is used under the change level operation must have at least one hierarchy. One must also be certain that the target level is a valid level of the hierarchy.

6. **Types.** All attributes that are mentioned in the condition of the selection operation must be of a type compatible to their use. For instance, Age in

Figure 5.6 is used in the > (Greater Than) comparison. Since the attribute Age is of type integer, we must ensure that the other operand is also a numeric type.

7. **Operation uses.** Each operation can appear at most once under the <data_query> element. The projection should exist under any <data_query> because it determines the schema of the output result. The CHANGE_LEVEL and the CHANGE_BASE cannot be used within the <data_query> due to the fact that they are applied to a result set.

Semantic checking relies extensively on the schema of the OLAP environment. In section 4.8, we explained in detail how the schema of the OLAP environment is natively written in XML. If all semantic tests are passed, then the parse tree is said to be valid and is sent to the logical query plan generator.

5.6 From Parse Trees to Logical Query Plans

In Section 5.4, we constructed the parse tree for an OLAP query. In this and the following section, we will explain how to turn this tree into a more efficient logical query plan. There are two steps, as sketched in Figure 5.3. The first step is to convert the parse tree into an initial OLAP algebra expression and the second step is to optimize this initial OLAP algebra expression prior to query execution. In this section, we will explain how the user's native OOP OLAP query, that is now represented in an internal parse tree, is translated into an OLAP algebra expression. We note that the primary advantage of using an OLAP algebra is that it makes alternative forms of an OLAP query easier to explore and optimize (e.g., push operations, pull up, etc.).

We note at the outset that due to the enormous effort required to implement a complete DBMS query optimization engine, the following sections focus primarily on the theoretical framework for this mechanism within an OLAP setting. In Chapter 6 (i.e., Experimental Results) we will explicitly clarify those components of the engine that have been physically implemented in the current system.

5.6.1 Conversion to Initial OLAP Logical Query Plan

We shall now describe basic rules for transforming the OLAP parse tree outlined in the previous section into an initial OLAP logical query plan. The first point to note is that the order of operations (<selection>, <change_level>, etc.) in the parse tree under the <data_query> element plays an important role in creating the initial OLAP logical query plan. Recall that the pre-processor of Section 5.5 ensures that a valid parse tree must have under any <data_query> element one Projection and/or one or more distinct OLAP operations (still taking into consideration that change level and change base cannot be used within the same line of operation). If we have an OLAP data query that has no binary operations (such as drill_across, union, intersection or difference), then we may replace the parse tree expression by an OLAP algebra expression that consists of the following (working from the bottom upwards):

1. The bottom of the tree is the cube mentioned under the <cube_name> element.
2. One must read the OLAP operation(s) under <data_query> from right to left and push its equivalent OLAP algebra operator into the logical parse tree. Note that we read from right to left to ensure that the projection operation that determines the schema of the result is on the top of the logical query plan.

For example, if we have an OLAP data query that consists of <projection>, <selection> and <change_level> — specifically expressed in this order — then we may replace the parse tree expression by an OLAP algebra expression consisting of, from bottom to top:

1. Cube name mentioned under the <cube_name> element.
2. CHANGE_LEVEL(LI $\rightarrow$ LO), where LI and LO are two lists of feature attributes specified under the <change_level> operation.
3. SELECTION(C), where C is the query restriction defined under the <selection> element. If we consider the parse tree of Figure 5.6, then C is equivalent to [**Customer (Age > 40) AND Time (Year = 2007 AND Month IN_RANGE May, October)**].
4. PROJECTION(L), where L is the list of feature and measure attributes mentioned under the <attribute_list> and the <measure_list> elements. For example, L is (**Quantity_Ordered, Product.Type, Customer.Age**) in Figure 5.6.

If we have an OLAP query that has binary operations (e.g., union, intersect), then we have to first apply the above rules for the unary operations (selection, projection, change_level etc.). The result is considered to be the current OLAP algebraic expression tree. From here, one should read the binary operations from left to right and place them in the OLAP algebraic expression tree relative to the current OLAP algebraic expression tree. For example, if a parse tree expression has the OLAP operations <projection>, <union>(DataQuery1), <selection>, <intersection>(DataQuery2),

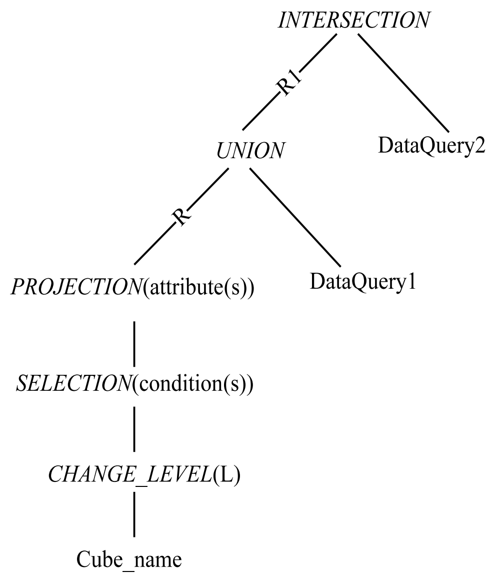


Figure 5.7: Initial OLAP Algebra.

and <change_level> (in this order) under <data_query>, then we may replace them by an initial OLAP algebra expression depicted in Figure 5.7, as follows:

1. Cube name mentioned in the <cube_name>
2. CHANGE_LEVEL(L)
3. SELECTION(condition(s))
4. PROJECTION(attribute(s))
5. The combination of the above four elements (called R) is considered as an argument to UNION. R1 = R UNION DataQuery1.
6. R1 INTERSECT DataQuery1, where R1 is the result of step 5.

Let us consider the OLAP parse tree of Figure 5.6. The bottom of the tree is the cube **Order**. We begin by taking the selection of the condition in the sub-tree rooted

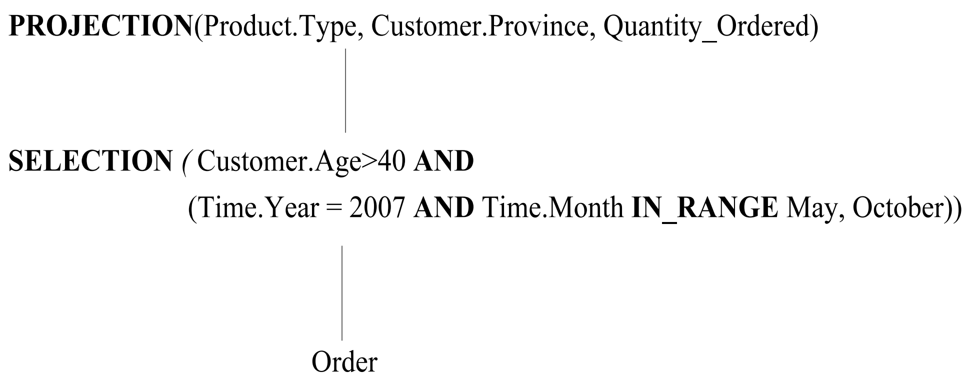

PROJECTION(Product.Type, Customer.Province, Quantity_Ordered)
SELECTION *(* Customer.Age>40 **AND**
(Time.Year = 2007 **AND** Time.Month **IN_RANGE** May, October))
Order

Figure 5.8: Translation of the parse tree of Figure 5.6 to an initial OLAP algebra tree

at <selection>, and projecting it into the <projection>, **product type, customer province** and **Quantity ordered**. The resulting algebraic representation is found in Figure 5.8. It is important to recognize that while an OLAP data query physically requires explicit joins between group-by (measure) and dimension tables in order to exclude cube rows for the cube content, there is no such requirement at this logical level.

5.7 Improving the OLAP Logical Query Plan

In Section 5.6, we explained how to convert our OLAP query into an initial OLAP algebra logical query plan. The next step, as sketched in Figure 5.3, is to rewrite the initial logical plan using the OLAP algebraic laws outlined in Section 4.7. In this section, our intent is to apply some of these algebraic laws to produce a better logical query plan. We note that in Chapter 6 we will provide experimental results that demonstrate the value obtained by enhancing the initial plan.

The following are the algebraic laws most commonly used in our OLAP query optimizer to improve OLAP logical query plans:

- As was discussed in Section 4.7, LAW_4 (Pushing law) states that a selection should be pushed down the expression tree as far as it can go. However, we saw in Section 4.7 (LAW_2) that in some situations it is necessary to first pull the selection up and then down. If we have two or more consecutive selections, we may then replace them by only one selection with the conjunction AND linking both conditions together (LAW_2). LAW_2 and LAW_4 are the most important rules for efficient query processing since they tend to reduce the size of intermediate cube results.

- A selection result depends on inner/natural joins between the cube and dimension tables in order to exclude cube rows that don't satisfy the query restriction. We use LAW_1 to eliminate the inner joins between cubes and dimension tables. This strategy significantly reduces cube processing time.

- Similarly, PROJECTION must be pushed down the tree in the case of CHANGE_LEVEL or CHANGE_BASE (LAW_6).

- We apply LAW_8 if we have a CHANGE_BASE(x)(PROJECTION(L) C). This rule allows us to reduce the number of OLAP operations in the OLAP expression tree.

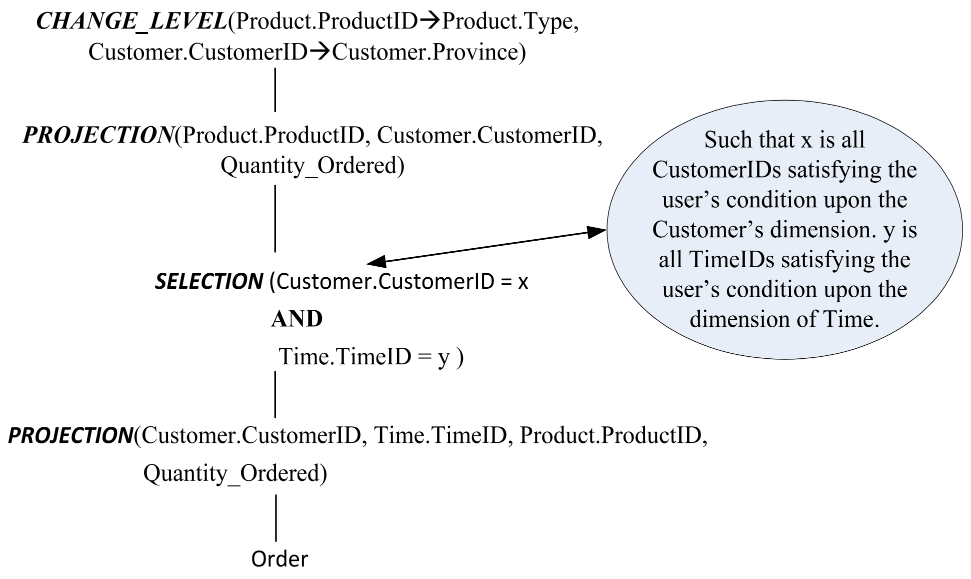
- We introduce a new PROJECTION if a PROJECTION follows a SELECTION (LAW_5). Introducing a new PROJECTION reduces the size of cubes because it eliminates dimension members from the cube and aggregates the resulting cube if necessary.

- We use LAW_7 to defer joins between cube(s) and dimension tables. We project onto the feature attributes that are stored in the cube instead of accessing dimension tables to project onto the required dimension attributes.

- We can replace several consecutive CHANGE_LEVEL operators by one CHANGE_LEVEL (LAW_11).
- We apply either LAW_9 (pulling up the CHANGE_LEVEL) or LAW_10 (pushing down the CHANGE_LEVEL) in order to reduce the size of intermediate results.
- Finally, we apply LAW_13 (trivial laws) if necessary.

Let us consider the query of Figure 5.8. The effect of applying our algebraic transformation laws is shown in Figure 5.9. The laws are applied as follows:

- We use LAW_1 to eliminate the inner join between **Order** and dimension tables (**Customer** and **Time**). The resulting selection operation can be written as: SELECTION(Customer.CustomerID = x AND Time.TimeID = y) where x is all CustomerIDs that have (Age > 40) and y is all TimeIDs in 2007 (Year) and between May and October (Month).
- We introduce a new PROJECTION below the SELECTION (LAW_5). The new projection is defined as: PROJECTION(**Customer.CustomerID, Time.TimeID, Product.ProductID, Quantity_Ordered**).
- Finally, we use LAW_7 to split PROJECTION(**Product.Type, Customer.Province, Quantity_Ordered**) into CHANGE_LEVEL(**Product.ProductID→ Product.Type, Customer.CustomerID⇒ Customer.Province**) and PROJECTION(**Product.ProductID, Customer.CustomerID, Quantity_Ordered**).



CHANGE_LEVEL(Product.ProductID→Product.Type, Customer.CustomerID→Customer.Province)
PROJECTION(Product.ProductID, Customer.CustomerID, Quantity_Ordered)
SELECTION (Customer.CustomerID = x AND Time.TimeID = y)
PROJECTION(Customer.CustomerID, Time.TimeID, Product.ProductID, Quantity_Ordered)
Order
Such that x is all CustomerIDs satisfying the user's condition upon the Customer's dimension. y is all TimeIDs satisfying the user's condition upon the dimension of Time.

Figure 5.9: The effect of query rewriting.

The above query consists of only PROJECTION and SELECTION. However, the same mechanism can, of course, be applied to more complex queries. As a concrete example we consider the OLAP query of Figure 5.11 against the cube (Sale) that is depicted in Figure 5.10. The cube Sale is a four dimensional cube with two measure attributes (Quantity_Ordered and Units_Sold) and four feature attributes. The data in the Sale cube stores the most specific details of each given dimension (e.g., ProductID, TimeID). The hierarchy in each dimension is indicated in bold. For example, Product has a hierarchy: ProductID (Base level) → ProductNumber → Type → Category(Root level). Moreover, the primary keys are underlined. The query in Figure 5.11 can be translated to English as (i) the total units sold grouped by the product category, store city, month, and year in year 2007, (ii) intersect the result in (i) with the total units sold in store located in the Monteral city, (iii) perform the union of (ii) with the total units sold in January and for all customers who are 40

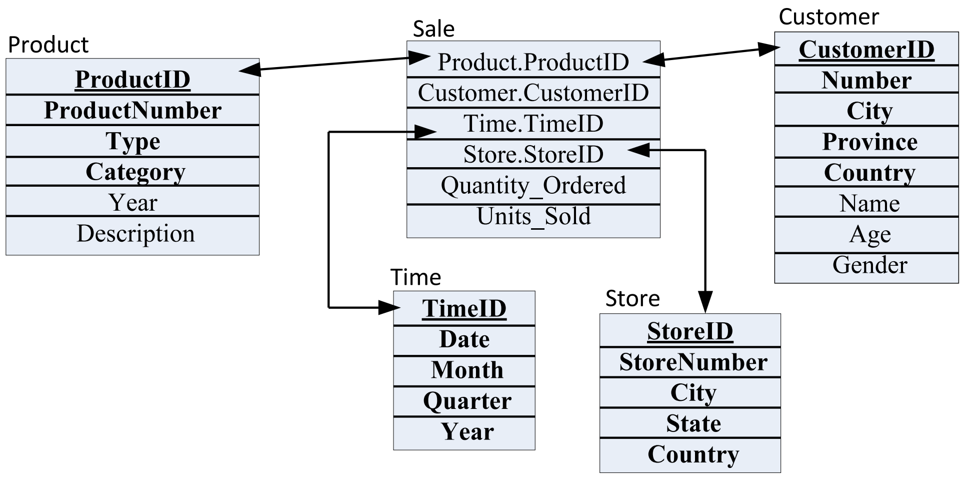
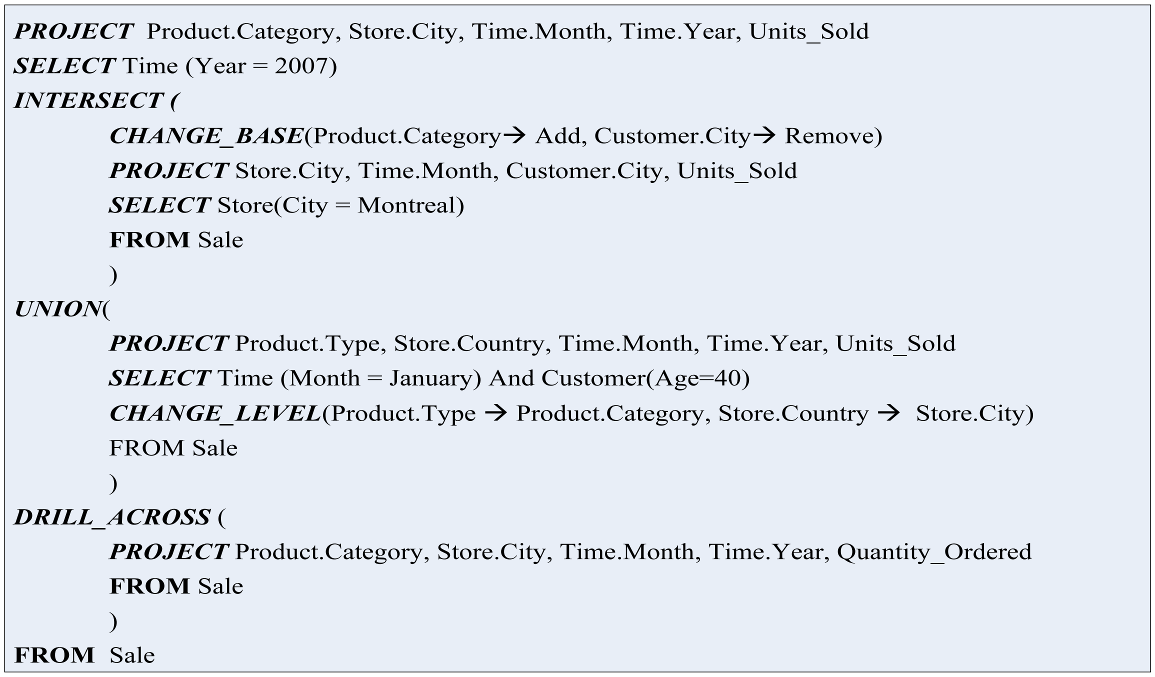


Figure 5.10: Cube Sale Schema.

years old. Finally, return the drill across of the total units sold in (iii) against the total Quantity ordered grouped by product category, store city, month, and year. In Figure 5.11, the query is represented in a simple string format for illustrative purpose. We turn the query of Figure 5.11 into an initial logical query plan by applying the rules that were discussed in Section 5.6. The resulting OLAP initial logical algebra is shown in Figure 5.12.

Now consider the query plan in Figure 5.12. Figure 5.13 illustrates the effect of our OLAP logical query optimizer. Laws are applied as follows:

- We use LAW_4 to push the SELECTION below the CHANGE_LEVEL operation.
- We pull up the two selections under the INTERSECTION operation.
- We use LAW_2 to combine the two SELECTIONS into one SELECTION with the conjunction of the arguments (Time.Year = 2007 AND Store.City = Montreal).



```
PROJECT  Product.Category, Store.City, Time.Month, Time.Year, Units_Sold
SELECT Time (Year = 2007)
INTERSECT (
          CHANGE_BASE(Product.Category→ Add, Customer.City→ Remove)
          PROJECT Store.City, Time.Month, Customer.City, Units_Sold
          SELECT Store(City = Montreal)
          FROM Sale
          )
UNION(
          PROJECT Product.Type, Store.Country, Time.Month, Time.Year, Units_Sold
          SELECT Time (Month = January) And Customer(Age=40)
          CHANGE_LEVEL(Product.Type → Product.Category, Store.Country →  Store.City)
          FROM Sale
          )
DRILL_ACROSS (
          PROJECT Product.Category, Store.City, Time.Month, Time.Year, Quantity_Ordered
          FROM Sale
          )
FROM  Sale
```

Figure 5.11: Complex OLAP query in a simple string form.

- We use LAW_4 to push the SELECTION down the tree because all attributes that are mentioned within the SELECTIONs are in the output of the INTERSECTION (Time.Year and Store.City).
- We use LAW_1 in order to eliminate the join between the Sale cube and the dimension tables (Customer, Product, etc.).
- We use LAW_6 to push the PROJECTION down the tree.
- We use LAW_8 to combine the CHANGE_BASE and the PROJECTION.
- We introduce a new PROJECTION by using LAW_5.
- LAW_7 is used to split and replace the PROJECTION into PROJECTION and CHANGE_LEVEL.

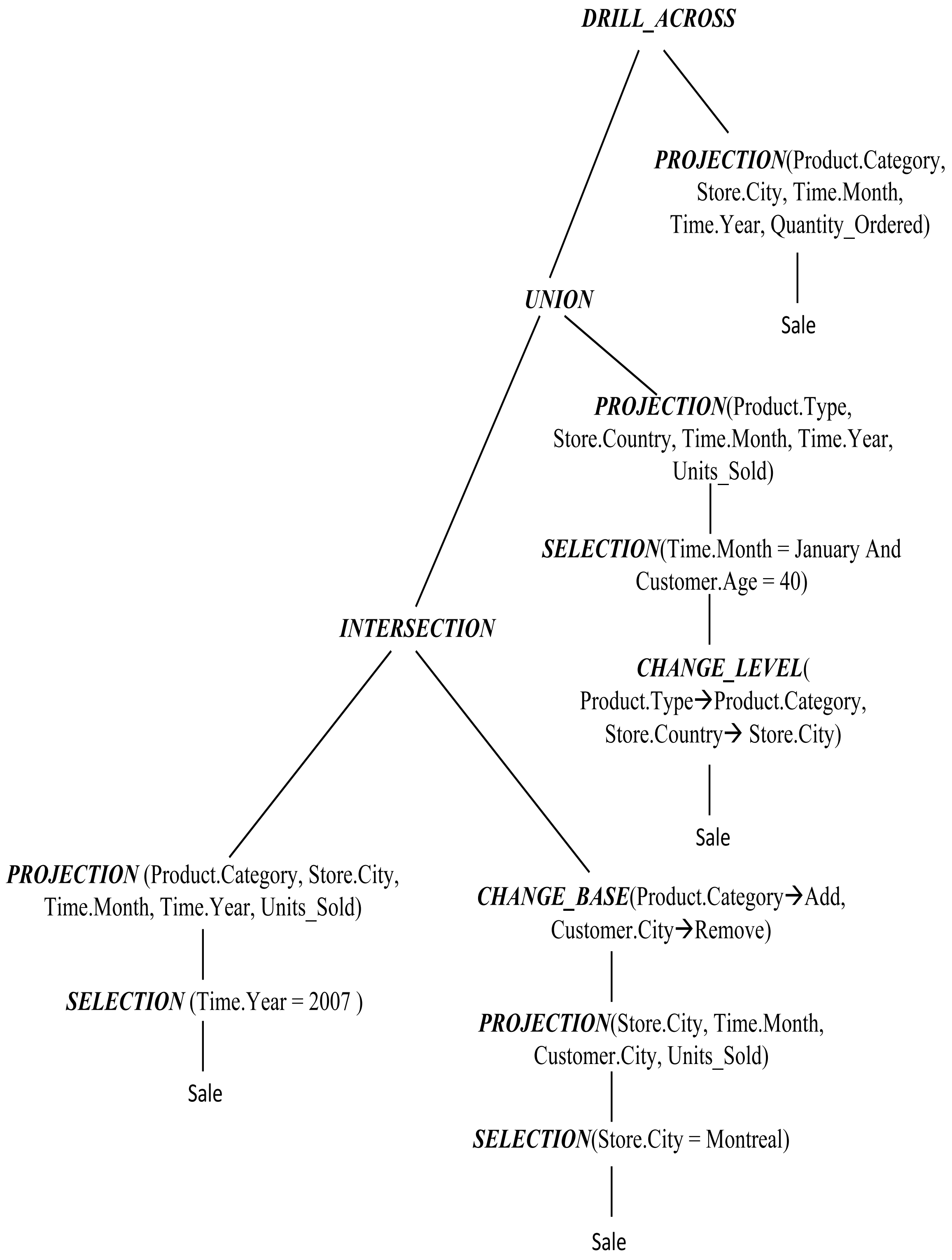

DRILL_ACROSS
PROJECTION(Product.Category, Store.City, Time.Month, Time.Year, Quantity_Ordered)
Sale
UNION
PROJECTION(Product.Type, Store.Country, Time.Month, Time.Year, Units_Sold)
SELECTION(Time.Month = January And Customer.Age = 40)
CHANGE_LEVEL(Product.Type→Product.Category, Store.Country→ Store.City)
Sale
INTERSECTION
PROJECTION (Product.Category, Store.City, Time.Month, Time.Year, Units_Sold)
SELECTION (Time.Year = 2007)
Sale
CHANGE_BASE(Product.Category→Add, Customer.City→Remove)
PROJECTION(Store.City, Time.Month, Customer.City, Units_Sold)
SELECTION(Store.City = Montreal)
Sale

Figure 5.12: Initial OLAP Logical Query Plan of Complex Query in Figure 5.11.

- we replace many CHANGE_LEVELs into only one CHANGE_LEVEL (LAW_11).
- Because the CHANGE_LEVEL involves only the Roll Up operation, then in this example we use LAW_10 to push this operation down the tree.
- Finally, we use the trivial law (LAW_13) between the INTERSECTION of two equivalent result cubes.

5.8 OLAP Physical Query Plan Generation

After the preferred OLAP logical query plan has been constructed, we must next transform it into a physical plan. The process is as follows: (i) we transform the logical query plan into several physical plans and (ii) estimate the cost of each. The physical plan with the least estimated cost is selected and passed to the query engine to be executed. We note that, in practice, we need not physically materialize each of the alternative plans. Rather we at each stage of the process, we choose from a number of possible alternatives.

The cost of each physical plan is affected by the following factors:

1. Sizes of intermediate cube results.
2. Order and grouping of commutative and associative OLAP operations: UNION, INTERSECTION, and DRILL_ACROSS.
3. An implementation for each OLAP algebraic operator in the preferred OLAP logical plan.
4. How the required dimensions and cubes are accessed, for example, whether there is a scan access or an index access.

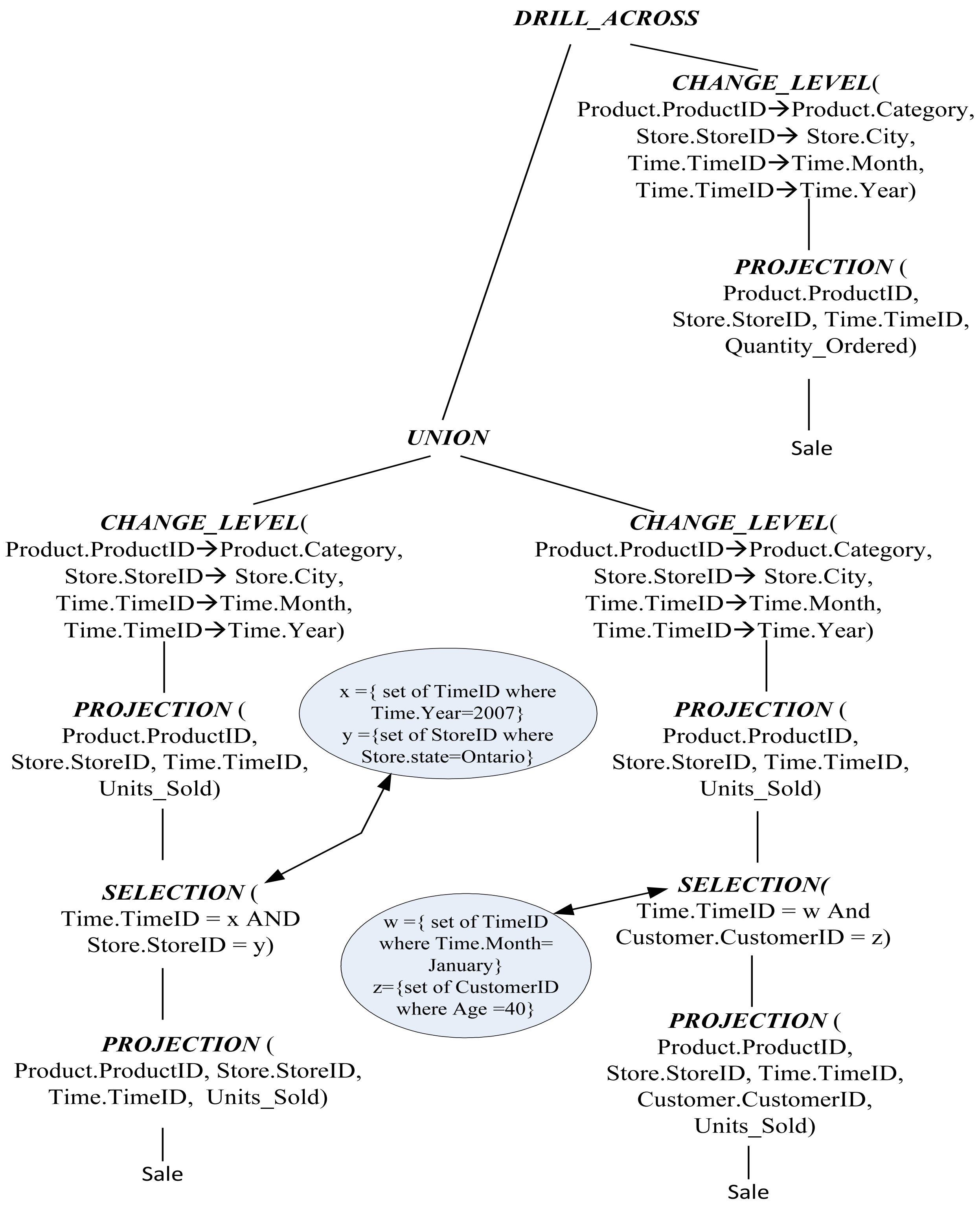

DRILL_ACROSS
CHANGE_LEVEL(Product.ProductID→Product.Category, Store.StoreID→ Store.City, Time.TimeID→Time.Month, Time.TimeID→Time.Year)
PROJECTION (Product.ProductID, Store.StoreID, Time.TimeID, Quantity_Ordered)
Sale
UNION
CHANGE_LEVEL(Product.ProductID→Product.Category, Store.StoreID→ Store.City, Time.TimeID→Time.Month, Time.TimeID→Time.Year)
PROJECTION (Product.ProductID, Store.StoreID, Time.TimeID, Units_Sold)
x ={ set of TimeID where Time.Year=2007}
y ={set of StoreID where Store.state=Ontario}
SELECTION (Time.TimeID = x AND Store.StoreID = y)
PROJECTION (Product.ProductID, Store.StoreID, Time.TimeID, Units_Sold)
Sale
CHANGE_LEVEL(Product.ProductID→Product.Category, Store.StoreID→ Store.City, Time.TimeID→Time.Month, Time.TimeID→Time.Year)
PROJECTION (Product.ProductID, Store.StoreID, Time.TimeID, Units_Sold)
SELECTION(Time.TimeID = w And Customer.CustomerID = z)
w ={ set of TimeID where Time.Month= January}
z={set of CustomerID where Age =40}
PROJECTION (Product.ProductID, Store.StoreID, Time.TimeID, Customer.CustomerID, Units_Sold)
Sale

Figure 5.13: A preferred OLAP logical query plan showing the effects of our OLAP query optimizer on Figure 5.12

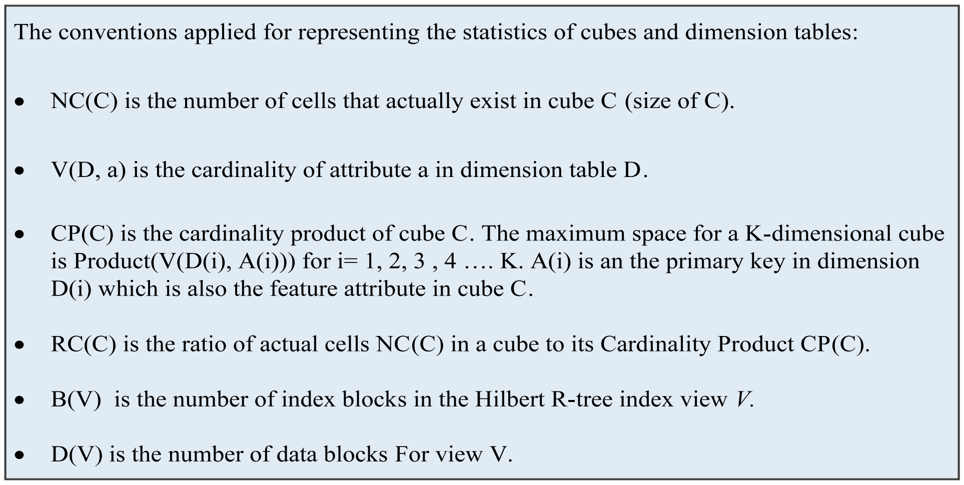
5. The introduction of physical operators (e.g., sorting, mapping) that are not explicitly defined in the logical OLAP query plan.

6. How results are passed from one operation to another.

The estimations of the physical plans are based on data parameters — see Figure 5.14 — in the cubes and dimension tables. As previously alluded to in Chapter 4, our OLAP algebra is implicitly read only. User-defined modifications cannot be directly conducted within the given database as database updates can only be performed via distinct ETL processes. Therefore, these parameters are precisely computed from the data itself. The goal of the query optimizer (Query rewriter and Physical plan generation) is to minimize the response time for a given query. The above mentioned points are used by the query optimizer to heuristically construct a good physical plan from the preferred logical query plan (discussed in the previous section). However, as it is the case with any heuristic approach, it is possible that the cost model does not result in the absolute best query plan. In fact, this is no different than the case with relational DBMS optimizers. In this section, we utilize the points mentioned above in order to choose one physical plan to be executed by our query engine. In practice, there is relatively little difference between the various *"good"* plans. The goal then is to avoid the obviously *"bad"* plans.

5.8.1 Estimating Sizes of intermediate Cubes

The cost of the physical plan is influenced by the size of the intermediate cubes in the query plan. We present a number of simple rules to estimate the number of cells that exist in an intermediate cube result. Ultimately, the size of each operation is the estimated number of cells in its output cube. Note that the size estimation


The conventions applied for representing the statistics of cubes and dimension tables:

- NC(C) is the number of cells that actually exist in cube C (size of C).
- V(D, a) is the cardinality of attribute a in dimension table D.
- CP(C) is the cardinality product of cube C. The maximum space for a K-dimensional cube is Product(V(D(i), A(i))) for i= 1, 2, 3 , 4 …. K. A(i) is an the primary key in dimension D(i) which is also the feature attribute in cube C.
- RC(C) is the ratio of actual cells NC(C) in a cube to its Cardinality Product CP(C).
- B(V) is the number of index blocks in the Hilbert R-tree index view *V*.
- D(V) is the number of data blocks For view V.

Figure 5.14: Notation for the size of cubes and dimension tables.

is used to help select a physical plan and not to return the exact plan size. It is important to mention here that this is similar to what is done when optimizing relational queries. In our case, however, we are optimizing multi-dimensional OLAP queries. As noted, the preferred OLAP logical query plan discussed in Section 5.7 does not have a CHANGE_BASE operator as it is replaced with other OLAP operators (LAW_8). Thus, we do not need to give an estimation of the result of this operation.

5.8.1.1 Estimating the Size of a PROJECTION

The projection is the identification of presentation attributes, including both the measure attribute(s) and dimension members. Since the aggregated values in the result of the PROJECTION must be re-calculated as required, we must estimate the number of cells in the output. In the extreme case, the size (number of cells) of **outputCube** = PROJECTION(Dim(s).Attribute(s), Measure(s)) C could be 1 if all cells are duplicated (the output cells have the same dimension member values) or as large as the size of cube C (if no duplicate in dimension members' values exists). Another way to get the maximum number of cells that could exist in the result of PROJECTION is the CP(outputCube) (i.e., the cardinality product of the result cube). That number could be greater than the number of cells in the input cube C. Consequently, we estimate the number of cells in outputCube by taking the smaller between (i) NC(outputCube) = NC(C)/2, and (ii) NC(outputCube) = RC(C) * CP(outputCube). Recall that NC is the number of cells in the cube, RC is the ratio of actual cell in the cube and CP is the cardinality product in the cube.

To illustrate the extreme cases of the PROJECTION operation, consider a three dimensional cube (CUBE) with three feature attributes (A,B,C) and only one measure attribute (M). Suppose that CUBE(A,B,C,M) has three cells which are: (5,2,3,200), (5,2,8,150), (5,2,9,100), then the result of (PROJECTION(A,B,M)CUBE) is only one cell which is (5,2,450). However the result of PROJECTION(A,C,M) is three cells which are (5,3,200), (5,8,150),(5,9,100).

Suppose that Sale(Product.Type, Time.Month, Location.Province, Store.Name , Units_Sold) is a four dimensional cube. Let V(Product, Type) = 5, V(Time, Month)= 6, V(Location, Province)= 4, and V(Store, Name) = 8. Also, let NC(Sale) = 384 cells. Then RC(Sale) = NC(Sale) /CP(Sale) = 384/5*6*4*8 = 0.4 (or 40%). Assume that we want to estimate the result of outputSale, such that

outputSale = PROJECTION(Product.Type, Time.Month,Units_Sold)Sale,

then we take the smaller of the following two estimates:

- NC(outputSale) = NC(Sale)/2 = 384/2= 192 cells
- NC(outputSale) = RC(Sale) * CP(outputSale) = 0.4*5*6 = 12 cells

Our best estimate for the number of cells in outputSale is 12.

5.8.1.2 Estimating the Size of a SELECTION

When we have a selection operation, then the number of cells in the result is reduced. As was illustrated in the previous section, SELECTION in the logical query plan involves only an equality comparison between a feature attribute(s) and a set of constant(s) (e.g., Product.ProductID = x, where x is all ProductIDs satisfying the user's restriction on dimension Product). For illustrative purposes, we consider the kind of a SELECTION where one feature attribute equals a set of values. Let selectionCube = SELECTION(D.F = x) C, where D.F is a feature attribute of cube C and x is a set of constants. If a value (v) of feature attribute (D.F) equals any value in x then the condition is satisfied and v will be in the result of the query. |x| is the number of constants in a set x. The logic behind the SELECTION estimate is: V(D,F) values of attribute F in dimension D, there are NC(C) cells. Here, with |x| values for attribute F in dimension D, there are NC(C)*|x|/V(D,F) cells. Then our estimate of NC(selectionCube) is: NC(selectionCube) = NC(C) * |x|/ V(D, F).

Several equality comparisons in the SELECTION operation are connected via logical operators (AND or OR). If several conditions in the selection are connected via AND(s), then we can treat them as a cascade of simple selection conditions. Let selectionCube = SELECTION(D1.F1 = x AND D2.F2 = y)C, then our estimate of the number of cells in selectionCube is: NC(selectionCube) = NC(C) * |x|/V(D1,F1) * |y|/V(D2, F2). When the SELECTION operator involves an OR, let selectionCube = SELECTION(D1.F1 = x OR D2.F2 = y) C. Here, we can simply estimate the size of the result as the sum of cells that satisfy D1.F1=x and those that satisfy D2.F2=y. That sum could be greater than the number of cells in cube C. If so, then we take the number of cells in cube C as an estimation of NC(selectionCube).

Consider the four dimensional cube (Sale) mentioned in the previous sub-section with all parameters. Assume that selectionCube = SELECTION(Location.Province= x AND Time.Month =y) Sale, where x is a set of constants with 3 different values(e.g., Quebec, Ontario, and Alberta), and y is a set of constants with 4 different values (e.g., March, April, July, and October). Our best estimate of NC(selectionCube) is:

NC(selectionCube) = NC(Sale) * |x|/V(Location, Province) * |y|/V(Time, Month)
= 384 * 3/4 * 4/6 = 192 cells.

5.8.1.3 Estimating the Size of UNION, INTERSECTION, DIFFERENCE

We have developed reasonable estimating techniques for the PROJECTION and the SELECTION operations. In this sub-section, we shall give some techniques for the estimation of the set operations: (UNION, INTERSECTION, and DIFFERENCE). These operations are called binary operations because they need two arguments that are themselves cubes.

UNION The number of cells in the result of UNION can be as large as the number of cells in both cubes or as small as the larger number of cells of the two cubes. One reasonable approach would to choose the average of the larger cube plus the average of both. Suppose we have two cubes C1 and C2, where NC(C1) > NC(C2), then the number of cells of resultUnion =C1 UNION C2 is:

NC(resultUnion) = (NC(C1)+NC(C2))/2 + NC(C1)/2 = NC(C1) + NC(C2)/2

INTERSECTION At the extremes, the result of INTERSECTION can have zero cells or can possess the number of cells in the smaller of the two arguments of INTERSECTION. We suggest taking the average of the extremes, which is half of the smaller. Consider two cubes, C1 and C2, the number of cells in resultIntersection = C1 INTERSECTION C2 is:

$$\text{NC(resultIntersection)} = \text{NC(C2)}/2$$

DIFFERENCE If no cells of the first argument appear in the second argument, then the number of cells of the result equals the number of cells of the first argument. If all cells of the first argument appear in the second argument, then the number of cells is equal to the number of cells of the first argument minus the number of cells of the second. So for resultDifference = C1 DIFFERENCE C2, the number of cells of the resultDifference is between NC(C1) and NC(C1) - NC(C2). Therefore, we suggest as an estimate the following:

$$\text{NC(resultDifference)} = \text{NC(C1)}/2 + (\text{NC(C1)} - \text{NC(C2)})/2 = \text{NC(C1)} - \text{NC(C2)}/2.$$

5.8.1.4 Estimating the Size of a CHANGE_LEVEL

As noted, we typically refer to this operation as the "roll-up" and the "drill down" analytical technique. Cells of the CHANGE_LEVEL operation must be re-calculated accordingly. Again, we don't know the number of cells of the result, so we must produce an estimate. In the extremes, the number of cells of resultChangeLevel = CHANGE_LEVEL(D.a1 $\rightarrow$ D.a2)C could be the same as the number of cells of C (non-duplicate cells in the result) or as small as 1 (all cells in the result are the same). Another upper limit on the number of cells of resultChangeLevel that could exist is CP(resultChangeLevel), which could be smaller than the number of cells in C (NC(C)). We have to consider three possible estimates:

1. Roll-up: Take the smaller of NC(C)/2 or CP(resultChangeLevel)*RC(C).

2. Drill-Down: Take the larger of NC(C)/2 or CP(resultChangeLevel)*RC(C).

3. Roll-up and Drill-Down: Take the average of the two numbers (i)NC(C) and (ii)CP(resultChangeLevel)*RC(C). The result equals (NC(C)/2 + (CP (resultChangeLevel) *RC(C))) / 2.

Consider the Sale cube discussed in sub-section 5.8.1.1, and assume that we have the following CHANGE_LEVEL operation:

$$\text{resultChangeLevel} = \text{CHANGE\_LEVEL}(\text{Product.Type} \rightarrow \text{Product.Category}, \text{Time.Month} \rightarrow \text{Time.Day})\text{Sale}.$$

Let V(Product,Category) = 2, and V(Time,Day)=16. CHANGE_LEVEL in this example consists of an OLAP roll-up operation from Product.Type to Product.Category and an OLAP drill-down operation from Time.Month to Time.Day. Our estimate of NC(resultChangeLevel) is the average of the following two numbers because the CHANGE_LEVEL involves Roll-up and Drill-down operations:

1. NC(C)/2= 384/2 = 192 cells

2. CP(resultChangeLevel) * RC(C) = 2 * 16 * 4 * 8 * 0.4 = 164 cells.

Our best estimate of NC(resultChangeLevel) is (192+164)/2 = 178 cells.

5.8.1.5 Reducing the Cost of the Logical Query Plan

We have already discussed in Section 5.7, how several OLAP algebraic laws (i.e., LAW_4 pushing SELECTION) can be applied to improve the cost of an OLAP logical query plan, independent of the cost estimation discussed in this section. When the preferred logical query plan is being generated, it may be possible to apply certain

Product: V(Product, ProductID) = 50, V(Product, Type) = 40
Time: V(Time, TimeID) = 60, V(Time, Month) = 40
Customer: V(Customer, CustomerID) = 5
Store: V(Store, StoreID) = 12, V(Store, City) = 4, and x is a set of StoreIDs where City equals Montreal. We Assume that |x|=2, where |x| is the number of distinct StoreIDs where City equals Montreal.
Sale: NC(Sale) = 36000 cells. RC(Sale) = 36,000 / 180,000= 0.2 (20%)

Figure 5.15: Cube Sale statistics.

transformations and measure the cost (of intermediate results) before and after. At this step, the cost of the logical query plan is the sum of all intermediate estimated cube result sizes (number of cells).

An example will illustrate the process. We use the same data cubes and dimensions of sub-section 5.8.1.1. Let the statistics for the cube Sale and dimensions Product, Customer, Time, and Store be as outlined in Figure 5.15. Consider the initial OLAP logical query plan of Figure 5.16. To generate the preferred logical query plan, we apply the rules mentioned in Section 5.7 to the initial logical query plan. However, we are not sure whether using LAW_9 or LAW_10 reduces the total cost (in terms of intermediate results sizes) of the logical plan. So we transform the initial OLAP logical query plan into the two logical query plans shown in Figure 5.17; they produce the same result but they differ in whether we pull or push the CHANGE_LEVEL operator.

We already explained how to estimate the size of the results of the SELECTION, PROJECTION, DIFFERENCE, etc. in the previous sections. We compare the two plans from Figure 5.17 by adding the estimated sizes for all the intermediate nodes (cube results). For plan 5.17(a), the estimated cost is 7200 + 1200 + 600 + 600 + 300 = 9900, while for plan 5.17(b) the estimated cost is 7200 + 1200 + 600 + 600

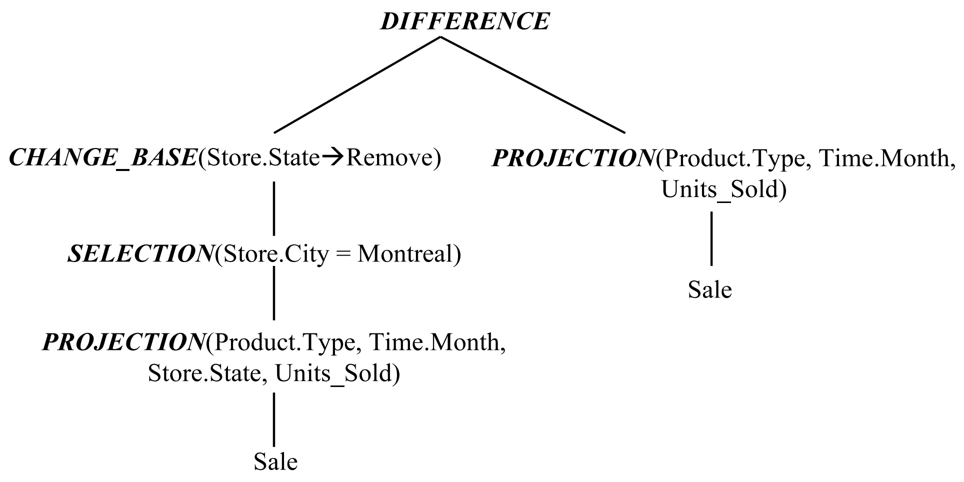
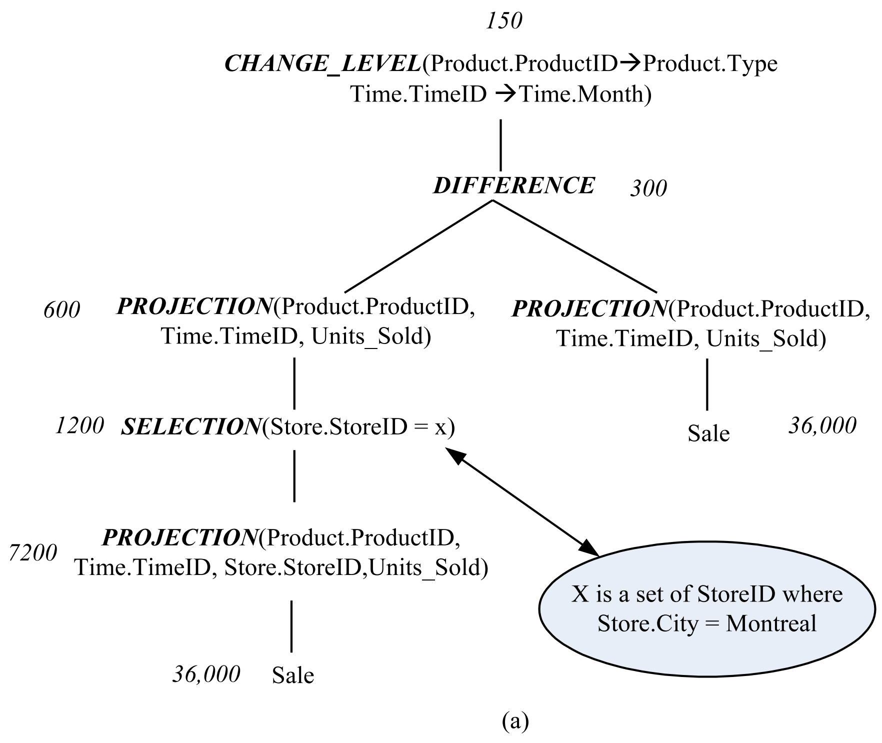

DIFFERENCE
CHANGE_BASE(Store.State→Remove)
SELECTION(Store.City = Montreal)
PROJECTION(Product.Type, Time.Month, Store.State, Units_Sold)
Sale
PROJECTION(Product.Type, Time.Month, Units_Sold)
Sale

Figure 5.16: Logical query plan for an OLAP query defined by our OLAP grammar.

+ 300 + 300 = 10200. Thus, we conclude that pulling the CHANGE_LEVEL is a better plan in this case. However, we would come to the opposite conclusion if the result of (V(Time, Month) * V(Product, Type) * RC(Sale)) is less than 150.

5.8.2 Choosing an order for binary operators

In this section, we consider a very important issue in our OLAP cost-based physical plan selection: ordering and grouping of similar binary operations on three or more cubes (such as DRILL_ACROSS, UNION and INTERSETCION). We discuss in detail how to determine an efficient processing order for evaluating the UNION of more than two cubes and also give examples showing how the good choice of the UNION order is important in terms of costs. Similar arguments can be applied to other associative and commutative binary operations in our OLAP algebra such as DRILL_ACROSS and INTERSECTION. For simplicity, we will assume that the left argument of the binary operator will be the one that possesses the least number of cells. The UNION's associativity and commutativity gives us many equivalent evaluation plans. Note that


150
CHANGE_LEVEL(Product.ProductID→Product.Type Time.TimeID →Time.Month)
DIFFERENCE
300
600
PROJECTION(Product.ProductID, Time.TimeID, Units_Sold)
PROJECTION(Product.ProductID, Time.TimeID, Units_Sold)
600
1200
SELECTION(Store.StoreID = x)
Sale
36,000
7200
PROJECTION(Product.ProductID, Time.TimeID, Store.StoreID,Units_Sold)
X is a set of StoreID where Store.City = Montreal
36,000
Sale
(a)

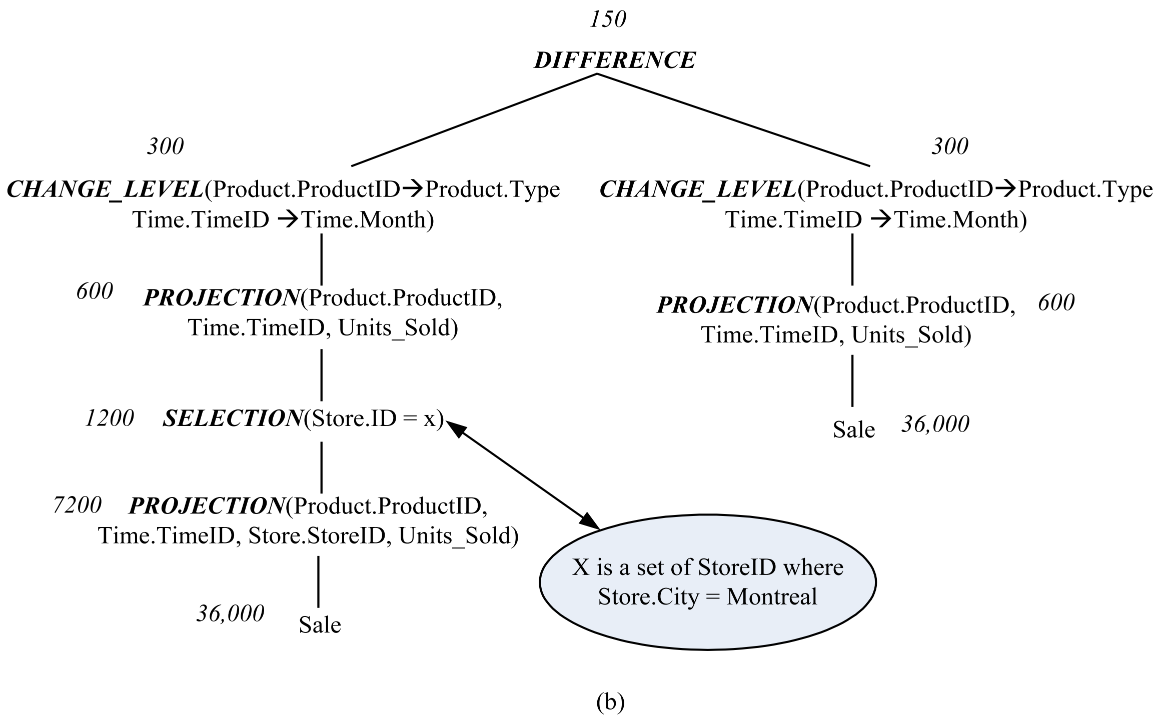

150
DIFFERENCE
300
300
CHANGE_LEVEL(Product.ProductID→Product.Type Time.TimeID →Time.Month)
CHANGE_LEVEL(Product.ProductID→Product.Type Time.TimeID →Time.Month)
600
PROJECTION(Product.ProductID, Time.TimeID, Units_Sold)
PROJECTION(Product.ProductID, Time.TimeID, Units_Sold)
600
1200
SELECTION(Store.ID = x)
Sale
36,000
7200
PROJECTION(Product.ProductID, Time.TimeID, Store.StoreID, Units_Sold)
X is a set of StoreID where Store.City = Montreal
36,000
Sale
(b)

Figure 5.17: Two candidates (a) and (b) for the preferred Logical query plan in Figure 5.16.

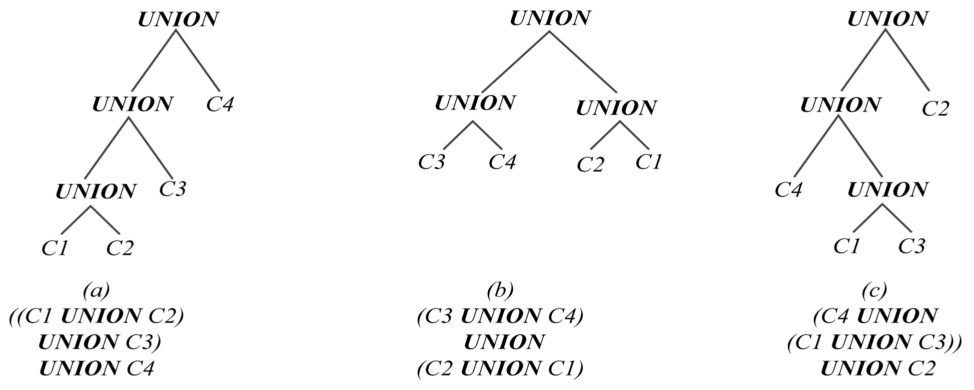


Figure 5.18: Three ways to union four cubes.

there are n! ways to order n elements. If a UNION is applied to n arguments (cubes), then the total number of possible ways to order and group the expression is n!*(n-1). For example, Figure 5.18 shows three of the (4!*3) possible ways to order the union of four cubes: C1, C2, C3, and C4. Note that a similar analysis is used with respect to the ordering of join operation in a relational query plan.

In our OLAP query optimizer, we have chosen to use a relatively simple heuristic to choose an order for the UNION of many cubes. Specifically, we use a “greedy algorithm” heuristic for UNION ordering. A greedy algorithm always makes the best choice at a specific moment rather than explicitly seeking a global optimum (i.e., dynamic programming techniques) [48, 33]. The greedy choice is between all possible unions at any given point in time. It works as follows:

- Select two cubes whose estimated UNION size is smallest. The union of these cubes becomes the current plan.

- Select a cube (C), not yet in the current plan, where the union of C with the current plan has the smallest estimated size. The new current plan has the old plan as its left argument and the selected cube (C) as its right argument.

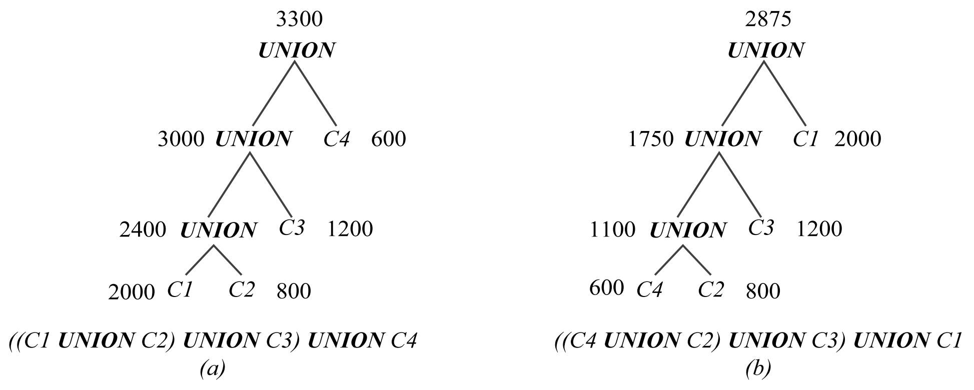


Figure 5.19: (a) Initial plan tree with cost (5400). (b) Optimized plan tree with cost (2850).

Consider the UNION of four data cubes (C1, C2, C3, and C4). The sizes (number of cells) of cubes are: NC(C1) = 2000, NC(C2) = 800, NC(C3) = 1200, NC(C4) = 600. The greedy algorithm begins by finding the pair of cubes that have the smallest estimated union size. As was illustrated, the estimated number of cells of the UNION operation is the size of the larger cube plus half the smaller. Therefore, we have currentPlan = C4 UNION C2, with the number of cells equal to (600 + 800/2) = 1100 cells. We now consider whether to UNION C1 or C3 with the currentPlan. We select C3 because it leads to a smaller estimated cost. Thus, the new currentPlan is (C4 UNION C2) UNION C3. The size of the currentPlan is 1100 + 1200/2 = 1750. Finally, there is no choice; we must union the currentPlan with C1. The final plan is ((C4 UNION C2) UNION C3) UNION C1, with a cost of 1750 + 1100 = 2850, the sum of the number of cells of the intermediate cube results.

Conversely, let us assume that the initial plan is ((C1 UNION C2) UNION C3) UNION C4. Here, the cost becomes (2000 + 800/2) + 2400 + 1200/2 = 2400 + 3000 = 5400. The optimized cost is 2 times smaller than the cost of the initial plan. Figure 5.19 describes the initial and the optimized plan trees.

5.8.3 Implementations for OLAP algebraic operators

In addition to all the steps mentioned in Section 5.7 and 5.8, the server must also select an algorithm for each OLAP operator in the OLAP logical plan in order to turn the preferred logical plan into a physical plan. In Chapter 3, we discussed the construction of an efficient OLAP storage engine that has various data structures and indexing components that allow for efficient and reliable execution of OLAP queries. We note that the algorithm for each OLAP operator (e.g., SELECTION, PROJECTION) depends on the functionality developed in Chapter 3. In this section, we discuss the selection of algorithms for OLAP operators defined in our OLAP algebra (SELECTION, PROJECTION, etc.). We note that the size of intermediate results could be more than the size of the main memory; however, we will assume that we have enough memory to store the intermediate query results. Extensions to external memory are expected to be undertaken in the future.

5.8.3.1 Choosing a SELECTION Method

The selection is the driving operation behind most analytical queries. Therefore, one of the important steps in choosing a physical plan is to select an implementation for each selection operator. As was illustrated, SELECTION in the preferred logical OLAP query plan is of the form: SELECTION(Dim.DimID = x AND Dim1.Dim1ID = y OR Dim2.Dim2ID ...)C. SELECTION is defined as a listing of dimensions related via AND and OR, where each dimension is associated with a condition. For simplicity, we consider the SELECTION with only one dimension (i.e., Dim) like SELECTION(Dim.DimID = x), such that x is a set of DimIDs that satisfy the user's query condition (UC) associated with one dimension called Dim (Note that DimID is the most detailed level of dimension Dim). The user's query condition associated with dimension Dim is of the form "Dim (A OP c)", where A can be a hierarchical or non-hierarchical attribute of dimension Dim, OP can be any comparison operator defined by our OLAP query grammar (e.g., <, >, =, IN_LIST), and c is a constant or set of constants. UC is a compound condition of one or more simple conditions against dimension Dim (connected via logical operators AND and OR). As was discussed in Section 4.7, we would like to eliminate the inner/natural joins between the cube and dimension tables that would ordinarily be required to exclude cube rows that do not satisfy the query restriction. The implementation of SELECTION is divided into the following three steps.

First, we need to find all dimension members (DimIDs) satisfying the query restriction called UC (defined by the user). For simplicity, we consider the query condition UC =Dim (A OP c).

1. If A is a hierarchical attribute level in dimension Dim, then we retrieve all DimIDs (most detailed integer values) that satisfy the comparison UC(AOP C), using the enhanced hierarchy manager (mapGraph) discussed in Sub-section 3.4.3.

2. If A is a non-hierarchical attribute level, then we retrieve all DimIDs that satisfy UC, using the FastBit compressed bitmap index created for each non-hierarchical attribute level in the dimension.

If UC is the AND/OR of simple conditions, then we use mapGraph and/or FastBit bitmap indexes to identify the set of DimIDs that satisfy UC. Using the mapGraph and the bitmap indexes ensure that the resulting DimIDs that satisfy the query condition (UC) associated with dimension Dim are sorted. This result is organized as an ordered set of contiguous ranges that is stored in a main-memory sorted array.

Given a DimID value v, we can directly apply a binary search within the sorted array to verify the existence of that given value. We can use similar techniques to find and store the dimension IDs for other user's dimension conditions mentioned in the SELECTION operator. An example of this will be provided shortly.

Second, the SELECTION at this step has the most detailed dimension values that satisfy the user's conditions on those given dimensions (e.g., SELECTION(Dim.DimID = x AND Dim1.Dim1ID = y OR ...) V, such that x and y are all DimIDs and Dim1IDs that satisfy the user's conditions on dimensions Dim and Dim1 respectively). We access the Berkeley database Hilbert R-tree index of view V, and use the Linear Breadth First (LBF) Search algorithm to efficiently answer the SELECTION operator. We stress that the initial LBF pre-dates the work in this research and answers very simple range queries. However we will soon see how the initial Sidera LBF is enhanced to answer complex range queries.

Finally, if no indexes are available for dimension tables and views, then we can answer the SELECTION operation by sequentially scanning dimension tables and views to find those rows that match the condition.

Consider the two dimensional view (Sales) of Figure 4.24 with the associated dimensions (Employee and Product) of Figures 3.3 and 3.4 respectively. Assume that we want to list the total sales for Product category (Automotive), and for all employees whose ages are greater than 25. The query restriction is: SELECTION(Product(Category = "Automotive") AND (Employee(Age > 25))Sales. Further, suppose that view Sales is indexed using the packed Hilbert R-tree index. Moreover, Figure 3.10 illustrates the mapGraph that represents hierarchical attribute levels of the Product dimension, while Figure 3.11(a) illustrates a simple bitmap index for the non-hierarchical attribute level (Age) of dimension Employee. Recall that the bitmap index for a non-hierarchical attribute level is compressed and implemented by means of using the FastBit implementation. The following are the steps for implementing the SELECTION operation:

1. Use the mapGraph to return all ProductIDs that have category = Automotive. This returns ProductID={1, 2, . . ., 7}. Keep the result in memory and store it as a set of contiguous ranges in a sorted array called ProductArray. ProductArray = {(1, . . . 7)} and has only one contiguous range.

2. Use the bitmap index for attribute (Age) to find all employeeIDs that are older than 25. The result is EmployeeID 2 and 3. Again, store this result as a sorted set of contiguous ranges in the EmployeeArray={(2, 3)}.

3. Use Linear Breadth First Search to find all records (cells) that intersect with ProductArray and EmployeeArray. There are 5 cells (records) that satisfy the user's condition and the schema of the output is the same as the schema of the input view (Sales). The output schema is (ProductID, EmployeeID, Units_Sold) and the result cells are: {(1, 1, 80), (1, 2, 41), (3, 1, 10), (3, 2, 28), (7, 1, 15)}.

5.8.3.2 Choosing a PROJECTION Method

Choosing a method to implement a PROJECTION operator depends on its position in the preferred logical query plan. The following are the options for implementing the PROJECTION operation:

1. If a PROJECTION(L,M)C is followed by a SELECTION, we can use Sidera's viewManager (outlined in Section 2.8) to obtain the best view from cube C —

the view itself must possess all attributes (L,M) that are mentioned within the PROJECTION operator — and then send it to the SELECTION operator.

2. If the dimension attributes L of (PROJECTION(L,M)V) are equivalent to the feature attributes of the view V, we scan all cells of V one at a time and pipeline them to the parent operation.

3. If the feature attributes of view V are greater in number than the dimension attributes L of PROJECTION(L,M)V, then the PROJECTION operator (PROJECTION(L, M)V) partitions the input cells (records) of view V into one or more groups. Each group consists of all cells that have the same values of dimension members (L). For each group G, we produce one cell that has the dimension member values for G and the aggregations of M. Therefore, in this case, the PROJECTION operator in our OLAP algebra is applied to cubes as a whole rather than one cell (record) at a time as in the SELECTION operation or the previous two cases (1 and 2).

We will see in the next section how algorithms can be applied to implement the PROJECTION for each given case.

5.8.3.3 Choosing Binary Operations Methods

We assume that we have enough memory to house any argument associated with our OLAP operators. Therefore, our approach for implementing the binary operators is to have the left argument in main memory. Then, one can read the right argument one cell at a time in order to perform the operation. We will describe the algorithms for binary operations in the next section.

5.8.3.4 Choosing a Method for CHANGE_LEVEL and CHANGE_BASE

Each one of these operations — CHANGE_LEVEL and CHANGE_BASE — are only relevant as a query against an existing set. Therefore, we assume that the inputs of these operations are result cubes that are housed in main memory. Recall that we don't consider an algorithm for CHANGE_BASE because of the use of LAW_8. For the CHANGE_LEVEL operator, we read the input cells of this operation one cell at a time, and then change the values for those attributes mentioned within this operation. Specifically, we use the Sidera mapGraph to change between arbitrary hierarchical level values. The resulting cell is stored in an in-memory hash table structure that supports efficient searching and insertion. In the next section, we will discuss an algorithm that implements the CHANGE_LEVEL operation.

5.8.4 Pipelining OLAP operations

There is an additional but important step needed in order to transform the preferred OLAP logical query plan into a complete OLAP physical plan. We must decide how to execute the entire OLAP expression tree. For each operation we have to choose one of the following alternatives:

- Materialization: The cube result (cells) of each intermediate OLAP operation is materialized (stored on disk) until it is needed by another operation.

- Pipelining: The intermediate cells (cube result) produced by one OLAP operation are passed to parent operations, even as an operation is being executed, without storing those cells on a disk.

The Pipelined evaluation is more rapid and cheaper than materialization due to the fact that one does not need to store intermediate results on disk while several operations can be evaluated simultaneously by passing on the results of one operation to the next. In this section, we discuss where and how we can use pipelining or materialization with our OLAP operators (SELECTION, UNION, etc.). Note that this is a theoritical presentation. In other words, physical materialization and pipelining are not yet implemented in the Sidera engine.

5.8.4.1 Pipelining SELECTION

Since SELECTION is one cell at a time, the SELECTION operator is a very good candidate for pipelining. We use the appropriate R-tree index view to return a set (S) of block numbers, one for each data block that intersects with the query restriction (Cond) mentioned within the SELECTION. We implement SELECTION(Cond)C in a pipelined fashion as depicted in Figure 5.20. The process is as follows:

1. For each block number in set S, read its corresponding data block b into the main memory.
2. For each cell c in b, check:
3. If c satisfies condition (Cond), then pass c directly to the parent operation and, if not, ignore it.

5.8.4.2 Pipelining PROJECTION

We shall discuss the following two different cases of the PROJECTION operator, (assuming outputCube = PROJECTION(L,M)inputCube)

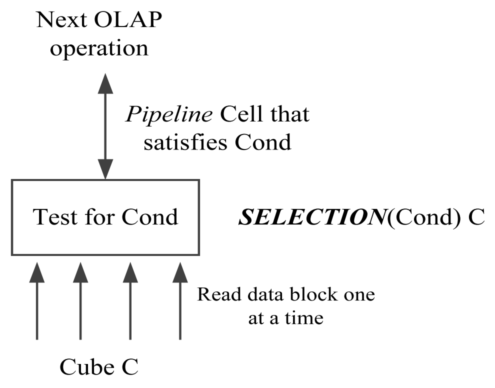

Next OLAP operation
Pipeline Cell that satisfies Cond
Test for Cond
SELECTION(Cond) C
Read data block one at a time
Cube C

Figure 5.20: Execution of a pipelined SELECTION.

1. If the feature attributes (L) of the outputCube are the same as the feature attributes of the inputCube, then we process PROJECTION by reading one cell (record) at a time, beginning with the inputCube and pipelining it to the parent OLAP operation by dropping the unnecessary measure attributes of the inputCube. In this case, we do not need the entire inputCube to be in the main memory to process the PROJECTION.

2. If the inputCube has more feature attributes than the dimension attributes L mentioned in PROJECTION, then we cannot produce the result of the PROJECTION until the last cell of the inputCube is seen because the duplicate output cells (cells with the same values of dimension attributes) must be re-calculated. The result of the PROJECTION is kept in a main memory data structure and is ready to be used in any other OLAP operator in the expression plan.

Figure 5.21(a) illustrates the pipelined PROJECTION process, while Figure 5.21(b) shows how the result of PROJECTION is blocked until the last cell of the input cube is seen.

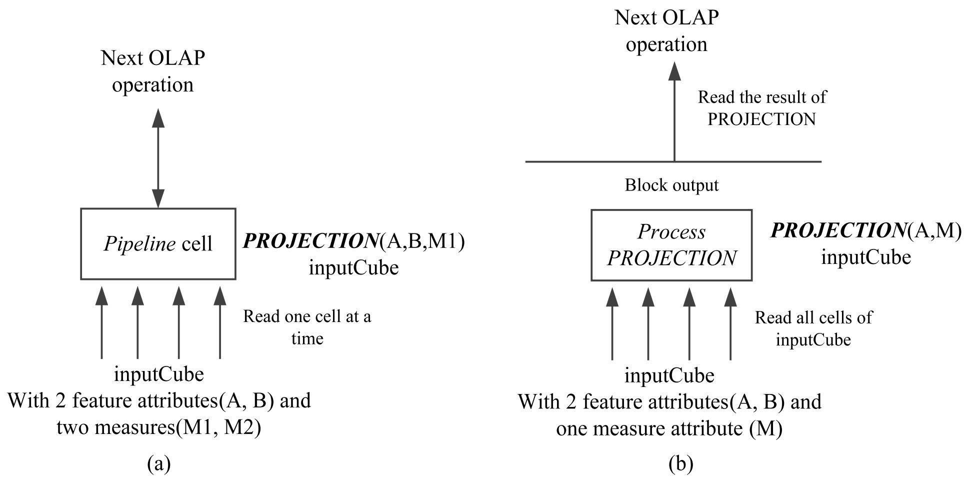

Next OLAP operation
Pipeline cell
PROJECTION(A,B,M1) inputCube
Read one cell at a time
inputCube With 2 feature attributes(A, B) and two measures(M1, M2)
(a)
Next OLAP operation
Read the result of PROJECTION
Block output
Process PROJECTION
PROJECTION(A,M) inputCube
Read all cells of inputCube
inputCube With 2 feature attributes(A, B) and one measure attribute (M)
(b)

Figure 5.21: (a) Execution of a pipelined PROJECTION. (b) Block PROJECTION result until the PROJECTION process terminates.

5.8.4.3 Pipelining Binary Operations

The results of our OLAP binary operations (UNION, INTERSECTION, DIFFERENCE and DRILL_ACROSS) can be pipelined. We assume that the left argument (cube) of any binary operation is organized in main memory as a hash table called hT (to be explained in Section 5.9). We implement a binary operation in a pipelined fashion as follows:

1. Read one cell (r) at a time from the right cube argument.
2. Execute the binary operation between the left argument and r.
3. If possible, pipeline a result cell, and if not, block the result and store it in a hash table data structure.

It is very important to recognize that pipelining a cell to the parent operation, even as the binary operation is being executed, depends on the current operation

(UNION, DIFFERENCE, etc.). We shall consider each one of the binary operations in turn.

UNION and DRILL_ACROSS: The results of UNION and DRILL_ACROSS operations can be pipelined. For example, the pipelined UNION operation is performed as follows:

1. If no search key in the hash table hT that contains the left cube argument equals the feature attribute values of r (r is a cell from the right cube argument), then we pipeline r to the parent OLAP operation.

2. If the values of feature attributes of r equal the search key of an entry e of the hash table hT, then we pipeline the cell that is the result of the UNION between two cells (r, e). Also, we remove the entry e from the hash table hT.

Finally, when the last cell of the right argument is seen, we pass all cells of the hash table hT to the parent operation. We can use a similar pipelined technique with the DRILL_ACROSS operation. But instead of executing UNION in step 2, we have to perform the DRILL_ACROSS operation.

INTERSECTION: If feature attribute values of r equal the search key of an entry e of the hash table hT, then we pipeline the cell that is the result of the INTERSECTION between the two cells (r, e).

DIFFERENCE: If the hash table hT contains an entry e where its search key equals the feature attribute values of r, then we remove e from the hash table and pipeline the result cell that is the DIFFERENCE between the two cells (e and r) to the next OLAP operation. After the last cell of the right argument is received, we pass all remaining cells in the hash table to the parent OLAP operation.

5.8.4.4 Pipelining CHANGE_LEVEL

We can pipeline the output of a CHANGE_LEVEL to other OLAP operations when all cells of the input argument (cube) have been read into the main memory and contributed to the result of the operation. The result of the operation is organized in the main memory as a hash table. After the operation is terminated, we pass the result to the next OLAP operation in the expression plan.

5.8.5 Physical Query Plan Notation

In this section, we describe the notation for our OLAP physical query plan. The final physical query plan must have details concerning access methods (R-tree index, bitmap index, etc.), pipelining or the materialization of intermediate results, and one or more physical algorithms for OLAP algebra operators. In the final OLAP physical query plan, each edge in the preferred OLAP logical query plan has to be marked to indicate that the intermediate result is materialized or pipelined. For simplicity, we insert the word *Blocking* close to the edge between the result that must be materialized and its parent operation. The *Pipelining* keyword is used otherwise.

5.8.5.1 Access method

Below are the access methods that will be used in our physical query plan:

1. BerkeleyRtreeAccess(C, F): here C is the best Hilbert R-tree index view/cuboid selected by Sidera's view manager to answer the current OLAP data query. F is a condition of the form Dim.DimID = x, where x is a set of DimIDs satisfying the user's condition(s) on dimension Dim. Cells of C satisfying condition F are returned by using the Linear Breadth First Search strategy of the R-tree indexed cuboid C that is stored as a Berkeley DB database.

2. CubeAccess(C, N): Sequentially reads all data blocks holding cells of view/cuboid C. For each cell, if the value of N is present, one should drop all unnecessary measure attribute(s) N from the measure attributes of the input cube C.

3. Dimension-scan (D, R): sequentially read the blocks containing the records of dimension D one by one to find dimension IDs satisfying the condition R.

4. Cube-scan(C, F): Sequentially read the blocks of view C one by one to answer the condition F.

5. mapGraphAccess(D, R): R is a condition on dimension D. We use mapGraph to answer the condition on dimension D if hierarchical attribute levels of dimension D are involved in R.

6. bitMapAccess(D,R): R is a condition on dimension D. Here condition R has non-hierarchical attribute(s), so we use the bitmap index to answer the condition.

5.8.5.2 Physical Operators for SELECTION

In Sub-section 5.8.3.1, we explained how the SELECTION operation is resolved. Specifically, we first use the hierarchy manager and the bitmap index manager to convert the user's condition to a condition that is in turn answered by accessing the appropriate R-tree index view/cuboid. Consider SELECTION(D(Cond)) C. Cond is a user's condition of the form A OP c, where A is an attribute of dimension D, OP is a comparison operator (IN_LIST, >, <, etc), and c is a constant or list of constants. C is the index cuboid/view that returns cells satisfying the user's condition Cond. We simply replace SELECTION(D(Cond)) C, by the following physical operators:

- **If** (A is a non-hierarchical attribute of dimension D) **THEN** F = bitMapAccess(D, A OP c) **ELSE** F = mapGraph(D, A OP c)
- BerkeleyRtreeAccess (C, F). Here F is a set of dimension IDs satisfying the user's condition on dimension D, C is the Hilbert R-tree index cuboid needed to answer the query.

5.8.5.3 Physical Operators for PROJECTION

PROJECTION in our logical query plan appears in several different positions. In the previous section, we discussed several methods used to implement the PROJECTION. We will discuss several implementations. Consider PROJECTION(L,M) C, where L is the list of dimension attributes, M is the list of measure attributes and C is the input cube. We have to take into consideration two cases:

1. If C were not stored on a disk, but a cuboid/view that was produced by another OLAP operation, such as SELECTION, then if the feature attributes of C are the same as the dimension attributes (L) of PROJECTION, then we use the operator in step a, and if not, we move to step b:

   a. dropUseless_Measure(N), N is the list of undesired measures. In other words, N is the list of all measure attributes of the input cube C without those measure attributes that are mentioned in list M.

   b. hashTable_PROJECTION(L,M).

2. If C is an index cube stored as one Berkeley DB physical database file that encapsulates the R-tree indexes of all cuboids in cube C, then the first operator that must be used is BV = viewManager_Projection(D,M, C), where BV is

the most efficient indexed cuboid/group-by from cube C to answer the current OLAP data query and D is the list of dimensions that are mentioned in L. Consequently, we have to consider three cases:

a. If the parent operation is a SELECTION, then we just send to it the appropriate view to answer the PROJECTION. This allows us to use the Hilbert R-tree index cube to answer the SELECTION.

b. If the parent operation is not a SELECTION and if the feature attributes of BV are equivalent to the dimension attributes L of the PROJECTION, then in addition to viewManager_Projection, we need one more physical operator — CubeAccess(BV, N) — to resolve the PROECTION operation. In other words, all cells in the input view contribute to the result of the PROJECTION.

c. If the number of feature attributes of BV are greater than the number of dimension attributes (L) of PROJECTION, then we need one more operator (hashTable_PROJECTION(L,M)) in addition to those operators of the previous step. Since the number of output attributes doesn't equal the number of input attributes in the input view, the output cells must be re-calculated accordingly.

5.8.5.4 Physical Operators for Binary Operations

OLAP binary operations are replaced by an appropriate physical operator. These physical operators indicate:

1. Current operation being performed (e.g., UNION, INTERSECTION).

2. The data structure that supports the execution of the operation, e.g., hash table.

We assume that the hash table supports the execution of our binary operators. We use the following notation for the OLAP binary operation: hashTable_OPERATOR, where OPERATOR can be any OLAP binary operator like UNION, DRILL_ACROSS, DIFFERENCE or INTERSECTION.

5.8.5.5 Physical Operator for CHANGE_LEVEL

The algorithm for implementing the CHANGE_LEVEL operator builds upon Sidera's hierarchy manager (mapGraph) and an in-memory hash table data structure. Consider CHANGE_LEVEL(Dim.DimID → Dim.AttLevel)C, where C is the input cuboid, Dim.DimID is a feature attribute in cuboid C and Dim.AttLevel is a hierarchical attribute level in dimension Dim. We replace the logical operator by the following physical operator:

mapGraph_HashTable_CHANGE_LEVEL(Dim.DimID → Dim.AttLevel).

Recall that in Section 3.4, we explain that each hierarchical attribute level has encoded values (integer values for user-supplied values); therefore, we use this physical operator in order to change the base level (Dim.DimID) attribute encoded (integer) values to the corresponding encoded values at attribute level Dim.AttLevel.

5.8.5.6 Additional Physical Operators (Get_realValues)

As discussed in our OLAP query grammar, the PROJECTION operation must exist in any OLAP data query. PROJECTION identifies the presentation attributes that are returned to the user, including both dimension attributes (L) and measure attribute(s). The result at the top of the physical query plan has only linear values (integer values) for the presentation attributes (e.g., Product Category 2 means Automotive). As was discussed, the result of each OLAP operator, excluding SELECTION, is stored into a hash table structure in the main memory. In the next section, we will explain how our hash function ensures that the entries are sorted according to the list of attributes of the search key. Therefore, we can say that the result of the OLAP operations is sorted.

Finally, we have to use an additional physical operator that converts the linear values for each dimension attribute (mentioned within the PROJECTION) to their real values. The hierarchy manager mapGraph contains multiple hash tables, one for each hierarchical attribute level. The hash table is simply used to provide O(1) conversion of system-specific integer values to real values. For example, Figure 3.10 illustrates two hash tables for the hierarchical attributes Type and Category. As an example, we can use the Type hash table to convert integer value 2 to Engine. Thus, at the top of the physical query, we use the following operator:

- Get_realValues(L), where L is the list of attributes that the user will see in the result. This operator replaces the linear values of the result with their real values.

5.8.5.7 Final Physical Query Plan

In this section, we illustrate the conversion of two preferred logical query plans to their final physical query plans. Figure 5.22 illustrates the OLAP physical query plan for the preferred logical query plan developed in Figure 5.9. In the OLAP physical plan of Figure 5.22, we use the view manager to get the best cuboid/view (BV) to answer the current OLAP data query. We use the hierarchy manager and the bitmap

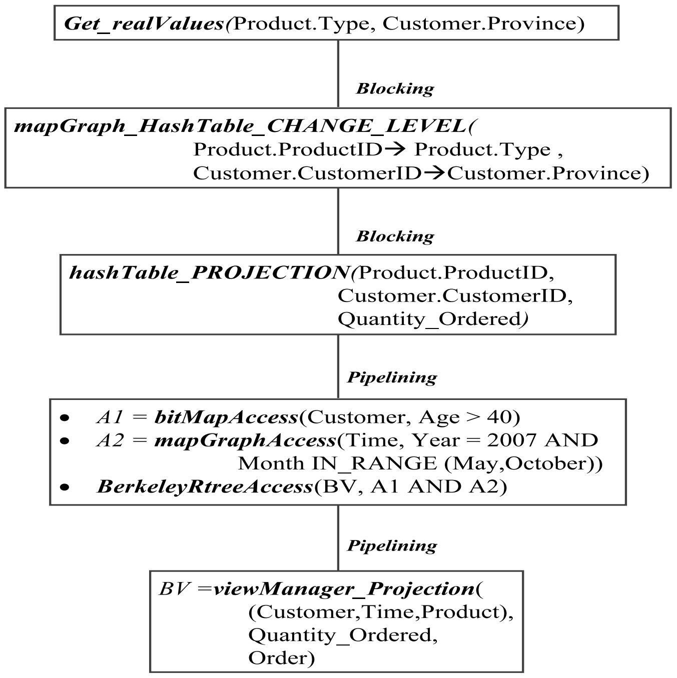

Get_realValues(Product.Type, Customer.Province)
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Type , Customer.CustomerID→Customer.Province)
Blocking
hashTable_PROJECTION(Product.ProductID, Customer.CustomerID, Quantity_Ordered)
Pipelining
A1 = bitMapAccess(Customer, Age > 40)
A2 = mapGraphAccess(Time, Year = 2007 AND Month IN_RANGE (May,October))
BerkeleyRtreeAccess(BV, A1 AND A2)
Pipelining
BV =viewManager_Projection((Customer,Time,Product), Quantity_Ordered, Order)

Figure 5.22: A physical plan for the logical query plan of Figure 5.9.

index manager to convert the user's conditions into a set of sorted arrays (A1 and A2). We use a linear breadth first search for the materialized indexed cuboid (BV) in order to retrieve all cells that satisfy the user's condition. After this, we project it onto the required feature attributes (Product.ProductID, Customer.CustomerID, Quantity_Ordered) and then use the hierarchy manager to implement the change level. Finally, the real values are obtained by using the physical operator (Get_realValues). The plan also illustrates the pipelined and materialized intermediate results.

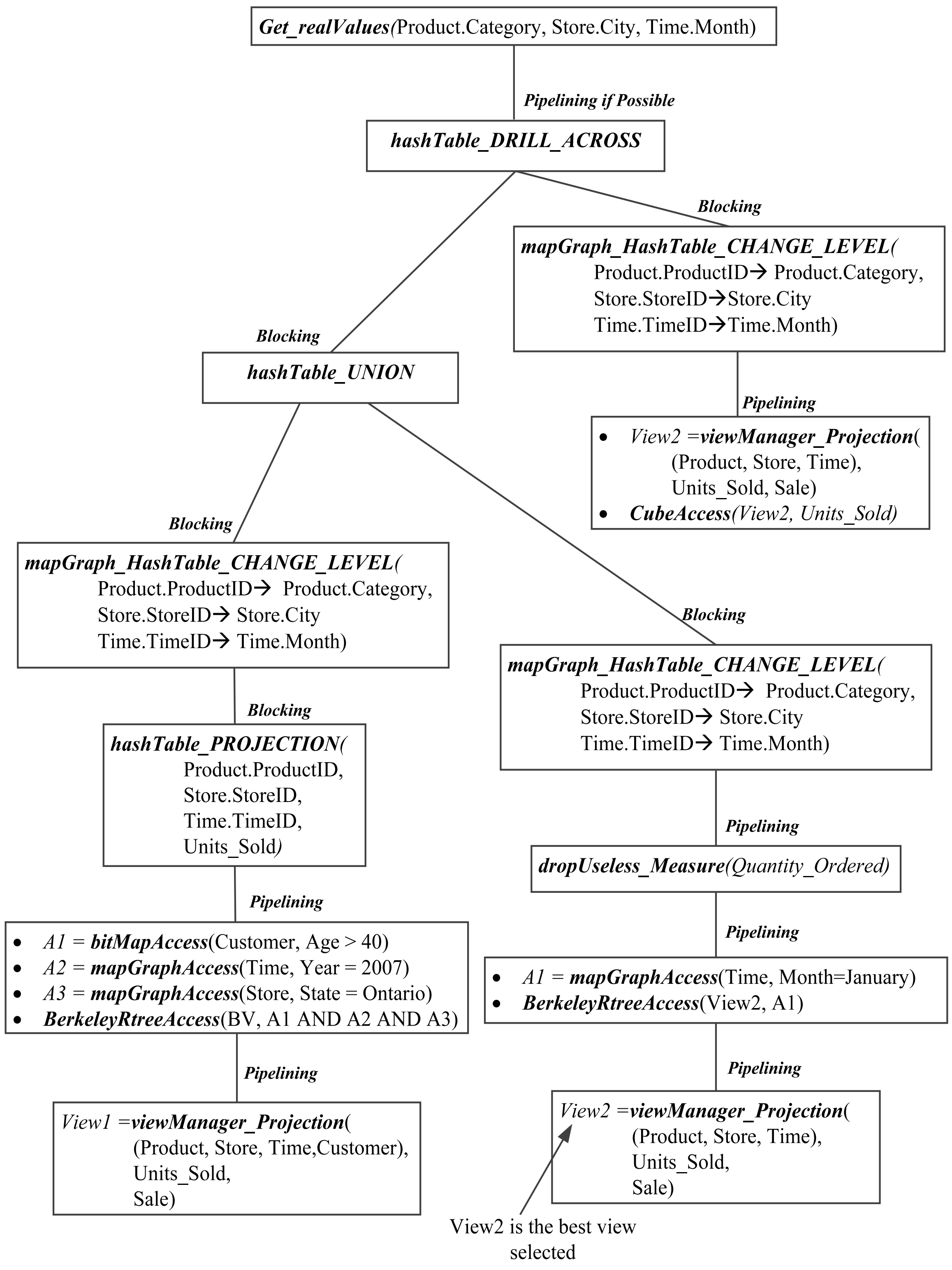
A more realistic OLAP physical query plan that covers most of our OLAP physical operators is shown in Figure 5.23. This plan is developed for the preferred OLAP logical query plan of Figure 5.13. We assume that the four dimensional cube Sale mentioned in Figure 5.10 has 3 materialized index cuboids/views. The following are their feature and measure attributes:

- View1: (Product.ProductID, Customer.CustomerID, Store.StoreID, Time.TimeID, Units_Sold, Quantity_Ordered)
- View2: (Product.ProductID, Store.StoreID, Time.TimeID, Units_Sold, Quantity_Ordered)
- View3: (Product.ProductID, Store.StoreID, Customer.CustomerID, Units_Sold, Quantity_Ordered)

Notice in Figure 5.23, View2 and View1 are selected by the view manager. We see how the dropUseless_Measure operation is used to drop the unnecessary measure attribute (Quantity_Ordered). It is very important to notice the different implementations of the PROJECTION operator in the physical tree (cubeAccess, hashTable_Projection, and dropUseless_Measure). Pipelining and blocking results are also mentioned in the physical plan. We can also see how the binary physical operations (hashTabe_UNION and hashTable_DRILL_ACROSS) are performed between two cube results.

5.9 OLAP Query Execution

The result of query compilation is an OLAP physical query plan (e.g., Figure 5.23) that defines an efficient execution plan for the received OLAP query. As mentioned in the previous section, this physical query plan is represented as a tree that implies the order of physical operations. Specifically, the cube data must flow up the physical query plan tree. We order the execution of all nodes of the physical plan tree in a bottom-up, left-to-right manner. In other words, we order the nodes of the tree in such that a pre-order traversal traverses the entire physical query tree. Following the pre-order traversal, our OLAP query optimizer can generate a sequence of function



Get_realValues(Product.Category, Store.City, Time.Month)
Pipelining if Possible
hashTable_DRILL_ACROSS
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Category, Store.StoreID→Store.City Time.TimeID→Time.Month)
Pipelining
View2 =viewManager_Projection((Product, Store, Time), Units_Sold, Sale)
CubeAccess(View2, Units_Sold)
Blocking
hashTable_UNION
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Category, Store.StoreID→ Store.City Time.TimeID→ Time.Month)
Blocking
hashTable_PROJECTION(Product.ProductID, Store.StoreID, Time.TimeID, Units_Sold)
Pipelining
A1 = bitMapAccess(Customer, Age > 40)
A2 = mapGraphAccess(Time, Year = 2007)
A3 = mapGraphAccess(Store, State = Ontario)
BerkeleyRtreeAccess(BV, A1 AND A2 AND A3)
Pipelining
View1 =viewManager_Projection((Product, Store, Time,Customer), Units_Sold, Sale)
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Category, Store.StoreID→ Store.City Time.TimeID→ Time.Month)
Pipelining
dropUseless_Measure(Quantity_Ordered)
Pipelining
A1 = mapGraphAccess(Time, Month=January)
BerkeleyRtreeAccess(View2, A1)
Pipelining
View2 =viewManager_Projection((Product, Store, Time), Units_Sold, Sale)
View2 is the best view selected

Figure 5.23: OLAP physical query plan for OLAP logical plan of Figure 5.10.

calls —one for each physical operation in the physical plan— and pass them to the OLAP query engine for execution.

In this section, we look at the algorithms that are used to access the data of the OLAP storage. Recall that Berkeley databases are used in our server to store the indexed cube in one physical file. Moreover, we shall cover the algorithms applied in the execution of our OLAP algebraic operators (SELECTION, PROJECTION, etc.) against the indexed cube (stored in the Berkeley DB) and associated dimension tables (e.g., FastBit bitmap indexes for non hierarchical attributes). We assume that we have enough memory to hold the result of any OLAP operator and any extra data structure (i.e., Bitmap index for an attribute). Note that extensions to external memory are expected in the future. Finally, for each physical operator in the OLAP physical query plan, we determine the appropriate algorithm(s) that can be used to answer them (e.g., algorithm x implements the physical operator BerkeleyRtreeAccess()).

5.9.1 In-memory Hash Table Representation

As was discussed in the previous section, some of our OLAP physical operators require an in-memory hash table data structure for efficient searching and inserting. In practice, the entry of a hash table is of the form (k,v), where k represents the search key of the hash table and v its associated value [56]. In our case, the value of the search key k is the value(s) of the feature attributes that will be in the result of a given OLAP operator, while v is the value of the measure attributes. In general, a hash table consists of an array of size N, and a hash function h that maps values of a given type (string, array of integers, etc.) to integers between [0, N-1].

In our case, for each physical operator that needs an internal hash table to be executed, we create a hash table (hT) of size N, where N is equivalent to the cardinality product of the result of the OLAP operator, and a hash function h that maps the values for one or more feature attributes to a specific integer between 0 and N-1. Algorithm 7 shows an implementation of our hash function. The input of the algorithm consists of a list of feature attributes fA, an array of cardinality products (aCP) and an array (aV) that possesses the values of the feature attributes to be mapped to an integer between [0, N-1]. Let the list of feature attributes be of the form fA = $\{f_1, f_2, \ldots, f_i, \ldots, f_n\}$, where n is the number of feature attributes in the result of a given OLAP physical operator. We can thus say that aCP can be written as $\{CPf_1, CPf_2, \ldots, CPf_n\}$, where the value of CPf_i represents the cardinality product of all subsequent feature attributes $\{f_{i+1}, f_{i+2}, \ldots, f_n\}$. Note that CPf_n equals 1. aV has n values $\{aV_1, aV_2, \ldots, aV_n\}$, one value for each feature attribute f_i in fA. It is crucial for one to maintain the exact sequential order of the numerical values in aV as they each represent a specific feature attribute. Algorithm 7 returns the hash key for the values of the feature attributes (aV). Our hash function ensures O(1) processor running time for searching, inserting and deleting entries from the hash table. Moreover, the example below will illustrate how our hash function ensures that the entries of the hash table are sorted according to the list of attributes in fA.

Consider the parameters for the Sale cube and dimensions: Product, Customer, Time and Store in Figure 5.10. Let the following OLAP physical operator be determined via the support of the hash table:

hashTable_PROJECTION(Customer.CustomerID, Store.StoreID,
Product.ProductID, Units_Sold)Sale

We first create a hash table array hA of size 5 * 12 * 50 = 3000 (Cardinality product of the output cube CP(CustomerID, StoreID, ProductID)). Let us assume that we

Algorithm 7 Hash Function algorithm

```
Input: List of Feature attributes fA{$d_1.d_1$ID, $d_2.d_2$ID, ...,$d_n.d_n$ID} where n is
    the number of feature attributes of the result, a list of cardinality products
    aCP{$CP_1$($d_2.d_2$ID, $d_3.d_3$ID, ..., $d_n.d_n$ID), $CP_2$($d_3.d_3$ID, $d_4.d_4$ID, ..., $d_n.d_n$ID),
    ..., $CP_n$(1)}, and the values of the feature attributes is v($d_1$ID, $d_2$ID, ..., $d_n$ID)
Output: an integer x between 0 and N-1, where N is the cardinality product for
    attributes in fA.
1: Initialize $x$ to 0
2: for each feature attribute in array fA stored at index i do
3:     $x = x +$ (v[i]-1) * CP[i]
4: end for
5: return $x$
```

need to find the hash value of the following set of feature attributes (CustomerID, StoreID, ProductID) (3, 10, 5). The input of Algorithm 7 is:

- fA = {Customer.CustomerID, Store.StoreID, Product.ProductID}
- Array aCP of cardinality products. aCP = {600, 50, 1}, 600 is the cardinality product of (StoreID, ProductID), while 50 is the cardinality product of ProductID.
- Array aV is the values of the feature attributes in fA, aV= {3, 10, 5}. In this case, 3 is the value of CustomerID, 10 is the value of StoreID and finally 5 is the value of ProductID.

Using the hash function outlined in Algorithm 7, the hash value of key (3,10,5) is: (3-1) * 600 + (10-1) *50 + 5-1 = 1200 + 450 + 4 = 1654 < 3000. This means that key(3,10,5) is stored in the array hA at the index of 1654. Moreover, consider the key (5, 12, 50), where the hash value is: (5 - 1) * 600 + (12 - 1) * 50 + 49 = 2999. Key (5,12,50) is stored at the very last index of the array hA. As mentioned, our hash function ensures the sorting of the entries according to their attributes in the key. Therefore, for the above example, key(1,1,1) is mapped at position 0, (1,1,2) is mapped at 1, (1,1,3) is mapped at 3, ..., (5, 12, 50) is mapped at position 2999 (last entry in the array).

Our hash table works well for attributes with small cardinalities. That being said, it is not fully scalable because the size of its array increases as the cardinalities of attributes increase and we also have to reserve memory space for non-existent values. However, our hash table is very useful after the execution of the SELECTION operation (i.e., the most important operator) as this likely reduces the size of the input cube by a large factor, as well as the cardinalities of feature attributes. In the future, we could extend our engine to use a more scalable balanced search tree — such as a red-black tree or AVL tree — as an internal data structure for result cubes with large cardinalities [11]. The balanced tree could also give fast lookup, insert, and remove (e.g., O(log n)) and would be sufficient for our purposes. In addition, the cube results could be sorted in the balanced search tree structure (i.e., working the tree could produce sorted key sets).

5.9.2 Index Based SELECTION Algorithm

Algorithm 8 is an algorithm applied to answer the SELECTION operator efficiently. Before we access the indexed cuboid/group-by to return the cube cells that satisfy the query restriction, we transform the user's query constraints that are specified on the attributes of the dimensions into the most detail-oriented level. Algorithm 8 utilizes a function called **transformSELECTION()** to convert the user's query restriction into a most detail-oriented value that can be utilized by our OLAP query engine. This algorithm is described in Algorithm 9. After this process, we open the Berkeley DB database object that represents the appropriate Hilbert R-tree index for the group-by (e.g., called V) to answer the selection operator. Finally, a **processSelection()** is applied which uses the Hilbert R-tree index for view V to answer the transformed user's condition and return the result. This process is described in Algorithm 10.

Algorithm 8 SELECTION Algorithm

```
Input: A user-defined OLAP selection condition $dC$, a hierarchy manager
    ($mapGraph$) containing the hierarchical attributes data, a cube $C$, an appro-
    priate view $V$ to answer the SELECTION operator, and a bitmap index manger
    $biM$ that contains the bitmap indexes for the needed non-hierarchical attributes.
Output: Fully resolved SELECTION ($I$ with all detailed level values satisfying $dC$).
 1: create a new array $OP$ of size $n$, where $n$ is the number of logical operators
    (AND and OR) that are used to form compound conditions, each associated with
    a dimension.
 2: Use $dC$ to get those logical operators and store them in $OP$.
 3: Invoke transformSELECTION($dC, mapGraph, biM$)
 4: Open the Berkeley database object called $db$ that contains the Hilbert R-tree
    index for group-by $V$.
    db.open(NULL, $C, V$, DB-RTREE, DB_RDONLY, 644);
 5: get result $I$ from disk, $I$ = processSelection($dC$, $db$, $OP$)
```

The primary focus of Algorithm 9 is to replace the user's query restrictions that are specified within the SELECTION operator into other restrictions (dC) that can be solved against the indexed data stored in the physical cube. As was illustrated, a SELECTION(Dim.A OP c) View is translated into a SELECTION(Dim.DimID = x)View where x is a set of DimIDs satisfying the condition (Dim.A op c).

Algorithm 10 is used to retrieve the data from the disk via the Berkeley database Hilbert R-tree index. Note that the core algorithm is the Linear Breadth First search strategy that was used in the old Sidera server [36]. However, our purpose here is to describe how to use the indexed cube that is stored in the Berkeley database (Chapter

Algorithm 9 SELECTION Transformation Algorithm

Input: A user-defined OLAP selection condition dC, a hierarchy manager $mapGraph$, OP array of logical operator, and a bitmap index manager biM.

Output: The user's condition in the most detail-oriented form (primary key form).

```
1:  for each dimension condition $C_i$ in $dC$ do
2:    for each expression $e_j$ in $C_i$ do
3:      if attribute ($A$) involved in $e_j$ is a hierarchical attribute level then
4:        $array_j = mapGraph.$getBaseID$(A, e_j)$
5:      else
6:        $array_j = biM.$getBaseID$(A, e_j)$
7:      end if
8:      if Logical operator between $e_j$ and $e_{j-1}$ equals AND then
9:        $array_j =$ setIntersection$(array_j, array_{j-1})$
10:     else
11:       $array_j =$ setUnion$(array_j, array_{j-1})$
12:     end if
13:   end for
14:   create a new range array $newR$ of size $|array_j|$
15:   store integer values in $array_j$ as a sorted set of contiguous ranges
16:   Remove the current SELECTION condition $C_i$ and replace it with $D_i.D_iID = newR$ such that $newR$ has all $IDs$ that satisfy condition $C_i$ associated with $D_i$.
17: end for
```

3) to answer the transformed query constraints (dC) attached to the SELECTION operator.

In Chapter 3, we discussed the general LBFS strategy that was used to answer simple OLAP query restrictions that had been defined in a proprietary syntax. More specifically, OLAP queries that have been answered in the old Sidera server are of the form:

Dim.DimID Dim1.Dim1ID(1 10, 20 25) C

where Dim.DimID and Dim1.Dim1ID are two feature attributes. The current Sidera engine selects the best Hilbert R-tree index from cube C in order to return all cells that have Dim.DimID between 1 and 10, and Dim1.Dim1ID between 20 and 25. As such, the query restriction that was answered by the original LBF outlined in Algorithm 3 from Chapter 3 consists of one contiguous range for each feature attribute specified in the query's restriction (i.e., Dim.DimID). Algorithm 10 uses the same search strategy as was discussed in Algorithm 3. However, the SELECTION operator condition can have many contiguous ranges for each feature attribute that is specified within the operator. For example, the SELECTION operator may have the following form: SELECTION(Dim.DimID = x and Dim1.Dim1ID =y) C, where x = 1, 2, 3, 8, 9, 10 and y=10, 11, 12, 13, 20, 21, 25, 26. We can see that the SELECTION operator has a set of contiguous ranges on each dimension (Dim and Dim1).

Algorithm 10 enhances the original LBF algorithm by adding two functions — **is_block_intersect()** and **is_record_intersect()** — that can be used to return all cells which satisfy the SELECTION restriction. These two functions are invoked from Algorithm 10 to verify the intersection between the selection condition (set of contiguous ranges on each dimension within the SELECTION) and an R-tree index

Algorithm 10 Linear Breadth First Search SELECTION Processing

Input: A Berkeley DB database object (db) that contains the appropriate Hilbert R-tree index group-by, OP array of logical operator, and a SELECTION condition dC.

Output: Fully resolved SELECTION condition dC.

```
1: Initialize $pageList$ with page/block number of the root index block
2: while not at the leaf level do
3:    $childList$ = new empty list
4:    for each page/block number in the page list do
5:       for each index block number $b$ at level $i$ do
6:          if $b$ is found in the $pageList$ then
7:             Using $b$ as an offset to read the relevant index disk block $B$ into memory
8:          end if
9:          for each child block $j$ of $B$ do
10:            if (is_block_intersect($j$, $dC$, $OP$)) then
11:               Add the number of block $j$ to $childList$
12:            end if
13:         end for
14:      end for
15:      $PageList$ = $childList$
16:   end for
17: end while
18: for each block number $i$ in the current page list do
19:    Using $i$ as an offset, read the relevant data disk block $B$ into memory
20:    for each record $r$ in block $B$ do
21:       if (is_record_intersect($r$, $dC$, $OP$)) then
22:          Add $r$ to $result$
23:       end if
24:    end for
25: end for
26: return $result$
```

block, and subsequently, the intersection with data records.

Algorithm 11 R-tree index block intersects with a complex condition

Input: An index block b, an array called OP of logical operator between the dimension conditions, and the selection condition dC as a set of sorted arrays.
Output: True if b intersects with dC, false otherwise

```
 1: create an array result $R$ of size $n$, where $n$ equals the number of sorted arrays
 2: for each feature attribute $Dim.DimID$ in $dC$ do
 3:     $Low$ = the range minimum from block $b$ for attribute $Dim.DimID$
 4:     $High$ = the range maximum from block $b$ for attribute $Dim.DimID$
 5:     $tmpArray$ = sorted set of contiguous range for attribute $Dim.DimID$
 6:     if (BinarySearch($tmpArray$, $Low$, $High$)) then
 7:         if ($OP$ has only OR logical operator) then
 8:             Return True
 9:         else
10:             $R[i]$ = True
11:         end if
12:     else
13:         if ($OP$ has only AND operators) then
14:             return False
15:         else
16:             $R[i]$ = False
17:         end if
18:     end if
19:     $i = i + 1$
20: end for
21: return the Boolean result of applying $OP$ to $R$ (array of Boolean values)
```

Algorithm 11 implements the **is_block_intersect()** function. It is used to verify if an index block b intersects with the selection condition dC. The algorithm works as follows. First, we need all ranges (low/high) from block b for all feature attributes mentioned in dC. Recall that for each feature attribute (Dim.DimID) in dC, we have a set of contiguous ranges obtained from Algorithm 9 stored in a sorted memory

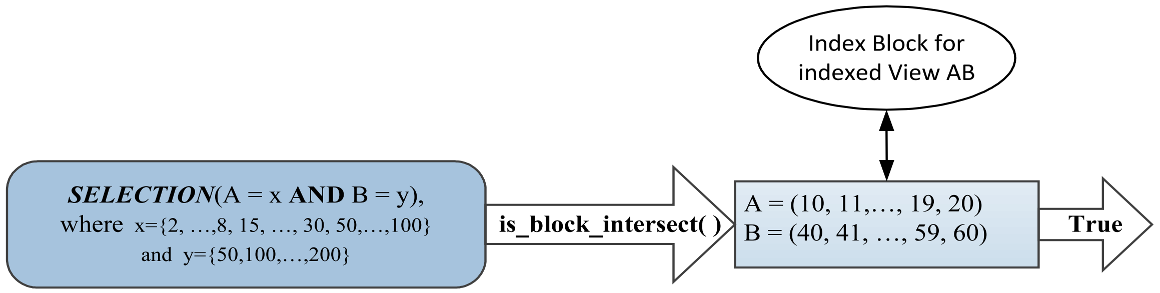

Index Block for indexed View AB
SELECTION(A = x AND B = y), where x={2, …,8, 15, …, 30, 50,…,100} and y={50,100,…,200}
is_block_intersect( )
A = (10, 11,…, 19, 20)
B = (40, 41, …, 59, 60)
True

Figure 5.24: Example of **is_block_intersect()** function.

array. Then, a binary search is performed for a set of contiguous ranges and returned in the affirmative if it has one value in common with its corresponding range from the index block b. Algorithm 11 is also aware of the logical operators between simple conditions (such as Dim.DimID = x AND Dim1.Dim1.ID). Figure 5.24 illustrates how this function works. For example, let block b represent an indexed block for two feature attributes (A and B) with two (low/high) ranges r1=[10, . . ., 20] and r2=[40, . . ., 60], and a condition of the form SELECTION(A = x **AND** B = y), such that x has three contiguous ranges x1=[2, . . ., 8], x2=[15, . . ., 30] and x3=[50, . . ., 100], while y has two contiguous ranges y1=[50, 50] and y2 = [100, . . ., 200]. Here, we can say that b intersects the condition because ranges r1 and x2 have common values (such as 16, 17 etc.) and also r2 and y1 have one value (50) in common.

Algorithm 10 uses the **is_record_intersect()** to check if a data record r matches the selection condition. Algorithm 11 can be used to implement this function; however the binary search returns affirmatively if a set of contiguous ranges has only one value in common with its corresponding value v from record r. An example to illustrate this function can be where r is a record (cell) with two values of the feature attributes (A = 4 and B = 10) and one measure value. We can say that r does not intersect the above mentioned SELECTION condition because B=10 is neither in y1 nor in y2.

5.9.2.1 Cost of the SELECTION operation

We must be able to estimate the cost of each OLAP physical operator that we use in the physical OLAP query plan. It is well-understood that it is slower to retrieve data from a disk than do anything with the data once it is in the main memory. Therefore, we use the number of disk I/O to estimate the cost of an OLAP operation. However, we shall also mention the processor running time when the amount of process time is proportional to a specific variable (i.e., n^2).

The input argument for the SELECTION operator is a Hilbert packed R-tree indexed group-by stored as a Berkeley DB database object on disk. Also, the SELECTION requires the data of non-hierarchical and hierarchical attributes in order to convert the user's query restriction to the most detail-oriented level form restriction. At run-time, the enhanced mapGraph hierarchy manager is used to represent the data of hierarchical attributes. In addition, we create another in-memory index manager called the Bitmap Index Manager to represent the data of each required non-hierarchical attribute in the SELECTION operator. We also assume that we have enough memory to store those two managers (mapGraph and indexManager). The result of the SELECTION is left in memory unless it is required to be returned to the disk. It is important to mention that the sorted arrays that are used to store the set of contiguous ranges are left in memory as well, until the SELECTION operation terminates. Recall that the sorted arrays represent the query restriction in the most detailed level form.

Theorem 1. *The cost of the SELECTION operator is bounded as the cost of sequentially scanning B(V) and D(V), where V is the appropriate packed R-tree index to answer the SELECTION, B(V) is the number of index blocks, and D(V) is the number of disk blocks. Cost = B(V) + D(V) I/O.*

Proof. SELECTION uses the Linear BFS strategy to retrieve records that satisfy its condition. LBFS uses a top-to-bottom/left-to-right search pattern for the packed R-tree indexed cube. As was discussed in Section 2.4, the indexed cube is stored physically on disk per consecutive disk IDs, using the same top-to-bottom/left-to-right fashion. Also, the data blocks follow this ordering. The worst case is to scan sequentially all index blocks and data blocks. Number of Disk I/O is B(V) + D(V) blocks. □

We note, however, there is also a large amount of processor time that may affect our assumption that only the disk I/O time is significant. If the condition of the SELECTION has k distinct feature attributes, then k sorted arrays are used to store IDs that satisfy the user's condition, where the larger sorted array has n IDs. We also assume that D(V) has m records (cells).

Theorem 2. *The worst case processor running time of the SELECTION operator has a bound of O(m * log(n)).*

Proof. In the worst case, we scan sequentially all index blocks and data blocks of view V. For each index block b, we perform a binary search to check if it intersects the selection condition that is stored as a set of sorted arrays. The worst case processor running time for the index scan is k * log(n) * B(V). Also, in the worst case, for each record (cell) of V we have to perform a binary search to check if it intersects the selection condition. The worst case running time for the data scan is k * log(n) * m. Finally, the worst case processor running time is k * log(n) * B(V) + k * log(n) * m which can be written as k * log(n) * (B(v) + m). This result can be re-written as k * log(n) * (O(m)) because m, number of records, dominates the number of index blocks. Finally, since k represents a small number of feature attributes, the worst case running time can be bounded as O(m * log(n)) in practice. □

The cost of the SELECTION algorithm can be determined by the sums of (a) the disk I/O and (b) the processor running time, as follows:

1. The worst case number of disk I/O is B(V) + D(V) disk I/O.

2. The worst case processor running time is O(m * log(n)).

In practice, we observe that for most queries the number of disk I/O dominates the processor running time. However, the processor time still has some effect on the total execution time.

Finally, note that Algorithm 9 is used as an implementation of the two physical operations (**bitMapAccess** and **mapGraphAccess**) in the final physical plan, while Algorithm 10 implements the physical operation (**BerkeleyRtreeAccess**).

5.9.3 Algorithms for the PROJECTION operator

In this subsection, we discuss three algorithms for the PROJECTION operator depending on its position in the preferred logical query plan. Choosing an algorithm for the PROJECTION operator depends on:

1. **A PROJECTION followed by a SELECTION:** if a PROJECTION operator in the logical query plan is followed by a SELECTION operator and the input cube of the PROJECTION is physically materialized (Hilbert R-tree index exists), then the implementation of the PROJECTION utilizes Sidera's view manager. In this implementation, we use Sidera's view manager to select the most cost effective materialized index cuboid/group-by that will be passed to the parent OLAP SELECTION operator. This algorithm requires no disk I/O for the PROJECTION since the view manager is found in the main memory. Algorithm 12 is used to implement the PROJECTION operator of this case. In other words, the query optimizer invokes Algorithm 12 as an execution of the physical operation **viewManager_PROJECTION()** in the final physical plan.

   Consider the Sale cube in Figure 5.10. We suppose that there are only two indexed materialized cuboids for the Sale cube (i)Sale1(Customer, Store, Time, Product), and (ii) Sale1(Customer, Store, Time). Consider the following OLAP query:

SELECTION(Customer.CustomerID = x)
PROJECTION(Customer.CustomerID, Store.StoreID, Units_Sold)Sale.

The implementation of the above PROJECTION operator is performed by using the view manager to move the best view to the SELECTION operator in order to answer the query. In this case, for example, Sale1 is selected and passed to the SELECTION operator.

2. **If the output attributes (L) of PROJECTION(L,M)V are equivalent to the feature attributes of view V**: Here, all cells of V will be in the output of the PROJECTION. In this case, we read one cell at a time and pipeline it to the parent operation by simply dropping the unnecessary measure attributes of V. No disk I/O is required if view V is a result of another operator. However, a sequential scan of the data set of view V (D(V)I/O) is required if view V is on disk. The implementation of this PROJECTION is described in Algorithm 13. Note that this algorithm is considered as the implementation of two physical operations: **dropUseless_Measure**(M) and **cubeAccess**.
Consider a three dimensional view schema V(Product.ProductID, Customer.CustomerID, Store.StoreID, Units_Sold, Quantity_Ordered), with the following cells {(1, 2, 5, 200, 250),(2, 3, 4, 300, 100),(1, 2, 3, 100, 200)}. The result of PROJECTION(Product.ProductID, Customer.CustomerID, Store.StoreID, Units_Sold) is {(1, 2, 5, 200),(2, 3, 4, 300),(1, 2, 3, 100)}.

3. **If the feature attributes of view V are greater in number than the list (L) of dimension attributes of PROJECTION(L,M)V,** then we create an in-memory hash table data structure that allows for efficient searching for a key value; however, extensions to more scalable data structures such as balanced search trees (i.e., AVL tree), are expected to be used in the future. The search key for the hash table is the list (L) of dimension attributes, while accumulated (i.e., sum function) values for each measure attribute in M are associated with their dimensions' members (search keys). Algorithm 14 illustrates the implementation of the PROJECTION operation that requires an internal hash table data structure. Algorithm 14 is used as an implementation of the physical operator **hashTable_PROJECTION**(L,M)V in the final physical plan. If the input cube of the PROJECTION is a stored indexed view, then instead of reading one cell at a time in Algorithm 14, we can read one data block at a time and then process each cell in turn. For each cell (record) r, we use the hash table in order to decide whether:

   a. No search key equals the values of the dimensions' attributes of r; therefore, we insert r into the hash table as a new entry.

   b. A search key (k) equals the values of the dimensions' attributes of r. Thus, we use the values of the measure attributes of r to accumulate the values of the measure attributes associated with key k. Recall that our OLAP server supports only measure attributes that can be aggregated with the sum function. The hash table in Algorithm 14 is implemented as an array of size N, where N is the cardinality product of the cube result and the hash function that was explained in subsection 5.9.1. For each cell c(fV, aM), where fV is the list of values of the feature attributes and aM is the list of values of the measure attributes of c, the algorithm invokes the function **hash_function**() to retreive the hash value hv of the key fV. From this point, it uses the function **get(fV)** to verify if the hash table array entry at position hv has a valid entry. If this is true, one can just update the values of the measure attributes of the entry at position hv with the values of the measure attributes of aM. From this, if the hash table array does not have an entry with hv, then one must use the put function to insert a new entry (fV, aM) in the hash table array at position hv.

Algorithm 12 View Manager Algorithm

```
Input: A set of feature and measure attributes ($L$) within the PROJECTION operator, a cube $C$ and the View Manager $vM$.
Output: Best view that can answer the PROJECTION operation with attributes $L$.
1: Retrieve the actual view $V$ from the view manager ($vM$), where $V$ contains the set of attributes $T$, such that $L \subset T$.
2: return $V$.
```

Algorithm 13 Drop the useless measure attribute(s) from view V

```
Input: Useless measure attribute(s) $M$ and a view/group-by $C$.
Output: All cells of view $V$ without attribute(s) $M$.
1: for each cell $c(fA, MA)$ in view $V$, where $mA$ is the list of measure attributes in V do
2:     $rc = c(fA, mA - M)$, (cell $c$ without the measure attribute(s) $M$)
3:     return $rc$
4: end for
```

Algorithm 14 Hash Table Projection algorithm

Input: A view $V(fA, mA)$, where fA and mA are lists of feature and measure attributes in V, a list of feature attributes L and measure attributes M that are specified within the PROJECTION, and an array CF that contains the cardinalities of all feature attributes in L.

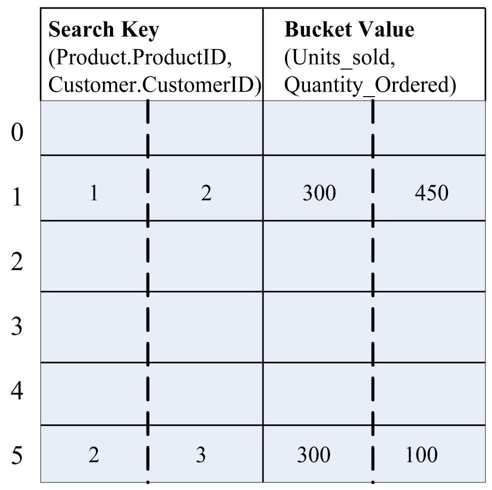
Output: An Array A containing the result of PROJECTION(L, M)V

```
1:  Create an empty Array $A$ with size $N$. $N$ is the cardinality product of the feature
    attributes ($L$). Array $A$ store entries as ($L$, $M$)
2:  Create an empty integer array $C$ of size $|L|$, where $|L|$ is the number of feature
    attributes specified in the PROJECTION
3:  Create an empty integer array $aM$ of size $|M|$, where $|M|$ is the number of distinct
    measure attributes mentioned in the PROJECTION
4:  Create an empty integer array tmpM of size $|M|$
5:  Create an empty integer array fV of size $|L|$
6:  $C[\ |L|\ ] = 1$
7:  for $i = |L|$ - 1 to 1 do
8:      $C[i] = C[i{+}1] * CF[i]$
9:  end for
10: for each cell $c$ of view $V$ do
11:     $fV$ = feature attributes values of cell $c$
12:     $hv$ = hash_function($L$, $C$, $fV$)
13:     $aM$ = measure attribute values of cell $c$
14:     $tmpM = A[hv]$.get($fV$)
15:     if ($tmpM$ is null) then
16:         $A[hv]$.put($fV, aM$)
17:     else
18:         for $i$ = 0 to —$M$— do
19:             $tmpM[i] = tmpM[i] + aM[i]$
20:         end for
21:         $A[hv]$.put($fV$, $tmpM$)
22:     end if
23: end for
24: return $A$.entries()
```

To illustrate how Algorithm 14 functions, consider the view from step (2) outlined above, where V(Product, ProductID) = 2 and V(Customer, CustomerID) = 3, V(Store,StoreID) = 8. Figure 5.25 shows a hash table array of size 6 (cardinality product of (ProductID,CustomerID)) which represents the output of PROJECTION(Product.ProductID, Customer.CustomerID, Units_Sold, Quantity_Ordered). We can derive from this figure that the hash value of key (1,2), where 1 and 2 are ProductID and CustomerID values respectively, is 1, while the hash value of key (2,3) is 5. Also, we notice that the values of the measure attributes (Units_Sold, Quantity_Ordered) associated with key (1,2) are calculated from the following two cells of the input view: (1, 2, 5, 200, 250) and (1,2,3, 100,200). The number of disk I/Os required for the PROJECTION that is described in Algorithm 14 is D(V) I/O. As was illustrated, the data blocks are stored sequentially in linear ordering. Thus, the worst case I/O running performance is equivalent to the time required to sequentially scan all data blocks in cube V. The worst case processor running time is O(n), where n is the total number of cells of the input cube V. As was discussed above, the worst case running time to search our hash table with n elements is O(n). Note that the worst case processor running time would be O(n log(n)) if the AVL balanced search tree is used instead of the hash table.

5.9.4 Hash Table Based Algorithms for Binary Operations

In this section, we discuss the algorithms for the binary operations: UNION, INTERSECTION, DIFFERENCE and DRILL_ACROSS. Previously, we showed in our OLAP query grammar that these operations require two operands that are themselves the results of two OLAP operations (such as PROJECTION, CHANGE_LEVEL, etc.). Also, these operations require one of the operands (result cube) to be in the



	Search Key (Product.ProductID,	Customer.CustomerID)	Bucket Value (Units_sold,	Quantity_Ordered)
0				
1	1	2	300	450
2				
3				
4				
5	2	3	300	100

Figure 5.25: Hash Table Array stores the result of the PROJECTION operation.

main memory. As in the previous section, a hash table data structure is used to allow for the efficient searching and inserting of cube cells. In our implementation, we assume that the left operand called C1(L,M), where L is a list of feature attributes and M is a list of measure attributes, is stored in a hash table data structure where the search key value is the combined values of the feature attributes of L and the associated data value is the combined values of the measure attributes (M).

5.9.4.1 UNION

We read the right operand cube called C2(L,M) from the main memory one cell (record/cell/tuple) r at a time. We see if the values of the feature attributes of r are present in the hash table, and if not, we insert r into the hash table. If r is in the hash table, use the values of the measure attributes of r to accumulate the required measure attributes. The result of this operation will be the values (L,M) of all search keys and their associated data in the hash table after the last cell of the right argument has been seen.

5.9.4.2 INTERSECTION

For each cell (record) r of the right operand, we search the hash table to see if the values of the feature attributes of r are found in an entry k of the hash table. If so, cell (r or k) with the smaller values of the measure attributes is stored as the output of this operation, and if not, we ignore r.

5.9.4.3 DIFFERENCE

Again, we read each cell (record) r of the right operand. We verify if the hash table has an entry e where its search key value equals the values of the feature attributes of r. If so, remove e from the hash table if its measure attribute (M) values are less than or equal to those of r(M). Furthermore, we can also subtract the values of the measure attributes of e from the values of the measure attributes of r. If r is not present in the hash table we are not required to do anything. Finally, after reading all cells of the right operand, the output of the difference operation is the content of the hash table (search keys + data).

5.9.4.4 DRILL_ACROSS

The measure attributes of both operands are different, as such, the data of the hash table contains two different sets of measure attributes. For each cell r of the right operand, we see if a search key of value k in the hash table is equal to the values of the feature attributes of r. If so, we update the hash table so that the bucket of k now contains its measure values and the measure values of r. If r is not in the hash table, then we insert r as a new entry into it. After the operation has been performed, the result is the content of the hash table.

5.9.4.5 Cost of Binary operations

No disk I/O is required for the binary operations since both operands (C1 and C2) are found in the main memory. However, if we allow both operands (C1 and C2) to be on the disk, then the number of disk I/O required is D(C1) + D(C2) I/O. In other words, the time performance in terms of disk I/O is equivalent to the sequential scan of the data blocks of C1 added to the sequential scan of the data blocks of C2.

The worst case processor running time for each one of the above binary operations is bounded as O(n), where n is the number of cells of the right operand. The worst case running time to find a key in the hash table of m search keys is O(1). For each of the n records of the right operand, we must verify the hash table for equality. Therefore, the worst case running time is O(1) * O(n) = O(n). However, if we use the AVL balanced tree as our internal structure in order to support the binary operations, the worst processor running time would be O(n * log(n)).

We shall not give an algorithm for each binary operation. Rather, we provide a representative example by presenting an algorithm that implements the UNION operation. Algorithm 15 defines the method. The input of this algorithm is a hash table array (A) that contains cells of the right argument view. This algorithm is executed by our OLAP query engine when a hashTable_UNION() physical operation is required. Other binary operations can be implemented in the same manner.

5.9.5 Algorithms for CHANGE_LEVEL Operation

Each one of these operations — CHANGE_LEVEL and CHANGE_BASE — is only relevant as a query against an existing set. Therefore, we assume that the input of these operations is a result cube that is housed in the main memory. Applying the

Algorithm 15 Hash Table UNION implementation

Input: A right view $RV(fA, MA)$, a hash table array A of size N (N is the cardinality product of feature attributes fA) which contains all cells of the left view LV, and an array (CF) of cardinalities of all feature attributes fA.

Output: A hash table array A that has the result of (LV UNION RV)

```
1: Create an empty integer array $C$ of size $|fA|$, where $|fA|$ is the number of feature
   attributes in the result of the operation (UNION)
2: Create an empty integer array $aM$ of size $|MA|$, where $|MA|$ is the number of
   distinct measure attributes of the result
3: Create an empty integer array $tmpM$ of size $|MA|$
4: Create an empty integer array $fV$ of size $|fA|$
5: $C[\ |fA|\ ] = 1$
6: for $i = |fA|$ -1 to 1 do
7:    $C[i] = C[i+1] * CF[i]$
8: end for
9: for each cell $c$ of view $RV$ do
10:    $fV$ = feature attributes values of cell $c$
11:    $hv$ = hash_function$(fA, C, fV)$
12:    $aM$ = measure attribute values of cell $c$
13:    $tmpM = A[hv]$.get$(fV)$
14:    if ($tmpM$ is null) then
15:       $A[hv]$.put$(fV, aM)$
16:    else
17:       for $i$=0 to $|MA|$ do
18:          $tmpM[i] = tmpM[i] + aM[i]$
19:       end for
20:       $A[hv]$.put$(fV, tmpM)$
21:    end if
22: end for
23: return $A$.entries()
```

merging laws (LAW_11) ensures that the CHANGE_LEVEL operator in the preferred OLAP query plan has a CHANGE_LEVEL(s) that corresponds to a Roll-Up operation (see Figure 5.23). This changes from the most detailed level values to other attribute level values. The implementation of this CHANGE_LEVEL (Roll-Up) is accomplished by using the enhanced Sidera mapGraph outlined in Section 3.4, and an in-memory hash table structure. First, the enhanced mapGraph is used to perform the following translation: (i) mapping from the most detailed encoded level value (base level attribute) to the corresponding sub-attribute linear encoded value and; (ii) mapping from a level encoded value to its real value. Second, the in-memory hash table is used to store the result in sorted order according to the order of the output dimension attributes.

Algorithm 16 provides the implementation of the CHANGE_LEVEL operation. We start by initializing some arrays that support the algorithm. Then, for each cell in the input cube, we use the mapGraph method (**mapGraph.get_encoding_value()**) to retrieve the encoded value for each output hierarchy level attribute. Finally, we use a function called **Get_RealValues()** to convert the cube result from its encoded values to its real values.

Algorithm 17 provides the implementation of the **Get_RealValues()** function. In this algorithm, we use the mapGraph hash table function (**mapGraph.get_RealValue**) that allows us to translate from the encoding level value to its real value. Specifically, we use the hash table associated with each hierarchical attribute level to convert in O(1) time an encoded/linear integer value on that level to its real value.

For example, consider the two dimensional cube (Sales) of Figure 4.24 and the hierarchy manager of the Product dimension in Figure 3.10. The following OLAP

Algorithm 16 CHANGE_LEVEL Algorithm

Input: An input cube $R(fA, MA)$, An array (CF) contains the cardinalities of attributes in the result of the CHANGE_LEVEL, Sidera's enhanced hierarchy manager (*mapGraph*), a list oA that include the output attributes of the operation (CHANGE_LEVEL), and two lists of attributes LI and LO mentioned within the CHANGE_LEVEL as $(LI \rightarrow LO)$.

Output: The result of CHANGE_LEVEL in a hash table structure

```
1: Create an empty hash table array $A$ with size $N$, where $N$ is the cardinality
   product of the output cube ($N = \prod_{1 \leq i \leq n} (CF_i)$ ), where n is the number of attributes
   in $oA$. Entry of $A$ is of the form $(oA, MA)$.
2: Create two empty integer arrays $rA$ and $C$ of size $n$.
3: Create two empty integer arrays $aM$ and $tmpM$ of size $m$, where $m$ is the number
   of measure attributes the input cube $R$.
4: Create an empty integer array $fV$ of size $k$, where $k$ is the number of feature
   attributes in $R$.
5: $C[n] = 1$
6: for $i = n$ - 1 to 1 do
7:    $C[i] = C[i+1] * CF[i]$
8: end for
9: for each cell $c$ in view $R$ do
10:    $fV$ = feature attributes values of cell $c$
11:    $aM$ = measure attribute values of cell $c$
12:    for $i = 1$ to $n$ do
13:       if $oA[i]$ equals to $fA[j]$, $j$= 1, ..., $k$ then
14:          $rA[i] = fV[j]$
15:       else
16:          $rA[i] = mapGraph.$get_encoding_value$(fA[j], fV[j], oA[i])$
17:       end if
18:    end for
19:    $hv$ = hash_function$(oA, C, rA)$
20:    $tmpM = A[hv]$.get$(rA)$
21:    if ($tmpM$ is null) then
22:       $A[hv]$.put$(rA, aM)$
23:    else
24:       for $i = 0$ to $m$ do
25:          $tmpM[i] = tmpM[i] + aM[i]$
26:       end for
27:       $A[hv]$.put$(rA, tmpM)$
28:    end if
29: end for
30: $result$ = Get_RealValues$(A, oA, mapGraph)$
31: return $result$
```

Algorithm 17 Get_RealValues implementation

```
Input: A hierarchy manager ($mapGraph$), a hash table array ($A$) that contains the
    cube result in encoding/linear integer form, and $oA$ list of output attributes
Output: Array that contains the result in real/native forms
1: Create an empty array $oR$ of size $m$, where $m$ is the number of attributes in $oA$.
2: Create an empty array $resA$ to store the result in its real form.
3: for each entry $e(Attribute, Measure)$ $j$ of $A$.entries() do
4:     for $i = 1$ to $m$ do
5:         $oR[i] = mapGraph$.get_RealValue($e$.Attribute[$i$], $oA[i]$)
6:     end for
7:     $resA[j] = (oR, e.Measure)$
8: end for
9: return $resA$
```

expression query, referred to as the preferred logical query plan, allows us to display the total units sold grouped by product category:

$$\text{CHANGE\_LEVEL(Product.ProductID} \rightarrow \text{Product.Category)}$$
$$\text{(PROJECTION(Product.ProductID, Units\_Sold)) sale.}$$

The result of PROJECTION(Product.ProductID, Units_Sold))sale is shown in Figure 5.26(a) as a one dimensional cube. For example, we can see the total units sold for product 1 is 128. In addition, the CHANGE_LEVEL is answered by using the enhanced mapGraph of Figure 3.10. For example, ProductID 1, ..., 7 corresponds to product category 1(Automotive), while ProductID 8, ..., 11 corresponds to product category 2 (HouseHold). Figure 5.26(b) illustrates the result of the above CHANGE_LEVEL operation. Finally, we can use the Category hash table (see Figure 3.10) to convert the linear values 1 and 2 of the product category to Automotive and HouseHold respectively.

No disk I/O is required because the hierarchy manager is found in the main memory and the input cube as well. Although the original Sidera hierarchy manager

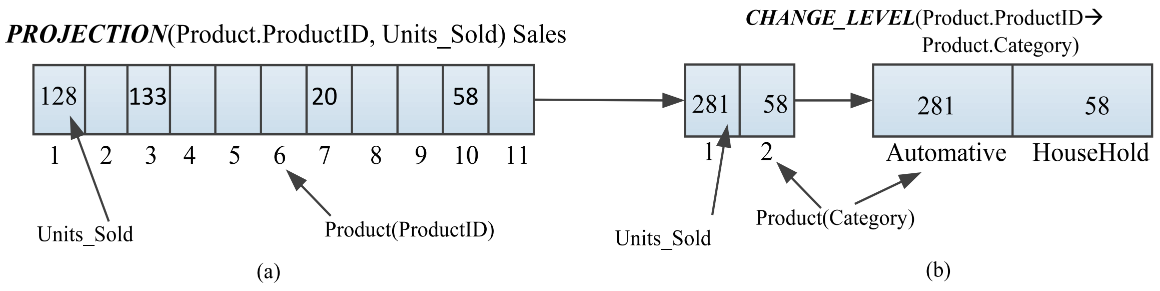


Figure 5.26: Result of PROJECTION. (b) Result of CHANGE_LEVEL.

supports various types of hierarchies such as ragged, simple, multiple, etc., we only consider in our research at this time the simple symmetric form of dimension hierarchy. Therefore, the worst case processor running time of the CHANGE_LEVEL in our logical query plan depends on the analysis of the enhanced hierarchy manager that represents the simple symmetric hierarchy [74, 34]. In other words, the worst processor running time is bounded as [*n * O(log(V(Dim.A))*], where n is the number of the base level that has to be transformed to the destination level attribute (A) in the dimension (Dim) and $V(dim.A)$ is the cardinality of the attribute A in the hierarchy dimension (dim).

Finally, it is important to mention that using Algorithm 16 without invoking the method (**Get_RealValues**) can be used as an implementation for any **mapGraph_HashTable_CHANGE_LEVEL()** physical operation in our physical plan. Algorithm 17 is an implementation for the physical operation (**Get_RealValues**) in our OLAP physical query plan.

5.10 The Sidera Server

As was mentioned in Section 2.4, the Sidera server is a high performance parallel ROLAP server. Figure 2.6 provided a simple depiction of the server architecture.

Note that the front end in Figure 2.6 is used to receive user queries that have a simple syntax specific to the Sidera server, to pass them along to the backend nodes for resolution, to collect the final result from the backend servers and to return it to the end-user. In turn, the Sidera backend nodes are fully responsible for query resolution and operate mostly independently. In Chapter 3, we described an efficient OLAP storage manager for the Sidera server. In other words, we explained how the data of the data warehouse (fact table, dimensions and cubes) is efficiently stored on disk and accessed quickly. Figure 3.20 illustrated how different Sidera components sit on top of Berkeley DB and FastBit in order to provide an efficient OLAP storage engine for the existing Sidera server. Note that the Query Processor component of Figure 3.20 is used to answer simple OLAP queries that have been hard-coded in a proprietary format.

In this chapter, we described a pure OLAP query processor (Compiler and Execution) for the Sidera server, which can be used to answer very complex OLAP queries received in XML format. Thus, we replace the old and very simple query processor with the new OLAP query processor. Figure 5.27 shows the new software model on each backend Sidera server after the integration of the new OLAP query processor. Since the new Sidera server needs the data of dimension tables in order to resolve real-world OLAP queries, the additional data is supported by the integration of new components (See Figure 5.27) into the backend Sidera server. In other words, the Bitmap Index Manager that represents the compressed bitmap indexes for the required non-hierarchical attributes are integrated on each backend node. Also, we can see the dimension table representation component that is responsible for encoding and indexing dimension tables (refer to Chapter 3).

Though our research in this thesis focuses on the components of the backend nodes, a brief description of how the query is executed in the parallel Sidera server will be presented in this section. In short, the new Sidera server is used to answer very complex user queries that first arrive in XML format on the front end node. The frontend node broadcasts the received XML OLAP query to the backend nodes. Each backend node operates independently to efficiently answer a portion of the received OLAP query from the OLAP storage data that is housed locally. Finally, a parallel service layer combines the local results and returns them to the user.

Front end processing proceeds as per Algorithm 18. Figure 5.28 provides an illustration of the frontend architecture and processing logic. For simplicity, we assume that the function of the frontend server is to receive only one OOP OLAP query in XML format at a time. However, from a practical perspective, the Sidera front end should receive several OLAP user queries at the same time. For this reason, the query is directly broadcasted to each of the backend Sidera processors, thereby avoiding unnecessary bottlenecks on the frontend.

Algorithm 18 Front End Sidera Server

```
1: receive an OLAP user query $Q$ in XML format
2: verify if $Q$ matches our DTD-encoding OLAP query grammar
3: if ill-formed syntax then
4:    error message to the user
5: end if
6: broadcast $Q$ to Sidera backend nodes
7: receive results into local buffers
8: create an XML-format $R$ for the result of $Q$
9: send $R$ to the user
```

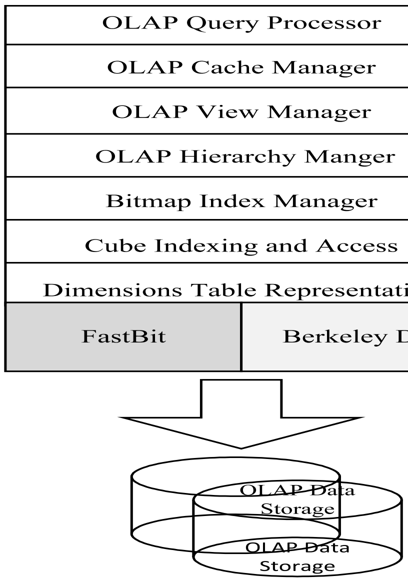


Figure 5.27: A block diagram of the software architecture on each Sidera Backend processing node.

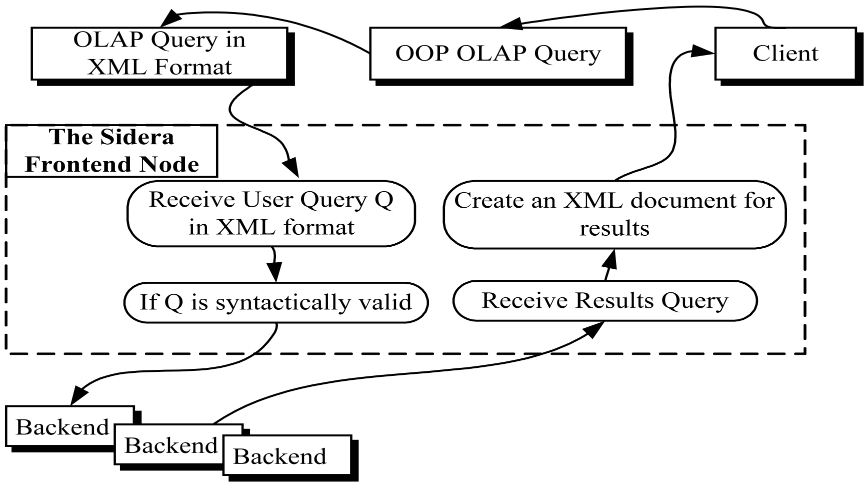


Figure 5.28: The Sidera frontend.

Algorithm 19 provides a high level description of the processing loop on the back-end server instances. A corresponding graphical depiction is provided in Figure 5.29. Specifically, it describes the query compiler and query engine modules used to process the user's OLAP query. To begin, each backend node receives the same OLAP query (uQ) from the front end. Then, the DOM parser creates the DOM tree (dT) for uQ if it matches our DTD-encoding OLAP query grammar, and if not, an error message is returned to the frontend node. Additionally, the Sidera parser turns the DOM tree dT into an internal parse tree (pT). The pre-processor component of our OLAP query compiler must be used at this step (step 9) to make sure that the received query is semantically valid. Specifically, it checks pT against the OLAP schema definition. Once this step is completed, the parse tree pT is converted into an OLAP expression tree (initial OLAP logical query plan) for our OLAP algebra (lQ). A preferred logical query plan (pLQ) is then constructed for the initial logical query plan (lQ) by applying several of the re-writing OLAP laws and techniques mentioned in Section 4.9. Next, pLQ must be turned into the most efficient physical plan (qP). Finally, the last step of the query compiler is to generate a sequence of n function calls (fC), one for each physical operation in qP.

After the query has been compiled, our OLAP query execution engine then executes the sequence of function calls by invoking the appropriate algorithms defined in Section 5.9 to answer the query. For example, if a function call is equivalent to the **bitMapAccess()** physical operation, then Algorithm 9 is invoked to answer this function call. In the previous section, we mentioned for each physical operation in our physical query plan its corresponding algorithm. After the execution of all functions, a Parallel Sample Sort is performed across the parallel machine to produce

Algorithm 19 Sidera Backend Query Resolution

```
 1: Receive the user's OLAP query ($uQ$) written in XML format from the frontend
 2: DOM Parser verifies if $uQ$ matches our OLAP query grammar
 3: **if** the syntax of $uQ$ is not valid **then**
 4:     Return error message
 5: **else**
 6:     DOM parser creates a DOM tree ($dT$) for $uQ$
 7: **end if**
 8: Sidera Parser creates a parse tree ($pT$) for $dT$
 9: Sidera Pre-Processor checks the semantic of $pT$
10: **if** the semantic of $pT$ is not valid **then**
11:     Return error message
12: **end if**
13: The initial logical query plan generator turns $pT$ into an initial OLAP algebraic
    expression tree ($lQ$).
14: The OLAP Query Optimizer must turn the initial OLAP logical plan ($lQ$) into
    the preferred logical query plan ($pLQ$).
15: The OLAP Query Optimizer transforms $pLQ$ into an OLAP physical query plan
    ($qP$).
16: The OLAP Query Optimizer generates a sequence of function calls ($fC$), one for
    each physical operation in $qP$. $fC$ has $n$ function calls $\{f_1, f_2, \ldots, f_n\}$.
17: Initialize the mapGraph Hierarchy Manager ($hM$) with the dimension hierarchy
    metadata.
18: Initialize the View Manager ($vM$) with the meta data about the physically stored
    views.
19: Initialize the bitmap index manager ($bI$) with the compressed bitmap indexes for
    all non-hierarchical attributes mentioned within the restriction of $uQ$.
20: **for** each function call $f_i$ in fC where $i$ = 1, ..., $n$ - 1 **do**
21:     $R_i$ = Invoke the appropriate algorithm(s) that implements $f_i$.
22:     Pass the result $R_i$ to the parent function $f_{i+1}$
23: **end for**
24: Do a parallel sort of $R_{n-1}$ across all backend Sidera nodes. Each backend node
    now contains a sorted result $sR$.
25: Invoke the function $f_n$ (**Get_realValues(L)**), result $R$ = **Get_RealValues(L)**
    $sR$, where $L$ is the list of dimension attributes in the output of the $uQ$.
26: Return result $R$ to the frontend (collect $R$ with **MPI_Allgather()**).
```

a distributed data set result that is now fully sorted across the backend nodes. At the conclusion of this phase, each node houses a segment of the final result in integer form. Moreover, the n records of the final result are partitioned across the p nodes of the network such that for records $\{i, j\}$, $1 < i < j < n$, and with locations $\{i(m), j(n)\}$, $1 < m < n < p$, we are guaranteed $m < n$. The last step of our query engine processing cycle is to invoke the **Get_RealValues()** function in order to convert the partial results on each backend from their encoded form (integer values) to their native form. Finally, because Sidera is a fully parallelized OLAP query engine, the partial results are returned to the frontend buffers. We use a standard Gather operation from the MPI libraries (Message Passing Interface). Note that the numbered sequence in Figure 5.29 indicates the processing cycle for an OLAP query. There are no numbers between the OLAP query execution component and the components (such as hierarchy manager, view manager, etc.) since these can be accessed any time during the execution of the physical plan. However, Number (8) indicates the execution of the function at the root of the physical plan (**Get_RealValues()**) after the parallel sample sort.

5.11 Result Sets

Eventually, after the query has been resolved, the final query results are collected and merged into one result buffer on the frontend node. Note that the Sidera client side transforms the query result received from the frontend into a multi-dimensional object that can be directly accessed. Therefore, the frontend must construct the query result set in a way that allows the client side to efficiently transform the result into a multi-dimensional array object. The frontend node packages the result into an XML

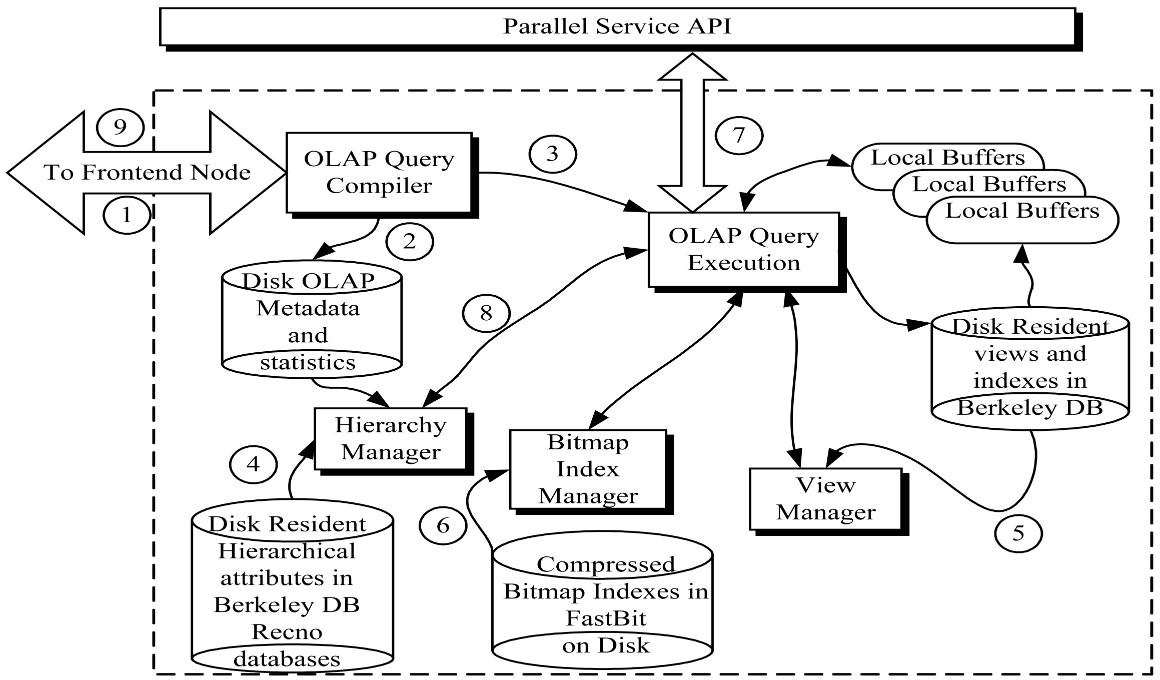

Parallel Service API
To Frontend Node
9
1
OLAP Query Compiler
3
2
7
Local Buffers
Local Buffers
Local Buffers
OLAP Query Execution
Disk OLAP Metadata and statistics
8
Disk Resident views and indexes in Berkeley DB
Hierarchy Manager
4
Bitmap Index Manager
View Manager
5
6
Disk Resident Hierarchical attributes in Berkeley DB Recno databases
Compressed Bitmap Indexes in FastBit on Disk

Figure 5.29: The Sidera Backend Node.

message. A DTD called *OlapResultSet* is used to define the format of the query result. A partial listing of the DTD is provided in Figure 5.30. In short, the *OlapResultSet* is structured as a combination of metadata and cell data. The metadata consists of the relevant dimensions, along with those dimension members actually included in the query result. The cell data, on the other hand, is listed in a compressed row format that maps cell values to the corresponding axis coordinates.

Figure 5.31 provides a partial representation of a simple result set. Note how each customer member is associated with a monotonically increasing Member ID, starting from zero. In the Raw Data section of file, we can see how each cell value is associated with the coordinates of three dimensions. The first row, for example, houses the values <0, 1, 2, 345.24>. Assuming that Customer is the first dimension in the meta data list, this implies that the cell value 345.24 is associated with Customer[0] = Joe.

```
<!ELEMENT RESULT CUBE (META DATA, RAWDATA)>

<!ELEMENT METADATA (CUBE NAME, DIM COUNT, DIMENSION LIST)>

<!ELEMENT DIMENSION LIST (DIMENSION+)>

<!ELEMENT CUBE NAME (#PCDATA)>
<!ELEMENT DIM COUNT (#PCDATA)>

<!ELEMENT DIMENSION (DIM NAME, MEMBER LIST)>
<!ELEMENT DIMNAME (#PCDATA)>

<!ELEMENT MEMBER LIST (MEMBER+)>
<!ELEMENT MEMBER (MEMBERNAME, MEMBER ID)>
<!ELEMENT MEMBERNAME (#PCDATA)>
<!ELEMENT MEMBER ID (#PCDATA)>

<!ELEMENT RAWDATA (ROW+)>
<!ELEMENT ROW ( ID LIST , VALUE)>
<!ELEMENT ID LIST (MEMBER ID+)>
<!ELEMENT VALUE (#PCDATA)>
```

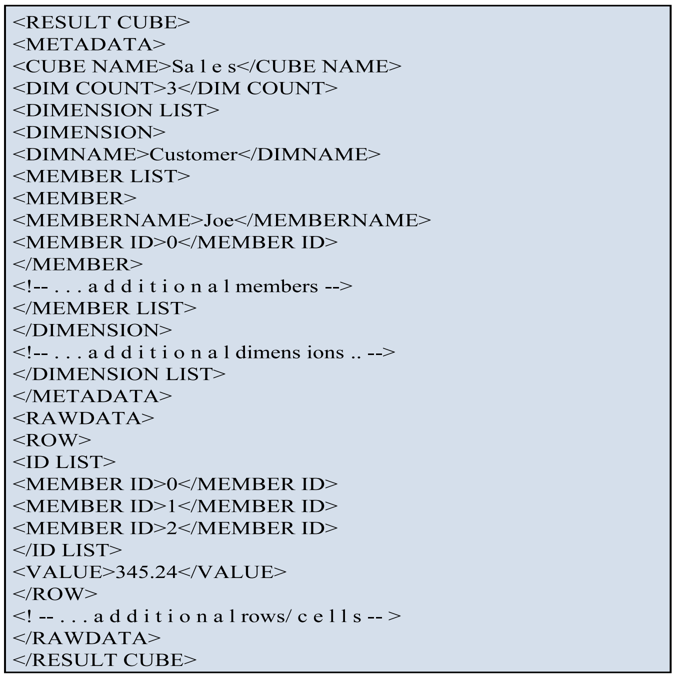
Figure 5.30: Result Set DTD Grammar.

We note that regardless of the storage format of the server (ROLAP, MOLAP, or otherwise), this XML is trivial to produce with a simple linear pass through the result. Once the XML result is received at the client side, it is immediately transformed into a multidimensional object.

5.12 Review of Research Objectives

In this section, we review the objectives that were identified in Section 5.3. In other words, we now check if those goals have in fact been achieved.

1. **Parse the received OOP OLAP query written in XML format such that an internal parse tree is created if the query is syntactically and semantically valid.** This goal is achieved by using two levels of parsing. First, a DOM parser was used to parse the received XML document, to check its syntax



```
<RESULT CUBE>
<METADATA>
<CUBE NAME>Sa l e s</CUBE NAME>
<DIM COUNT>3</DIM COUNT>
<DIMENSION LIST>
<DIMENSION>
<DIMNAME>Customer</DIMNAME>
<MEMBER LIST>
<MEMBER>
<MEMBERNAME>Joe</MEMBERNAME>
<MEMBER ID>0</MEMBER ID>
</MEMBER>
<!-- . . . a d d i t i o n a l members -->
</MEMBER LIST>
</DIMENSION>
<!-- . . . a d d i t i o n a l dimens ions .. -->
</DIMENSION LIST>
</METADATA>
<RAWDATA>
<ROW>
<ID LIST>
<MEMBER ID>0</MEMBER ID>
<MEMBER ID>1</MEMBER ID>
<MEMBER ID>2</MEMBER ID>
</ID LIST>
<VALUE>345.24</VALUE>
</ROW>
<! -- . . . a d d i t i o n a l rows/ c e l l s -- >
</RAWDATA>
</RESULT CUBE>
```

Figure 5.31: Result sent from the frontend to the client side as an XML message.

against our query grammar and to create the corresponding DOM graph/tree. Second, the Sidera parser was utilized to translate the DOM graph/tree into an internal parse tree and to check if the received query is semantically valid.

2. **Provide an initial OLAP logical query plan that can be easily optimized later.** The user's OOP OLAP query is translated internally into an OLAP algebra expression tree that consists of our OLAP algebraic operators. As outlined in Section 5.6, the main advantage of using a comprehensive OLAP algebra is that it makes alternative forms of an OLAP query easier to explore, manipulate, and subsequently optimize.

3. **Produce an OLAP logical query plan that requires less time to execute.** This goal is achieved by applying the OLAP algebraic laws outlined in

Section 5.7 to the initial OLAP logical query plan in order to produce a better plan. Note that “better” means it is likely to require less time to execute and possibly use less memory if intermediate results are smaller. Experimentally, we will validate this goal in the next chapter. Specifically, we will compare the execution running time of OLAP queries before and after applying some of our OLAP algebraic laws.

4. **Pick the OLAP physical query plan with the least estimated cost.** We propose an OLAP cost model (similar to those used in relational database) in order to estimate the cost of a physical plan. The cost is estimated in terms of the number of disk I/Os and, if necessary, the processor running time. We expect that the physical query plan with the least estimated cost should require the least actual execution time and should be selected by our optimizer. In the next chapter, experiments will be conducted to validate the correctness of our OLAP cost model.

5. **Efficient algorithms for implementation of the operations of our OLAP algebra.** In Section 5.9, we proposed a number of algorithms for efficient implementation of the operations of our OLAP algebra. In other words, we defined how we should execute each of the individual steps of an OLAP logical query plan (e.g., SELECTION, PROJECTION). These algorithms are divided into index-based (i.e., SELECTION) and sequential-based (e.g., PROJECTION) methods. Moreover, we proposed in-memory data structures (such as hash tables, Bitmap Index Manager, etc.) that support efficient execution of OLAP operations.

6. **Allow for the simple manipulation of the cube results.** The frontend node packages the result into an XML message and returns it to the client side. This XML document can be efficiently translated into multi-dimensional OLAP cube results that can be directly and easily accessed by the end-users.

5.13 Conclusions

In this chapter, we have presented a pure OLAP query processor component for compiling and resolving Object-Oriented OLAP queries in the Sidera server. The processor is divided into two main components: an OLAP query compiler and an OLAP query execution engine.

For query compilation purposes, we use a process similar to that of traditional relational database systems in that we use two main approaches to optimize queries (rule-based and cost-based). In our OLAP environment, an internal OLAP parse tree representing the user's OLAP query is created. The query compiler employs an OLAP optimizer that is based upon two mechanisms (i) OLAP multi-dimensional rules and (ii) an OLAP cost model. We have showed that our multi-dimensional rules-based engine (Section 4.9) can be applied to multidimensional databases by rewriting each OLAP query to obtain an efficient OLAP logical query plan. Again, these techniques are similar to the basic approaches taken in traditional DBMS systems; however, here they are specific to OLAP environments (cubes, views, hierarchies, cells, measures, etc.). The output of the OLAP query compiler is a set of OLAP physical operations that can be resolved by the OLAP query execution engine.

For query execution, we have defined a number of algorithms for execution of the operations of our OLAP algebra. These algorithms build upon the efficient OLAP

Sidera data storage and data structures (e.g., hierarchy manager) previously described in Chapter 3. Moreover, the query engine uses an in-memory hash table structure that allows efficient implementation of these algorithms. In the next chapter, we will discuss various experimental results that support the design decisions that we have made.

In addition, we have described in Section 5.10 how several components fit together to support the high performance parallel ROLAP server. The front end Sidera server receives and broadcasts a user's OLAP query, collects results from Sidera backend nodes, and subsequently returns the final result to the end user in an XML document. In turn, each backend node compiles and executes the received OLAP query on the portion of data that is housed on its disk and returns the results to the front end. We have illustrated how the final query result is returned to the end-user in XML format. This XML is constructed in a way that allows the client side to efficiently transform the result into a multi-dimensional array object that can be directly and easily accessed.

In summary, our OLAP query processor complements the efficient OLAP storage engine (Chapter 3) and the OLAP query grammar and algebra (Chapter 4) by providing the final piece that the Sidera DBMS requires in order to support high performance OLAP DBMS within the ROLAP environment.

Chapter 6

Experimental Results

6.1 Introduction

The end result of the previous three chapters is a comprehensive OLAP DBMS server that can be used to answer complex native-OOP OLAP queries in a very efficient way. Specifically, we developed a multi-dimensional algebra that is used by our OLAP query optimizer to optimize OLAP queries that match our robust OLAP grammar. In the current chapter, our focus is to check the effectiveness of our new methods and data structures. We provide experiments that demonstrate the importance of the application of our algebraic laws. Moreover, we show experimental results that assess the ability of our OLAP query engine to efficiently resolve native-OOP OLAP queries in a responsive manner. We will present a series of tests; each is designed to explore a particular feature of our OLAP query processor. In addition, we provide direct comparisons with commercial DBMS servers to highlight the viability of our prototype. Finally, we discuss parallel query resolution time in the Sidera server.

The chapter is organized as follows. In Section 6.2, we discuss the test environment as it relates to the hardware, software, and data that we use in our evaluation. Section 6.3 describes the current query processor components that have actually been implemented. In Section 6.4, we will look at a sequence of tests in order to highlight the importance of our query compiler and execution engine with respect to the resolution of OLAP queries. In Section 6.5, we compare our query engine against two well known enterprise database servers: (1) MySQL — currently the world's most popular open source relational database management system — and (2) Microsoft SQL Analysis services — currently the world's most popular commercial OLAP system. In Section 6.6, we will look at parallel query resolution as the number of processors increases. Section 6.7 is the conclusion of this chapter.

6.2 The Test Environment

In terms of the test platforms, the majority of these tests were conducted on a Linux-based workstation running a standard copy of the 2.6.x kernel, with 1GB of main memory and a pair of 3.2 GHz dual CPU boards. Disks are 160 GB drives running at 7200 RPM. Moreover, one additional but important test focuses on the comparison between Microsoft SQL server (Microsoft Analysis Services) and our OLAP server (Sidera). Since Microsoft SQL server 2008 can be installed on a windows-based machine only, we therefore use a dual-boot, user managed machine (windows and Linux), with one GB of main memory and a pair of 2.6 GHZ dual CPU processors. In the experiments that follow, we make sure that all comparisons are conducted on the same hardware resources.

All software components of our server have been implemented using C++, STL (the Standard Template Library) and the MPI communication libraries. In addition to the integration of the source codes of the Berkeley DB and the FastBit core with the Sidera server components (mapGraph, ViewManager, cube indexing, etc.), our

OLAP query processor (except the optimizer) was fully implemented in C++ and replaced the old Sidera query processor.

In terms of dimensions, we create six dimensions as depicted in Table 6.1. The underlined attributes in Table 6.1 are the primary keys (e.g., CustID, ProdID). The table shows simple metadata for each dimension as follows:

1. The dimension name (e.g., Customer, Product).
2. The dimension attributes (e.g., CustID, Region, Age)
3. The number of records.
4. The dimension's hierarchy (i.e., CustID is the base attribute).

Data for each dimension was generated using an open source data generator called Spawner [108]. It is a tool for generating suitable test data for populating databases. Specifically, the schemas were defined and then Spawner was utilized to generate the appropriate number of records for each dimension. Moreover, we supply Spawner with text files (i.e., a country file that contains the name of all countries) to ensure that the data of dimensions makes sense (e.g., Quebec is a province in Canada and not in USA). All dimensions were encoded and physically stored on disk. As was discussed in Section 3.4, a number of extra attributes must be added to the encoded dimension (one for each hierarchical attribute level). We use the following conventions for naming those extra attributes: (i) the name of the encoded primary key (i.e. the name of the most detailed encoded level) is constructed as the first letter of the dimension name, "_", and ID (for example, in dimension Customer the name of the encoded primary key is C_ID, while P_ID is used in dimension Product), and (ii) the name

Dimension Name	**Attributes**	**Number of Records**	**Hierarchy Description**
Customer	(CustID, Name, Age, Country, Region)	One Million	CustID → Region → Country
Product	(ProdID, Quantity, ProdDesc, Category, Type)	200,000	ProdID → Type → Category
Time	(Day, DayName, DayOfWeek, Year, Quarter, Month)	3650	Day → Month → Quarter → Year
Store	(StoreID, StoreName, StoreState, StoreCity, StoreCountry)	655	StoreID → StoreCity → StoreState → StoreCountry
Vendor	(VendorNumber, VendorName, Phone, CountryName, StateName, City)	416	VendorNumber → City → StateName → CountryName
Employee	(SIN, FirstName, LastName, Phone, Email)	300	No Hierarchy

Table 6.1: Dimension Tables.

of the encoded hierarchical sub-attribute is the attribute name concatenated with "ID" (e.g., RegionID, CountryID). Hierarchical attributes were stored in Berkeley DB databases with the Recno access method (e.g., the Region attribute in dimension Customer). Finally, FastBit was used to create compressed bitmap indexes, one for each non-hierarchical attribute (e.g., Age in dimension Customer, Quantity in dimension Product).

Regarding the fact table, we generate a six-dimensional fact table (the dimension count varies with some tests), with cardinalities chosen to be equivalent to the cardinalities of the corresponding dimensions (1000000, 200000, 3650, etc.) in Table 6.1. Data sets are generated using a custom data generator developed specifically for our

OLAP environment. The distributions of data sets are not skewed. Data values are randomly generated and uniformly distributed. We note that while skewed data is important in the cube generation, it has little impact on query resolution evaluation. Depending on the test involved, row counts typically vary from 100,000 records to 10,000,000 records. The primary fact table is then used to compute a fully materialized data cube containing hundreds of additional views or cuboids. Once the cube is materialized, we index the views using the R-tree mechanism provided by the Sidera server and store the index cube in Berkeley DB. The schema of the fact table (feature attributes and measure attributes) along with the schemas of its associated dimensions (dimension name, hierarchies, etc.) is stored in Berkeley DB XML. Note that the format of the schema matches the grammar defined in Section 4.8.

In terms of OLAP queries, there is no standard business oriented ad-hoc OLAP query generator (something similar to TPC-H that is used for OLTP systems) that can be used to evaluate the performance of a database server [110]. We must therefore develop our own query framework. As illustrated in Figure 5.28, our OLAP server receives the user's OOP query in XML format where it must be parsed, optimized and executed. To carry out the experiments, we require batches of OLAP queries written in XML format. Moreover, each of these OLAP queries must be translated by hand into the appropriate SQL syntax and MDX-syntax so that they can be executed on MySQL server and the SQL Server DBMS in order to conduct the comparison with those database servers. Because of the complexity of constructing OLAP queries in XML format and the labour-intensive nature of translating them into SQL and MDX syntax, and finally, the fact that no standard set of OLAP queries exists, we are limited in terms of the size of the query batches that we use. Specifically, we rely on batches of 10 to 20 OLAP queries per test. Appendix A and B illustrate the SQL-syntax of OLAP queries used to evaluate our OLAP server. We note that these OLAP queries are divided into queries on low-dimensional views (less than four dimensions) that can be easily visualized and high-dimensional views (views with more than 3 dimensions). Recall that our OLAP server supports the simplest (and most common) data warehouse schema (star schema); therefore, queries that are used in our experimental results are of the form of star queries. In the star query, one table (fact table or view) serves as the central table and all other tables (dimensions) are joined to this table. Finally, we note that the "drop_caches" option available in the newer Linux kernels was used to delete the OS page cache between the executions of each query.

6.3 Current Query Processor Implementation

Before digging in to the details of the experimental results, it is useful to first provide an overview of the components (compiler elements and execution elements) that have actually been implemented for the OLAP query processor that was discussed in Chapter 5. The following is the list of query processor elements that are fully implemented at the current time:

1. **XML Query Parser:** We have used the Xerces-C++ to create the DOM/-graph and check the syntax of the received OLAP XML query. Moreover, we can convert this DOM graph into an internal parse tree.

2. **Pre-processor:** The semantic checking element is fully implemented and used to make sure that the received queries are semantically valid.

3. **Initial Logical Plan:** Our current query compiler is able to convert the validated query into an initial logical plan of OLAP logical operators.

4. **SELECTION Algorithms:** This operation was fully implemented in our server. In other words, we have implemented Algorithms 8, 9, 10 and 11 which, as a group, can be used to process arbitrary OLAP query restrictions.

5. **PROJECTION Algorithms:** This operation is also fully implemented in our engine. Specifically, we have implemented Algorithms 12, 13 and 14 that provide different implementations of this operation, depending upon its location in the query.

6. **INTERSECTION Algorithm:** This operation is fully implemented.

7. **CHANGE_LEVEL Algorithm:** Algorithm 16 and 17 were also implemented to change the base encoded level values to sub-attribute native values.

For the query compiler, our logical (Section 5.7) and physical query (Section 5.8) optimizations are not yet fully implemented due to the extensive implementation effort required to produce a robust query optimizer. In the subsequent sections, we manually provide various tests to check the effectiveness of our query optimization policies. For example, we provide the engine with optimized and non-optimized queries. Moreover, in order to check our cost model, we create several physical plans that are sent to the engine and subsequently report their running times.

For query execution, future projects will include implementation for other binary operations such as DRILL_ACROSS, UNION, and DIFFERENCE. Also, we will eventually require implementation of the balanced search tree (instead of the hash table) that can support larger intermediate cubes. Finally, the query result set is currently returned as a simple array; however, this should be as an XML message in the future.

6.4 Single Node Experimental Evaluation

Though Sidera (our OLAP server) is a fully parallelized system, we have used a single node test environment in this and the next section for two reasons. First, the components on each node — see Figure 5.29 (OLAP query optimizer, execution engine, mapGraph, bitmap index manager, etc.) — of the parallel server are designed to work independently. Second, it is far more convenient to conduct single node tests against the commercial DBMS (such as MySQL DBMS and Microsoft Analysis server). We will now look at a series of results that show the importance of our multi-dimensional OLAP query processor (optimization and execution) in terms of query resolution. Specifically, we discuss the running time of OLAP queries before and after using our OLAP techniques (OLAP algebraic laws and physical plan).

6.4.1 Query Engine with and without LAW_5

Since the base cuboid of a cube contains all dimensions, it can be used to answer any OLAP query against the cube. Recall that the purpose of LAW_5 is to select the best view to answer the current OLAP query. In the absence of LAW_5, our OLAP query engine utilizes the base cuboid to answer a given OLAP query. In this section, we discuss the performance of our OLAP query engine with and without LAW_5 as the number of records in the fact table increases and also as the number of dimensions changes.

Figure 6.1 illustrates the performance of our query engine as the number of records in the fact table varies from (100K, 1M and 10M records) with the same set of dimensions (6 dimensions). In terms of OLAP queries, we create 16 OLAP queries in XML format which match our OLAP query grammar. Appendix B illustrates the SQL-syntax of those 16 OLAP queries used in this section. In Figure 6.1, we can observe that the running time increases dramatically as the size of the data cube increases. Again, this is because the base cuboid is by far the largest view amongst the cuboids in the cube. Using LAW_5, the same views are selected to answer the query at each fact table size; however, they have different sizes because the size of the base cuboid (fact table) varies. We can see the importance of LAW_5 when the number of records increases in the fact table. For example, the running time without this law increases by a factor 1 to 3 when the number of records is less than 1 million records; however, this factor increases to 10 when the number of records is 10 million.

Figure 6.2 illustrates the performance of the query engine (with and without LAW_5) as the number of dimensions that surround the fact table increases. Figure 6.2 shows the running time of 12 OLAP queries in XML format for three data cube density levels. All data cubes have the same number of records in the base cuboid (1 Million records), but they differ in the number of dimensions (4, 6 and 8). In terms of OLAP queries, SQL queries in Appendix A illustrate the equivalence of the OLAP queries used in this test. Note that only four dimensions and the fact table are involved in those queries (Customer, Product, Time and Store). In Figure 6.2, using LAW_5 in our server ensures that the running time of the same set of queries remains the same as the number of dimensions increases in the cube. This is due to the fact that the same views are selected for the same queries. However, in the absence of LAW_5, we can observe that the resolution time for the same set of queries

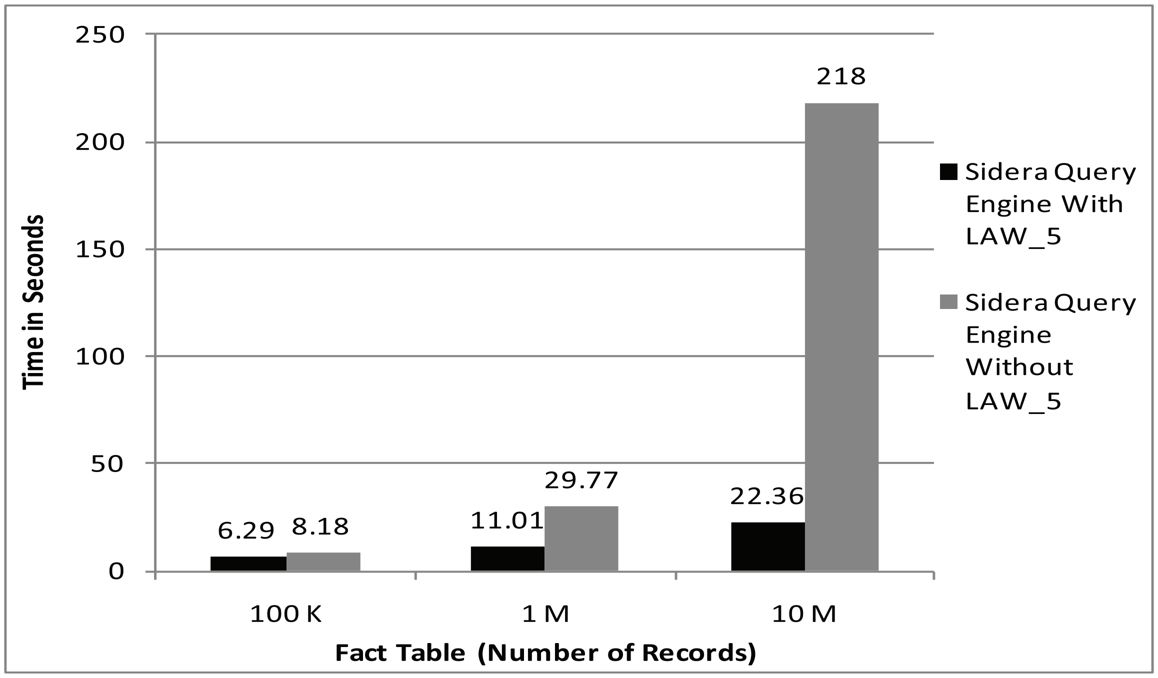


Figure 6.1: Query running time with and without LAW_5 as the number of records in the data cube increases.

is increased as the number of dimensions in the cube is increased, because the number of dimensions in the base cuboid is increased. Moreover, the running time without LAW_5 is increased by a factor of 10% to 20% as the number of dimensions increases; however, it is very likely to be the same when using LAW_5.

6.4.2 Sidera Query Engine with and without LAW_2, LAW_4 and LAW_13

In this test, we create four different OLAP queries that can be optimized by the following algebraic laws: Law_4, LAW_2 and LAW_13. In other words, each query has the INTERESECTION operator between two OLAP queries, which can be optimized in four steps. First, our optimizer uses LAW_4 to pull the SELECTION up. Second, if necessary, LAW_2 is utilized to combine many conditions by using the AND operator.

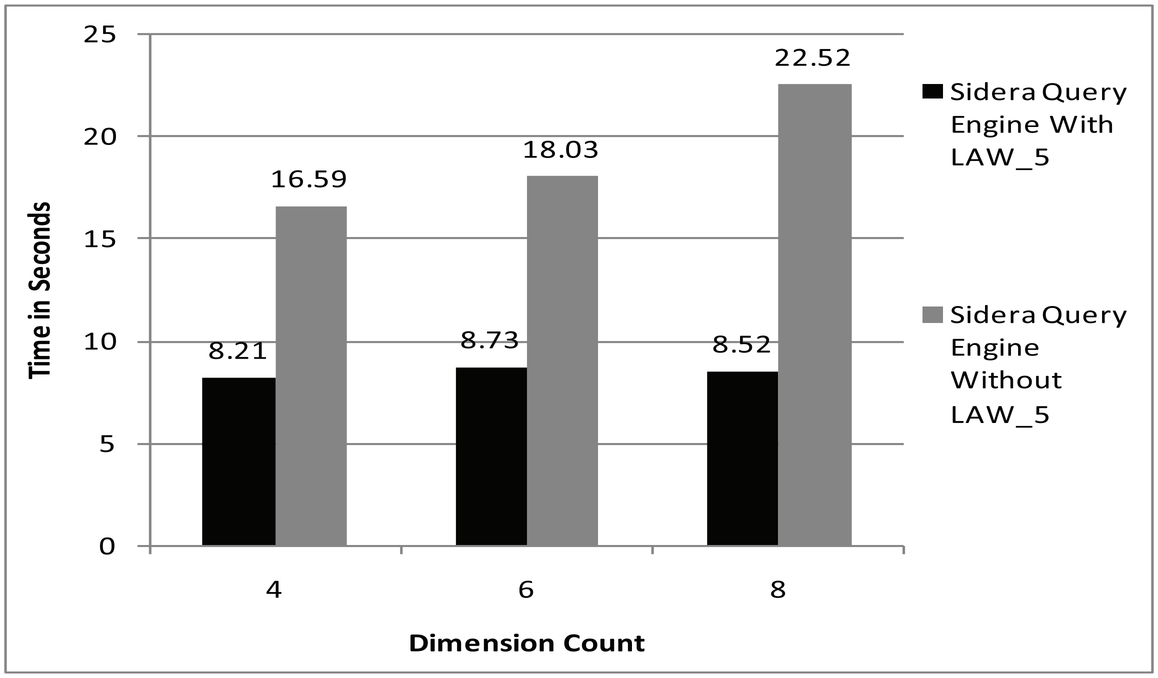


Figure 6.2: Query running time with and without LAW_5 as the number of dimensions increases in the data cube.

Third, LAW_4 is used again to push the SELECTION down to both arguments of the INTERSECTION. Finally, LAW_13 is used to replace the intersection of two identical queries with only one of them. In this test, the OLAP queries were divided into two groups: (i) those containing four attributes or less, and (ii) those drawn from the complete 6-dimensional space.

Figure 6.3 provides the test results. Here, we present the total response time of four OLAP queries before and after applying: LAW_2, LAW_4 and LAW_13. Note that in this test, a fact table with 6 dimensions is used to create the indexed cube (64 index views) as 64 Berkeley DB database objects that are stored in one Berkeley DB physical file on disk. Results of Figure 6.3 are shown for two indexed cubes that are created from initial fact tables of one million and ten million records.

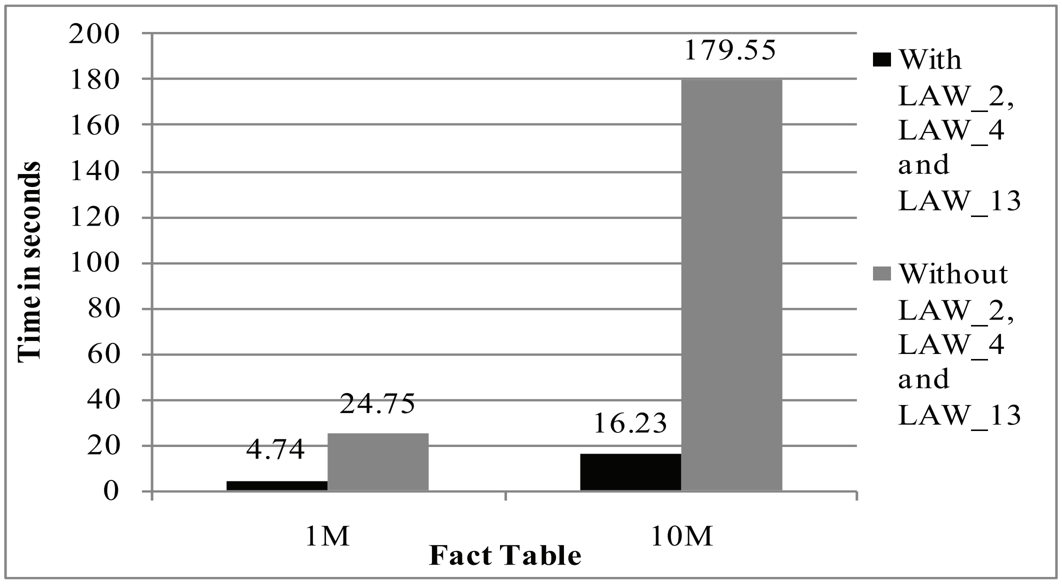


Figure 6.3: Query response time with and without LAW_2, LAW_4 and LAW_13 as the size of the fact table varies.

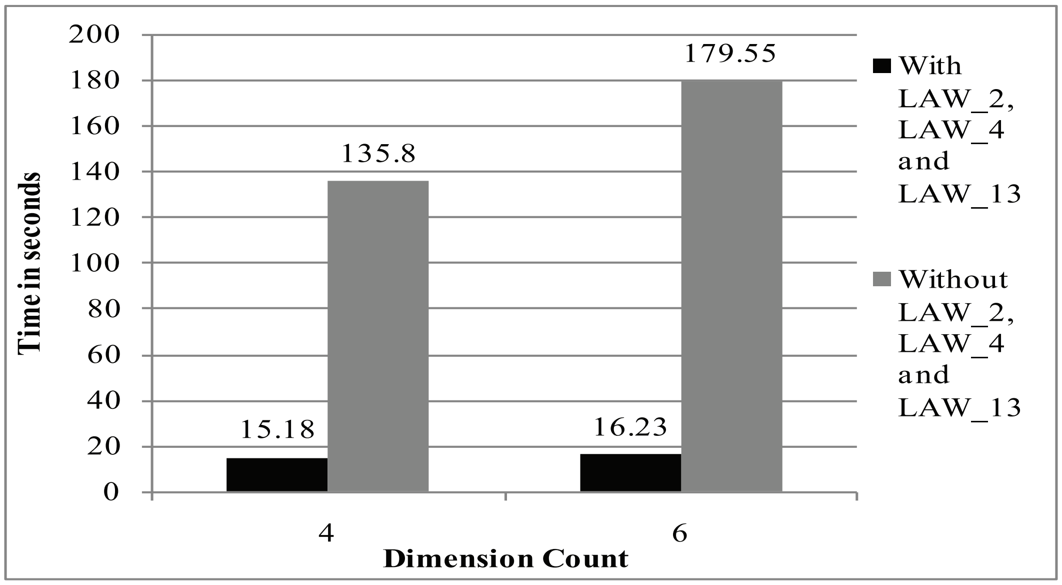


Figure 6.4: Query response time with and without LAW_2, LAW_4 and LAW_13 as the size of the number of dimension varies.

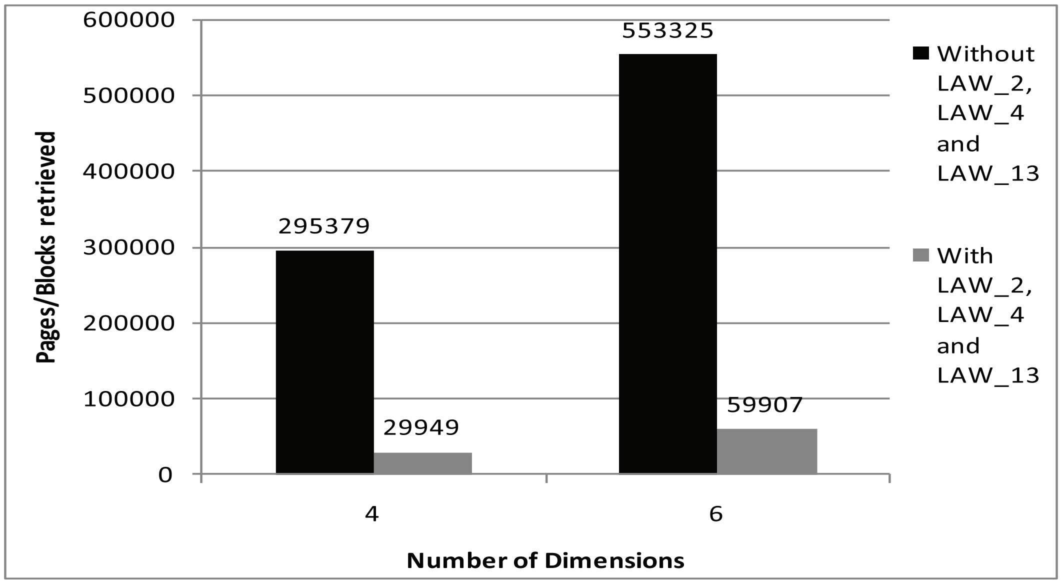


Figure 6.5: Comparison of number of blocks retrieved for the query engine before and after the use of LAW_2, LAW_4 and LAW_13 for the same queries.

Figure 6.4 illustrates the significance of our laws when the number of dimensions increases from four to 6. In this test, the indexed data cubes are created from ten million records with four and six dimensions respectively. We can see in this figure also how the running time of four OLAP queries decreases with the use of these laws. Figure 6.5 presents a comparison of the number of blocks retrieved before and after using the optimization laws (LAW_2, LAW_4 and LAW_13). The graph suggests that our engine without these optimization laws results in more than ten times as many block retrievals on both dense and sparse views.

We can see in Figure 6.3 and 6.4 that the running time is 5 to 15 times faster if our engine utilizes LAW_2, LAW_4 and LAW_13 to optimize queries. This improvement is due to three main factors (i) the intersection of two identical data queries is replaced by only one (LAW_13), (ii) LAW_2 changes a large query (in terms of the number of records in the result) to a smaller one because the conjunction of multiple conditions requires less disk I/O (see Figure 6.5) and (iii) finally, pushing down the new SELECTION with the conjunction of all conditions. We note the improvement factor increases as the number of dimensions and records in the fact table increases.

6.4.3 Index scan Versus Sequential cube scan

In this test, we compare the running time of queries when we sequentially scan the cube against the Berkeley DB R-tree index scan. Figure 6.6 illustrates the performance of our query engine for R-tree indexing and sequential scans for two data cube density levels (4 and 6 dimensions). Each data cube is generated from a fact table of 10 Million records. In terms of OLAP queries, we use 12 OLAP queries (equivalent to those queries in Appendix A) to compare index versus sequential cube scan in the case of a four-dimensional cube. However, 13 OLAP queries equivalent to the following SQL queries — 1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 14, 15 and 16 — in Appendix B are used in the 6-dimensional cube test result. We can see in Figure 6.6 that the running time for indexed view access is four times faster than sequential view access. We note that there is a point where no index can improve upon a sequential scan, as the result of a query increases sharply relative to the view size. In our server this happens when the result of the query exceeded 20% to 25% of the records in the view that is used to answer the query. However the penalty associated with pathologically large queries is small because of the indexing model (Linear BFS) that is used in our server. Therefore, we can conclude that sequential scans are not important in our OLAP server. This is important as it minimizes the complexity of query execution planning.

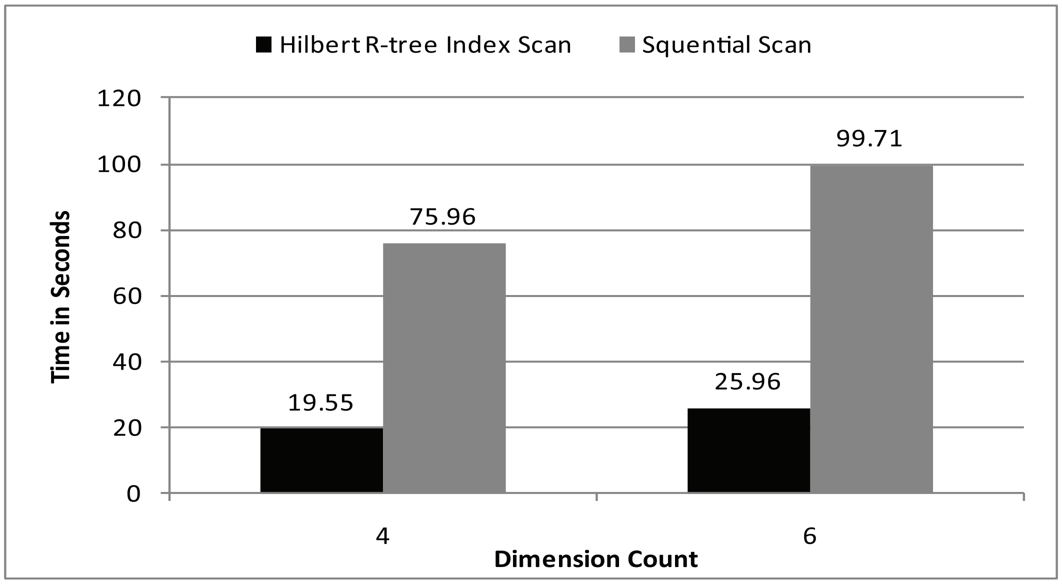


Figure 6.6: Sequential Scan versus Hilbert R-Tree index Scan.

6.4.4 Query Compilation Time versus Query execution Time

In almost all cases, the cost of query compilation (parser and optimizer) is significantly less than the cost of executing a query plan. To demonstrate the efficiency of our OLAP query compiler, we compare the compilation time with the execution time of a set of mixed OLAP queries. We illustrate that the compilation time (Parsing and Optimizing) in our server is a fraction of that of the execution time. Recall that our query compiler is not yet fully implemented. Consequently, we cannot report the final query optimization time because we are still implementing some of the query optimizer components. Therefore, in this section we compare the compilation time without the optimization time against the execution time. Currently, for each received OLAP query in XML format, the following times can be reported as part of the compilation time:

- Time to create the parse tree and to check its syntax.

- Time to check the semantic meaning of the query.
- Time to create the initial OLAP logical query plan.

For this test, we create a six-dimensional indexed full cube. In other words, 64 Berkeley DB R-tree database objects are created and stored on disk. The base view/cuboid or fact table has ten million records and it is surrounded with the six dimensions shown in Table 6.1. We create twelve different OLAP queries. These OLAP queries were divided into six groups. Each group contains a different number of dimensions from the 6-dimensional space. For example, group 1 includes those queries containing one attribute, while group 6 comprises those containing 6 attributes. The OLAP queries that were used for the test of this section are considered to be small OLAP queries. Specifically, the result set of each query doesn't exceed 1% of all of the records in the data set (these records are stored between 5% and 10% of the data blocks). Figure 6.7 shows the compilation time and the execution time for each query. We can observe that the compilation time in our server is less 1.6% of the execution time. We should note here that the compiler requires more time with the use of the optimizer; however, the optimization time is expected to be small compared to the execution because of the way our optimizer works (greedy algorithms to choose order, simple heuristics, etc.).

6.4.5 Validating the Cost Model

To demonstrate the accuracy of our OLAP multi-dimensional cost model, we compare the estimated cost in terms of disk I/O and CPU running time of several OLAP physical query plans against their actual execution time. We create a six-dimensional fact table (i.e., called Fact-Sales) with 10 Million records and surrounded with the six

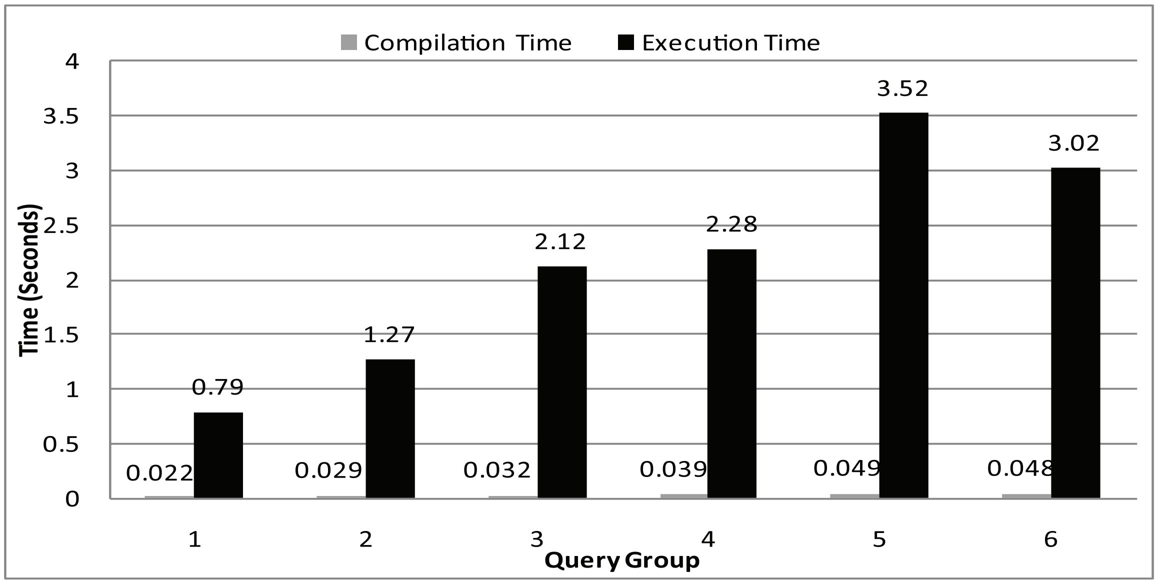


Figure 6.7: Query compilation time compared to Query execution time for different query groups (group i is answered from i-dimensional cuboids).

dimensions of Table 6.1. Moreover, the R-tree index cube (Called Sales) is created and stored in Berkeley DB. In this test, we report on one OLAP query, illustrated in SQL-syntax in Figure 6.8, which runs against the indexed cube (Sales) and dimension tables stored in our server. Figure 6.8 illustrates the initial OLAP logical query plan with the preferred logical query plan. Recall that our physical query model is not yet implemented. We restrict our work in this test to only one query because it is very time consuming to convert the XML OLAP query into a set of physical queries and then execute and report the running time of all of them.

Since our query optimizer is not yet implemented, we manually use the methods and theories that were described in Section 5.7 and 5.8 in order to convert the initial logical query of Figure 6.8 into four physical query plans that are depicted in Figure 6.9. These physical query plans generate the same result but they differ on how each plan is executed. For example, in plan-1 a dimension scan is performed to find

	Plan-1	**Plan-2**	**Plan-3**	**Plan-4**
Estimated Block I/O	89689	91689	85690	127145
Estimated CPU running Time	511767	511767	511767	50082027
Actual Execution Time	13.15	13.36	10.8	28.23

Table 6.2: Estimated cost (Block I/O and processing time) versus actual time.

all product IDs with category equal to Household, while a bitmap scan is used to get all customer IDs with age equals 30. On the other hands, Plan-4 uses sequential scans for the appropriate view and dimension tables (Product and Customer) to resolve the query. In Table 6.2, we can see the estimated cost and the actual execution time for each physical plan. We observe that dimension Customer and Product have 6000 and 4000 blocks respectively. Moreover, the appropriate view that is utilized to answer the query is the two dimensional view (Customer-Product) that contains 9999739 cells. In addition, the number of disk blocks and index blocks is 117644 and 2355 respectively. Our estimate for the SELECTION result for this query is 83332 cells. We observe that the actual number of cells for the SELECTION result when running this query is 72038 cells.

To summarize, at present our server does not physically compute these four plans; however, we instruct the system to use the appropriate algorithms for each plan in order to report its actual execution time. Our optimizer, when fully implemented, will of course pick Plan-3 because it has the least estimated cost.

6.4.6 Scalability

In production environments, it is quite likely that OLAP users will be accessing systems that are larger than the ones that can be conveniently tested in academic

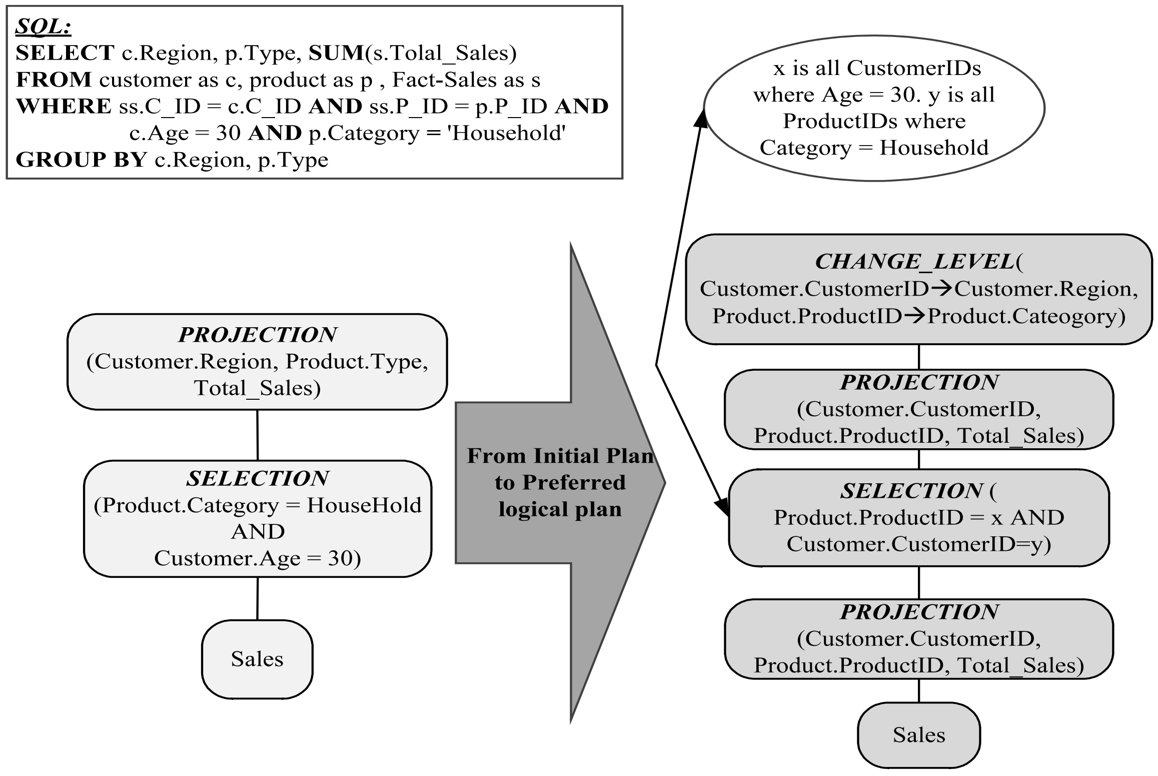

SQL:
SELECT c.Region, p.Type, SUM(s.Tolal_Sales)
FROM customer as c, product as p , Fact-Sales as s
WHERE ss.C_ID = c.C_ID AND ss.P_ID = p.P_ID AND
c.Age = 30 AND p.Category = 'Household'
GROUP BY c.Region, p.Type
PROJECTION (Customer.Region, Product.Type, Total_Sales)
SELECTION (Product.Category = HouseHold AND Customer.Age = 30)
Sales
From Initial Plan to Preferred logical plan
x is all CustomerIDs where Age = 30. y is all ProductIDs where Category = Household
CHANGE_LEVEL(Customer.CustomerID→Customer.Region, Product.ProductID→Product.Cateogory)
PROJECTION (Customer.CustomerID, Product.ProductID, Total_Sales)
SELECTION (Product.ProductID = x AND Customer.CustomerID=y)
PROJECTION (Customer.CustomerID, Product.ProductID, Total_Sales)
Sales

Figure 6.8: Initial and preferred OLAP query plan and its equivalent in SQL.

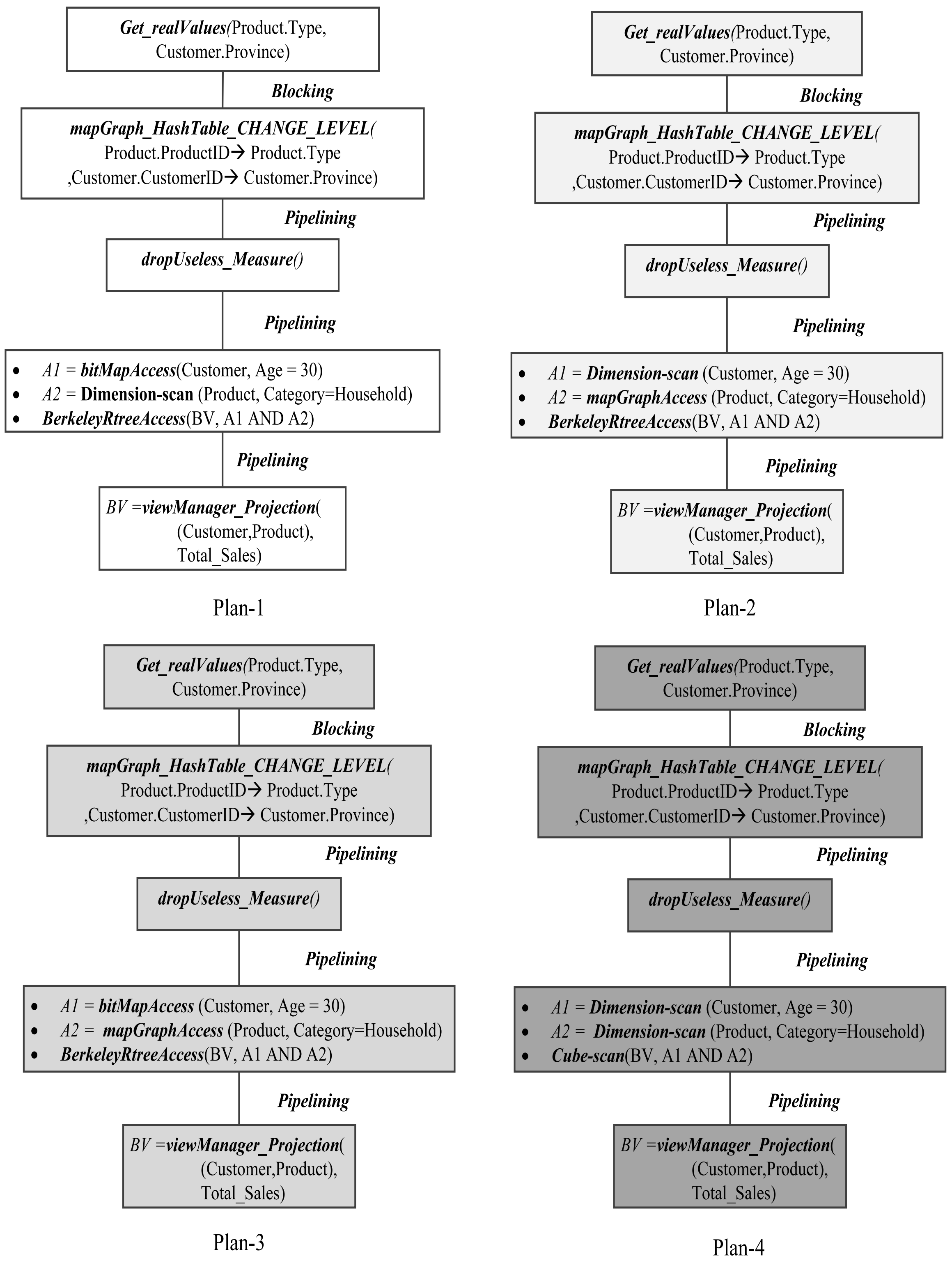

Get_realValues(Product.Type, Customer.Province)
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Type ,Customer.CustomerID→ Customer.Province)
Pipelining
dropUseless_Measure()
Pipelining
A1 = bitMapAccess(Customer, Age = 30)
A2 = Dimension-scan (Product, Category=Household)
BerkeleyRtreeAccess(BV, A1 AND A2)
Pipelining
BV =viewManager_Projection((Customer,Product), Total_Sales)
Plan-1
Get_realValues(Product.Type, Customer.Province)
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Type ,Customer.CustomerID→ Customer.Province)
Pipelining
dropUseless_Measure()
Pipelining
A1 = Dimension-scan (Customer, Age = 30)
A2 = mapGraphAccess (Product, Category=Household)
BerkeleyRtreeAccess(BV, A1 AND A2)
Pipelining
BV =viewManager_Projection((Customer,Product), Total_Sales)
Plan-2
Get_realValues(Product.Type, Customer.Province)
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Type ,Customer.CustomerID→ Customer.Province)
Pipelining
dropUseless_Measure()
Pipelining
A1 = bitMapAccess (Customer, Age = 30)
A2 = mapGraphAccess (Product, Category=Household)
BerkeleyRtreeAccess(BV, A1 AND A2)
Pipelining
BV =viewManager_Projection((Customer,Product), Total_Sales)
Plan-3
Get_realValues(Product.Type, Customer.Province)
Blocking
mapGraph_HashTable_CHANGE_LEVEL(Product.ProductID→ Product.Type ,Customer.CustomerID→ Customer.Province)
Pipelining
dropUseless_Measure()
Pipelining
A1 = Dimension-scan (Customer, Age = 30)
A2 = Dimension-scan (Product, Category=Household)
Cube-scan(BV, A1 AND A2)
Pipelining
BV =viewManager_Projection((Customer,Product), Total_Sales)
Plan-4

Figure 6.9: Four Possible physical plans.

settings. As a result, it is important to provide some understanding of performance as core parameters grow. Our scalability assessment begins with a look at performance patterns as the number of records increases from 100K to ten million. We use a six-dimensional fact table and define the dimensions as per the dimensions mentioned in Table 6.1. In this test, we create 16 OLAP XML queries equivalent to those SQL-based queries defined in Appendix B. That being said, these queries can be answered from different indexed views from the 6-dimensional space (64 indexed views). Specifically, ten of them can be answered from views with 3 dimensions or less, while views with more than 3 dimensions were used to answer six of them. We note in this test that the result sets of 14 queries did not exceed 10% of the original data blocks in the views, while two unusually large queries (more than 25% of the data blocks are involved) were also included.

Figure 6.10 shows the execution time as a function of data cube size. As can be seen in the figure, an increase of a factor of ten in the number of records in the fact table is associated with an exponential increase in execution time. The result in Figure 6.10 is expected because of the high cardinalities of the dimensions. In other words, we observe that the sizes of views with more than one dimension increase by roughly the same factor as the fact table/base cuboid (10 in our case). Observe that the running time doubles as the number of records in the fact table increases by a factor of ten.

In practice, the sizes (number of records) of lower dimensionality views in different cubes would be almost the same as the number of records in the base view increases. Consequently, our OLAP server is very scalable in that an increase in the number of records in the fact table (base view) is associated with nearly the same execution

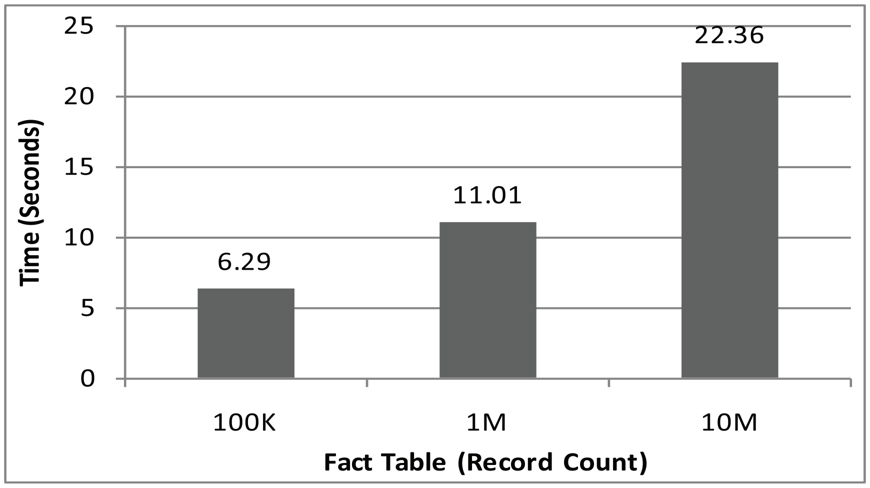


Figure 6.10: Execution time for sixteen OLAP queries as a function of the data cube size.

time. If the data is skewed, then view sizes would grow more slowly and thus the increase in time would be sub-linear.

6.5 The Sidera OLAP Query Engine versus Query Engines

As discussed, our main goal is to have a very efficient OLAP server to answer analytical OLAP queries in a very responsive way. Consequently, we provide several tests in this section to evaluate our query engine against two enterprise query engines. First, we evaluate our engine against the open source MySQL database server. Second, a comparison against the Microsoft SQL Server 2008 OLAP component (Microsoft Analysis Services MAS) will be discussed. Here, we build a 6-dimensional fact table with ten million records. Dimensions are described in Table 6.1. In terms of the queries, we first generate small batches of OLAP queries in XML format by hand. Then, each of these Sidera OLAP queries is translated into the appropriate

SQL syntax and MDX syntax so that they can be executed on the MySQL server and Microsoft Analysis Services. Because of the labour-intensive nature of this task, we cannot use batches of large numbers of queries, instead relying on batches of 12 to 20 queries.

6.5.1 Sidera Query Engine versus MySQL

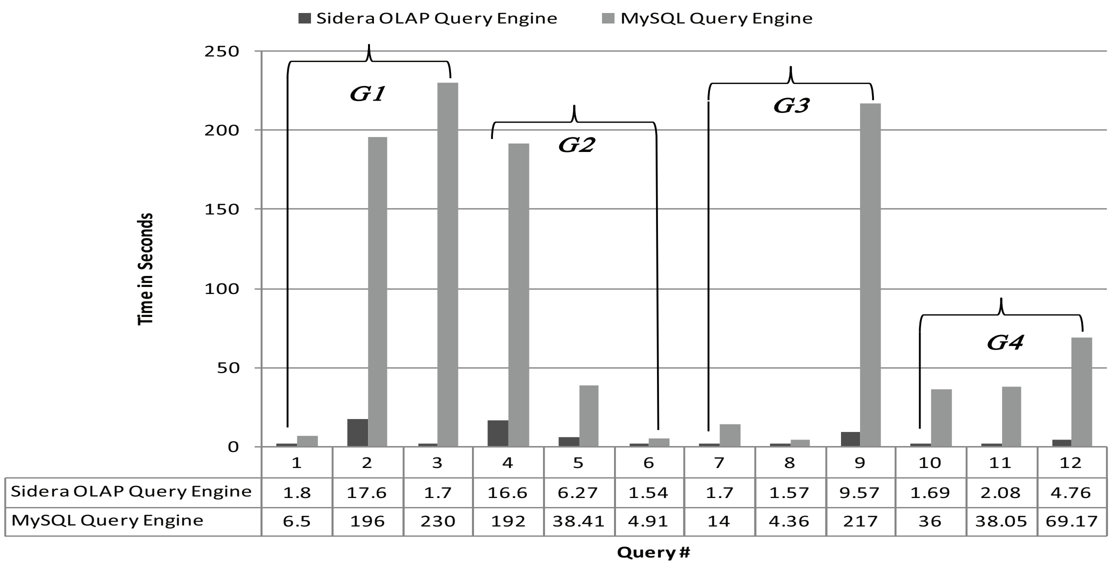
Here in this test, we first installed the open source MySQL 5.1 DBMS server on our Linux-based workstation. We build a 4-dimensional fact table with ten million records. Dimensions are the first four dimensions in Table 6.1 (Customer, Product, Time and Store). The fact table is called sales and is connected with its associated dimensions with a foreign key attribute (feature attribute). Each dimension is stored as a relation in MySQL with consecutive integer values as a primary key. A B-tree index for each dimension is created on the primary key. Each primary key in a given dimension is represented in the fact table as a foreign key/feature attribute. The foreign keys are concatenated to create one composite key that forms the primary key of the fact table. Then, a B-tree index is created for the primary key in the fact table. In the fact table, we consider only one measure that is aggregated with the sum function (i.e., Total_Sales in the fact table sales).

Note that in this test we do not materialize any aggregate group-bys. In other words, all tests are conducted simply against the fact table/base view and its associated dimension tables in order to allow a fair comparison of the results. In terms of the queries, we constructed a batch of twelve OLAP queries that match our OLAP query grammar defined in Section 4.6. These OLAP queries are divided into four groups G = G_i where i = 1, 2, 3, 4, such that G_i contains queries that can be answered from the fact table and |i| dimensions. Note that all groups have an equal number of queries (3 queries each). Then, each of these OLAP queries is translated by hand into the appropriate SQL syntax so that they can be executed on the MySQL Server DBMS. Appendix A shows these twelve queries in SQL-syntax based. We note in this test that query 2, 4 and 9 are unusually large queries because their result sets exceed 20% of the original records in the fact table.

Figure 6.11 illustrates the test results. Here, we present the total response time for the twelve OLAP queries — divided into four groups — executed by our OLAP query engine versus their equivalent in SQL-syntax answered by the MySQL query engine. We can clearly see the difference between our engine's capabilities and MySQL's capabilities to resolve complex analytical queries (Appendix A). This large difference is due to many different factors. First, MySQL doesn't have an OLAP-aware multi-dimensional indexing mechanism such as Hilbert Packed R-tree. The composite primary key is useful when the key combination uniquely defines the record but it is not ideal for multidimensional query environments. Second, MySQL doesn't have hash or merge join techniques which are frequently used for these kinds of analytical queries (star queries). Finally, the MySQL query optimizer has very limited plan choices because of the limited indexing techniques available.

6.5.1.1 Dimension Count

In addition to the previous experiment, we also compare our OLAP query engine against the MySQL query engine when the number of dimensions increases. For the current test, the fact table holds a constant number of ten million records, while its associated dimension tables vary from four (first four dimensions from Table 6.1) and six (six dimensions from Table 6.1). In terms of queries, the twelve queries of the previous section are resolved against both dimensional fact tables (four and



	1	2	3	4	5	6	7	8	9	10	11	12
Sidera OLAP Query Engine	1.8	17.6	1.7	16.6	6.27	1.54	1.7	1.57	9.57	1.69	2.08	4.76
MySQL Query Engine	6.5	196	230	192	38.41	4.91	14	4.36	217	36	38.05	69.17

Figure 6.11: Comparison of resolving OLAP queries using our OLAP query engine against MySQL Query Engine.

six). Moreover, we use five and six dimensional queries — query 3, 8 and 16 — from Appendix B which are resolved only from the six-dimensional fact table. In the Sidera server, the indexed base view and its associated dimension tables are used to answer those queries. The same data from the fact table and dimension tables are stored in MySQL.

Figure 6.12 illustrates the high performance of the Sidera query engine compared to the MySQL database server as the number of dimensions increases. In other words, the running time of answering multi-dimensional queries using the Sidera server is 10 to 15 times faster than the MySQL server. This result is due to many reasons. First, the data clustering favours the first dimension in the B-tree index on the composite key (primary key) of the fact table; however, in our server, we use a multi-dimensional indexing scheme (Hilbert R-tree) that is ideal for multi-dimensional queries. Second, in the Sidera server, the Sidera hierarchy manager (mapGraph) and

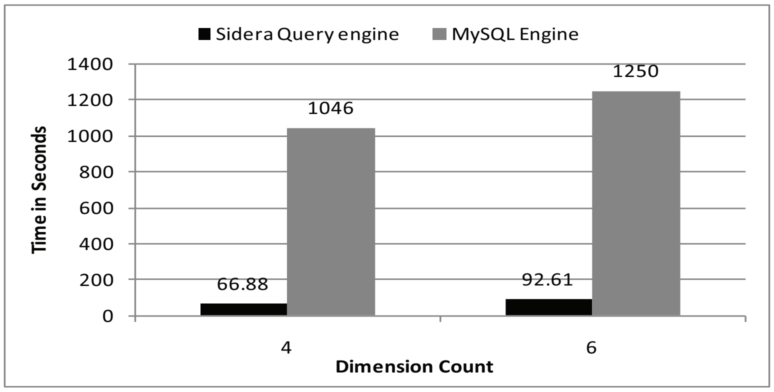


Figure 6.12: Comparison of the Sidera query engine with MySQL server.

bitmap index manager are very efficient at resolving the user's query restrictions; however, in MySQL all records of the appropriate dimension tables must be read and tested for a match with the user's query.

6.5.1.2 Fact Table Size

In this test, we create four-dimensional fact tables of different number of records that vary from 100K to 10M records. Moreover, each fact table is surrounded with the first 4 dimension tables in Table 6.1. In the Sidera server, we create the R-tree index for the base view in order to have a fair comparison. The same data are loaded into MySQL relational tables, where each contains a B-tree index for its associated primary key. In terms of queries, we create 12 OLAP-XML queries equivalent to those queries defined in Appendix A. Figure 6.13 shows the performance of our query engine as the number of records in the fact table (base view) increases. Using our server, the running time is 10 to 15 times faster than MySQL on schemas of equivalent six. This result is again due the factors mentioned above.

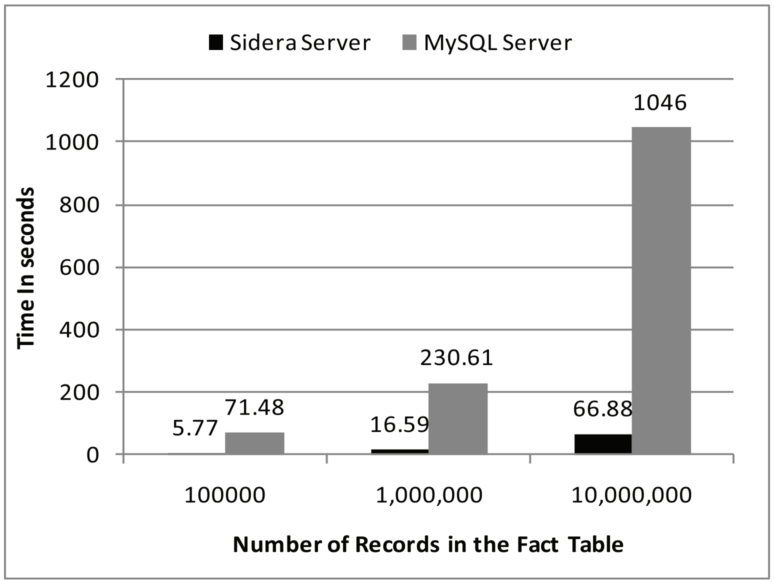


Figure 6.13: Sidera Query Engine against MySQL Engine as the number of records increases.

6.5.2 Sidera Engine versus Microsoft SQL Analysis Service

In this test, we first installed Microsoft SQL server 2008 on the Windows's partition of test system. Dimension tables (Table 6.1) are created in SQL server as a set of relational tables and populated with the same data available in the Sidera server. Moreover, the fact table is also created as a relational table with a composite primary key. Each attribute in this composite key connects the fact table with a corresponding dimension. We then use the Microsoft SQL Analysis services component available in SQL server 2008 to create the cube that represents the fact table and dimension tables stored in the related relational tables. The cube is created in SQL Analysis services (SSAS) by defining the source fact table and its surrounding dimensions data. Feature attributes are the composite key, while only one measure attribute (Total_Sales) of type sum is used. Hierarchies and non-hierarchical attributes are also defined for dimensions in the cube. SSAS allows us to choose the storage partition of the cube

(ROLAP, MOLAP or HOLAP). Because our server is a ROLAP server, we initially choose the ROLAP data storage engine to conduct the comparison against our server. However a MOLAP test will be presented in the following section.

We installed the necessary software (FastBit, Berkeley DB) along with the Sidera code on the Linux-partition of the test system. We make sure that the same dimensions and fact table data are used to compare our engine against SSAS. Note that in this test we do not materialize any aggregate group-bys to be used with our server. In other words, all tests are conducted simply against the fact table/base view and its associated dimension tables in order to allow a fair comparison of the results. In terms of queries, we translate the queries outlined in Appendix A into MDX-based queries to be resolved against a 4-dimensional cube. For the 6-dimensional cube, we translate three extra queries (3, 8, 16) from Appendix B into MDX format. As such the queries involve a variable number of dimensions (one to six). We note in this test that query 2, 4 and 9 in Appendix A are unusually large queries because more than 20% of the original data blocks were accessed.

6.5.2.1 Dimension Count

In this test, we evaluate our OLAP query engine against the SSAS query engine when the number of dimensions in the cube increases. Again, the SSAS ROLAP storage engine is used. For the current test, the fact table holds ten million records, while its associated dimension tables vary from four (first four dimensions from Table 6.1) to six (six dimensions from Table 6.1). Figure 6.14 presents the total response time for OLAP queries executed by our query engine versus equivalent MDX-based queries answered by Microsoft SQL Analysis Services as the dimension count increases from four to six. Recall that 12 queries (equivalent to queries in Appendix A) were used

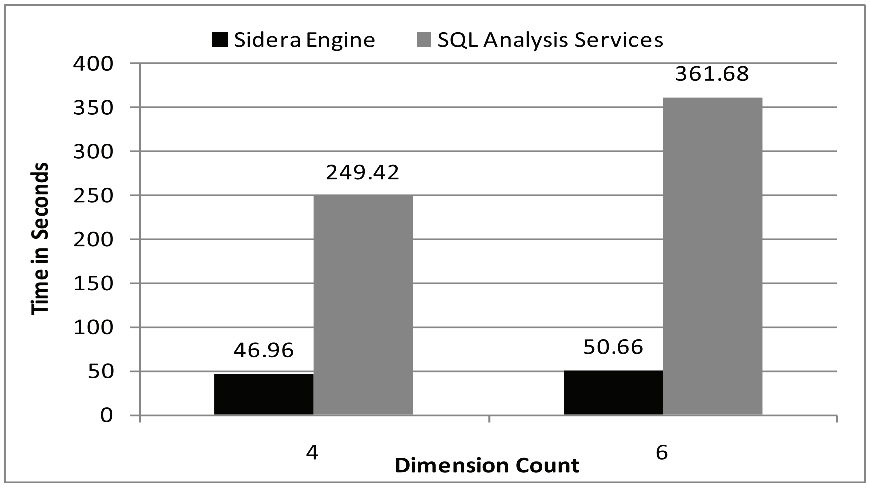


Figure 6.14: Comparison of our query engine versus Microsoft SQL Analysis Services.

in the case of four dimensions, while 15 queries were used in case of six dimensions. Figure 6.14 illustrates that our engine is 5 to 7 times faster than the SQL Analysis services engine. Again, there are a number of contributing factors. Our storage engine provides a very efficient multi-dimensional indexing scheme (Hilbert R-tree) for the cube which is not available in the Microsoft product. Moreover, the in-memory structures (hierarchy manager (mapGraph) and bitmap index manager) are very efficient at answering complex query from the main memory without reading all records of the dimension tables. Finally, SQL server has a complex query compilation step that includes the translation of the MDX-based query into SQL format that, in turn, must be parsed, optimized and executed.

6.5.2.2 Fact Table Size

In addition to the impact of the number of dimensions, we also look at the impact of an increase in the number of records in the fact table. In this case, we create a fact table with four dimensions (first four dimensions in Table 6.1). The number of

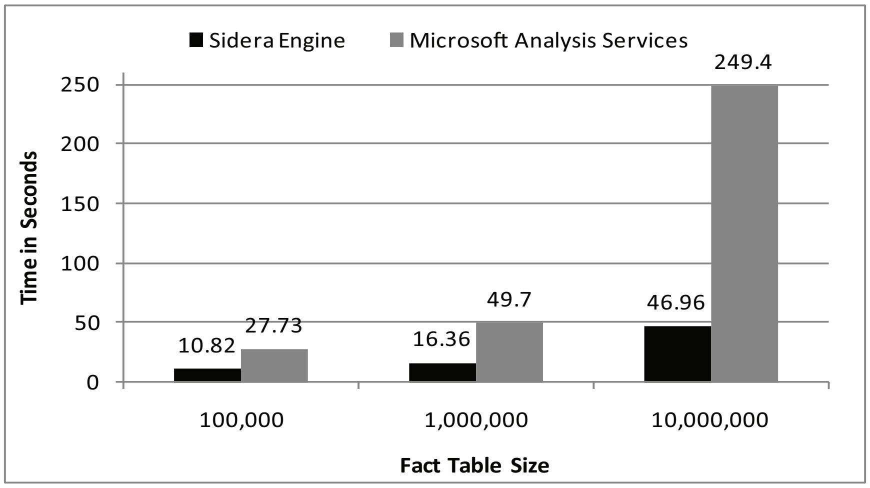


Figure 6.15: Comparison of the Sidera engine versus Microsoft Analysis Services for three different cube sizes.

records varies from 100,000 to 10,000,000. We use batches of 12 MDX queries and 12 OLAP XML queries, equivalent to those queries in Appendix A, to be answered from Microsoft SQL server and our engine respectively.

Figure 6.15 provides the test results. Here, we present the total response time for equivalent queries answered by our engine and Micrososft Analysis services. Observe that our OLAP query engine is 3 to 6 times faster than the engine of Microsoft Analysis Services. This result is expected since our engine provides very efficient methods to store and access the cube (Hilbert R-tree index in Berkeley DB) and dimension tables (bitmap indexes). Specifically, we can match the condition of the query from in-memory data structures (mapGraph and bitmap manager) and only use the R-tree index for the fact table to return those records satisfying the query condition. Moreover, the SQL server query compilation adds some processing overhead because, as previously noted, the MDX query is internally translated into SQL format that must then be parsed and optimized.

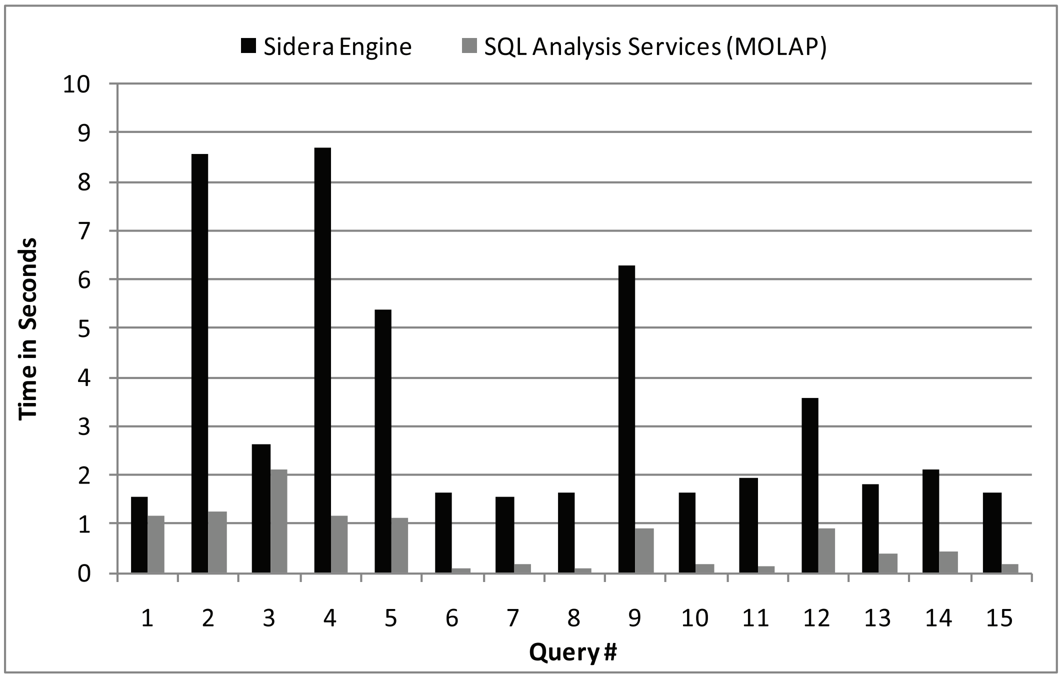


Figure 6.16: Running time in the Sidera Server versus the MOLAP SQL server.

6.5.2.3 Sidera Engine versus MOLAP SQL Analysis Services

In this section, we examine the performance of our engine compared with SQL Analysis service when the MOLAP data partition is used for the cube. Here, we create a six-dimensional fact table with 10,000,000 records. In terms of queries, we use 15 queries equivalent to those queries in Appendix A and query 3, 8 and 16 from Appendix B.

Figure 6.16 illustrates the test results. Here, we present the response time for each one of the 15 queries executed by our OLAP query engine versus SQL Analysis Services when it uses the MOLAP storage mode for data partitioning. We can see in Figure 6.17 that SQL Analysis Services using MOLAP storage is 4-5 times faster than our engine.

At first glance, this may appear to be disappointing result. However, we note the

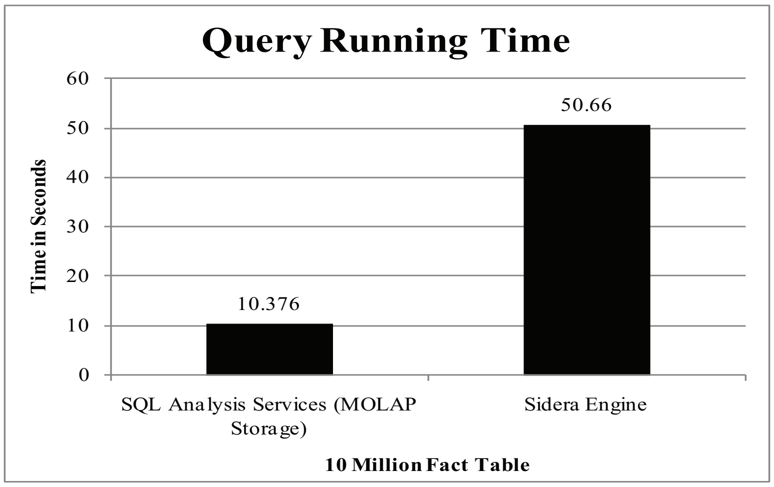


Figure 6.17: Sidera Engine versus Microsoft Analysis Services (MOLAP storage).

following:

1. MOLAP is ideally suited to small databases (i.e., those that fit entirely in memory as in the case here). It is well understood, both from academic and industrial testing, that MOLAP does not scale well. Our ROLAP system however is explicitly designed to scale to very large datasets. Unfortunately, some of the original Sidera components cannot support larger datasets at the present time, so we cannot do a more realistic MOLAP/ROLAP comparison.

2. Sidera's caching framework (a separate project) is currently not integrated with the server. Eventually, this caching module will provide many MOLAP-like benefits for in-memory data.

3. Microsoft's DBMS software obviously benefits from years of optimization and performance tuning. We are quite limited in this sense.

4. Most importantly, this test restricts Sidera to using a fact table only. In other

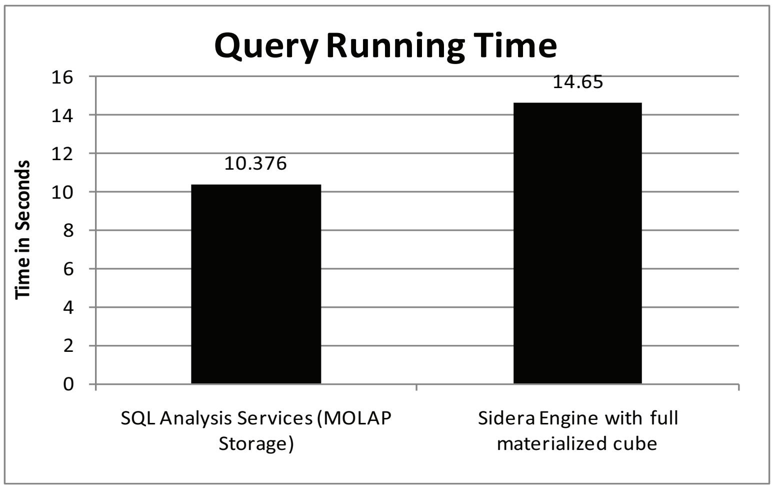


Figure 6.18: Sidera Engine (fully materialized cube) versus Microsoft Analysis Services (MOLAP storage).

words, it is not using any of the additional summary views that Sidera is specifically designed to produce. This severely handicaps Sidera's performance in this kind of test.

Given the fourth point listed above, we conducted a second test in which Sidera was permitted to generate a fully materialized data cube (note that even partial materialization can produce similar results). Figure 6.18 shows the results of this comparison. Here, we see that while the Microsoft MOLAP system still comes out on top, the difference is now less than 30%. Given that the first three points listed above still hold, we can conclude that Sidera can effectively compete with state of the art OLAP solutions, even in environments ideally suited to MOLAP.

6.6 Parallel Query Engine Resolution

Recall that our compilation and execution components in the Sidera server (Storage engine, query processor, etc.) are designed to work independently on each node.

In Section 5.10, we explained how several components mentioned in this thesis are integrated into the larger Sidera parallel server framework. In this section, we will show the parallel wall clock time for distributed query resolution as a function of the number of processors. In this test, we create a six-dimensional fact table surrounded by the 6 dimension tables described in Table 6.1. Moreover, the fact table contains 10 million records. The indexed cube is generated and striped equally to each processor. Note that the Sidera server uses the combination of Hilbert ordering with round-robin striping that almost perfectly balances the query results over the parallel machine. The same dimensional queries in Appendix B were used in this test. However, we changed the conditions in queries (2, 4, 6 and 16) so that they require more blocks to be accessed in order to answer those queries. For example, we replace the condition (Age $=$ 30) with (Age $<$ 30) in query 2.

Figure 6.19 illustrates parallel time for distributed query resolution as a function of the number of processors used, while Figure 6.20 presents the corresponding speedup. For example, on 16 processors, a speedup of 10.12 is achieved. We note that the speedup values are quite good compared to the ideal speedup. The difference between our observed speedup and the best speedup is due to a couple of factors. Perhaps not surprisingly, it doesn't arise from the workload of the query compiler on each backend node. Specifically, at present the query compilation time is insignificantly relative to the query execution time. Instead, the difference in the speed up values arises from:

1. The dimension table processing that is used to build the mapGraph and bitmap index manager. Specifically, in order to match the query condition each backend node must do exactly the same processing on the same data in order to convert the query condition to a form that can be answered directly from the appropriate

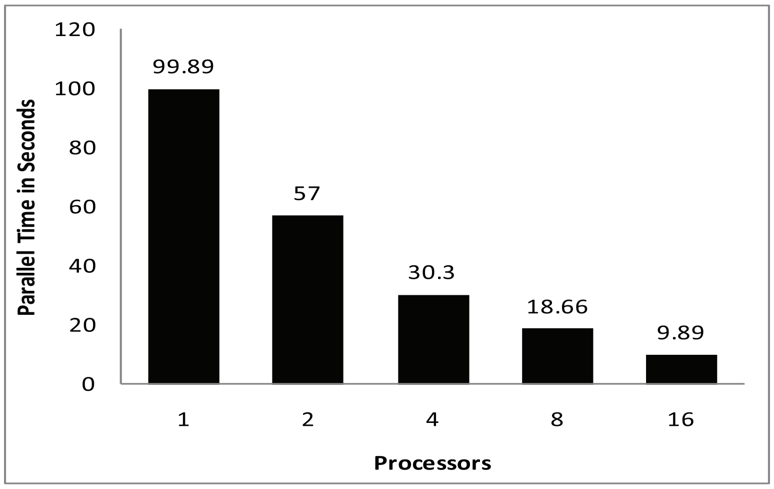


Figure 6.19: Parallel clock time for distributed query resolution.

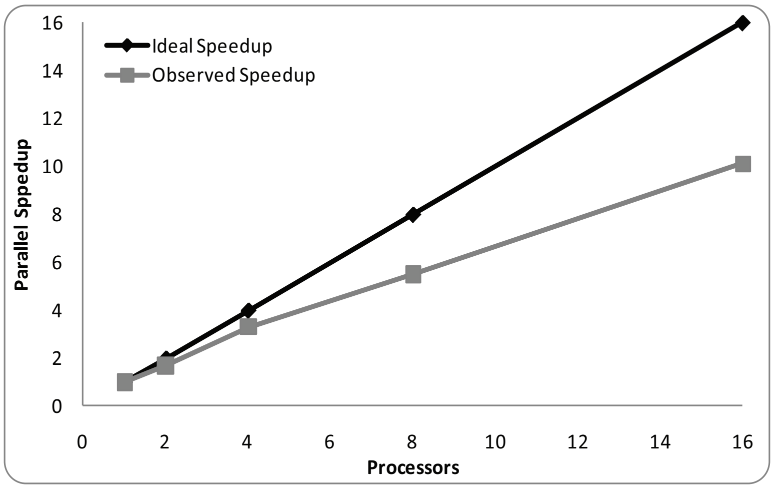


Figure 6.20: Parallel speedup.

view. In other words, each backend node has a copy of the smaller dimension tables. So each backend node must do similar dimension processing.

2. The Parallel Sample Sort used to order the query results. Specifically, this mechanism cannot guarantee that the records in the intermediate processing steps are perfectly balanced across the nodes.

The above points suggest that these speedup results might be further improved by finding a better way to minimize the dimension processing and by the use of a more balanced sort code.

6.7 Conclusions

In this chapter, we have provided experimental results that assess the significance and viability of our storage engine, query optimization and execution models. Experimentally, our results support the design decisions that we have made. Test results demonstrate the query running time with and without using algebraic laws to optimize the query. Specifically, we described the improvement when our engine uses several of the core laws. Moreover, we show that at present the query compilation time is likely to add very little overhead to the total query response time. We also validate the accuracy of the cost model that is used to select the physical plan. Scalability, in terms of record size, is also assessed. In addition, experimental results relative to two enterprise database products (MySQL and SQL server) were conducted and showed very high performance in query resolution. Finally, we looked at the speedup of our parallel engine. Experimentally, the end result of our research is a very efficient OLAP server that has the potential to significantly boost run-time OLAP query performance for the large problem domains typically found in enterprise environments.

Chapter 7

Conclusions and Future Work

7.1 Summary

In this thesis we have discussed the design, implementation, and evaluation of an OLAP DBMS that can efficiently resolve queries written in native client side programming languages. Basically, we are trying to demonstrate that it is possible to provide MOLAP performance with ROLAP scalability. Given that Sidera is ultimately designed as a scalable parallel system, our focus in this research was to provide performance. For this we decided to create an OOP conceptual model, comprehensive OLAP algebra and grammar that would allow us to control query processing from end to end (client programs right to the backend disks). This allowed us to exploit the right data structures and indexes, optimize query planning and provide efficient execution of those plans. Moreover, we have discussed an efficient OLAP storage engine that is responsible on how data is efficiently stored and accessed quickly. Specifically, we have focused on the following three issues in order to have the potential to provide MOLAP-style performance with ROLAP-style scalability.

1. **OLAP Storage Management.** We have demonstrated how the primary components of the data warehouse — fact and dimension tables — are efficiently

stored on disk in order to facilitate rapid access. Additionally, we discussed how this data is encoded into a more compact integer format. In terms of the physical architecture, we described the integration of the Berkeley DB components into our OLAP server so as to significantly enhance storage layer functionality. Specifically, this integration allows us to store the hundreds of files comprising the R-tree indexed cube into a more intuitive and easy to manage Berkeley DB environment. Moreover, we have suggested storing non-hierarchical attributes as a set of compressed FastBit bitmap indexes that, in turn, can be exploited by the query optimizer to provide more timely OLAP analysis. We also discussed how hierarchical attributes are stored and accessed via the runtime hierarchy manager. Finally, we discussed DTD-encoded OLAP metadata that defines the format of the schema for our OLAP environment. Schema storage is done natively in Berkeley XML DB, allowing us to utilize standard XML tools such as XQuery and XPath. Justification for our Berkeley DB integration, as well as hierarchical and non-hierarchical storage techniques, was provided by way of analysis and experimentation.

2. **OLAP Query Language.** We have described a comprehensive OLAP query algebra (operations and laws) and grammar that can be used to enhance the process of resolving OLAP queries. Ultimately, the algebra reduces the complexity of writing OLAP queries relative to the relational algebra. In the current case, the comprehensive OLAP algebra allows for the optimization of OLAP queries written in native OOP languages. To that end, we have also described a robust DTD-encoded OLAP grammar that provides the concrete foundation for client language queries, thereby potentially eliminating the reliance on intermediate,

string-based embedded languages. Finally, a set of algebraic laws were defined that would allow an OLAP query optimizer to convert algebraic expressions into equivalent, but more efficient, plans.

3. **OLAP Query Processing.** We have discussed how OLAP queries can be executed efficiently. Specifically, we have defined two main components for processing OLAP queries: an OLAP query compiler and an OLAP query execution engine. The query compiler uses a process similar to that of traditional relational database systems in order to parse and optimize OLAP queries. In short, an internal parse tree representing the OLAP query is first created. We then apply an optimization process that relies upon two mechanisms: (i) OLAP multi-dimensional rules and (ii) an OLAP cost model. In terms of query execution, a number of algorithms have been defined for the execution of the operations of our OLAP algebra. We also have described how several DBMS components (e.g., storage, query language, and processing) fit together to support the larger functionality of the parallel server (i.e., Sidera). Finally, we have described how the result of the query is returned in an XML format that can be directly and easily accessed on the client side. Extensive testing of the new OLAP query processor demonstrated the significance and viability of our query optimization engine, in terms of functionality and query execution time.

7.2 Future Work

The research described in this thesis represents a foundation for the development of a pure OLAP DBMS for the large problem domains typically found in enterprise environments. Below we identify a number of possible projects or research themes that would significantly extend the functionality of the current research:

1. **Integration of additional Sidera components.** The Sidera project provides several independent components that can be integrated into the current server in order to provide a more complete OLAP DBMS server. In particular, the new R^3-cache, a natively multi-dimensional caching framework designed specifically to support sophisticated warehouse/OLAP environments, offers opportunities to further optimize query response time [37]. Also, the integration of Sidera's R-tree compression methods into the server would dramatically reduce the storage footprint for the underlying data cubes [35].

2. **Full Implementation of our Query Optimizer.** Recall that our query optimizer (logical and physical) is not yet fully implemented. Various optimizations — discussed purely in a theoretical context in this thesis — could have a significant impact upon the performance of the current prototype.

3. **Support for additional hierarchical forms.** At present, the server can be used to model only simple symmetric dimension hierarchies. However, in the real world we find several additional types of hierarchies (e.g., Asymmetric hierarchies, Non-Strict hierarchies, Parallel hierarchies, Multiple Hierarchies). It would be important to extend our mapGraph (hierarchy manager) to deal with these kinds of dimension hierarchies.

4. **Support additional measure attributes.** At present, our storage engine supports only the distributive form of measure attribute that can be aggregated with the "sum" function. However, in practical settings we find several

additional types of measure attributes such as Algebraic (e.g. average, standard deviation, variance) and Holistic (e.g. mean and rank) forms. It would be important to extend our storage engine to deal with these kinds of measure attributes.

5. **External memory algorithms.** Recall that our current query processor assumes that there is enough main memory to store intermediate query results and any additional data structures. In practice, this assumption may not be true for extremely large data sets. To properly support such large initial data sets, it will be necessary to extend the current query engine algorithms into external memory. Although conceptually straightforward, the "systems" work required would be quite significant.

7.3 Final Thoughts

The research presented in this thesis describes a number of the core elements one would require if implementing a robust, high performance OLAP-specific data management system. We have discussed the motivation, design, implementation, and evaluation of our research and have emphasized their application to practical query environments. Given the significance of the problem domain, both from a commercial and academic perspective, we believe that our research represents a meaningful contribution to the OLAP literature.

Bibliography

[1] Microsoft analysis services. http://www.microsoft.com/sqlserver/2008/en/us/analysis-services.aspx.

[2] Oracle essbase. http://www.oracle.com/us/solutions/ent-performance-bi/business-intelligence/essbase/index.html.

[3] Sap. http://www.sap.com/services/education/catalog/netweaver/bi.epx.

[4] Xml for analysis specification v1.1, 2002. http://www.xmla.org/index.htm.

[5] Cwm, common warehouse metamodel, 2003. http://www.cwmforum.org/.

[6] Jsr-69 javatm olap interface (jolap), jsr-69 (jolap) expert group, 2003. http://jcp.org/aboutJava/communityprocess/first/jsr069/index.html.

[7] Haskelldb, 2010. http://www.haskell.org/haskellDB/.

[8] Ruby programming language, 2010. http://www.ruby-lang.org/en/.

[9] F.N. Afrati, C. Li, and J.D. Ullman. Generating efficient plans for queries using views. *SIGMOD*, pages 319–330, 2001.

[10] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. *Proceedings of the 22nd International VLDB Conference*, pages 506–521, 1996.

[11] Alfred V. Aho and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[12] M. O. Akinde and M. H. Bohlen. Efficient computation of subqueries in complex olap. *In International conference on Data Engineering (ICDE)*, pages 163–174, 2003.

[13] C. Bauer and G. King. Java persistence with hibernate. *Manning Publications Co., Green-wich, CT, USA*, 2006.

[14] R. Bayer. Binary b-trees for virtual memory. *Proceeddings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*, 1971.

[15] N. Beckmann, H. Kriegel, R. Schneider, , and B. Seeger. The r-tree: an efficient and robust method for points and rectangles. *ACM SIGMOD*, 1990.

[16] L. Bellatreche, M. Schneider, H. Lorinquer, and M. Mohania. Bringing together partitioning, materialized views and indexes to optimize performance of relational data warehouses. *DAWAK*, pages 15–25, 2004.

[17] Ladjel Bellatreche, Arnaud Giacometti, Dominique Laurent, Patrick Marcel, and Hassina Mouloudi. Olap query optimization: A framework forcombining rule-based and cost-based approaches. *EDA*, 2005.

[18] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *Proceedings of the 1999 ACM SIGMOD Conference*, pages 359–370, 1999.

[19] J. A. Blackard. The forest covertype dataset. *ftp://ftp.ics.uci.edu/pub/machine-learning-databases/covtype.*

[20] J. A. Blakeley, V. Rao, I. Kunen, A. Prout, M. Henaire, and C. Kleinerman. .net database programmability and extensibility in microsoft sql server. *In ACM*

SIGMOD International conference on Management of Data, pages 1087–1098, 2007. New York, NY, USA.

[21] L. Cabibbo and R. Torlone. Querying multidimensional databases. *Proceedings of the 6th DBLP Workshop*, pages 253–269, 1997.

[22] S. Chaudhuri. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1919–1323, 2004.

[23] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26:65–74, 1997.

[24] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. *in ICDE*, pages 190–200, 1995.

[25] E. Codd, S. Codd, and C. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E.F. Codd and Associates, 1992.

[26] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. *in PODS*, pages 155–166, 1999.

[27] http://www.mathsisfun.com/associative-commutative-distributive.html.

[28] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. *In International conference on Software Engineering (ICSE)*, pages 97–106, 2005.

[29] C. Cunningham, G. Graefe, and C. A. Galindo-Legaria. Pivot and unpivot: Optimization and execution strategies in an rdbms. *In International conference on Very Large Data Bases (VLDB)*, pages 998–1009, 2004.

[30] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *International Conference on Database Theory*, 2001.

[31] B. Dinter, C. Sapia, G. Hofling, and M. Blaschka. The OLAP market: State of the art and research issues. *ACM First International Workshop on Data Warehousing and OLAP*, pages 22–27, 1998.

[32] http://www.w3schools.com/Dom/dom_parser.asp/.

[33] http://en.wikipedia.org/wiki/Dynamic_programming.

[34] E.Zimanyi E. Malinowski. Hierarchies in a conceptual model: From conceptual modeling to logical representation. *Data & KNowledge Engineering*, 2005.

[35] T. Eavis and D. Cueva. A hilbert space compression architecture for data warehouse environments. *22nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK 07)*, 2007.

[36] T. Eavis and D. Cueva. The lbf r-tree: Efficient multidimensional indexing with graceful degradation. *22nd International Database Engineering and Applications Symposium (IDEAS 07)*, 2007.

[37] T. Eavis and R. Sayeed. High performance analytics with the r3-cache. *Data Warehousing and Knowledge Discovery (DaWak)*, 2009.

[38] Todd Eavis, George Dimitrov, Ivan Dimitrov, David Cueva, Alex Lopez, and Ahmad Taleb. Sidera: A cluster-based server for online analytical processing. *International Conference on Grid computing, high-performAnce, and Distributed Applications (GADA)*, 2007.

[39] Todd Eavis and Ahmad Taleb. Mapgraph: efficient methods for complex olap hierarchies. *Conference on Information and Knowledge Management*, pages 465–474, 2007.

[40] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of DATABASE SYSTEMS/Fouth Edition*. Pearson/Addison Wesley, 2006.

[41] Fastbit. https://sdm.lbl.gov/fastbit/.

[42] Y. Feng, D. Agrawal, A. Abbadi, and A. Metwally. Range cube: Efficient cube computation by exploiting data correlation. *ICDE*, 2004.

[43] E. Franconi and A. Kamble. The gmd data model and algebra for multidimensional information. *In: Proc. of the 16th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2004).*, 3084:446–462, 2004.

[44] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[45] Arnaud Giacometti, Dominique Laurent, Patrick Marcel, and Hassina Mouloudi. A new way of optimizing olap queries. *BDA*, pages 513–534, 2004.

[46] J. Goldstein and Per-Ake Larson. Optimizing queries using materialized views: A practical, scalable solution. *in SIGMOD*, pages 331–342, 2001.

[47] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. *In International conference on Data Engineering (ICDE),Washington, DC, USA*, pages 152–159, 1996. IEEE Computer Society.

[48] http://en.wikipedia.org/wiki/Greedy_algorithm.

[49] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for OLAP. *Proceeding of the 13th International Conference on Data Engineering*, pages 208–219, 1997.

[50] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57, 1984.

[51] M. Gyssens and L. V. S. Lakshmanan. A foundation for multi-dimensional databases. *In International conference on Very Large Data Bases (VLDB)*, pages 106–115, 1997. Morgan Kaufmann Publishers Inc.

[52] M.S. Hacid and U. Sattler. Modeling multidimensional database:a formal objectcentered approach. *In: Proc. of the 6th European Conference on Information Systems (ECIS 1998)*, 1998.

[53] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, 2000.

[54] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. *In SIGMOD*, 2001.

[55] Sivakumar Harinth and Stephen Quinn. Professional sql server analysis services 2005 with mdx. *Microsoft Business Inteligence Platform*, 2005.

[56] http://en.wikipedia.org/wiki/Hash_table.

[57] C. Hurtado, A. Mendelzon, and A Vaisman. Maintaining data cubes under dimension updates. *Proceedings of the IEEE Interantional Conference on Data Engineering*, 1999.

[58] D. Kossmann J.-P. Dittrich and A. Kreutz. Bridging the gap between olap and sql. *In International conference on Very Large Data Bases (VLDB)*, pages 1031–1042, 2005.

[59] H. Jagadish. Linear clustering of objects with multiple attributes. *ACM SIGMOD International Conference on Management of Data*, pages 332–342, 1990.

[60] H. V. Jagadish, Laks V. S. Lakshmanan, and Divesh Srivastava. What can hierarchies do for data warehouses? *The VLDB Journal*, pages 530–541, 1999.

[61] Jsr 243: Java data objects 2.0 - an extension to the jdo specification, 2003. http://java.sun.com/products/jdo/.

[62] I. Kamel and C. Faloutsos. On packing r-trees. *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 490–499, 1993.

[63] Ralph Kimball and J. Caserta. *The Data Warehouse ETL Toolkit*. John Wiley and Sons, 2004.

[64] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit*. John Wiley and Sons, 2002.

[65] N. Kotsis and D. R. McGregor. Elimination of redundant views in multidimensional aggregates. *In DaWak 2000*.

[66] Teo Lachev. Applied microsoft analysis services. *Microsoft Business Inteligence Platform*, 2005.

[67] L.V.S. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. *VLDB 2002*.

[68] W. Lehner. Modelling large scale olap scenarios. *In International conference on Extending Database Technology (EDBT 1998)*, pages 153–167, 1998.

[69] H. Lenz and A. Shoshani. Summarizability in OLAP and statistical data bases. *Proceedings of the Ninth International Conference on Scientific and Statistical Database Management*, pages 132–143, 1997.

[70] A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. *in PODS*, pages 95–104, 1995.

[71] Witold Litwin. Linear hashing: A new tool for file and table addressing. *Proc. 6th Conference on Very Large Databases*, pages 212–223, 1980.

[72] J. Pei L.V.S. Lakshmanan and Y. Zhao. Qctrees: An efficient summary structure for semantic olap. *SIGMOD 2003.*

[73] R. Zare M. Whitehorn and M. Pasumansky. Fast track to mdx. *Springer-Verlag New York, Inc., Secaucus, NJ, USA*, 2005.

[74] E. Malinowski and E.Zimanyi. Olap hierarchies: A conceptual perspective. *Advanced Information Systems Engineering, 16th International Conference, CAiSE*, 2004.

[75] Volker Markl and Rudolf Bayer. Processing relational olap queries with ub-trees and multidimensional hierarchical clustering. *DMDW*, 2000.

[76] Analysis Services: MDX, 2006. http://msdn.microsoft.com.

[77] J. Melton. Advanced sql 1999: Understanding object-relational, and other advanced features. *Elsevier Science Inc., New York, NY, USA*, 2002.

[78] Mondrian. http://www.mondrian.pentaho.org.

[79] K. Morfonios and Y. Ioannidis. Cure for cubes: cubing using a rolap engine. *In International conference on Very Large Data Bases (VLDB)*, pages 379–390, 2006. VLDB Endowment.

[80] S. Muto and M. Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. *ACM 2nd Annual Workshop on Data Warehousing and OLAP*, pages 67–72, 1999.

[81] Mysql. http://www.mysql.com/.

[82] Thomas P. Nadeau and Toby J. Teorey. Olap query optimization in the presence of materialized views. *HICCS*, 2003.

[83] R. Ng, A. Wagner, and Y. Yin. Iceberg-cube computation with PC clusters. *Proceedings of 2001 ACM SIGMOD Conference on Management of Data*, pages 25–36, 2001.

[84] T. Niemi, J. Nummenmaa, and P. Thanish. Logical multidimensional database design for ragged and unbalanced aggregation hierarchies. *International Workshop on Design and Management of Data Warehouses, DMDW 2001*, pages 1–8, 2001.

[85] Olap4j. http://www.olap4j.org/.

[86] Olapdml. http://oracle.com/.

[87] P. ONeil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD*, pages 8–11, 1995.

[88] P. ONeil and D. Quass. Improved query performance with variant indexes. *in ACM SIGMOD*, pages 38–49, 1997.

[89] Oracle olap. http://www.oracle.com/technology/products/bi/olap/index.html.

[90] T. Pedersen, C. Jensen, and C. Dyreson. A foundation for capturing and querying complex multidimensonal data. *Information Systems Journal*, 26(5):383–423, 2001.

[91] Nathan Goodman Philip A. Bernstein, Vassos Hadzilacos. Concurrency control and recovery in database systems. *Addison Wesley Publishing Company*, 1987.

[92] JR. Pottinger and A. Levy. A scalable algorithm for answering queries using views. *VLDB*, pages 484–495, 2000.

[93] A. Gupta R. Agrawal and S. Sarawagi. Modeling multidimensional databases. *In International conference on Data Engineering (ICDE)*, pages 232–243, 1997. IEEE Computer Society.

[94] Andreas Reuter and Theo Haerder. Principles of transaction-oriented database recovery. *ACM Computing Surveys (ACSUR)*, 1983.

[95] O. Romero and A. Abello. On the need of a reference algebra for olap. *In International conference on Data warehousing and Knowledge Discovery (DaWak)*, pages 99–110, 2007.

[96] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, pages 266–283, 1976.

[97] N. Roussopoulos, Y. Kotidis, and M. Roussopolis. Cubetree: Organization of the bulk incremental updates on the data cube. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 89–99, 1997.

[98] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. *ACM SIGMOD Conference*, pages 17–31, 1985.

[99] Hans Sagan. Space-filling curves. *Springer-Verlag*, 1994.

[100] A. Sanjay, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, 2004.

[101] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996.

[102] Sas olap server. http://www.sas.com/technologies/dw/storage/mddb/index.html.

[103] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree - a dynamic index for multidimensional objects. *VLDB*, pages 507–518, 1987.

[104] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1990.

[105] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. *Proceedings of the 22nd VLDB Conference*, pages 522–531, 1996.

[106] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the PetaCube. *Proceedings of the 2002 ACM SIGMOD Conference*, pages 464–475, 2002.

[107] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Hierarchical dwarfs for the rollup cube. *DOLAP 03: Proceedings of the 6th ACM international workshop on Data warehousing and OLAP*, pages 17–24, 2003.

[108] Spawner. http://sourceforge.net/projects/spawner/.

[109] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy. Answering queries with aggregation using views. *VLDB*, pages 318–329, 1996.

[110] Tpc-h. http://www.tpc.org/tpch/.

[111] W.Wang, H.Lu, J.Feng, and J.Xu Yu. Condensed cube: An effective approach to reducing data cube size. *ICDE 2002*.

[112] http://xerces.apache.org/xerces-c/.

[113] http://www.w3schools.com/Xml/xml_dtd.asp/.

[114] http://www.w3schools.com/schema/schema_intro.asp.

[115] Chendong Zou, Betty Salzberg, and Rivka Ladin. Back to the future: Dynamic hierarchical clustering. *ICDE*, 1998.

Appendix A

Twelve SQL Queries

This appendix illustrates the syntax of 12 SQL queries that are used in our experimental tests. They are as follows:

1. **SELECT** c.Region, SUM(s.Tolal_Sales)
   FROM customer as c , sales as s
   WHERE s.C_ID = c.C_ID and c.Age = 50 and c.Region = ‘Quebec’
   GROUP BY c.Region

2. **SELECT** t.Month, SUM(s.Tolal_Sales)
   FROM time as t, sales as s
   WHERE s.T_ID = t.t_ID and t.Year = 2005 and DayName =‘Monday’ and t.Quarter =‘Q1’
   GROUP BY t.Month

3. **SELECT** p.Type, SUM(s.Tolal_Sales)
   FROM product as p, sales as s
   WHERE s.P_ID = p.P_ID and p.ProdDesc = ‘Urna’ and p.Category =‘Automotive’ and p.Quantity=200
   GROUP BY p.Type

4. **SELECT** s.StoreCity, t.Month, SUM(ss.Tolal_Sales)

FROM time as t , store as s, sales as ss
WHERE t.T_ID = ss.T_ID and ss.S_ID = s.S_ID and ((t.year=2005 and t.DayName = ‘Monday’) and (t.Quarter=‘Q1’ or t.Quarter=‘Q2’)) and s.StoreState=‘Ontario’
GROUP BY s.StoreCity, t.Month

5. **SELECT** c.Region, p.Category, SUM(ss.Tolal_Sales)
   FROM customer as c, product as p, sales as ss
   WHERE ss.C_ID = c.C_ID and ss.P_ID = p.P_ID and c.Age = 40 and c.Country = ‘Canada’ and p.Quantity=200 and p.Category=‘Automotive’
   GROUP BY c.Region, p.Category

6. **SELECT** c.country, t.Month, SUm(ss.Tolal_Sales)
   FROM customer as c , time as t, sales as ss
   WHERE ss.C_ID = c.C_ID and ss.T_ID = t.T_ID and (((t.year=2005 and t.DayName=‘Monday’) and (t.Month=‘May’ or t.Month=‘June’)) and (c.Age = 40 and c.Region = ‘Quebec’))
   GROUP BY t.Month,c.country

7. **SELECT** c.Region, p.Type, s.StoreCity, SUM(ss.Tolal_Sales)
   FROM customer as c, product as p , store as s, sales as ss
   WHERE ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = ‘Ontario’ and s.StoreState= ‘Ontario’ and p.Category = ‘Household’
   GROUP BY c.Region, p.Type, s.StoreCity

8. **SELECT** s.StoreCity, t.Month, SUM(ss.Tolal_Sales)
   FROM time as t,customer as c , store as s, sales as ss
   WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and (t.year=2005 and t.DayName=‘Monday’) and (c.Age = 40 and c.Region = ‘Quebec’) and s.StoreState = ‘Ontario’
   GROUP BY s.StoreCity, t.Month

9. **SELECT** t.Quarter, p.Type, s.StoreCity, SUM(ss.Tolal_Sales)
   FROM time as t, product as p , store as s, sales as ss
   WHERE t.T_ID = ss.T_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and t.Year = 2005 and t.Quarter = 'Q1' and s.StoreState= 'Ontario' and p.Category ='Household'
   GROUP BY t.Quarter,p.Type, s.StoreCity

10. **SELECT** t.Quarter,c.Region, p.Type, s.StoreCity, SUM(ss.Tolal_Sales)
    FROM time as t,customer as c, product as p , store as s, sales as ss
    WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = 'Ontario' and s.StoreState= 'Ontario' and p.Category = 'Household'
    GROUP BY c.Region, p.Type, s.StoreCity, t.Quarter

11. **SELECT** p.Type, s.StoreState, SUM(ss.Tolal_Sales) **FROM** time as t,customer as c, product as p , store as s, sales as ss
    WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = 'Ontario' and t.Quarter= 'Q1' and p.Category = 'Household'
    GROUP BY p.Type, s.StoreState

12. **SELECT** t.Quarter, SUM(ss.Tolal_Sales)
    FROM time as t,customer as c, product as p , store as s, sales as ss
    WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Age = 40 and t.year =2005 and s.StoreState= 'Ontario' and p.Type = 'Engine'
    GROUP BY t.Quarter

Appendix B

Sixteen SQL Queries

This appendix illustrates the syntax of 16 SQL queries that are used in our experimental tests. They are as follows:

1. **SELECT** c.country, p.Type, t.Month, v.CountryName, Sum(ss.Tolal_Sales)
   FROM customer as c, product as p , time as t, salessix as ss, vendor v
   WHERE v.V_ID=ss.V_ID and ss.C_ID = c.C_ID and ss.T_ID = t.T_ID and ss.P_ID = p.P_ID and c.Age = 40 and c.Region = 'Quebec'
   GROUP BY c.country, p.Type, t.Month, v.CountryName

2. **SELECT** c.Region, p.Type, sum(ss.Tolal_Sales)
   FROM customer as c, product as p , salessix as ss, employee as e
   WHERE e.E_ID = ss.E_ID and ss.C_ID = c.C_ID and ss.P_ID = p.P_ID and c.Age = 30 and c.Region = 'Quebec' and e.FirstName = 'Thor'
   GROUP BY c.Region, p.Type

3. **SELECT** t.Quarter,c.Region, p.Type, s.StoreCity,v.CountryName, sum(ss.Tolal_Sales)
   FROM time as t,customer as c, product as p , store as s, salessix as ss, vendor as v
   WHERE v.V_ID= ss.V_ID and t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = 'Ontario' and s.StoreState=

‘Ontario’ and p.Category = ‘Household’
GROUP BY c.Region, p.Type, s.StoreCity, t.Quarter,v.CountryName

4. **SELECT** c.country, p.Type, t.Quarter, SUm(ss.Tolal_Sales)
   FROM employee as e, customer as c, product as p , time as t, salessix as ss
   WHERE e.E_ID = ss.E_ID and ss.C_ID = c.C_ID and ss.T_ID = t.T_ID and ss.P_ID = p.P_ID and (((t.year = 2005 or DayName = ‘Monday’) and (t.Month=‘June’ or t.Month=‘May’)) AND (c.Age = 40 and c.Region = ‘Quebec’)) and e.LastName = ‘Vinson’
   GROUP BY c.country, t.Quarter

5. **SELECT** c.country, t.Year, v.City , sum(ss.Tolal_Sales)
   FROM vendor as v, customer as c , time as t, salessix as ss
   WHERE ss.v_ID = v.V_ID and ss.C_ID = c.C_ID and ss.T_ID = t.T_ID and t.Quarter=‘Q2’and (c.Age = 40 and c.Region = ‘Ontario’)
   GROUP BY c.country, t.Year,v.City

6. **SELECT** p.Type, SUM(ss.Tolal_Sales)
   FROM employee as e, product as p , salessix as ss, vendor as v
   WHERE v.V_ID= ss.V_ID and e.E_ID = ss.E_ID and ss.P_ID = p.P_ID and e.LastName = ‘Moore’ and v.StateName = ‘Quebec’
   GROUP BY p.Type

7. **SELECT** v.StateName,t.Month, sum(ss.Tolal_Sales)
   FROM vendor as v, time as t, salessix as ss **WHERE** ss.V_ID = v.V_ID ss.T_ID = t.T_ID and v.CountryName = ‘Australia’
   GROUP BY v.StateName, t.Month;

8. **SELECT** t.Quarter,c.Region, p.Type, s.StoreCity,v.CountryName, SUM(ss.Tolal_Sales)
   FROM employee as e, time as t,customer as c, product as p , store as s, salessix as ss, vendor as v

WHERE v.V_ID= ss.V_ID and e.E_ID = ss.E_IDand t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = 'Ontario' and s.StoreState= 'Ontario' and e.LastName = 'Moore'
GROUP BY c.Region, p.Type, s.StoreCity, t.Quarter,v.CountryName

9. **SELECT** t.Quarter, SUM(ss.Tolal_Sales)
   FROM time as t, salessix as ss
   WHERE ss.T_ID = t.T_ID and t.Year='2004' and t.DayName = 'Monday'
   GROUP BY (t.Quarter);

10. **SELECT** p.Category,v.CountryName, sum(ss.Tolal_Sales)
    FROM vendor as v, product as p, salessix as ss
    WHERE ss.V_ID=v.V_ID and ss.P_ID = p.P_ID and v.VendorName = 'International' and p.Quantity=200
    GROUP BY v.CountryName, p.Category

11. **SELECT** s.StoreCity, v.CountryName, SUM(ss.Tolal_Sales)
    FROM time as t , store as s, salessix as ss, vendor as v
    WHERE t.T_ID = ss.T_ID and v.V_ID=ss.V_ID and ss.S_ID = s.S_ID and (t.year=2005 and t.DayName='Monday') and s.StoreState='Ontario'
    GROUP BY s.StoreCity,v.CountryName

12. **SELECT** c.Region, SUm(ss.Tolal_Sales)
    FROM customer as c , salessix as ss
    WHERE ss.C_ID = c.C_ID and c.Age = 50 and c.Country = 'Canada'
    GROUP BY c.Region

13. **SELECT** s.StoreCity,v.StateName, sum(ss.Tolal_Sales)
    FROM vendor as v, salessix as ss,store as s
    WHERE v.V_ID = ss.V_ID and s.S_ID = ss.S_ID and v.CountryName =

‘Canada’ and s.StoreState = ‘Quebec’

GROUP BY v.StateName, s.StoreCity

14. **SELECT** s.StoreCity,t.Quarter, SUM(ss.Tolal_Sales)

    FROM employee as e, time as t, store as s, salessix as ss

    WHERE e.E_ID = ss.E_ID and t.T_ID = ss.T_ID and ss.S_ID = s.S_ID and e.LastName = ‘Moore‘and s.StoreState=‘Quebec’

    GROUP BY s.StoreCity, t.Quarter

15. **SELECT** t.Quarter,c.Region, p.Type, s.StoreCity, SUM(ss.Tolal_Sales)

    FROM time as t,customer as c, product as p , store as s, salessix as ss

    WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and c.Region = ‘Ontario’ and s.StoreState= ‘Ontario’ and p.Category = ‘Household’

    GROUP BY c.Region, p.Type, s.StoreCity, t.Quarter

16. **SELECT** v.City, p.Type, s.StoreCity, SUM(ss.Tolal_Sales)

    FROM employee as e, customer as c, time as t,product as p , store as s, salessix as ss, vendor as v

    WHERE t.T_ID = ss.T_ID and v.V_ID= ss.V_ID and e.E_ID = ss.E_ID and ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and ss.P_ID = p.P_ID and e.LastName = ‘Moore’ and c.Region = ‘Ontario’ and c.Age=60 and t.Year = 2007 and t.DayName=‘Monday’

    GROUP BY v.City, p.Type, s.StoreCity