NATIVE LANGUAGE OLAP QUERY EXECUTION

Hiba Tabbara

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

September 2012

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Miss Hiba Tabbara**

Entitled: **Native language OLAP query eXecution**

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

Doctor of Philosophy (Computer Science)

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

Signed by the final examining commitee:

Dr. Chun-Yi Su	Chair
Dr. Rokia Missaoui	External Examiner
Dr. Rachida Dssouli	Examiner
Dr. Peter Grogono	Examiner
Dr. Joey Paquet	Examiner
Dr. Todd Eavis	Supervisor

Approved by Dr. Volker Haarslev
Chair of Department or Graduate Program Director

20 June 2012 Robin Drew, Ph.D., Dean
Faculty of Engineering and Computer Science

Abstract

Native language OLAP query eXecution

Online Analytical Processing (OLAP) applications are widely used in the components of contemporary Decision Support systems. However, existing OLAP query languages are neither efficient nor intuitive for developers. In particular, Microsoft's Multidimensional Expressions language (MDX), the de-facto standard for OLAP, is essentially a string-based extension to SQL that hinders code refactoring, limits compile-time checking, and provides no object-oriented functionality whatsoever.

In this thesis, we present Native language OLAP query eXecution, or NOX, a framework that provides responsive and intuitive query facilities. To this end, we exploit the underlying OLAP conceptual data model and provide a clean integration between the server and the client language. NOX queries are object-oriented and support inheritance, refactoring and compile-time checking. Underlying this functionality is a domain specific algebra and language grammar that are used to transparently convert client side queries written in the native development language into algebraic operations understood by the server. In our prototype of NOX, JAVA is used as the native language. We provide client side libraries that define an API for programmers to use for writing OLAP queries. We investigate the design of NOX through a series of real world query examples. Specifically, we explore the following: fundamental `SELECTION` and `PROJECTION`, set operations, hierarchies, parametrization and query inheritance. We compare NOX queries to MDX and show the intuitiveness and robustness of NOX. We also investigate NOX expressiveness with respect to MDX from an algebraic point of view by demonstrating the correspondence of the two approaches in terms of `SELECTION` and `PROJECTION` operations.

We believe the practical benefit of NOX-style query processing is significant. In short, it largely reduces OLAP database access to the manipulation of client side, in memory data objects.

Acknowledgments

It was always the insightful comments and positive criticism of my supervisor Dr. Todd Eavis that helped me be on the right track to complete my thesis work. Without his continuous support the research presented in this thesis would not have been possible.

I would like to thank all my colleagues in the Computer Science and Software Engineering department at Concordia University, as well as the staff and faculty of the department for their commitment to further education.

This thesis will not have been completed without the full support and love of my family especially my mom, dad, brother and sister. Special thanks go to my study mates and friends Bassel Bitar, Rania Khattab, Mazen El Masri, Genevieve Turmel, Rasha Samaha and Jocelyne Faddoul for their continuous encouragement.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Information is often seen as a kind of digital treasure in the current era, with captured data providing a wealth of information and analytical opportunities. In industrial settings, data warehousing and Online Analytical Processing (OLAP) have become two of the most significant technologies in this regard. Together, they enable efficient, multidimensional analysis of data in a multitude of industries such as retail sales, telecommunications, financial services and real estate [CD97] [SBSR08]. In practice, consumer-focused companies collect terabytes of information on past transactions that, in turn, enables them to define and target both new and potential customers. Real world examples of the value and scope of the data analysis process include:

1. WalMart uses approximately half a petabyte of customer transaction data to forecast demand and increase revenue [Hay04]. Analysis of sales transactions after a hurricane resulted in the discovery that the normal volume of pop-tarts and beer sold increased by a factor of seven. An analysis of cold medicine purchases revealed that they are often accompanied with purchases of soup and

orange juice.

2. Pharmaceutical companies use data mining techniques to discover and extract useful patterns from their large sets of data. Manipulation and classification of this data helps improve the quality of drug discovery processes and delivery methods while still competing on lower costs [Ran05].

3. Financial companies rely on data warehousing to explore new customer opportunities. For examples, users employ tools such as Microsoft Analysis Services and SAP's Business Information Warehouse for the analysis of data held in the data warehouse. Ultimately, OLAP allows decision-makers to quickly and interactively analyze the multi-dimensionally modeled data relevant to various business considerations [Hil10].

Because of its impact, effective data collection and analysis has grown into a multi-billion dollar industry that is dominated by some of the world's largest software companies. Still, the supporting data management applications and interfaces remain complex and unintuitive, particularly for users and developers with little OLAP experience. For this reason, important opportunities exist for improved — or even completely new - data access and query models in this domain.

1.1 Motivation for the current research

Over the past three decades, relational database management systems (RDBMS) have secured their place as the cornerstone of contemporary data management environments [Sel08]. During that time, logical data models and query languages have matured to the point whereby database practitioners can almost unequivocally identify common standards and best practices. In particular, the ubiquitous relational data model and the Structured Query Language (SQL) have become synonymous with the notion of efficient storage and access of transactional data.

That being said, a number of new and important domain-specific data management applications have emerged in the past decade. At the same time, general programming languages have evolved, driven by a desire for both greater simplicity, modeling accuracy, reliability, and development efficiency. As such, a motivation to explore new data models, as well as the languages that might exploit them, has emerged [CW00].

One particular area of interest is the aforementioned Business Intelligence (BI)/OLAP domain. Typically, such systems work in conjunction with an underlying relational data warehouse that houses an integrated, time sensitive, repository of one or more organizational data stores. At its heart, BI attempts to abstract away some of the often gory details of the large warehouses so as to provide users with a cleaner, more intuitive view of enterprise data. Very often, in fact, BI applications effectively serve as wrappers for the supporting warehouses and, with varying levels of success, seek to

hide some of the warehouse's physical and design complexity. Beyond trivial exploitation of the BI facilities, however, meaningful analysis can become quite complex and can necessitate a considerable investment of the developer's time and energy [SC05].

We note, however, that although BI has long been recognized as providing the technologies, applications and practices for the collection, integration and analysis of data, no standard query interface for OLAP DBMSs has been developed. In practice, Microsoft's Multidimensional Expressions query language (MDX) — extended SQL — has become a de-facto choice in many production environments. Still, as will be discussed later in the thesis, use of such languages (MDX) can have a negative impact on programmer productivity. In particular, they force the programmer to become an expert in two very different languages (the implementation language and the query language) with completely different mental models. Moreover, the embedded query strings cannot be checked at compile time and the code cannot easily be refactored when the backend data model changes.

For this reason, there is a growing belief that the "one size fits all" approach does not and cannot meet current data management demands [SC05]. We believe that there is a need for more intuitive and powerful access languages that have the potential to dramatically enhance productivity, particularly in domains such as Business Intelligence that have unique but fairly well understood data models and query patterns.

1.2 Core Research Objectives

As noted, the OLAP/BI domain has not achieved the same level of standardization as seen in the world of transactional or operational databases. Of particular significance in this context is the awkward relationship between the development language and the data itself. For systems building directly upon an underlying relational data warehouse, BI querying still often relies upon non-procedural SQL or one of its proprietary variations. 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, OLAP concepts such as *data cubes*, *dimensions*, *aggregation hierarchies*, *granularity levels*, and *drill down* relationships map poorly at best to the standard logical model of relational systems.

A second related concern is the relative difficulty of integrating non-procedural query languages into application level source code. Larger development projects typically encounter one or more of the following limitations:

- The non Object-Oriented nature of the model minimizes the ability to separate the application's interface from its implementation.
- There are few possibilities for the code re-use that is afforded by OOP concepts such as inheritance and polymorphism.
- Utilizing two fundamentally distinct programming models concurrently (i.e., procedural OOP versus non-procedural non-OOP) complicates development.

- The use of embedded query strings (i.e., JDBC/SQL) severely limits the developer's ability to efficiently *refactor* source code in response to changes in schema design.

- Comprehensive compile-time type checking is often impossible since queries are simply passed to the backend DBMS at run time.

A final concern relates to the MDX language specifically. While it is true that the syntax of MDX is certainly more "OLAP friendly" than the set based SQL, it is important to note — particularly from an academic perspective — that MDX lacks any real formal basis. OLAP operators are not well-defined and no clean conceptual model is recognized. MDX is simply based on an ad hoc grammar that lacks an algebraic backbone. Not only is this aesthetically unappealing, it also limits query optimization opportunities by the supporting DBMS since it is difficult to cleanly represent the core operations of the language and the potential relationship between them.

Given the above, we may briefly list the primary research objectives of the current thesis as follows:

- We would like to provide an OLAP-specific algebra and associated language grammar that defines the core operations associated with the OLAP domain.

- The algebra should be backed by a conceptual data model that directly supports these operations.

- The combination of algebra, grammar and data model should then provide or permit the following:
  - An intuitive Object-Oriented query model,
  - Associated code re-use afforded by OOP concepts such as inheritance and polymorphism,
  - The ability of developers to efficiently *refactor* source code in response to changes in schema design,
  - Comprehensive compile-time type checking.
- The formal elements of the framework (algebra, grammar, data model) should be supported by a practical implementation (i.e., language libraries) providing the following features:
  - Developers should be able to write queries that interact with massive, remote data repositories using standard OOP principles and practices.
  - It should be possible to pass run-time parameters in a simple and intuitive way.
  - Query functionality should include support for the hierarchical access patterns typical of OLAP settings.
  - In terms of usability, the new approach should compare favorably to current languages such as MDX and SQL.

– Object-Oriented manipulation of results sets should be a component of the API.

1.3 Overview of Proposed Solution

In practice, the introduction of new database query languages or models requires the implementation of significant infrastructure. In the current case, we note that our OLAP research was initially inspired by the Safe Query Object (SQO) approach first introduced by Cook in 2005 [CR05, CR06]. There, query functionality was encapsulated in the native language of the application developer (e.g., Java), with a series of classes and methods that allowed the developer to conceptually represent the database as a local, in-memory data object. While Safe Query Objects were proposed for general relational environments — and were actually quite limited as a result — the general idea maps well to environments with more consistent conceptual data models. OLAP, in fact is one such domain.

Building upon this initial concept, we have proposed what we now refer to as the Native language OLAP query eXecution system (NOX). Briefly, NOX consists of the following elements:

- **OLAP conceptual model**. NOX allows developers to write code directly at the conceptual level; no knowledge of the physical or even logical schema is required.

- **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 OLAP algebra.

- **OLAP grammar**. Closely associated with the algebra is a DTD-encoded OLAP grammar that provides a concrete foundation for client language queries.

- **Client side libraries**. NOX provides a small suite of OOP classes corresponding to the objects of the conceptual model. Collectively, the exposed methods of the libraries form a clean programming API that can be used to instantiate OLAP queries. In the prototype, we note that Java is used as the development language.

- **Augmented compiler**. At its heart, NOX is a query re-writer. During a pre-processing phase, the framework's compilation tools (JavaCC/JJTree) effectively re-write source code to provide transparent model-to-DBMS query translation.

- **Cube result set**. OLAP queries essentially extract a subcube from the original space. The NOX framework exposes the result in a logical, read-only multi-dimensional array.

In practice, each of these elements plays a role in the definition, instantiation, and execution of a NOX query. Specifically, a developer would access the database as follows. Using the client side API, the query is encoded in the native language.

In addition to the fundamental Query class(es), the API exposes model elements such as dimensions, hierarchies, cube cells, etc. At compile time, the NOX pre-parser (JavaCC/JJTree) analyzes the source code to identify query elements (i.e., API components). Query logic, as well as query types, are verified. If valid, the query is converted into an algebraic representation that is physically encoded in an XML grammar. The XML string is then encapsulated within a network call to the DBMS and the updated source is recompiled by the standard (Java) compiler. At run-time, the network call to the backend DBMS is automatically invoked and query results are returned to the client and loaded into a result set object. It is important to note that the entire process, except of course the initial query specification, is entirely transparent to the developer.

Finally, we note that while NOX can be seen as a standalone framework whose core principles could be applied to existing DBMSs, it is currently implemented as a component of a larger research system known as Sidera. This DBMS system, described by Eavis et al [EDD+07], provides a robust parallel server for high performance OLAP environments. As illustrated in Figure 1, the NOX infrastructure, including the libraries and compiler tools, is accessible on the client PC/workstation. The output of the compilation phase is then transferred to the backend DBMS for optimization and execution, before the result is returned again to the client.

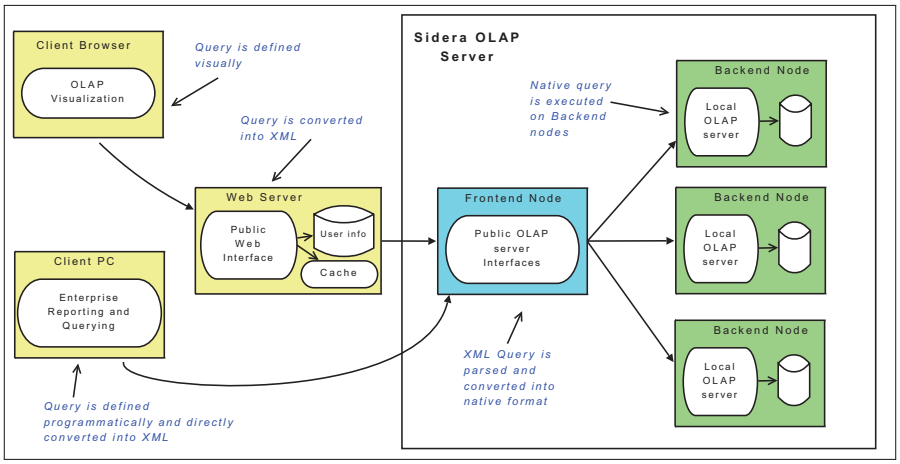


Figure 1: The Sidera system model

1.4 Research Evaluation

Because the purpose of NOX, and client side querying in general, is to provide an intuitive and accessible query environment, it is important to demonstrate that the research does indeed provide this functionality. Our evaluation takes two forms. First, we provide extensive examples of NOX queries on common OLAP access patterns. In particular, we provide examples of OLAP operations such as “slice and dice”, “roll up” and ”drill down”, and pivot. We also demonstrate the ease with which aggregation hierarchies can be traversed. Query examples illustrating the use of run-time parameters are listed as well. In many cases, we provide comparative examples using the MDX language so that readers can assess the relative simplicity/complexity of the two models. We emphasize the fact that because NOX provides a fully functionality prototype, all NOX queries listed in this thesis have been parsed, converted, and compiled using the structural components described above.

In addition to the implementation itself, we also provide an analysis of the language elements of NOX and MDX. Because MDX is associated with no formal algebra, we have performed the formal evaluation by way of a comparative classification of common OLAP query forms. In other words, we examine fundamental query patterns or classes, defining the algebraic features of each. We then show that the NOX native language model is in fact capable of supporting the primary forms found in practical settings. We note, of course, that NOX is also able to provide functionality that MDX can not, such as OOP-style inheritance, simplified refactoring, and compile time checking.

1.5 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 introduces the basic data warehouse architecture, the OLAP multidimensional model and its grammar, JavaCC and JJTree language parsing, and the Document Type Definition (DTD) schema. In Chapter 3, we follow this up with a literature review of query languages in both the relational databases world and the OLAP world. We also look at the differences between existing string based query languages that are still much used nowadays, as well as native language facilities utilized for querying relational databases.

We then present our new framework for querying OLAP systems in Java, namely Native language OLAP query eXecution (NOX), in Chapter 4. Chapter 5 illustrates the fundamentals of NOX application programming and demonstrates its usage through examples that have been implemented and tested in Java. Next, in Chapter 6,

we explore how NOX manipulates OLAP hierarchies and compare its performance to that of MDX in this context. Chapter 7 describes how passing parameters is done in NOX. We then investigate the formal basis by which we map the slicing and dicing operations of the NOX grammar to those of the MDX grammar in Chapter 8. Finally, Chapter 9 concludes the thesis and provides some pointers to future work.

Chapter 2

Background Material

In this chapter, we introduce some concepts that we need to be familiar with before discussing the details of our research. This thesis core material is considered to combine ideas from a number of different fields: `Data warehouses`, `OLAP systems`, `OLAP hierarchies` and `OLAP operations`, `Multidimensional modeling`, `JavaCC and JJTree`, and `DTD schema`.

Section 2.1 gives an overview of a typical Data warehouse, its architecture, its materialization and its star schema implementation. Section 2.2 introduces OLAP systems, data cubes, and the grammars used for OLAP, while Section 2.3 illustrates *multidimensional* modeling and its materialization. Then, the essential hierarchical structure of dimensions in a data warehouse is investigated in Section 2.4. Section 2.5 describes the commercial OLAP operations, while Section 2.6 introduces JavaCC and JJTree parsing in Java. Finally, Document Type Definition (DTD) definition of legal building blocks of XML-format documents is presented in Section 2.7.

2.1 Data Warehousing

Decision Support Systems (DSS) are defined as interactive computer-based systems intended to help decision makers utilize data and models in order to identify and solve problems and make decisions [Pow99]. A *Data Warehouse* is a repository of multiple heterogeneous data sources, organized under a unified schema in order to facilitate management decision making [HK06]. Data warehouse technology includes data cleansing, data integration, and OLAP analysis techniques with functionalities such as summarization, consolidation, and aggregation, as well as the ability to view information from different perspectives. In warehouses, data is typically represented in the form of decision cubes.

2.1.1 The Data Warehouse Architecture

A data warehouse can be seen as a three-tier architecture [CD97, HK06]. The canonical data warehouse architecture is shown in Figure 2 [SH98], with the possible data sources shown at the bottom of the figure. Information is extracted from various legacy systems and operational sources, and is then consolidated, summarized, and loaded into the data warehouse using a process commonly known as ETL (Extract, Transform, and Load). Strictly speaking, this first step is not one of the three tiers, as its functionality is external to the warehouse proper.

At the first tier, there is the data warehouse server, along with several data marts. Essentially, each data mart is a small warehouse designed for a specific department or business process. At this stage, we can assume that the ETL processing is complete

and the data warehouse is fully loaded and contains the data required for basic "decision support". The second tier houses the OLAP server/engine that allows users to access and analyze data in the warehouse, typically using more advanced techniques. Finally, the third tier includes the front end tools that provide a graphical interface for top managers and decision makers.

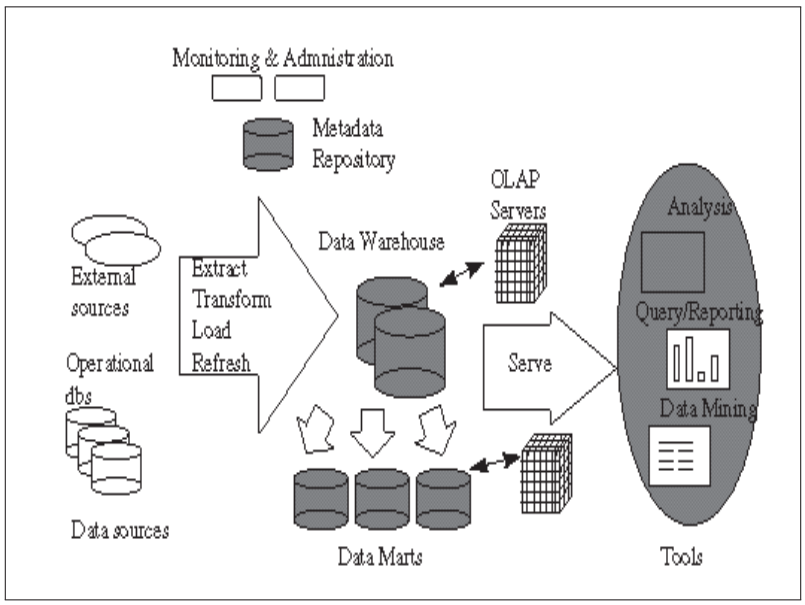


Figure 2: Typical data warehouse architecture

2.1.2 The Star Schema

The Star Schema, proposed by Kimball [KR02], is perhaps the simplest and most intuitive logical model for data warehouse design. The term "Star Schema" is derived from the fact that a graphical depiction of the schema resembles a star. Star Schemas consist of two basic table types: *dimension* tables and *fact* tables. A fact table contains measurement records such as the "total sales" in the fact table of the star schema given in Figure 3. These records model the business process and provide us

with measurements (or facts) in terms of the key dimensions in our data warehouse. In effect, these are the numbers that allow decision makers to actually make decisions. Dimensions are data warehouse "subjects". Dimensions in our example are Location, Product, Customer and Date tables. In practice, Fact tables are typically massive, holding perhaps billions of records (or facts), while Dimension tables are relatively small and contain information about the entries of a particular attribute in the fact table.

Note that the dimension tables are generally *denormalized*, meaning that the tables maintain some of the redundancy that a good OLTP (OnLine Transaction Processing) system typically eliminates. An example of a denormalized table, where some data is repeated, is given in Figure 6. At query time, each dimension table is joined to the fact table as necessary. In this setting, denormalizing the dimension tables significantly decreases the number of costly joins that would otherwise be required with a normalized schema. Since the dimension tables are comparatively small when compared to the enormous fact tables, the redundancy produced by the denormalization is of little interest in most OLAP contexts.

2.2 What is OLAP?

The term *OLAP* was first presented by E. F. Codd in 1992. It was presented in the context of a vendor sponsored paper called "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate" [CCS92], where he described twelve rules of OLAP. Codd indicated twelve features that should be present in any OLAP

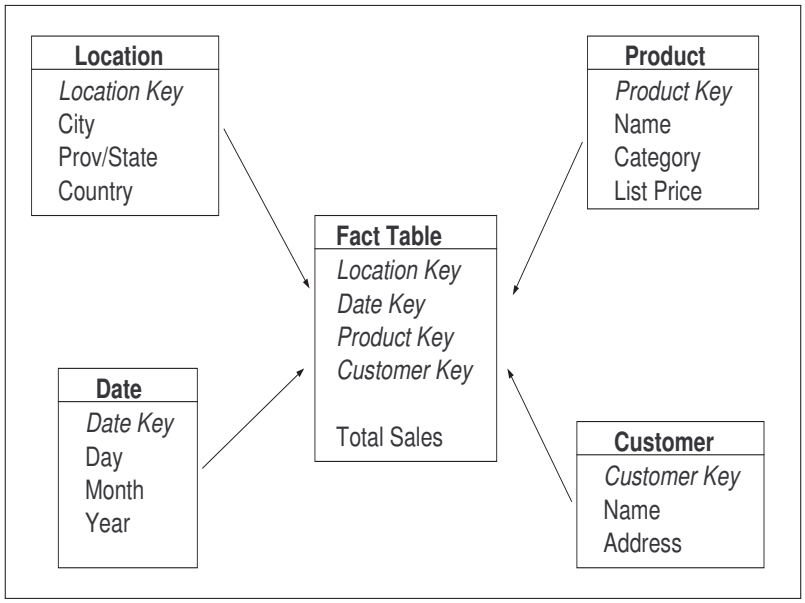


Figure 3: Star schema example

application. The following four points, taken from that report, are probably the most significant of the twelve:

- Multidimensional conceptual view. In contrast to relational databases that manipulate individual records or concepts, the focal point in OLAP is the relationship between multiple dimensions.

- Transparency. The end user should not have to worry about the details of data access or conversions. In addition, OLAP systems should be part of open systems that support heterogeneous data sources. Ultimately, the system should present a single logical schema of the data.

- Flexible reporting. Reporting must present data in a fully integrated manner, and minimize any restrictions in the way that basic data elements of dimensions are combined.

- Unlimited dimensional and aggregation levels. A serious tool should support more than just a few concurrent dimensions (Codd actually indicated that 15 - 20 would be ideal).

2.3 Multidimensional Modeling

Both data warehouses and OLAP systems are based on a *multidimensional* model. Specifically, we logically represent data in a d-dimensional space such as the one depicted in Figure 4. In this context, the multidimensional model can be described as a data abstraction allowing one to view aggregated data from a number of perspectives (dimensions). In fact, for a d-dimensional space, there are exactly 2^d distinct dimension combinations that represent the underlying Star Schema, each from a unique perspective. In OLAP terminology, we refer to this as the *data cube*.

As previously noted, low level information is divided into facts and dimensions. An individual fact represents an item or transaction of interest to the user. In the multidimensional data cube model, facts are aggregated into *measures* that are contained within cells of the data cube. In Figure 4, one can see the measure values on the front face of the cube. Simply put, a given measure represents a series of fact values that have been aggregated for a given combination of dimensions. In Figure 4, for example, if we assume that the measure represents "Total Sales", then we can see that total sales for Customer 3 in Location 1 for Product 2 has the value 7.

We note that the MD (Multi Dimensional) model is logical in nature. In other words, it makes no assumptions about how the data is physically stored. Advanced

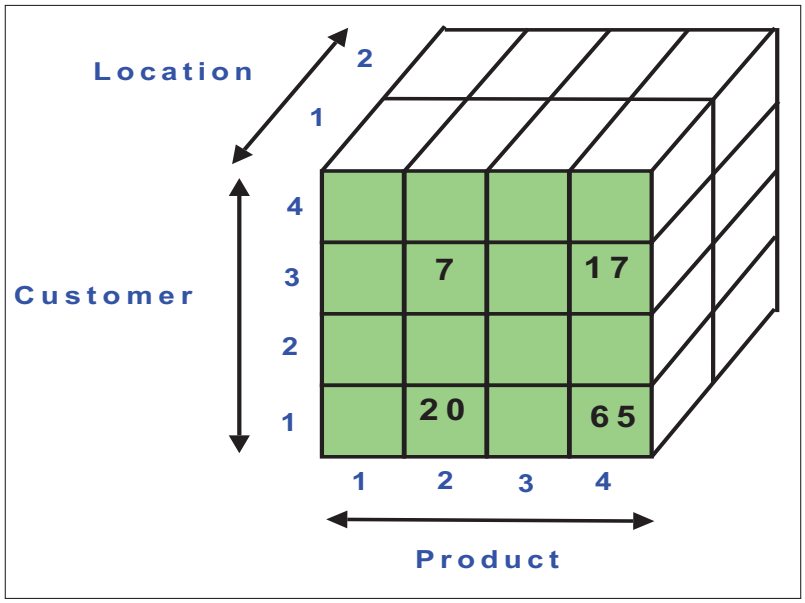


Figure 4: A three dimensional data cube example

OLAP servers may in fact take the data from the tables of the original Star Schema and further process it. The new data may be stored in a series of new tables or even a multi-dimensional array that represents a one-to-one mapping between the logical data cube and the physical storage. We refer to the first type of system as ROLAP (relational OLAP), while the second is known as MOLAP (multi-dimensional OLAP). We will not go into details of the physical storage format, as it is distinct from the primary focus of our research.

2.4 OLAP Hierarchies

Data granularity refers to the level of detail at which measures are presented. This is determined by a combination of the granularities within each dimension of the cube. For example, in Figure 4 the lowest level of granularity or detail in of the Customer dimension is Customer ID. However, the vast majority of common business

and scientific dimensions actually have a hierarchal structure. As a concrete example, the customer hierarchy, given in Figure 5, can be thought of in terms of NAME, TYPE, and REGION. In OLAP environments, the traversal of such "aggregation hierarchies" is perhaps the most fundamental of all query forms. Usually, OLAP tools only cope with hierarchies that ensure summarizability or that can be transformed so that summarizability conditions hold [LS97]. Summarizability refers to the correct aggregation of measures where a higher hierarchy level takes into account existing aggregations in a lower hierarchy level [MZ04].

As it turns out, there are in fact many different types of hierarchies in real-world applications. In the simplest case, we can think of a tree of dimension levels that is

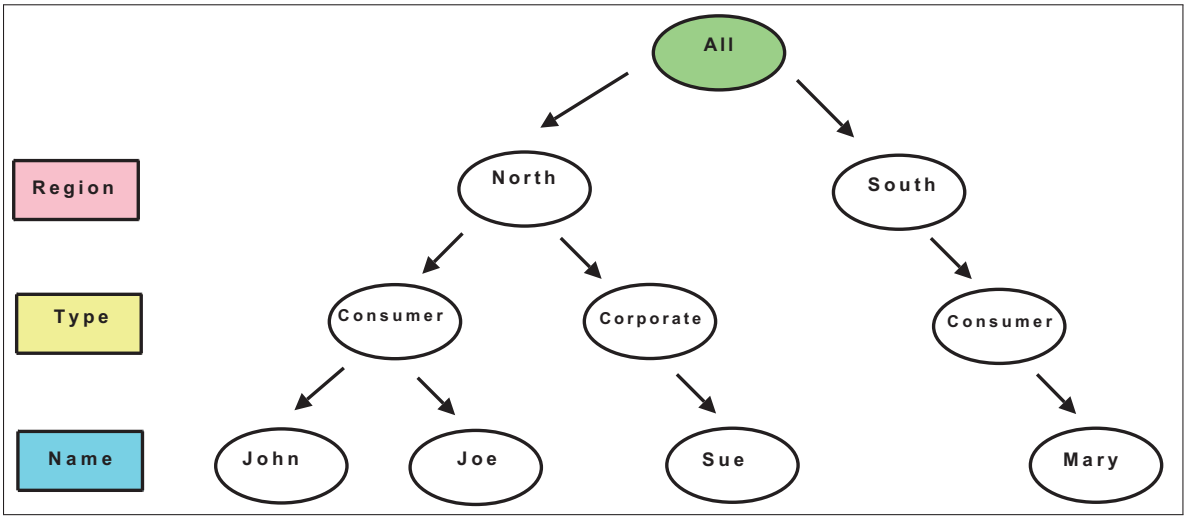


Figure 5: A *Customer* hierarchies example

constructed as a series of *one-to-many* relationships. An example of such a hierarchy is shown in Figure 5. Physically, simple trees like this are represented by additional columns in the associated dimension table, as depicted in the example in Figure 6. In fact, this is what we call a *denormalized* dimension. In a *normalized* model, there

would be three separate tables. In data warehouses, we typically denormalize the separate tables into a single table in order to improve performance by eliminating table joins.

Customer			
ID	Name	Type	Region
1	John	Consumer	North
2	Joe	Consumer	North
3	Sue	Corporate	North
4	Mary	Consumer	South

Figure 6: A dimension table corresponding to the *Customer* hierarchies example

2.5 OLAP Operators

Commercial OLAP systems may provide many OLAP functions and analytical extensions. In practice however, there are five fundamental operations that represent the bulk of query processing: *Slice*, *Dice*, *Roll-up*, *Drill-down* and *Pivot*. In the following section, we emphasize the slice and dice operations as they are the most relevant to the current thesis. Other operations are described briefly.

2.5.1 Slice

The slice operation performs a *selection* on one dimension of the given cube, thus resulting in a subcube. A slice is a subset of a multi-dimensional cube corresponding to a single attribute on one of the dimensions of the cube while allowing the other

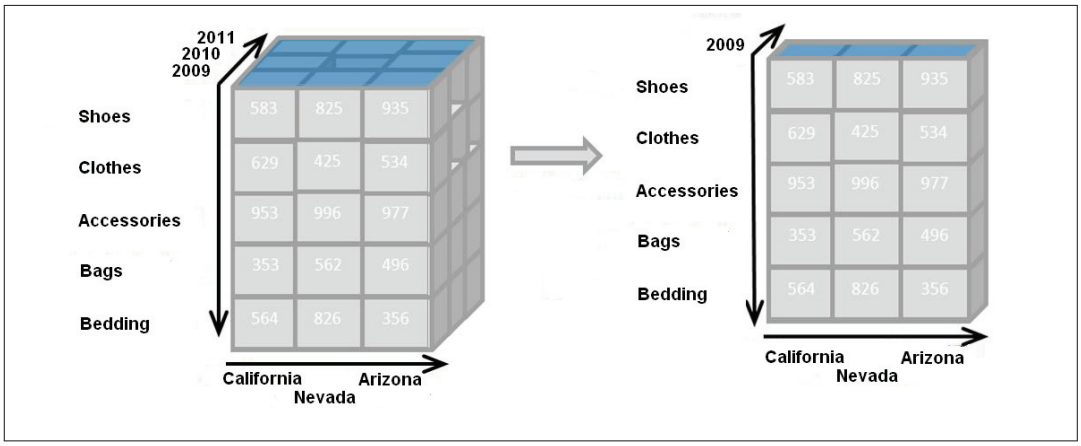


Figure 7: An OLAP slice

dimensions to vary. Figure 7 shows a slicing operation where the sales figures of all states and all product categories of the company in the year 2009 are "sliced" out of the data cube.

2.5.2 Dice

The dice operation is a slice on more than two dimensions of a data cube (or more than two consecutive slices). Figure 8 shows a dicing operation where the sales figures of a limited number of product categories are returned, and the time and region dimensions cover the same range as before.

2.5.3 The other Algebraic Multidimensional Operators

- Roll-up: The Roll-up operation acts on the hierarchical structure of a dimension. It aggregates values at a coarser level of granularity. Figure 9 shows a roll-up operation where values referring to insect protection, sun protection and first aid are summed up to values referring to outdoor protective equipment at a

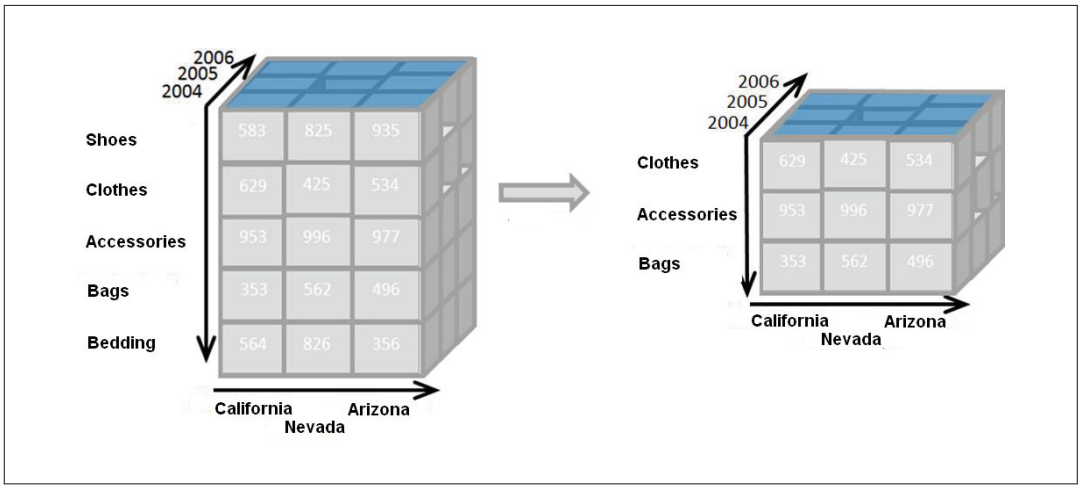


Figure 8: An OLAP dice

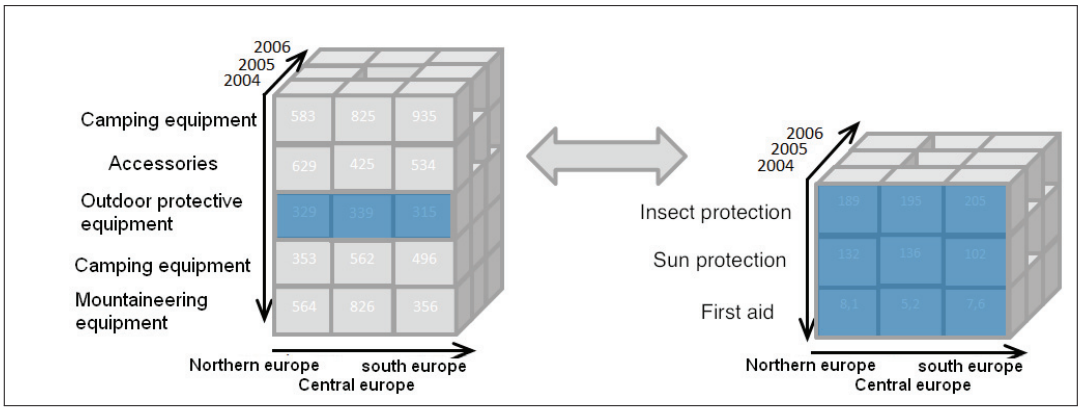


Figure 9: OLAP drill-down and roll-up

coarser level of the hierarchy of the dimension.

- Drill-down: The Drill-down operation also acts on the hierarchical structure of a dimension. It performs the opposite of what Roll-up does. It decomposes the aggregation at a finer level of detail. Figure 9 shows the drill-down operation where values referring to outdoor protective equipment are decomposed into values referring to insect protection, sun protection and first aid at a finer level of the hierarchy of the dimension.

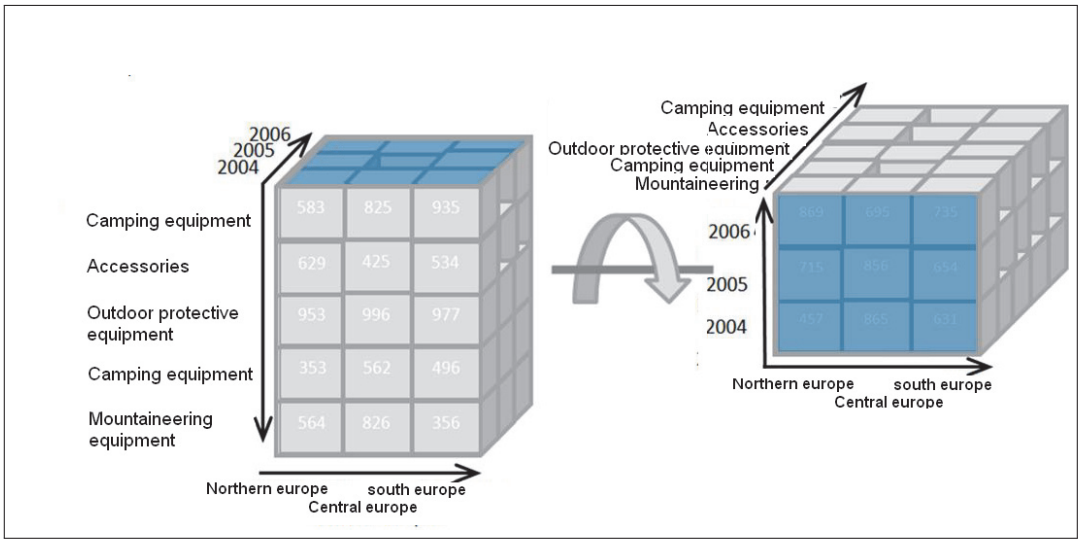


Figure 10: OLAP pivoting

- Pivot: The pivot operation acts on a cube by re-organizing its axes. The result can be more dramatic with a tabular representation. Figure 10 shows the pivot operation where the years dimension and the equipments dimension switched places.

2.6 JavaCC and JJTree Parsing

Java Compiler Compiler (JavaCC) and JJTree are language design tools that play a fundamental role in the Java prototype at the heart of this research. In this section, we give an overview of the structure and processing logic of both JavaCC and JJTree. At least a basic grasp of their processing logic is required for a meaningful understanding of the material presented in the thesis. JavaCC is the most popular parser generator for use with Java applications. In short, a *parser generator* is a tool that reads a grammar specification and converts it to a program that can recognize matches

to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions and debugging. The generated tree is known as AST (Abstract Syntax Tree) or parse tree.

JJTree is a pre-processor to JavaCC that inserts parse tree building actions at various places in the JavaCC source. The output of JJTree is run through JavaCC to create the parser and its parse tree. Each node of the tree denotes a construct found in the source code. By default, JJTree generates code to construct parse tree nodes for each nonterminal in the language. This behavior can be modified so that some nonterminals do not have nodes generated, or so that a node is generated for a part of a production expansion [Jav, JJT]. An example of a parse tree is depicted in Figure 11, where a node is denoted by an oval shape with the name of the node written inside the shape.

JJTree defines a Java interface Node that all parse tree nodes must implement. The interface provides methods for operations such as setting the parent of the node, and for adding children and retrieving them. Now, the structure of the trees gives the abstract syntax of the input, but not, by default, the tokens. We can capture the tokens as needed.

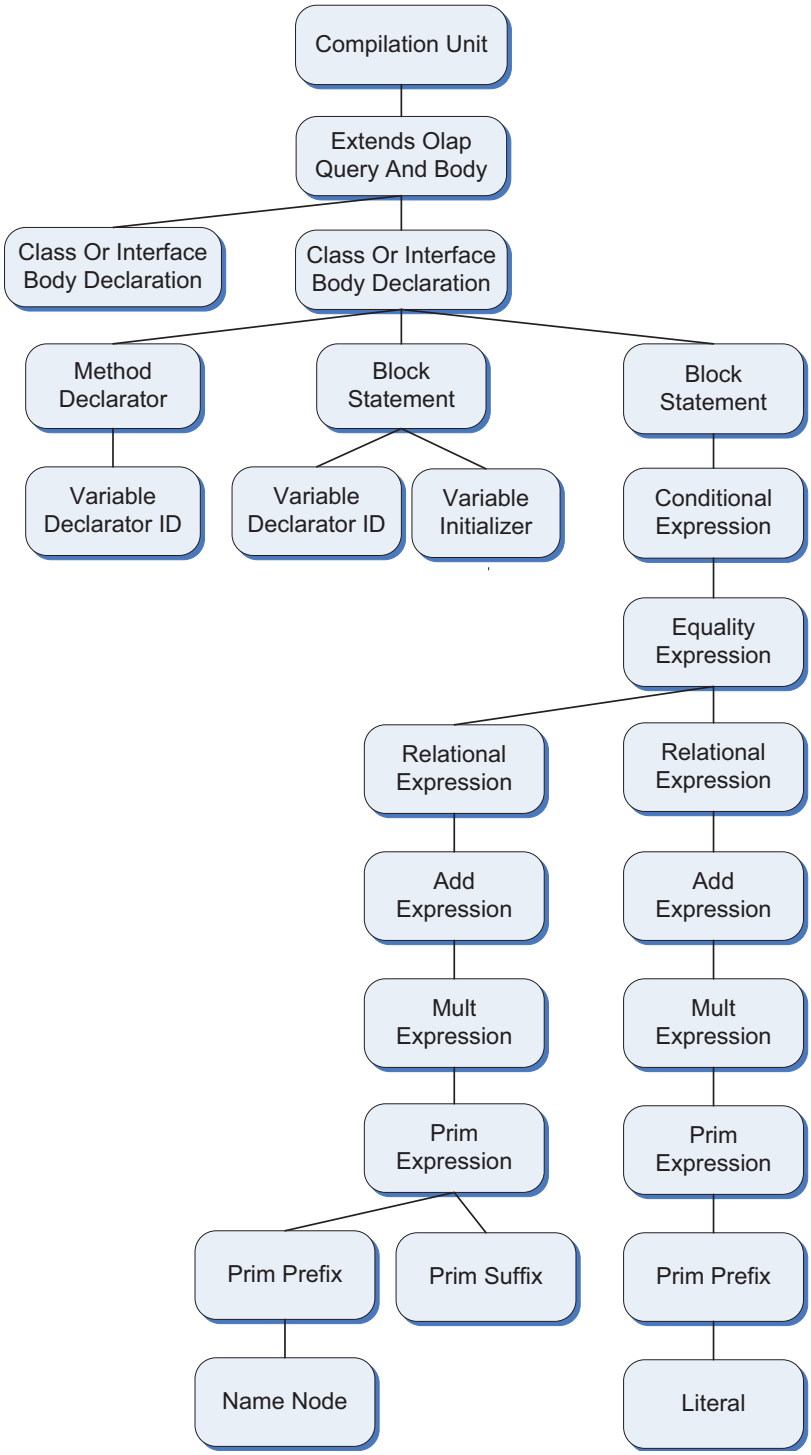


Figure 11: Simple parse tree

2.7 What is Document Type Definition (DTD) Schema?

As we will see, XML plays an important role in the concrete specification of our OLAP queries. A *Document Type Definition* (DTD) defines the legal building blocks of an XML document. It defines the document structure with a list of legal elements and attributes. A DTD describes the structure of XML documents by declaring each eligible element and its attribute list. Element declarations name the allowable set of elements within the document, and specify whether and how declared elements and character data may be contained within each element. Attribute list declarations name the allowable set of attributes for each declared element, including the type of each attribute value, if not an explicit set of valid value(s). A DTD is associated with an XML document via a Document Type Declaration, which is a tag that appears near the start of the XML document. The declaration establishes that the document is an instance of the type defined by the referenced DTD [DTDb]. An example of a Document Type Declaration is given in Listing 2.1.

```
<?xml version=``1.0'' encoding=``UTF-8'' standalone=``no''?>
<!DOCTYPE QUERY SYSTEM ``dtd/ClientQuery.dtd''>
```

Listing 2.1: DTD declaration

For more information about DTD schema, refer to Appendix B

2.8 Conclusion

We introduced in this chapter the concept of a data warehouse, its architecture and its design. A data warehouse is a repository of multiple heterogeneous data sources, organized under a unified schema in order to facilitate management decision making. The Star Schema is perhaps the simplest and most intuitive logical model for data warehouse design. Online Analytical Processing (OLAP) was introduced, including core functionalities such as aggregation, as well as the ability to view information from different angles. We also introduce the multidimensional modeling of OLAP systems, namely the data cube logical model. In addition, we discussed OLAP hierarchies as well as fundamental OLAP operations such as slice, dice, roll-up, drill-down and pivot. Next, we introduced the JavaCC (Java Compiler Compiler) and JJTree parsing. These are parser generator and tree building tools, respectively. We concluded the chapter with a brief discussion of the DTD, a mechanism that defines the legal building blocks of an XML document.

Chapter 3

Related Work

Much research has been done in the area of query frameworks for relational database systems (RDBMS). Our research is inspired by Cook's work who introduced the notion of native querying language in RDBMS [CR06]. Traditionally, a popular approach has been to utilize Object Relational Mapping (ORM) Frameworks. In fact, the limitations of these frameworks led to Cook's native querying language. Other approaches include those that have language specific database libraries that allow queries to be written in the embedding language itself. While these techniques as well as Cook's native language model targeted the relational database environment, we target the multidimensional database domain and propose the NOX framework.

For OLAP systems, Multidimensional Expression (MDX) language provides a specialized syntax for querying and manipulating the multidimensional data stored in data cubes. MDX has been supported by many OLAP vendors and has become the de-facto standard for OLAP systems. However, MDX is still a string-based language with many limitations. A string-based language is a language whose code is introduced as

strings when inserted within another language code. We compare and evaluate the NOX language against MDX considering it is widely used among OLAP developers.

Concerning the multidimensional algebras, there are many in research. One algebra YAM^2, created by Abello and Romero [RA07], is the product of comparing many existing algebras and finding their backbone algebra. We refer to this approach and use related concepts to develop the NOX algebra.

In this chapter, Section 3.1 explores recent relational databases querying languages that influence the industry and research work. Languages that query OLAP systems are investigated in Section 3.2. Section 3.3 illustrates the multidimensional OLAP algebraic operators and compares them to the relational operators.

3.1 Relational Databases Querying Languages

For more than 30 years, Structured Query Language (SQL) has been the de-facto standard for data access within the relational DBMS world. In conjunction with APIs such as ODBC and JDBC, it has served as the "query backbone" for small data management environments and massive enterprize settings alike. That being said, SQL despite numerous updates to the standard is now a relatively *old* language. For this reason, numerous attempts have been made to modernize database access mechanisms. Two themes in particular are noteworthy in the current context:

- The Object Relational Mapping (ORM) frameworks presented in Subsection 3.1.1
- Simplified database access extending the development languages themselves.

This is discussed in Subsection 3.1.2

3.1.1 Object Relational Mapping (ORM) Frameworks

In an attempt to minimize the *impedance mismatch* associated with tuple-to-object integration, ORM frameworks have been successfully used to define type-safe mappings — typically with XML configuration files or languages-based annotations — between the tuples of the DBMS and the native objects of the external applications. As much as possible, the ORM framework attempts to provide *transparent persistence*, the illusion that the DBMS-backed data is nothing more nor less than a simple object. With respect to the Java language, JDO (Java Data Objects) [JDO] became the early standard, with EJB [EJB] and its Java Persistence API (JPA), emerging shortly after. The standards are now quite similar, with both providing POJO (Plain Old Java Objects) style persistence for individual objects. JDO, for instance, accomplished this with a compile-time *enhancement* that modifies the byte-code to insert the appropriate mapping information. OQL is another ORM query language for databases that influenced the design of some of the newer database query languages such as JDOQL and EJB. JDOQL is an object-based query language that lets programmers write in SQL while retaining the Java object relationship. Listing 3.1 provides an example of a query written in JDOQL. The query is created by querying the *Student* class with the condition "age $<$ 20". Though JDOQL is "more" object oriented than other ORM languages, it is still partially string-based as shown in the example.

```
Query query = persistenceManager.newQuery(Student.class,
              "age < 20");
Collection students = (Collection)query.execute();
```

Listing 3.1: In JDOQL [JDO, Rus03]

An OQL query is given in Listing 3.2. The query asks for `students` who are younger than 20 years. The OQL language is modeled after SQL as shown in the example and is string-based.

```
String oql = "select * from student in Students where
             student.age < 20";
OQLQuery query = new OQLQuery(oql);
Object students = query.execute();
```

Listing 3.2: In OQL [GBB+00, ODM]

The open source community has also been active in this area, with the Hibernate framework [BK06] being the most mature project to date. Originally developed with its own proprietary API, it now also supports the JPA. Unlike JDO's compile-time enhancers, Hibernate uses run-time *Reflection* as the basis of its tuple-to-object conversions.

While the aforementioned mechanisms provide certain advantages to the developer, particularly in terms of basic OOP syntax, they are far from a perfect solution. Primary limitations include:

- While the ORM frameworks do provide transparent persistence for individual objects, this transparency largely vanishes in the face of more complex query requirements. Here, the systems employ string based query languages such as JDOQL (JDO), JPQL (JPA), or HQL (Hibernate) to execute joins, complex selections, subqueries, etc. In practice, this leaves the ORM models in something of a grey area between pure transparent persistence and glorified SQL substitution.

- Since modern Integrated Development Environments (IDEs) will not automatically refactor field names that appear in strings, refactorings can cause class models and query strings to be inconsistent.

- Developers are constantly required to switch contexts between implementation language and query language, and they have to learn the two languages.

- There is no explicit support for creating reusable query components.

To address the above problems, Cook describes in [CR05] how to express a query in the *native* language like plain Java or C# using Safe Query Objects. Note that by native language, we are referring to the application development language, rather than the database access language. The goals for these native queries are:

- 100% native: Queries should be *completely* expressed in the implementation language, rather than a mix of two distinct languages
- 100% object-oriented: Queries should provide encapsulation and inheritance functionality
- 100% refactor-able: Queries should be fully accessible to modern IDE refactoring functionality (i.e., class/method updates)
- 100% type-safe: Query specifications should be checked for type safety at compile time

In [CR06], Cook presented Safe Query Objects, a technique for representing queries as statically typed objects while still supporting remote execution by a database server. To illustrate this idea, in the Java code of Listing 3.3, Cook expresses a query written in the Java programming language itself. He uses an abstract base class for queries, the "Predicate" class, and a method named "match" that defines the query. Of course, a way to pass a Student object to the expression, as well as a way to pass the result back to the query processor are also needed. Cook does this by defining a STUDENT parameter and by returning the result of the expression as a boolean value.

```
public abstract class Predicate <ExtentType> {
  public <ExtentType> Predicate () {}
  public abstract boolean match (ExtentType candidate);
}

Predicate<Student> predicate = new Predicate <Student > () {
    public boolean match(Student student) {
```

```
        return student.getAge() < 20 &&
               student.Name.Contains("f");
 }

List <Student> students = database.query <Student> (predicate);
```

Listing 3.3: Predicate class and match method for querying the *Student* table

The underlying idea here is to allow programmers to think of the target as though it were merely an object(s) residing in memory. In the example, it is as if we have an "in-memory" list of students and we want to "query" this list to find those students under the age of 20 and whose names contain the letter "f". Because in Cook's paper [CR06] they are dealing with arbitrary relational databases, they cannot use a loop to access the objects of the database since the database in fact is not an object and it is certainly not local. Instead, the method "match" is defined that returns a boolean value representing the success of the query operation against each possible data value in a given table. The proper type checking is performed by the native language's regular compiler. The value of the boolean result indicates whether a given student in the Student table meets the criteria.

The key to safe query objects is that type-checked class definitions are translated into code to call standard database interfaces such as JDBC [HC97] or JDO [Rus03]. This new code is added to the class that contains the query to override a method (responsible for sending the new new code) in the base class. The translation could be performed on the classes during compilation, on byte-codes after compilation, or

during loading. Cook's prototype uses OpenJava [TCIK99], which follows the first approach.

While Cook's full representation of a query is used for relational databases, our work will be applied in OLAP systems. In fact, we are adapting this idea of translating the programmer's code into a new querying format that can be delivered to the server, though it must be noted that we have a different set of problems and concerns which are specific to OLAP.

3.1.2 Language Specific Database Libraries

Another approach to simplified database access extends the development languages themselves. In fact, this has been an ongoing research theme, with work stretching back more than 20 years [AB87]. We look briefly at a few of the more interesting examples. The Ruby language [Rub] provides one of the simplest interfaces by employing an *ActiveRecord* which is a library built for Ruby that dynamically examines method invocations against the database schema. Explicit field/member mappings are not even required. The Haskell language has been extended with HaskellDB [Has]. Its monad-based syntax expression is intriguing in that queries are "decomposed" into a series of distinct algebraic operations (e.g., restrict, project). Even C++ has been extended to support native database access. ARARAT [GL07] is a C++ template library whose objective is type safe, and largely transparent, generation of SQL statements. With each of these examples, we note that the expressive power of the query facilities is limited and that an "SQL backdoor" may be needed in more sophisticated

environments.

Perhaps the most notable of the language-centric approaches is Microsoft's LINQ extensions for its .NET family of languages (C# and VisualBasic) [BRK+08]. Syntactically, LINQ resembles embedded SQL in that, for more complex queries at least, the standard SELECT-FROM-WHERE format is employed (for better or for worse). While LINQ has been quite popular with developers, it has been subsumed under the new ADO.NET model [AMM07]. The overarching theme of ADO.NET is the Entity Framework (EF), a comprehensive attempt to pull back the abstraction level of development projects from the object-oriented logical level to the entity-focused conceptual level. In other words, use of EF and its Entity Data Model makes it possible, in theory, to program directly against user level concepts. Source code, possibly written with LINQ, is then parsed into an internal command tree, which can subsequently be used to generate optimized SQL. While the move towards greater abstraction is quite appealing, initial reaction has been mixed, with many developers concerned about the design and development complexity associated with the EF. Db4o (Database for objects) is another database language that allows to use the native program language to query the database [NGD+08]. It is an embeddable open source object database for Java and .NET developers. In .NET, LINQ support is fully integrated in db4o. Although db4o offers nice language integrated queries, it suffers from some drawbacks of which a notable one is the difficulty to overcome its slow performance when retrieving a lot of objects.

Listing 3.4 presents a query in db4a that is used in Java context. A list of students

who are less than 20 years old and whose grade is gradeA is returned. Note the match method that is common to db4o and Cook's safe queries.

```
Predicate<Student> predicate = new Predicate <Student > () {
    public boolean match(Student student) {
       return student.age() < 20 && student.grade() == gradeA;
 }

List<Student> students = db.query <Student> (predicate);
```

Listing 3.4: In Java using db4o [DB4]

A query written in LINQ is given in Listing 3.5. Again, it returns students who are less than 20 years old and whose grade is gradeA. Note the usage of FROM-WHERE-SELECT which is similar to SQL syntax.

```
var result = from Student s in container
             where s.Age < 20 && student.Grade == gradeA
             select s;
```

Listing 3.5: In .NET using LINQ [LIN]

In addition to the disadvantages mentioned earlier for individual database query languages, the main hindrance of db4o and LINQ, as is the case with many of the other tools, is the lack of interoperability that is taken for granted in the SQL world, such as industry standard connectivity, reporting tools, backup and recovery standards and OLAP functionality!

3.2 Multidimensional Databases Querying Languages

In terms of OLAP, there was also a flurry of interest in the design of supporting algebras [AGS97, GL97]. The primary focus of this work was to support an algebraic application programming interface (API) that would ultimately lead to transparent, intuitive support for the underlying data cube. In a more general sense, these algebras identified core elements of the OLAP conceptual data model. Recently, the various algebras have been directly compared so as to extract the operations common to each model [RA07].

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 in an attempt to more intuitively support standard OLAP query patterns [Mel02]. It must be noted, however, that support for these operations in DBMS platforms is inconsistent at best, leading most OLAP/BI vendors to provide their own proprietary implementations [DKK05]. In addition to SQL, many commercial applications support Microsoft's MDX query language [WZP05]. MDX provides a specialized syntax for querying and manipulating the multidimensional data stored in data cubes. MDX has been embraced by wide majority of OLAP vendors and has become the de-facto standard for OLAP systems [SHW+06]. Still, MDX remains an embedded string based language with an irregular structure and is somewhat representative of the language philosophy of the 1980s and 1990s.

To give the reader a better sense of the MDX language, we will now present a series of simple MDX examples. The cube we are working with contains the following feature attributes:

- Customers
- Time
- Product
- DMA

The Measure attributes consist of:

- Store Cost
- Profit
- RunningTotalSubs

The MDX query depicted in Listing 3.6 is a very simple query for finding store costs (a measure attribute in the data cube) associated with all customers in 1997.

The MDX query given in Listing 3.7 is similar to the previous MDX query, but this time the query is doing a drill down on the customers in the USA region.
In the query given in Listing 3.8, we are doing a crossjoin on Customer and Gender to get all combinations by year, and we are using a different measure value Profit.

We note that while the queries included here are quite simple, and consequently quite readable, more sophisticated MDX queries can be virtually incomprehensible to

```
SELECT
      { [Time].[1997] } ON COLUMNS ,
      { [Customers].[All Customers] } ON ROWS
FROM  [Sales]
WHERE ( [Measures].[Store Cost] )
```

Listing 3.6: MDX query 1

```
SELECT
      { [Time].[1997] } ON COLUMNS ,
      { [Customers].[All Customers].[USA].CHILDREN }
ON ROWS
FROM  [Sales]
WHERE ( [Measures].[Store Cost] )
```

Listing 3.7: MDX query 2

```
SELECT
      { [Time].[1997] } ON COLUMNS,
      {
        { [Customers].[All Customers].[USA].CHILDREN } *
        { [Gender].[All Gender].[F], [Gender].[All Gender].[M] }
      }  ON ROWS
FROM  [Sales]
WHERE ( [Measures].[Profit] )
```

Listing 3.8: MDX query 3

anyone other than the developers themselves. An example of a rather incomprehensible MDX query is depicted in Listing 3.9. In the query, measure VariantPercentage is created and defined using a formula in terms of RunningTotalSubs and some hierarchical attributes. Tuples are used here which complicates the query even more. They are used to indicate that RunningTotalSubs of a hierarchy path in the time hierarchy which refers to some date in 2004 is subtracted from that of 2005 date in the time hierarchy then divided by the total RunningTotalSubs of "all" "time" hierarchy. The "SELECT" on COLUMNS is similar to what we saw before, where the RunningTotalSubs and the VariantPercentage are displayed. However, "SELECT" on ROWS is more complex in this example. There is a CROSSJOIN of TopCount ([DMA] .children , 5000 ,([RunningTotalSubs])) and [Time].[2004].& [1].[1].[1] ,[Time].[2005].&[1].[1].[1] ,[Time]). This means that the top 5000 RunningTotalSubs of the children of [DMA] are crossjoined with the hierarchy path referring to 2004 date of the time hierarchy, the hierarchy path referring to the 2005 date of the time hierarchy and "All Time". These are all what will be displayed on ROWS and COLUMNS. They are selected FROM "Customers" cube where the slicing operation performs selection of "product ID" to be equal to 14.

```
WITH

MEMBER [ Measures ] . [ VarientPercentage ] AS

'(([Time].[2004].\&[1].[1].[1],
```

```
[RunningTotalSubs]) -
([Time].[2005].\&[1].[1].[1],
[RunningTotalSubs])) /
([Time].[All Time], [RunningTotalSubs] )'

SELECT {
[RunningTotalSubs] ,
[Measures].[VarientPercentage] }
ON COLUMNS ,

CrossJoin(
TopCount (
 {[DMA].CHILDREN}, 5000 ,
( [RunningTotalSubs] ) ) ,

{[Time].[2004].\& [1].[1].[1] ,
[Time].[2005].\&[1].[1].[1] ,
[Time] } )
ON ROWS

FROM [Customers]
WHERE ( [Product].[Product ID].\&[14] )
```

Listing 3.9: A more sophisticated MDX query

Finally, we note that no discussion of OLAP query languages and models would be complete without a brief reference to the ill-fated JOLAP standard [JOL03]. Delivered in 2003, the JOLAP JSR-69 was an industry-backed attempt to define an enterprise-ready, Java-oriented meta data and query framework for OLAP applications. Drawing upon the Common Warehouse Metamodel [CWM03], JOLAP introduced a purely compositional query API that layered itself on top of elements of the CWM's logical

metamodel. JOLAP object model provides a core layer of services and interfaces that are available to all clients. While intuitively appealing, the JOLAP specification proved to be extraordinarily complex for both vendors and query writers. To date, no client or server side application has ever been developed around JOLAP. It currently serves as both an inspiration for OLAP centered projects and a cautionary tale.

3.3 OLAP Algebras in Research

A great deal of effort has been devoted to multidimensional modeling in OLAP settings with several models having been introduced in the literature [ASS01, VS99]. A multidimensional algebra is as crucial for satisfactory data warehouse querying as the relational algebra (select, project, join, etc.) is for satisfactory relational database querying. Romero and Abello, in [RA07], compare existing multidimensional algebras in the literature so that their common backbone is discovered.

In terms of the models themselves, Romero and Abello highlight the following:

- [LW96] introduces a multidimensional algebra of five operators, namely "Add Dimension", "Transfer", "Cube Aggregation", "Join", "Union", representing mappings between either Cubes or relations and Cubes. The authors illustrate, in their paper, how the multidimensional algebra gets translated to SQL. In fact, this algebra was one of the first multidimensional algebras introduced in the literature and its aim was to construct Cubes for local operational databases.

- [AGS97] presents an algebra of six operators which are "Push", "Pull", "Destroy

Dimension", "Restriction", "Join", "Merge". These operators are invented to be translated to SQL. They are minimal. No operator can be expressed in terms of other operators and no operator can be excluded without affecting the algebra.

- [GL97] presents an algebra of seven operators that are based on the relational algebra operations. The seven operators are "Selection", "Projection", "Cartesian product", "Union/Difference/Intersection", "Fold/Unfold", "Classification", "Summarization". They also define a calculus that is equivalent to the proposed algebra.

- [TD97] et al and [TD01] et al present an algebra with eight operators based on [AGS97]. These operators are "Restriction", "Metric Projection", "Aggregation", "Cartesian Product", "Join", "Union/Difference", "Extract", "Force". The authors presume the algebra to express complex OLAP queries in a concise way.

In addition to the above, additional algebras were presented by Romero and Abello, including but not limited to [CT98], [HS98], [VS99], [GMR98], [FS00], [FBV00] and [FK04]. In addition to reviewing the existing work in the area, Romero and Abello propose a multidimensional *reference* algebra that we will present in the next few paragraphs. In this thesis, we essentially adapt Romero and Abello's reference algebra as the underlying mechanism for OLAP query transformation.

In their framework, Romero and Abello describe the following concepts that are common to virtually all OLAP models.

- A **Dimension**: A dimension contains a hierarchy of `Levels` where a level contains `Descriptors`.
- A **Fact**: A fact table contains `Cells`. These cells contain `Measures`.
- A **Base**: A base is a minimal set of levels that identify a cell that may be a primary key in the database.
- A **Cube**: A cube is a set of cells placed in the multidimensional space. It should be positioned with regard to the Base.
- A **Star**: A star is one `Fact` and several `Dimensions`.

The reference algebra of Romero and Abello is presented as a framework called YAM^2 [ASS05]. The YAM^2 algebra was introduced in detail in [ASS03], where it was proven to be complete, meaning that any other multidimensional operation can be expressed in terms of it. These algebraic operations are as follows:

- **Selection**: This operation selects the subset of points of interest out of the whole n-dimensional space by means of a logic clause C over a Descriptor.
- **Projection**: This operation selects a number of Measures from the Cube.
- **Roll-up and Drill-down**: The "Roll-up" operation groups cells in the Cube based on an aggregation hierarchy. It modifies the granularity of data. The "Drill-down" operation is the inverse of Roll-up. It can only be performed if a Roll-up has been previously applied and the correspondence between cells is preserved.

- **ChangeBase**: This operation reallocates exactly the same instances of a Cube into a new n-dimensional space with exactly the same number of points, by means of a one-to-one relationship.

- **Drill-across**: This operation changes the subject of analysis of the Cube by means of one-to-one relationship.

- **Set Operations**: These operations operate on two Cubes (like in set theory) if both are defined over the same n-dimensional space. **Union**, **Difference** and **Intersection** are the usual set operations performed.

Figure 12 shows the table given by Romero and Abello [RA07] that depicts the mapping between the two sets of algebraic operators:

- the set of relational operators as the columns names

- the set of multidimensional operators as the rows names

In the figure, the intersection of the columns and rows means that the corresponding two operators are equivalent when applied on the subscript names of the tick sign. `SELECTION` as the multidimensional operator is equivalent to `SELECTION` as the relational operator when applied over Descriptors (features) fields. `PROJECTION` as the multidimensional operator is equivalent to `PROJECTION` as the relational operator when applied over Measures fields. The tick sign without restriction means both operators are equivalent. In the table, the set operators including Union and Difference are equivalent as both relational operators and multidimensional operators.

Reference Operator		Selection	Projection	Join	Union / Diff	Group by	Aggregation
Selection		$\surd_{Descs}$					
Projection			$\surd_{Measures}$				
Roll-up						$\surd_{Descs\_ID}$ +	$\surd_{Measusres}$ +
Drill-across			$\surd_{Descs\_ID}$ +	$\surd_{Descs\_ID}$ +			
changeBase	Add Dim.			$\surd_{Descs\_ID}$			
	Remove Dim.		$\surd_{Descs\_ID}$				
	Alt. Base		$\surd_{Descs\_ID}$ +	$\surd_{Descs\_ID}$ +			
Union/Difference					$\surd$		

Figure 12: Reference operator matching between multidimensional and relational algebra operations

The + sign means that one multidimensional operator is equivalent to more than one relational operator. In the table, we see that Roll-up, Drill Across and ChangeBase multidimensional operators involve equivalence to more than one relational operators. Roll-up operator is equivalent to the two relational operators Group-by and Aggregation. Drill-across operator is equivalent to the two relational operators Projection and Join. ChangeBase specifically Alternate Base operator is equivalent to the two relational operators Projection and Join. In our research, we focus mostly on `SELECTION` and `PROJECTION`. We also cover, from a pratical point of view, the set operations and the manipulation of navigational hierarchies (Roll-ups and Drill-downs).

3.4 Conclusion

We presented in this chapter the important related work to this thesis. In RDBMS, ORM languages play a critical role in querying the database, as they define type-safe mappings between the tuples of the DBMS and the native objects of the external applications. However, as we demonstrated by example, these languages are partially or totally string-based. To tackle this problem, we introduced Cook's work that describes how to express a query in the native language itself using Safe Query Objects. Cook's work in querying relational databases instigated our work in querying OLAP systems. To complete the review of work that is done in RDBMS, we introduced the language specific database libraries that extend the development languages themselves such as db4o and LINQ.

In the OLAP world, the MDX language is the de-facto language to query OLAP. We illustrated its usage through examples. MDX, being a string-based and often obscure language, has motivated us to build a framework where querying an OLAP system is done in the native language itself.

Finally, we tried to emphasize that a very important part of any querying framework is the algebra that is used. We presented in this chapter a number of multidimensional algebraic operations that were introduced by contemporary researchers. Romero and Abello derived a common algebra they called YAM^2 which is the algebra we adapted in our work.

Chapter 4

Native language OLAP query eXecution (NOX)

The native language OLAP query eXecution system, abbreviated as NOX, has been constructed from the ground up so as to emphasize the *transparency* in the term "transparent persistence", which is the illusion that the server's data is nothing more nor less than a simple object. Doing so, of course, requires considerable infrastructure.

Our current research work focuses on building the client side libraries and parsing infrastructure that allows programmers to write OLAP queries in the native programming language used. OLAP queries, written by the programmer, then become accessible to IDE and compiler features like compile-time type checking, auto-completion, and refactoring. Moreover, we avoid the requirement for the programmer to learn a second programming language, for example, SQL or MDX.

In this chapter, we begin in Section 4.1 by discussing the Sidera system server architecture to which NOX sends its OLAP queries. We then discuss the design,

implementation, and use of the NOX framework where Section 4.2 introduces the primary NOX components, and Section 4.3 discusses the underlying conceptual model. Section 4.4 illustrates the core operations of the OLAP algebra in the NOX model. Section 4.5 expands the model to include the NOX grammar and its DTD representation. Finally, we present in Section 4.6 the full details of the client architecture, specifically the NOX pre-processor.

4.1 The Sidera System Architecture

In this section, we describe the Sidera system architecture. Eavis et al in their paper titled "Sidera: a cluster-based server for Online Analytical Processing" presented a comprehensive architectural model for a fully parallelized OLAP server [EDD+07]. The model consists of a network-accessible frontend server and a series of protected backend servers. Each backend server handles a portion of the user request. Other architectures have utilized existing DBMS servers to provide backend storage and query resolution services with minimal implementation efforts, but have limited support for advanced OLAP functionality such as cubing and hierarchical querying. Another limitation is that they return local results to the primary server where the data will then be merged and aggregated. In relatively large production systems, the bottleneck on the frontend becomes significant. Sidera eliminates this bottleneck by operating within a fully coordinated architecture that allows each node to participate in global sorting, merging and aggregation operations. This brings the full computational capacity of the whole cluster for every OLAP query. Note that prototypes of

the backend and the frontend implementations have been developed and published by Eavis et al. in [EDD+07] and [ETT10] simultaneously. A description of the whole framework was published by Taleb et al. in [TET11].

Figure 13 illustrates the fundamental design of Sidera. Here, the frontend node serves as an access point for user queries. Query reception and session management is performed at this point but the frontend does not participate in query resolution, other than to collect the final result from the backend instances and return it to the user. In turn, the backend nodes are fully responsible for storage, indexing, query planning, I/O, buffering, and meta data management. In addition, each node houses a Parallel Service Interface (PSI) component that allows it to hook into the the global PSI layer.

Figure 14 is an illustration of the Sidera frontend, a multi-threaded head node that handles logins, authentication, and transfer of queries to the backend nodes. The head node represents the server's public interface. Its core function is to receive user requests and to pass them along to the backend nodes for resolution. It does not participate in query resolution directly, and thus does not represent a performance bottleneck for the system. The numbered sequence in the figures indicates the processing cycle for a typical query. Figure 15 depicts the processing loop on the backend server instances. While the frontend provides the public interface, it is of course the backend network that performs virtually all of the query resolution.

We note that Sidera has been used as the target platform as it allows us to explore both query processing on the client and query optimization on the server (a separate

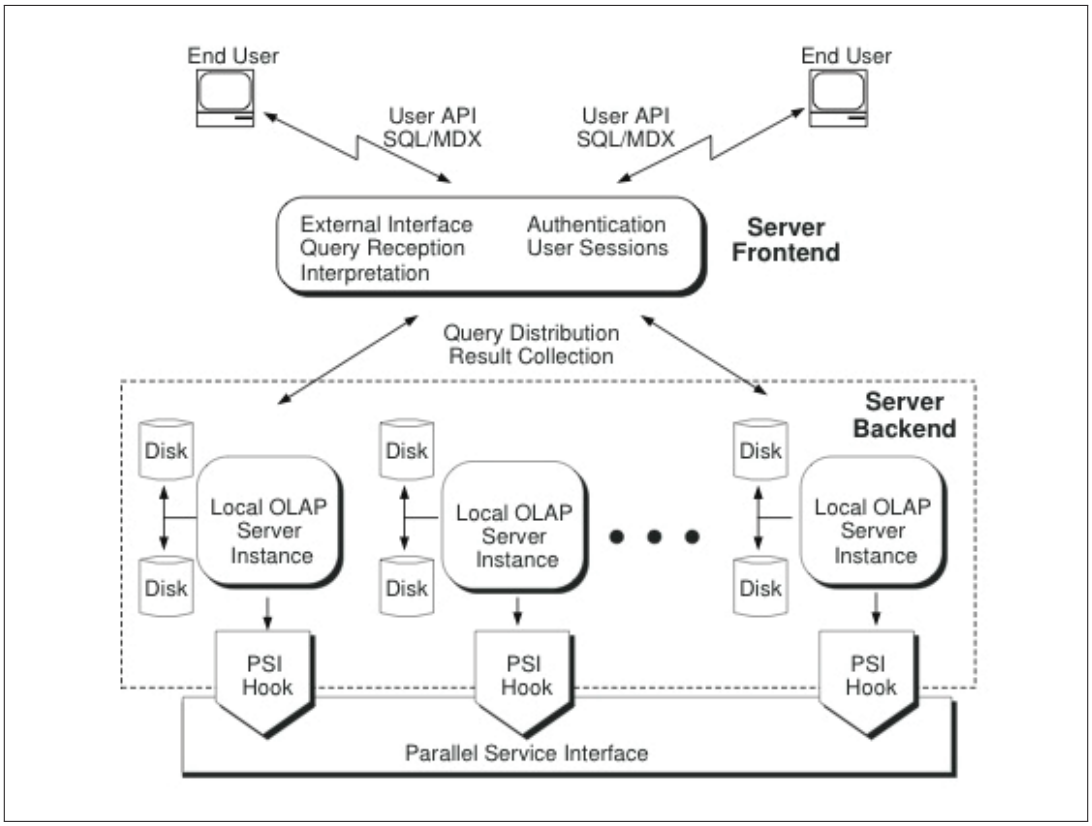


Figure 13: The core architecture of the parallel Sidera OLAP server [EDD+07]

research project). The NOX framework was implemented and tested to send queries to the Sidera framework and receive results back. Having complete freedom with the code base is a distinct advantage for this kind of research. That being said, it is important to note that the principles discussed in this thesis can in theory be applied to existing DBMS platforms, assuming the implementation of suitable mappings to the given DBMS backend.

4.2 The NOX Framework

We now turn to the problem of providing native language functionality in the OLAP setting, the key contribution of this thesis. The NOX framework, being a query

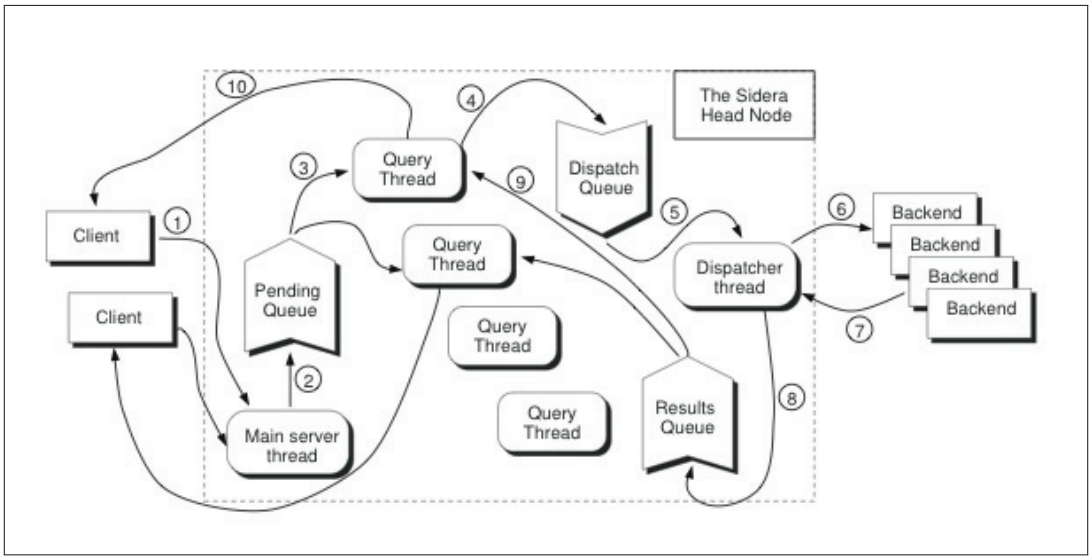


Figure 14: The Sidera frontend [EDD+07]

language framework, we begin with a brief overview of its primary physical and logical elements. They include the following:

- **OLAP conceptual model**. As with the Entity Framework, NOX allows developers to write code directly at the conceptual level. No knowledge of the physical or even logical level is required.

- **OLAP algebra**. Given the complexity of directly utilizing the relational algebra, in the OLAP context, we define fundamental query operations against a cube-specific OLAP algebra.

- **OLAP grammar**. Closely associated with the algebra is a DTD-encoded OLAP grammar that provides a concrete foundation for client language queries.

- **Client side libraries**. NOX provides a small suite of OOP classes corresponding to the objects of the conceptual model.

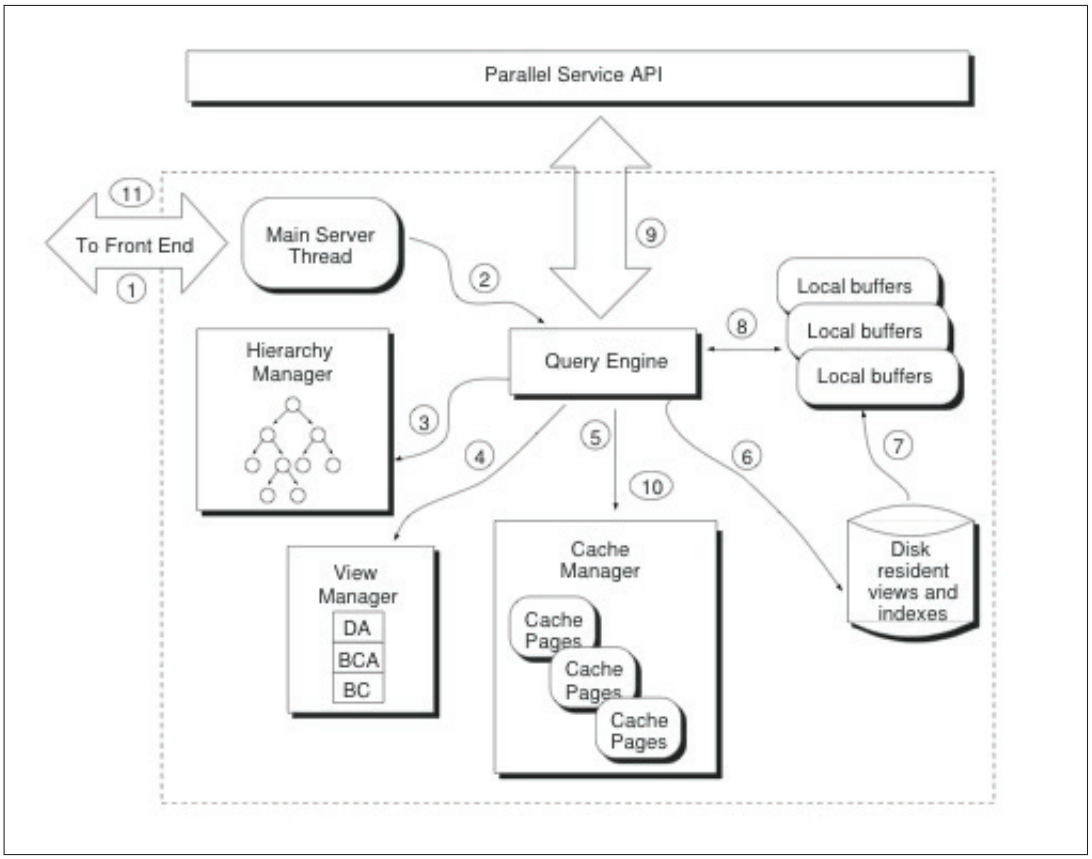


Figure 15: The Sidera backend node [EDD+07]

- **Programming API**. Collectively, the exposed methods of the libraries form a clean programming API that can be used to instantiate OLAP queries.

- **Augmented compiler**. At its heart, NOX is a query re-writer. During a pre-processing phase, the framework's compilation tools (JavaCC/JJTree) effectively re-write source code to provide transparent model-to-DBMS query translation.

- **Cube result set**. OLAP queries essentially extract a subcube from the original space. The NOX framework exposes the result in a logical read-only multi-dimensional array.

Figure 16 provides a concise illustration of the NOX processing stack. In short, the developer's view of the OLAP environment consists of the elements of the top three levels. From the developer's perspective, all OLAP data is housed in a series of one or more cube objects housed in local memory. The fact that these repositories are not only remote, but possibly Gigabytes or even Terabytes in size, might be irrelevant to the developer since he is querying a cube as if it is an object residing in memory. However, the time needed to get the results back depends on the size of the subcube result.

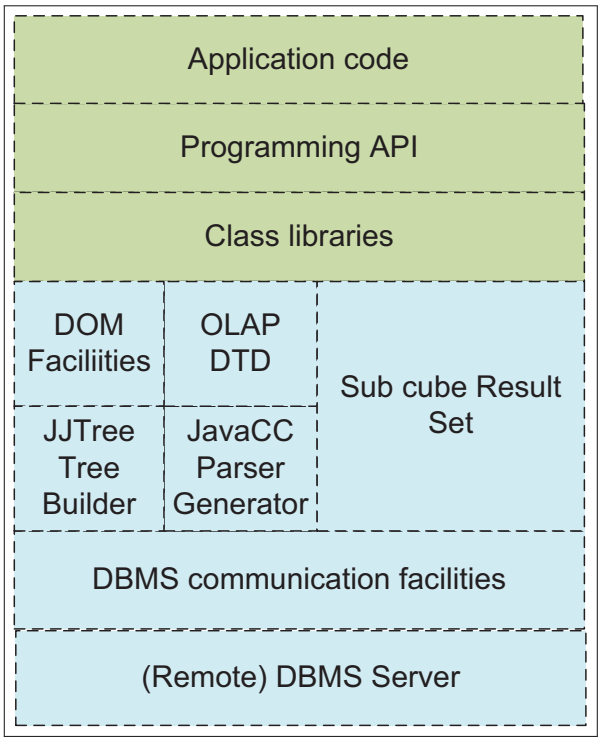


Figure 16: NOX processing stack

4.3 Conceptual Model

In the OLAP context, 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 17 provides an illustration of a very simple three dimensional cube. We can see, for example, that 20 units of Product FH1 were sold in the Berkeley location during the month of January (assuming a Count measure). Beyond the basic cube, however, the conceptual

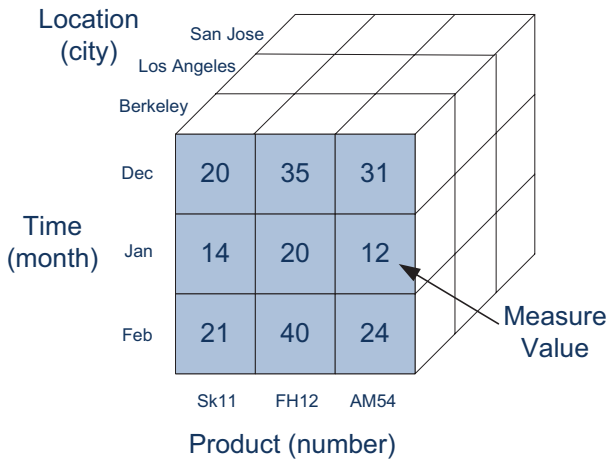


Figure 17: NOX conceptual query model

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 [MZ06] (e.g., symmetric, ragged, non-strict). For our purposes, however, it is enough at this point to supplement the NOX conceptual model with the notion of an arbitrary graph-based hierarchy that may be used to decorate one or more cube dimensions. Figure 18 illustrates a simple geographic hierarchy that an organization might use to identify intuitive customer groupings. The path in yellow is an example of an OLAP path where each value on the OLAP path comes from a different level in the hierarchy.

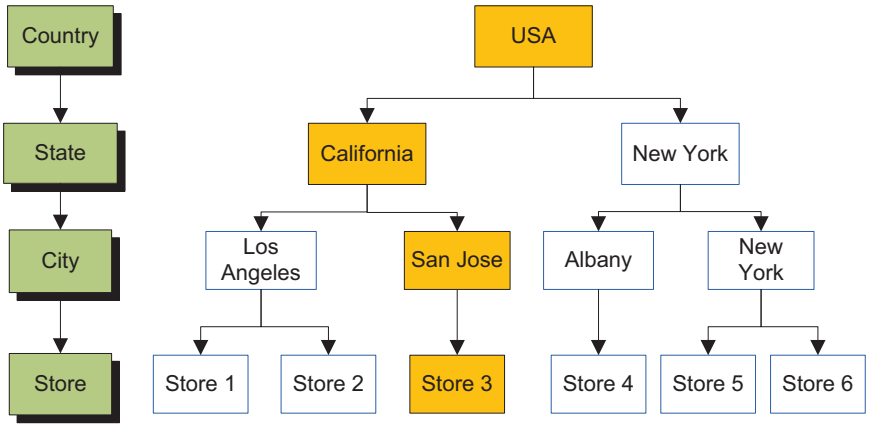


Figure 18: A simple symmetric hierarchy

4.4 The NOX Algebra

Given the clean, conceptual model described above, it is possible to consider the application of an OLAP algebra that directly exploits the model's structure. A number of researchers have identified the core operations of such an algebra as detailed in Section 3.3. We will shortly see how the exploitation of a formal algebra ultimately

allows developers to program directly against the conceptual model, rather than to a far more complex physical or even logical model.

As indicated, a core set of operations for NOX common to virtually all proposed OLAP algebras has been identified. Below, we list and briefly describe these operations.

- `SELECTION` ($\sigma_p$cube): provides the identification of one or more cells from within the full d-dimensional search space. This is one of the two core OLAP operations and is commonly referred to as "slicing" and "dicing". A logic predicate `p` defines cells of interest within the d-dimensional space. The logic predicate has the syntactical form where mathematical expressions can be compared to each other and different conditional expressions can be combined with logical operators such as AND and OR. The selection operation is given in Figure 19.

- `PROJECTION` ($\pi_{measure_1,\ldots,measure_n} cube$): provides the identification of presentation attributes, including *both* measure attributes and feature attributes. This is the second core OLAP operation and it is mainly concerned with screening results in an output mechanism such as diagrams, objects or simply text. In other words, it does selection of a subset of display attributes (measures or features). The projection operation is depicted in Figure 20.

- `DRILL-ACROSS` ($\text{cube}_1 \infty \text{cube}_2$): performs the integration of two independent cubes, where each cube possesses common dimensional axes. In effect, this is a cube "join" (possibly a self join) that changes or extends the subject of analysis,

by showing measures regarding a new fact. The n-dimensional space remains exactly the same, only the data placed in it change so that new measures can be analyzed. For example, if the cube contains data about profits, this operation can be used to analyze data regarding expenses using the same dimensions. Figure 21 illustrates the drill-across operation.

- `UNION` ($\text{cube}_1 \cup \text{cube}_2$): performs the union of two cubes over the same n-dimensional space sharing common dimensional axes. The union operation is presented in Figure 22.

- `INTERSECTION` ($\text{cube}_1 \cap \text{cube}_2$): performs the intersection of two cubes over the same n-dimensional space sharing common dimensional axes.

- `DIFFERENCE` (cube_1 - cube_2): performs the difference of two cubes over the same n-dimensional space sharing common dimensional axes.

- `CHANGE LEVEL` ($\gamma^{level_i \to level_j}_{f(measure_1),\ldots,f(measure_n)}$) does the modification of the granularity of aggregation for the current result set. This process is typically referred to as "drill down" and "roll up". The roll-up operation groups cells in a Cube based on an aggregation hierarchy while the drill-down goes down through an aggregation hierarchy, and showing more detailed data. The gamma representation of the change level operation means that as the level of the data is changing, the measure values are changing according to functions that aggregates or decomposes data values (along the levels of the hierarchy). The change level operation is given in Figure 23.

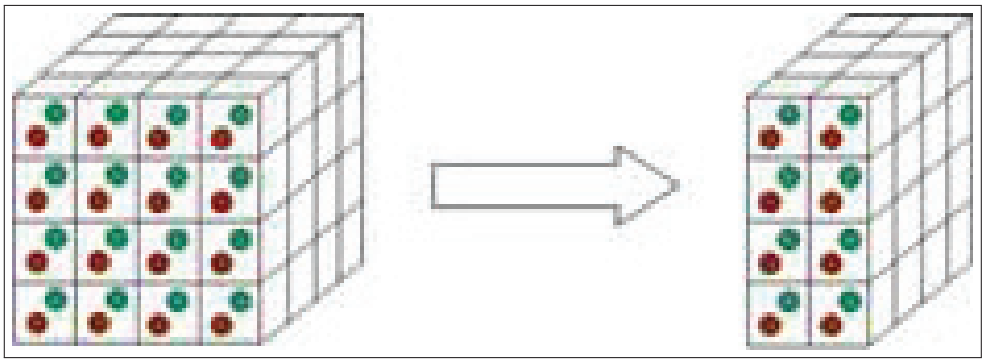


Figure 19: Selection operation [AR]

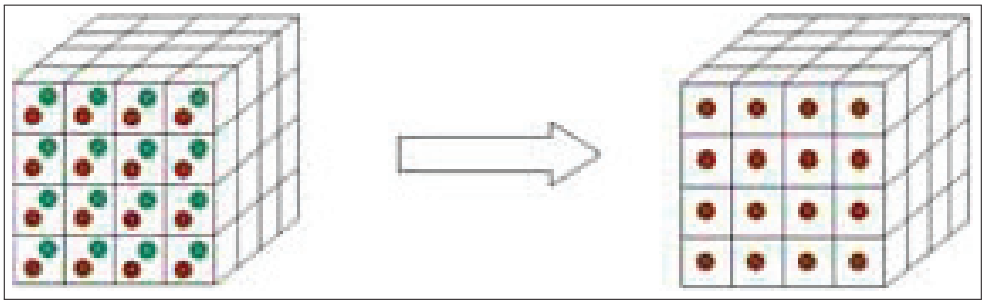


Figure 20: Projection operation [AR]

- `CHANGE BASE` ($\chi_{base_1 \rightarrow base_2}$): allows two different kinds of changes in the n-dimensional space: it performs the addition or deletion of one or more dimensions from the current result set or just rearranges the multidimensional space by reordering the dimensions (this is also known as Pivoting). When addition or deletion of dimensions is done, aggregated cell values must be re-calculated accordinglY. Figure 24 depicts the change base operation.

- `PIVOT` (ϕ_{base}): does the rotation of the cube axes to provide an alternate *perspective* of the cube. No recalculation of cell values is required. Figure 10 given in Chapter 2 illustrates the pivot operation.

Several explanatory notes are in order at this stage. First, the `SELECTION` is

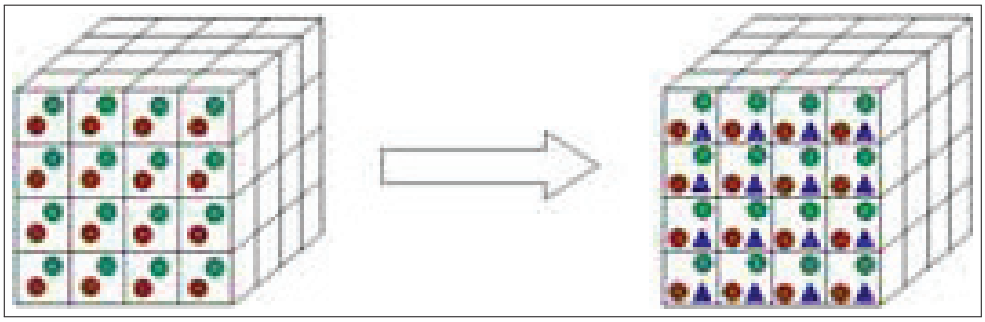


Figure 21: Drill-across operation [AR]

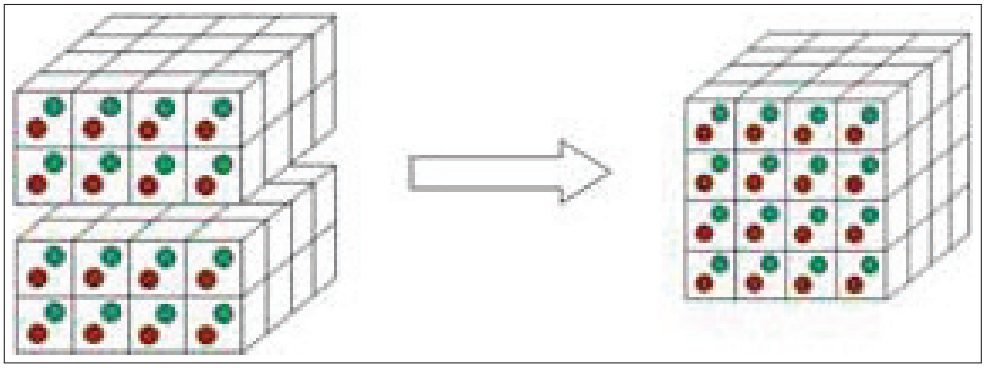


Figure 22: Set operations (Union) operation [AR]

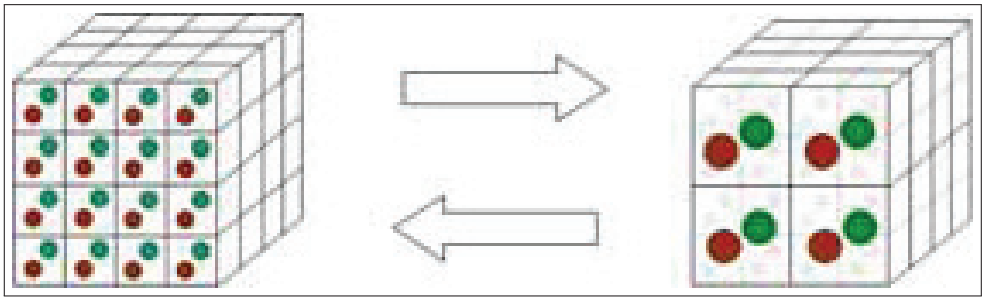


Figure 23: Change Level operation [AR]

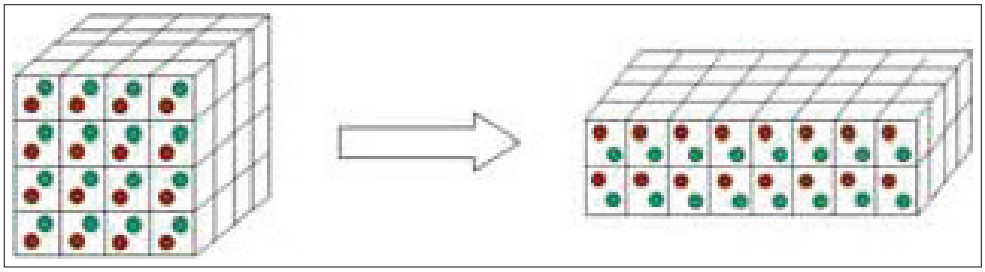


Figure 24: Change Base operation [AR]

the driving operation behind most analytical queries. In fact, if suitable defaults for identifying presentation attributes, such as a certain key or a set of keys, are available for the `PROJECTION`, many queries can be expressed with nothing more than a selection. Second, the final two operations `CHANGE BASE` and `PIVOT`, are distinct from the first seven in that each is only relevant as a query against an existing result set. This is because `CHANGE BASE` and `PIVOT` provide alternative presentations of data such as rearranging or rotating the result cube. Third, it is important to recognize that while logical data warehouse models typically require explicit joins between fact (measure) and dimension tables, there is no such requirement at the conceptual level. Data is viewed at the conceptual level as objects residing in memory. The result is a dramatic reduction in complexity for the developer. Depending upon the architecture of the supporting analytics server, of course, join operations may still be performed at some point. Finally, and perhaps most importantly, the OLAP algebra is implicitly *read only*, in that database updates (change in the data or schema) are performed via distinct ETL processes. Remember from Chapter 2 that information is loaded into the data warehouse using a process known as ETL (Extract, Transform, and Load). It is well known that updates significantly complicate the logic of ORM frameworks.

As discussed in Section 3.3, Abello and Romero provide the YAM^2 reference framework for OLAP algebras. Our algebra is similar, with the addition of the `PIVOT` operation which is a special case of the `CHANGE BASE` operation. The special case is when `CHANGE BASE` changes its visual orientation by rotating the cube along its axis with a one-to-one correspondence between its dimensions. Finally, we note that YAM^2 has

in fact been proven to possess the following properties [RA05] and [ASS03]:

- Closed, meaning when the algebra operations are applied to a cube-query, its result is another cube-query.
- Complete, meaning that any valid cube-query can be the result of a combination of a finite set of the algebra operations applied to the right cell, and
- Minimal, meaning that none of the algebra operations can be dropped without affecting the algebra and none of the operations can be expressed in terms of the others.

Therefore, the NOX algebra is also closed, and complete. However, it is not minimal as the `PIVOT` operation can be expressed in terms of `CHANGE BASE`.

4.5 The NOX Grammar

NOX encapsulates the operations of the algebra in a formal grammar encoded by a Document Type Definition (DTD). Section 2.7 gave an overview of what a DTD schema is. (We note that the XML Schema could be used as well). The NOX grammar DTD is relatively complex as it effectively represents the foundation for an expressive, XML-based analytics language.

Listing 4.1 depicts the current DTD-encoded grammar of the NOX query language. While NOX is very much a research prototype, the grammar is nonetheless quite sophisticated as it is required to support most of the features of a DBMS access

language. Below, we look more closely at a few of the most significant grammar elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT QUERY (DATA_QUERY | META_QUERY)>

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

<!-- Selection -->
<!ELEMENT SELECTION (DIMENSION_MEASURE_LIST)>
<!ELEMENT DIMENSION_MEASURE_LIST
   ((DIMENSION,(LOGICAL_OP,(DIMENSION || MEASURE))*) ||
    (MEASURE,((LOGICAL_OP,DIMENSION),(LOGICAL_OP,(DIMENSION ||
        MEASURE))*)) ||
    (MEASURE,((LOGICAL_OP,(DIMENSION ||
        MEASURE))*,(LOGICAL_OP,DIMENSION))))>
<!ELEMENT DIMENSION (DIMENSION_NAME, EXPRESSION)>
<!ELEMENT DIMENSION_NAME (#PCDATA)>


<!-- Projection -->
<!ELEMENT PROJECTION (MEASURE_DIMENSION_LIST)>
```

```
<!ELEMENT MEASURE_DIMENSION_LIST ((MEASURE,(LOGICAL_OP,(DIMENSION
    || MEASURE))*) ||
    (DIMENSION,((LOGICAL_OP,MEASURE),(LOGICAL_OP,(DIMENSION ||
        MEASURE))*)) ||
    (DIMENSION,((LOGICAL_OP,(DIMENSION ||
        MEASURE))*,(LOGICAL_OP,MEASURE))))>
<!ELEMENT MEASURE(MEASURE_NAME,(COND_OP,SIMPLE_EXP)?)>
<!ELEMENT MEASURE_NAME (#PCDATA)>

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

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

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

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

<!ELEMENT COND_OP (
    RELATIONAL_OP |
    EQUALITY_OP |
    OLAP_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 ARITHMETIC_OP (#PCDATA)>
<!-- ADD | SUBTRACT | MULTIPLY | DIVIDE) -->

<!-- Generic Functions -->
<!ELEMENT FUNCTION_LIST (FUNCTION+)>
<!ELEMENT FUNCTION (PARENT, FUNCTION_NAME, ARGUMENT_LIST?)>
<!ELEMENT PARENT (#PCDATA)>
<!ELEMENT FUNCTION_NAME (#PCDATA)>
<!ELEMENT ARGUMENT_LIST (ARGUMENT+)>
<!ELEMENT ARGUMENT (#PCDATA)>

<!-- Hierarchies -->
<!ELEMENT HIERARCHY_LIST (HIERARCHY+)>
<!ELEMENT HIERARCHY (HIERARCHY_NAME, HIERARCHY_OP,
    OLAP_PATH_LIST)>
<!ELEMENT HIERARCHY_NAME (#PCDATA)>
<!ELEMENT HIERARCHY_OP (#PCDATA)>
<!-- IN_RANGE | IN_LIST)> -->

<!-- Hierarchies Paths -->
<!ELEMENT OLAP_PATH_LIST (OLAP_PATH+)>
<!ELEMENT OLAP_PATH (VALUE+)>
<!ELEMENT VALUE (#PCDATA) >

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

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

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

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

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

<!-- Drill across -->
<!ELEMENT DRILL_ACROSS (DATA_QUERY)>
<!-- <!ATTLIST DRILL_ACROSS
    output (BOTH | REPLACE) #REQUIRED> -->

<!-- Meta data queries: this will be extended later-->
<!ELEMENT META_QUERY (CUBE_NAME)>
<!ATTLIST META_QUERY
   scale (FULL | PARTIAL) #REQUIRED>
```

Listing 4.1: "ClientQuery.dtd" used to validate NOX XML files

In Listing 4.1, #PCDATA stands for Parsed Character Data and specifies character data. #REQUIRED stands for values that must be given, meaning they may not be empty strings.

Each query is associated with a single cube (though references to other cubes are possible), as well as an optional Operation List and an optional Function List.

The Operation_List contains the algebraic elements of the query, and each may occur exactly zero or one time in a single query. One important operation is the selection which is defined as a listing of one or more dimensions, each associated with an expression, and possibly one or more measures. In effect, the expression represents a query restriction on the associated dimension or measure (this will become clearer in Chapter 5). 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. An example of a Selection XML string is given in Listing 4.2 where an expression "age > 40" is defined on the "Customer" dimension.

```
<SELECTION>
<DIMENSION_MEASURE_LIST>
<DIMENSION>
<DIMENSION_NAME>
Customer
</DIMENSION_NAME><EXPRESSION>
<RELATIONAL_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>
</RELATIONAL_EXP>
</EXPRESSION>
</DIMENSION>
</DIMENSION_MEASURE_LIST>
</SELECTION>
```

Listing 4.2: Example of a Selection XML string

There are several elements such as LOGICAL_OP, RELATIONAL_OP and EQUALITY_OP that are defined as #PCDATA, so they are free to be any sequence of characters. However, their values should be relevant to the meaning that they hold. For example, LOGICAL_OP should be either AND or OR, RELATIONA_OP should be either GT, GTE, LT or LTE, and EQUALITY_OP should be EQUALS or NOT_EQUAL.

Listing 4.1 also shows the FUNCTION_LIST and HIERARCHY_LIST elements that can be values of EXP_VALUE. When an expression value is a FUNCTION_LIST, it is associated with a PARENT dimension, a FUNCTION_NAME such as in_range and an ARGUMENT_LIST which consists of one or more arguments such as number values. When an expression value is a HIERARCHY_LIST, it is made of one or more hierarchies. A HIERARCHY consists of a HIERARCHY_NAME, a HIERARCHY_OP and an OLAP_PATH_LIST. A HIERARCHY_OP can be in_range and

in_list. An OLAP_PATH_LIST is made of one or more OLAP_PATH where each OLAP_PATH defines a path in a hierarchy. An OLAP_PATH consists of one or more values, where each value come from a different level of the hierarchy. Listing 4.1 also illustrates the simplicity of the *set operation* specifications. Three kinds of set operations are given in Listing 4.1: intesection, union and difference. Each operation acts on some data query. An example of a set operation `INTERSECTION` is given in Listing 4.3. In this example, intersection is done on two selection criteria, namely "Customer.getAge $>$ 40" and "Customer.getAge $<$ 60". From programming point of view, consider for example, a string equality check in a language such as Java, where we would write `myString.equals("Joe")`, rather than something like `myString == "joe"`. This same approach allows us to represent set operations simply as a nested data query, defined relative to the current query. The way this will be implemented in NOX is given in the next chapter.

```
<QUERY>
<DATA_QUERY>
<OPERATION_LIST>
<OPERATION>
<INTERSECTION>
<DATA_QUERY>
<OPERATION_LIST>
<OPERATION>
<SELECTION>
<DIMENSION_MEASURE_LIST>
<DIMENSION>
<DIMENSION_NAME>
Customer
```

```
</DIMENSION_NAME><EXPRESSION>
<RELATIONAL_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>
</RELATIONAL_EXP>
</EXPRESSION>
</DIMENSION>
</DIMENSION_MEASURE_LIST>
</SELECTION>
</OPERATION>
<OPERATION>
<SELECTION>
<DIMENSION_MEASURE_LIST>
<DIMENSION>
<DIMENSION_NAME>
Customer
</DIMENSION_NAME><EXPRESSION>
<RELATIONAL_EXP>
<SIMPLE_EXP>
<EXP_VALUE>
<ATTRIBUTE>
```

```
age
</ATTRIBUTE>
</EXP_VALUE>
</SIMPLE_EXP><COND_OP>
<RELATIONAL_OP>
LT
</RELATIONAL_OP>
</COND_OP><SIMPLE_EXP>
<EXP_VALUE>
<CONSTANT>
60
</CONSTANT>
</EXP_VALUE>
</SIMPLE_EXP>
</RELATIONAL_EXP>
</EXPRESSION>
</DIMENSION>
</DIMENSION_MEASURE_LIST>
</SELECTION>
</OPERATION>
</OPERATION_LIST>
</DATA_QUERY>
</INTERSECTION>
</OPERATION>
</OPERATION_LIST>
</DATA_QUERY>
</QUERY>
```

Listing 4.3: Example of INTERSECTION XML string

As for CHANGE_LEVEL, CHANGE_BASE, DRILL_ACROSS and META_DATA, they are included the in grammar in basic form. They are part of future work and further development of these operations are needed.

4.6 The Client Side API

Within the NOX query language framework, the conceptual model and its associated grammar are intended to provide an abstract development environment for expressive analytical programming. The NOX framework was implemented and tested to send queries to the Sidera framework and receive results back. In this section, we provide a detailed overview of the NOX query transformation model.

In a nutshell, NOX provides persistent transparency via a source code re-writing mechanism that interprets the developer's OOP query specification in JAVA and decomposes it (by NOX pre-processor) into the core operations of the OLAP algebra. Persistent transparency means that the programmer queries a cube as if it is an object residing in local memory. These operations are given concrete form within the NOX grammar and then transparently delivered (via standard socket calls) at run-time to the backend analytics server for processing. Results are again transparently injected back into the running application. In our proposed framework, OLAP compilation is a multi step process. This process is described in Listing 4.4.

```
1. Find the OLAP queries (that need to be executed) in the
   source code. OLAP queries are located by the parser through
   special keywords. This will be explained in further detail in
   the next chapter.

2. Parse each operation (such as select, project, ...) method
   and convert it into an algebraic form represented in XML. How
   the conversion is done will be illustrated in the next
   subsection.

3. Rewrite part of the programmer's source code to include new
   network methods that connect to the server and transfer the
   corresponding XML. The rewriting process will be explained in
   the next chapter. It is important to note here that the
   original programmer's own source code that will not actually
   be executed. The rewritten code is the one that will be
   executed in the next step.

4. Recompile the new source code.

5. The server receives the XML, extracts the grammatical
   elements and hands off the results to the underlying query
   engine.

6. Eventually, query results will be transparently passed back
   to the client application via the same network mechanism. In
   other words, the results will be sent back from the server in
   XML format. Then, on the client side, they will be converted,
   using result set manipulation by our prototype (will be
   presented in detail in the next chapter), into the appropriate
   type for the native language (JAVA) and ``inserted'' back into
   the program itself.
```

Listing 4.4: Pseudocode for OLAP compilation

We note at this point that we have chosen to provide external libraries for NOX rather than direct language modification. This is partly to encourage portability between languages, as we consider the NOX model to be broadly applicable to any modern OOP language. In our prototype, we use JAVA as the OOP language to implement NOX functionality. However, it is also due to the fact that while OLAP/BI is an immensely important commercial domain, OLAP-specific language extensions would have virtually no relevance to the vast majority of developers working in arbitrary domains.

Figure 25 depicts the UML class diagram for NOX. The diagram shows three parts that are separated by dashed lines. The first part shows the NOX API client query classes that make up the client side libraries of NOX. These are the classes that are ultimately imported by the programmer in order to specify a specific OLAP query. Specifically, the classes will define the query's dimensions, hierarchies and measures and are created by extending the existing classes OlapDimension, OlapHierarchy and OlapMeasure respectively. The second part of the diagram illustrates an example of NOX program-specific query classes used for an OLAP query instance. In this example, classes CustomerDimension, StoreDimension and DateDimension extend the OlapDimension class. The classes CustomerHierarchy, GeographicHierarchy and StoreHierarchy extend the OlapHierarchy class. Finally, classes ProfitMeasure and SalesMeasure extend the OlapMeasure class.

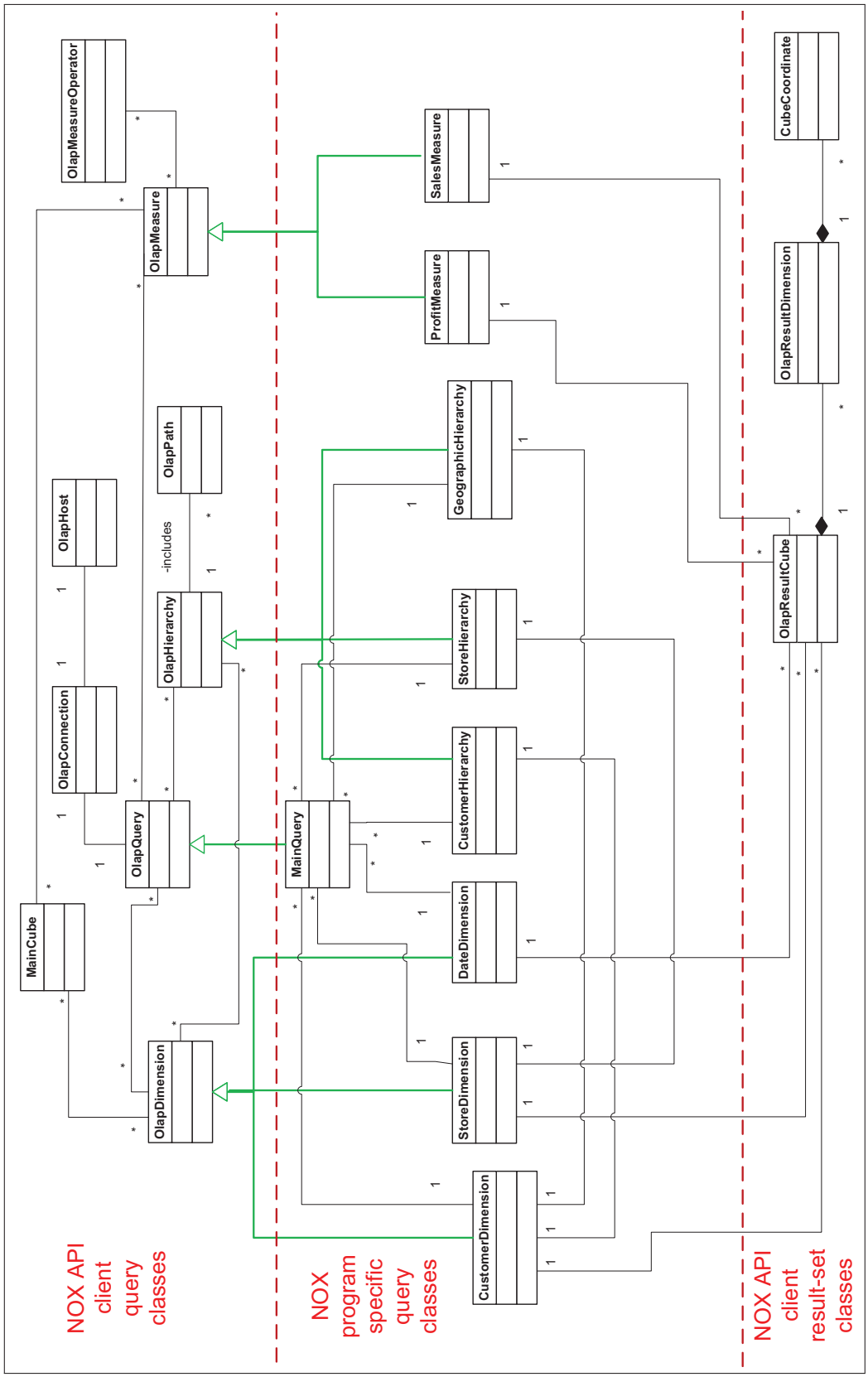


Figure 25: UML class diagram for NOX

A more thorough UML class diagram for the NOX API classes, along with their attributes and methods is given in Figure 26. The fields and methods of each class are provided. Briefly explaining the diagram:

- The OlapQuery class (parent of all OLAP queries) has many-to-many relationships with the OlapDimension class, the OlapMeasure class and the OlapHierarchy class with these three classes being used to build the query. In other words, an OLAP query can examine one or more dimensions, their hierarchies and one or more measures (implicitly from fact tables). Dimensions, hierarchies and measures can, of course, be used in more than one query.

- The OlapQuery class has a one-to-one relationship with the OlapConnection class which in turn has a one-to-one relationship with the OlapHost class. Note that each query gets delivered via standard socket calls to the backend analytics server for processing.

- The MainCube class — which can be any cube name — has a many-to-many relationship with the OlapDimension class and the OlapMeasure class, with a cube consisting of one or more dimensions (features) and one or more fact tables (measures). Features and measures can be common to other cubes in the hypercube space.

- The OlapMeasureOperator class has a many-to-many relationship with the OlapMeasure class, where operators are applied on measures.

- The OlapPath class has a many-to-one relationship with the OlapHierarchy

class, where an OLAP hierarchy includes one or more OLAP paths but an OlapPath belongs to only one hierarchy. Remember an OLAP path is a sequence of values where each value on the OLAP path comes from a different level in the hierarchy. An example of an OLAP path is given in yellow in Figure refsymmetricc.

- Finally, we note that due to its fundamental significance to warehousing and OLAP processing, a pre-defined Date class is included in the NOX API. The class extends the OlapDimension class and includes sub classes for Days, Months and Years. Of course, the developer is free to further extend the class to add additional functionality.

We note that additional class diagrams and UML representations for the program specific query classes will be presented in Chapter 5.

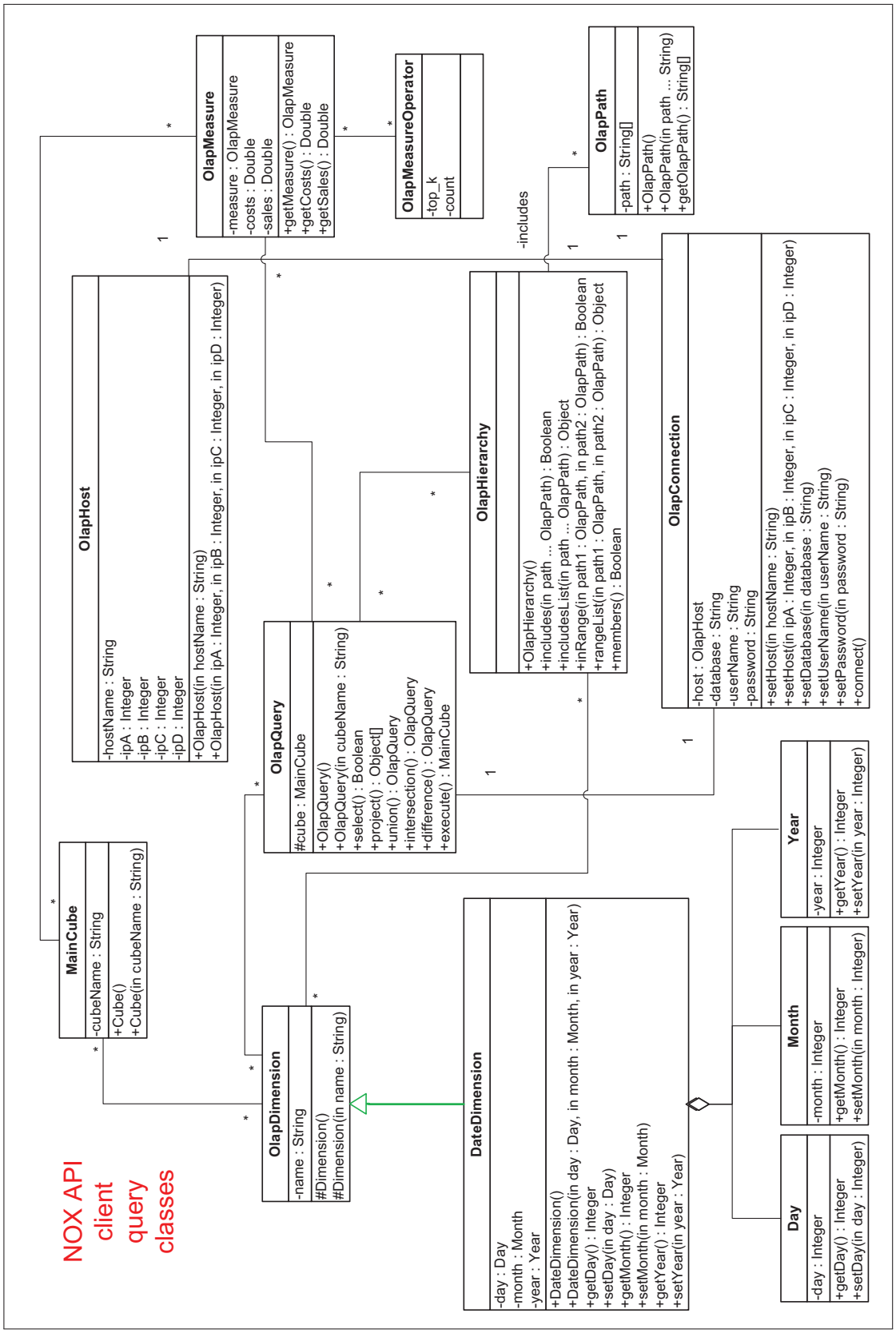


Figure 26: UML class diagram for the NOX API library

4.6.1 The NOX Pre-processor

NOX must identify query-specific elements of the source code (JAVA in our prototype) and transform them as required. To accomplish this, NOX includes a pre-processing module that transforms code before passing it to the standard Java compiler. The pre-processor is produced with the JavaCC parser generator and its JJTree Tree builder plug-in [Jav, JJT]. Briefly, JJTree is used to define parse tree building actions that are executed during the later parse process. In the NOX case, JJTree identifies query-specific code constructs (e.g., class definitions) that should be augmented. The output of JJTree is then used by JavaCC to construct a Java parser that actually locates and transforms appropriate methods. We note that although NOX utilizes a complete Java 1.5 grammar for its parser, the pre-processor only examines and/or processes tree nodes defined by JJTree. In practice, this makes the pre-processing step extremely fast.

So what is the pre-processor looking for? NOX is supported by client libraries that define the relevant query components. The fundamental structure is the OlapQuery class. Listing 4.5 provides a partial listing of its contents. Use of this structure allows programmers to over-ride the OlapQuery and provide only the operations necessary for the query at hand (often just selection). The remaining methods are effectively no-ops. Note that these methods never actually get executed. They are only stubs that are used to allow the regular programming language compiler to verify that the structure of the query is valid. The body of these methods will be replaced by some programmer-specific code. The "execute" method would then serve as being both the

```
public abstract class OlapQuery {

 public boolean select() {return false;}
 public Object[] project() {return null;}
 public OlapQuery drill_across() {return null;}
 public OlapQuery union() {return null;}
 public OlapQuery intersection() {return null;}
 public OlapQuery difference() {return null;}

 public ResultSet execute(){
  return new ResultSet();
 }
}
```

Listing 4.5: Base class OLAP query with stub methods

invocation mechanism and the element of the class definition that would be re-written during parsing the query.

Figure 27 graphically illustrates the process described thus far. In the box at the top left, we see the parser generation tools that produce the *translating pre-processor*. The dashed line to the pre-processor itself indicates that this association is static, and the parser building tools are not invoked directly at either compile time or run-time.

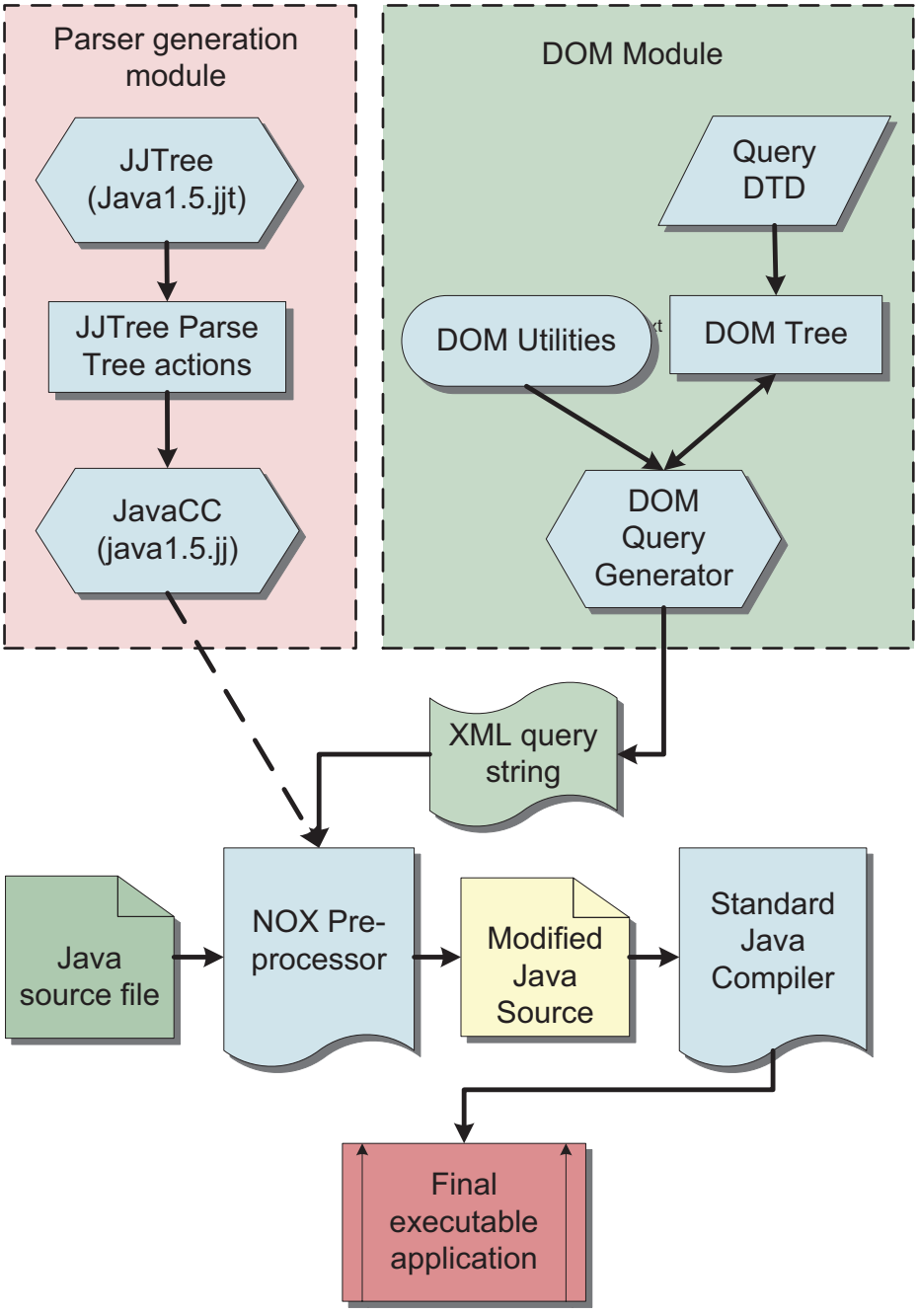


Figure 27: The client compilation model.

In terms of the compilation process, the pre-processor takes as input the original Java source file and then, using the parse tree constructed from this source, converts source elements into an XML decomposition of the OlapQuery. Examples of source elements that get converted are select, project, intersection, union and difference. These will be converted into selection, projection, intersection, union and difference. Other elements are mapped according to the children / parent relationships and according to the stored values. Different combinations are checked and mapped to the proper XML elements and values. 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, the NOX translation step would be integrated into a build task (ANT, makefile, IDE script, etc.) and would be completely transparent to the programmer. The details of the components of Figure 27 are as follows:

- Parser Generation Module:

    1. **JJTree (Java1.5.jjt)**: This component is part of the Java Compiler. It acts like a pre-processor to the JavaCC parser generator and is mainly used to build the program parse tree. In fact, we can add some functionality to this component to allow us to choose which parts of the parse tree to build. Ultimately, JJTree generates code to construct parse tree nodes during the parsing process. We can also rename the nodes and choose to highlight

tokens that help us during the process of parsing the client program.

In practice, the Java1.5.jjt is compiled with JavaCC and it produces a JJTree Parse Tree that corresponds to the client program. It also produces the JavaCC (Java1.5.jj) component that produces a Java parser for the client program. A more detailed description of how JJTree is used in our system is given in Section 4.6.2.

2. **JJTree Parse Tree Actions**: A parse tree is generated by JJTree. Nodes in the tree correspond to grammar rules in the Java language. More details and examples are given in Chapter 5.

3. **JavaCC (Java1.5.jj)**: This is the parser generator that is produced by JJTree (Java1.5.jjt) and is compiled by JavaCC to produce the Java Parser for the client source file.

- DOM Module:

1. **Query DTD**: This is an XML Schema that defines the various components of the Sidera systems such as OLAP queries, meta queries, database structure and query results. It is used both on the client side and the server side of the Sidera architecture. On the client side, it validates the XML string generated during the process of OLAP query parsing. The grammar DTD used in our system was given in Section 4.5.

2. **DOM Tree**: The DOM Tree is an intermediate component between the JJTree parse tree and the XML corresponding to the OLAP query. The

DOM Tree is useful for two main reasons. The first is that it can be directly validated against the OLAP query grammar DTD schema. After finishing construction of the DOM query tree, the DOM tree translation to an XML query string is a relatively straightforward step. The second reason is that its construction is flexible and intuitive when using the DOM methods in the DOM Query Generator.

3. **DOM Utilities**: A library of DOM related utilities is used to manipulate DOM trees and XML strings. They provide functions to build, access and modify DOM tree objects. Also, the processes of transforming DOM trees into XML strings and vice versa are well supported.

4. **DOM Query Generator**: This component contains a number of methods used by the modified JavaCC compiler to generate DOM nodes, where these nodes make up the DOM tree corresponding to the client OLAP query. The DOM tree is then tanslated to an XML string using a DOM utility.

- **Java Source file**: This is written by the programmer where he defines a query to extend the OlapQuery class. In the extended query class, the programmer over-rides the "operation" method(s) needed to implement the OLAP query and then instantiates and "executes" the query object. An example of a Client Java program that implements a "select" operation is given in Listing 5.1 of Section 5.2.

- **XML Query String**: This contains the tags and values that were translated from the OLAP query given by the programmer.

- **NOX Pre-processor**: This is the parser that is the product of the JavaCC Java Compiler. It is executed to parse the Client Java Program using DOM utilities. The NOX pre-processor traverses the parse tree to find the subtree that corresponds to the OLAP query. While searching the subtree for components of the query in a depth-first fashion, it builds the corresponding XML DOM tree. Keywords in the subtree guide the search process. Methods of the DOM Query Generator are used to produce the DOM tree. Then, the DOM tree is validated using the OLAP query DTD and translated to XML string using a DOM utility. The pre-processor locates the OLAP queries in the source code, parses the OLAP operations methods and converts them into an XML string. In addition, the NOX pre-processor will rewrite the programmers "execute" method to send the XML to the server. Rewriting of the "execute" method is done using JavaCC and JJTree actions. Hence, the Client Java Program and the XML query string are both input to the NOX pre-processor and the output will be the modified Client Java Source. Although the XML query string is produced from the OLAP query given by the programmer, it is not part of the client source file. The pre-processor with the help of DOM utilities produce the XML query string from the client source file that will be included in the over-writing "execute" method. All of this is transparent to the programmer.

- **Modified Java Source**: This is the program that is produced after parsing and compiling the Client Java Program. This program will have the programmer's rewritten "execute" method that establishes a connection to the OLAP server and sends it the XML string corresponding to the OLAP query. This updated client Java program needs to be recompiled. Again, this entire process is transparent to the programmer. In fact, the programmer does not even know that an updated Java program exists. Debugging in this case becomes a problem and it is interesting to tackle this problem in future work.

- **Standard Java Compiler**: Obviously, this is the regular Java Compiler that just needs to be invoked against the updated client Java program.

4.6.2 JJTree in the NOX Pre-processor

As described earlier, JavaCC is a parser generator for Java applications and JJTree is a pre-processor to JavaCC that inserts parse tree building actions at various places in the JavaCC source. JJTree can generate code to construct parse tree nodes for each nonterminal in the language. In the NOX pre-processor, we have modified this behavior so that some nonterminals do not have nodes generated, while other nonterminals have nodes generated for parts of their productions' expansion. Hence, the parse tree is built so that nodes, needed in parsing the programmer's OLAP query and building the corresponding DOM tree and eventually the XML corresponding string, are generated.

We use an example of a parse tree, shown in Figure 28, that is generated by

JJTree in the NOX pre-processor to illustrate how parsing of the associated OLAP query is done. In this figure, a node is denoted by an oval shape with the name of the node written inside the shape. We added a new reserved word to the JavaCC/JJTree parser in NOX, which is denoted by the token called "OlapQuery" that is preceded by the word "extends". This is done so that the parser locates each class that extends "OlapQuery". Then, it parses the class code that describes the OLAP query by locating words (that gets transformed to nodes in the parse tree) to translate it to XML.

When parsing the OLAP query code, NOX is actually parsing the subtree (of the parse tree produced by NOX) corresponding to the query. Hence, while the client Java program is being parsed using JavaCC and JJTree, each time the parser finds an "OlapQuery" query, it generates a node called "ExtendsOlapQueryandBody" and the subtree under this node will be the located subtree that will be parsed by NOX parser to generate the XML query. Also, the first and the last token of the body of the "ExtendsOlapQueryandBody" class body will be located by the NOX JJTree methods so that the "execute" method will be rewritten to include the XML query before sending it to the server. The method to do this using JJTree is given in Listing 4.6. The method will not be valid from a Java point of view. JJTree has its own syntax for writing code.

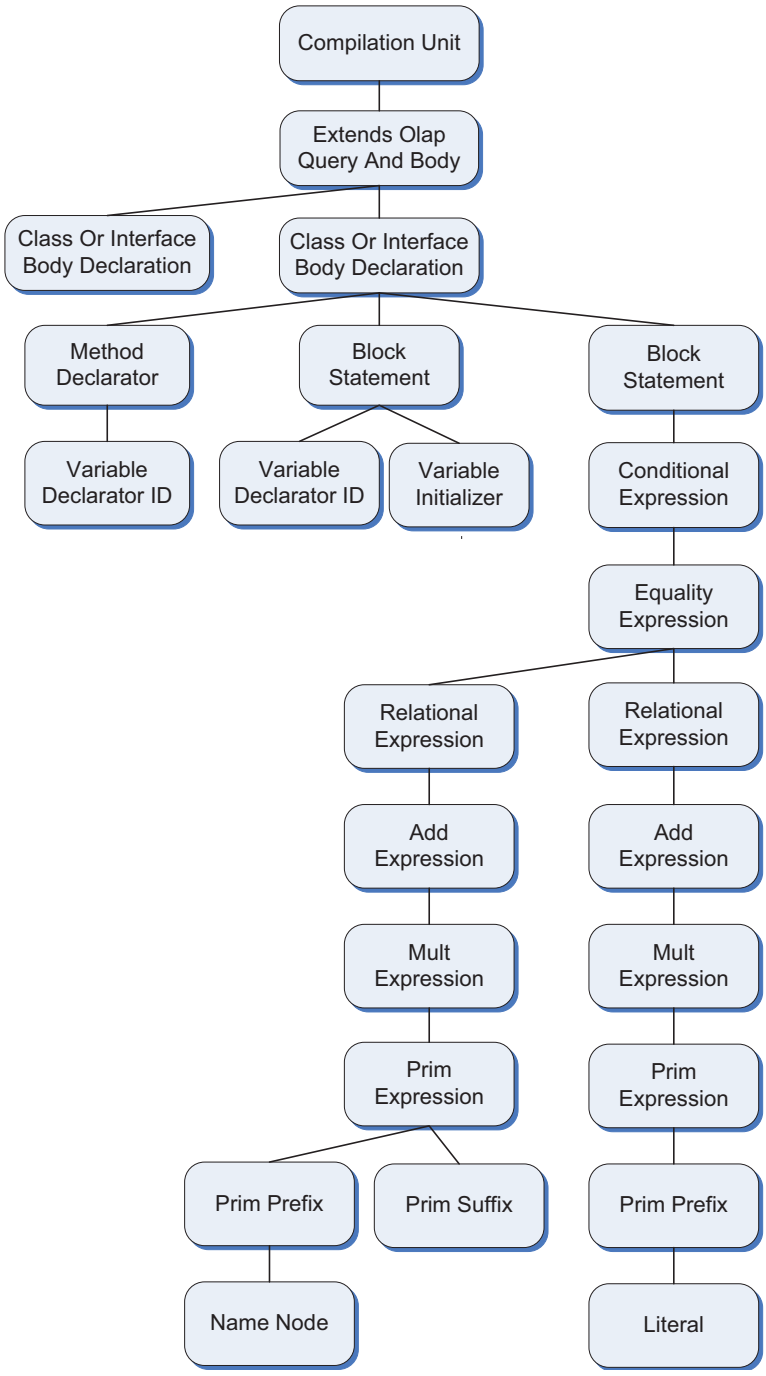


Figure 28: Simple query parse tree.

```
void ExtendsOlapQueryandBody(boolean isInterface):
{
   boolean extendsMoreThanOne = false;
   Token t;
}
{
 "extends" <OLAPQUERY>
  { t = getToken(1);  }
   "{" ( ClassOrInterfaceBodyDeclaration(isInterface) )*
  { jjtThis.jjtSetFirstToken(t);
    jjtThis.jjtSetLastToken(getToken(0));
  }
  "}"
}
```

Listing 4.6: Saving first and last tokens of a class that extends OlapQuery using JJTree

Figure 28 shows the root of the parse tree generated by JJTree in the NOX preprocessor along with some of the tree's branches. This tree corresponds to the query given in Lisitng 5.1 (will be presented in Chapter 5. The tree's root node is named "Compilation Unit", which is the default name given by JJTree. The child of the root node is an "ExtendsOlapQueryandBody" node, which is the root of the subtree that contains the OLAP query. The parser generated by JavaCC and JJTree in NOX recursively visits the nodes of the "ExtendsOlapQueryandBody" subtree, and when it finds a method name (saved as a token in a node of the tree), it checks its value if it is one of the OLAP operators, such as select and project. Then, the parser has found an OLAP query operation. This is an example of how NOX detects names that are

used to build the OLAP operation's XML query string. A middle step, which is an implementation detail, is that the parser first generates a DOM tree, and then the DOM tree is translated to XML.

Finally, we present two pieces of pseudocode to illustrate building the complete process of a parse tree and preprocessing the input file. We note, however, that we will not go through much implementation detail as it becomes somewhat tedious for the reader. The first pseudocode is given in Listing 4.7 to show the steps of how to build a parse tree. This pseudocode is implemented in NOX in JavaCC1.5.jjt using JavaCC and JJTree. In JJTree, we can add additional tokens as part of the grammar. We can also return a tree and set a root of a tree. Some nonterminal variables that we do not need to have their nodes (in the parse tree) produced will have their producion rules set to void.

```
1. Add a new reserved word ``OLAPQUERY'' as a token in the
   grammar
2. Set ``CompilationUnit'' as the name of the root node of the
   Abstract Syntax Tree
3. Set #void for some nonterminals that we do not need to
   produce nodes in the parse tree
4. Mark queries that extend ``OlapQuery'' with a special name
   node ``ExtendsOlapQueryAndBody'' and save the token of the
   query class name (as in Listing 4.5).
5. Set names to certain nonterminals in the parser and save the
   tokens so they are manipulates while walking the parse tree
   by doing the following:
      { jjtThis.jjtSetFirstToken(getToken(1));}
```

```
        {((SimpleNode)n).end = getToken(0);  }
  6. Return the root of the AST tree of the java input file by
     embedding a Java action ``return  jjtThis'' at the end of
     ``CompilationUnit'' production of the JavaCC grammar.
   7. Get the ``CompilationUnit'' root node of the parse tree and
      walk the tree by calling the interpret() method as follows:
      (parser.jjtree.rootNode().interpret();)
```

Listing 4.7: Pseudocode for constructing the parse tree in Java1.5.jjt (using JavaCC and JJTree)

4.7 Conclusion

In this chapter, we presented mainly the client side libraries and parsing infrastructure of NOX. We first described the Sidera System Architecture, a comprehensive architectural model for a fully parallelized OLAP NOX queries are sent, by the client, to the Sidera system in XML format. Then, the Sidera system will process the data and return its result to the client. We also presented the components of NOX both at the primary physical and logical levels. The primary components are the NOX conceptual model, its OLAP algebra and related grammar, client side libraries, programming API, augmented compiler, and cube result set. The OLAP algebra in our framework is similar to the YAM^2 algebra proposed by Romero. Finally, we illustrate the usage of JJTree in the NOX pre-processor. Some pseudocode is given to describe building the parse tree and preprocessing the input file.

Chapter 5

NOX Application Programming

The NOX framework, as described in the preceding chapter, provides a clean and intuitive development model for the (Java) programmer. In the prototype, we provide object-oriented programming libraries of interface/abstract classes that the programmer uses to construct queries. Developers then are able to make use of object-oriented concepts in building their queries. Simply put, they can think of the cube simply as an object residing in memory. In fact, it is one of the primary advantages of this framework that programmers can visualize an entire Terabyte size OLAP database as a series of objects in local memory. We can do this easily in our design because the server provides an OOP domain model, with the underlying code verification translation steps completely transparent to the client side programmer.

In this chapter, we demonstrate the practical use of NOX through a number of query examples. Section 5.1 uses UML notation to graphically illustrate the structure of a basic OLAP query. Section 5.2 describes the **select** method and illustrates its use through a small but typical `SELECTION` example, as well as a more sophisticated

query. Section 5.3 depicts the **project** method and illustrates its use through a small but typical `PROJECTION` example. In Section 5.4, we discuss how OLAP set operations are represented in NOX. In Section 5.5, we expose the query inheritance feature of the framework. Section 5.6 explains how NOX manipulates the results of the OLAP query returned from the server. Finally, the last section evaluates the NOX framework in comparison to the MDX language.

5.1 UML of a Sample OLAP Query

Figure 29 shows the UML class diagram of the dimensions, hierarchies and measures for a specific OLAP query example. In this example, the name of the OLAP query is MainQuery and it inherits the library class OlapQuery. It takes 3 parameters as input to the query and it has a **select()** method (i.e., the `SELECTION` algebraic operation) and a **project()** method (the `PROJECTION` algebraic operation). These methods use the Customer dimension, the Store dimension and the Date dimension and their corresponding hierarchies. The **project()** method also displays values for the `PROFIT` measure and the `SALES` measure.

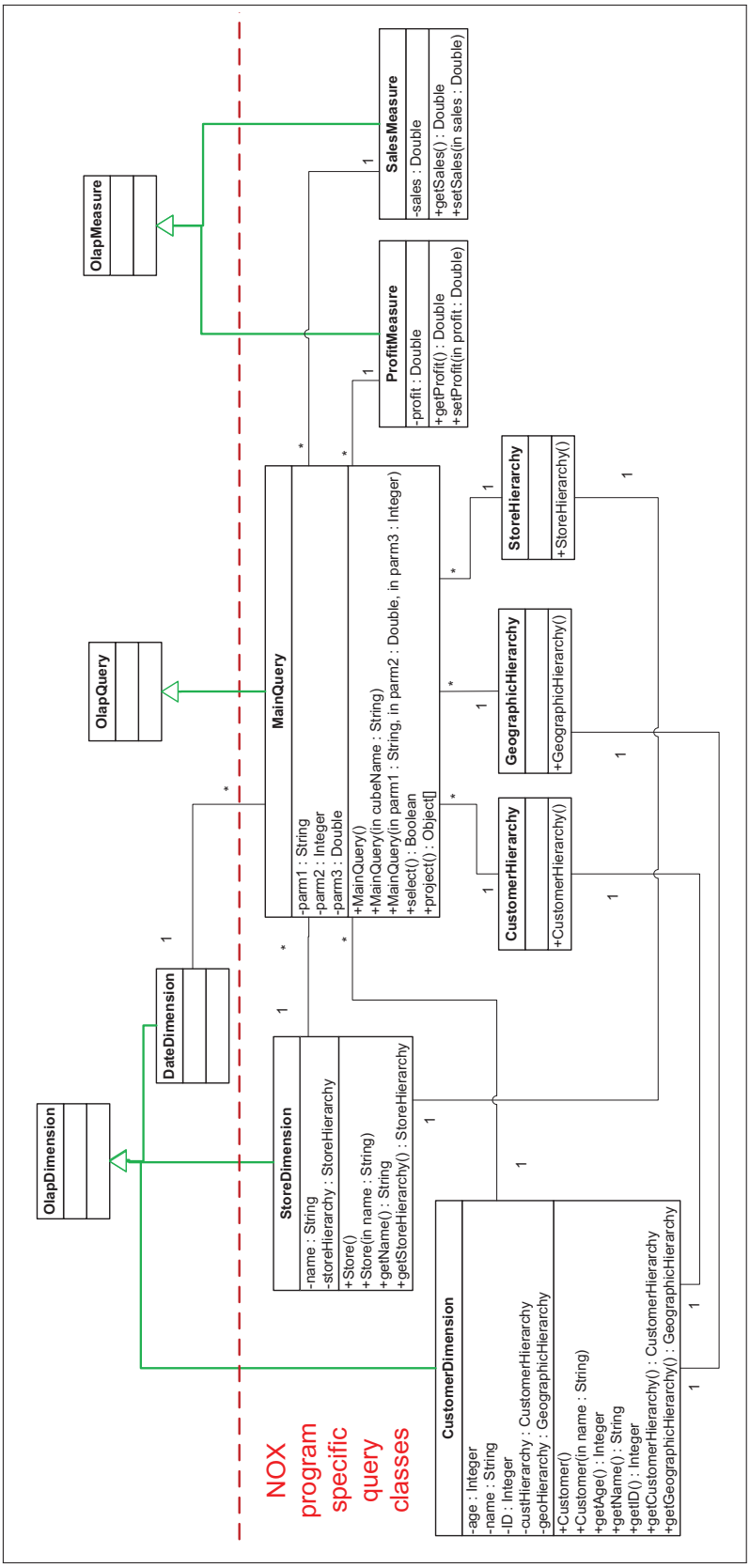


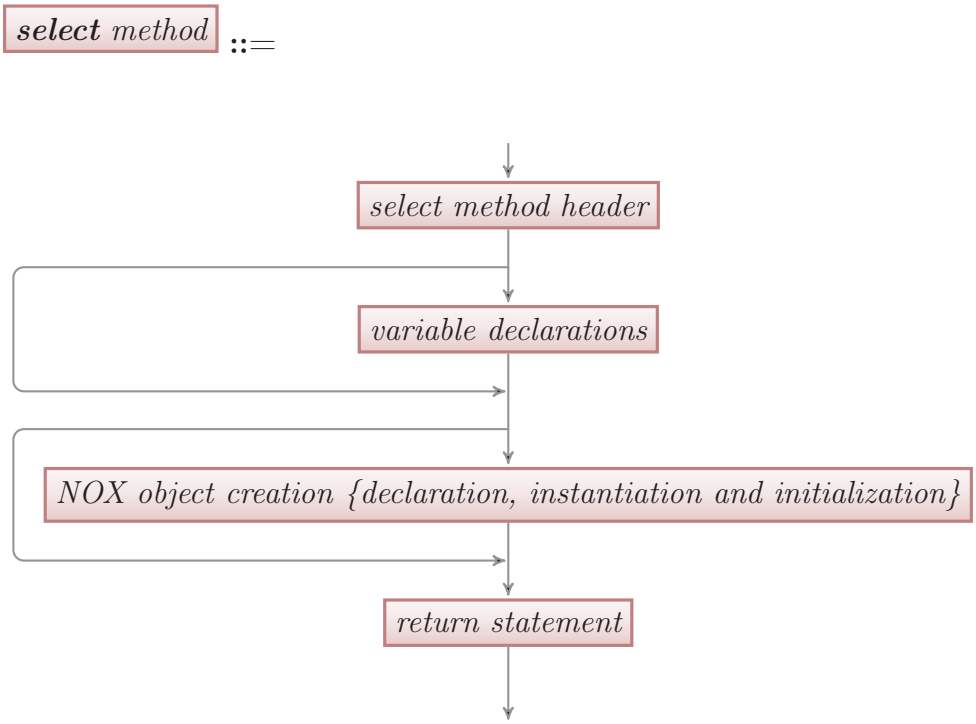
Figure 29: UML class diagram for NOX programmer OLAP classes

5.2 SELECTION

The **select()** method is the method responsible for the SELECTION algebraic operator ($\sigma_p$cube) presented in Section 4.4. SELECTION is identifying one or more cells from the d-dimensional space by a logic predicate p. It is known as "slicing" and "dicing" in the industry.

5.2.1 SELECTION Syntax in NOX

The syntax diagram for the **select** method in NOX is given as follows:



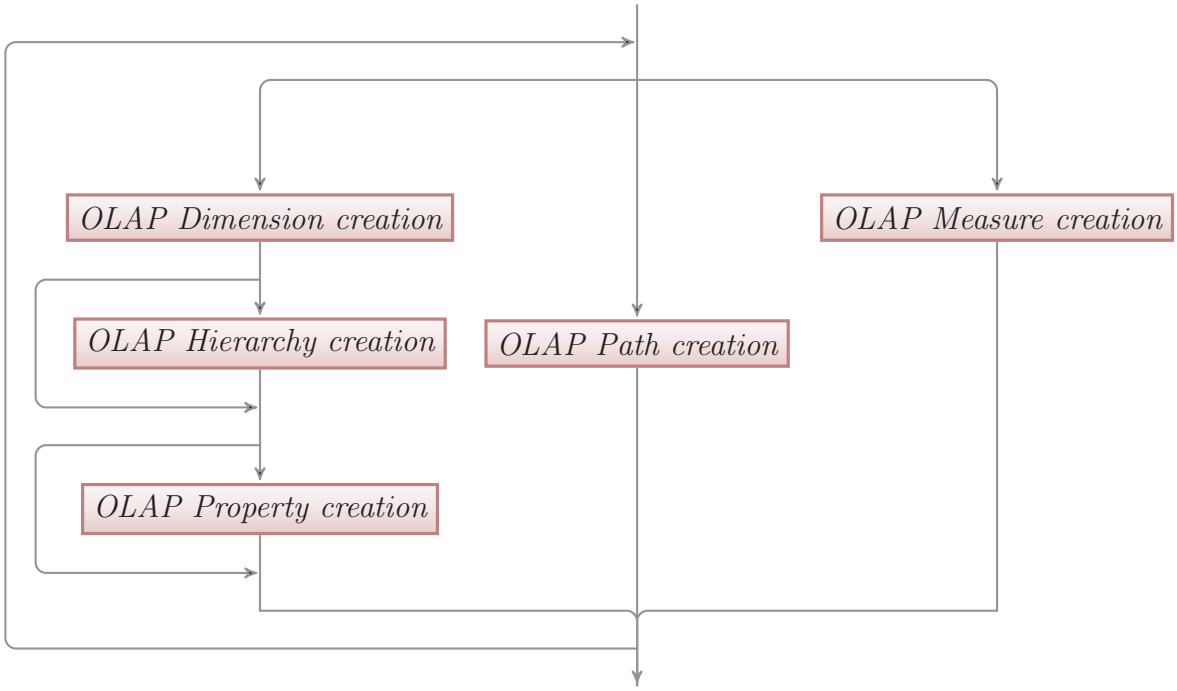
The **select method header** is followed by some optional variable declarations and NOX object creation that are used in the OLAP query. The NOX objects are declared

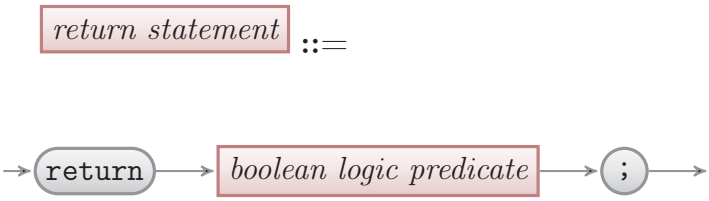
and instantiated and possibly initialized. The last statement returns the selection criteria, formulated as a boolean expression.

Next, we give the syntax diagrams for the **select method header**, the **NOX object creation {declaration, instantiation and initialization}** and the **return statement**.

select method header ::=

NOX object creation {declaration, instantiation and initialization} ::=





The first syntax diagram shows the return type of the **select** method which is boolean. The return type of a selection operator is always a single boolean value. The second syntax diagram identifies the type of NOX objects that are created. There are the OLAP dimensions and their properties, OLAP hierarchies and their paths, and OLAP measures. The OLAP hierarchies cannot exist without their dimensions. Also, each OLAP property is defined as a member of a certain dimension. The parallel lines corrspond to parallel constructs creation. The last syntax diagram returns the boolean logic expression that corresponds to the `SELECTION` criteria of the OLAP query. Examples in the following subsections demonstrate how these programming constructs define an OLAP query in NOX.

5.2.2 A Simple `SELECTION`

NOX is used on the client-side and is responsible for compiling, translating, sending the query in a certain format and receiving the final result from the server side. This is all transparent to the programmers. It allows them to think of the target of the "query" as though it were merely an object(s) residing in local memory.

To illustrate how application programming is done in NOX, we begin with a query that specifies a simple selection criteria, namely that we would like to list total sales

for the year 2001. Listing 5.1 provides the corresponding OlapQuery definition, along with a small **main** method that demonstrates how the query's execute method would be invoked. In this example, we will ignore the `PROJECTION` method that would specify the measure and display attributes in order to focus our attention on the `SELECTION` operation. We also ignore the network connection and authentication methods. We can see that the **select** method instantiates a DateDimension and invokes its getYear() method. Because Dates are virtually universal in analytical processing, NOX provides a fully functional Date class "out of the box" (with the standard empty method bodies). In terms of the `SELECTION` criterion, note how it is specified simply via a boolean-generating return statement.

In Lisiting 5.1, the select() method is not called as it is only defined to contain the query code and then translated to XML query string. It is the execute() method that contains the corresponding XML query string that will be called. The execute() method is defined in Listing 4.5 and called in Listing 5.1. The select() method is defined in Listing 4.5 and over-ridden in Listing 5.1.

It is crucial that we understand why such an approach is used. From the programmer's perspective, the query is executed against the physical data cube such that the selection criteria will be iteratively evaluated against each cell. If the selection test evaluates to true, the cell is included in the result; if not, it is ignored. In reality, of course, the server would almost certainly not resolve a query in this manner. After the source code has its parse tree produced, constructs such as select, project, intersection, union and difference get converted into XML elements such as selection,

```
class SimpleQuery extends OlapQuery {
   public SimpleQuery(String cubename) {
     super(cubeName);
   }

   public boolean select() {
     DateDimension date = new DateDimension();
     return date.getYear() == 2001;
   }
   //...projection excluded
}

public class Demo1 {
   public static void main(String[] args) {
    //...DBMS network connection
    SimpleQuery myQuery = new SimpleQuery(``SalesByDate'');
    ResultSet result = myQuery.execute();
    // ...manipulate result set
    }
}
```

Listing 5.1: Simple OLAP query

projection, intersection, union and difference. Other elements are mapped according to the children / parent relationships in the parse tree and according to values stored. Different combinations of nodes and tokens are checked and mapped to the proper XML elements and values. Throughout this process, various DOM utilities and services are used in order to generate and verify the XML. Finally, once the query is decomposed and sent to the server, the backend DBMS is free to resolve the query.

In terms of the decomposition itself, it is of course represented in an XML string generated by the pre-processor. This string is inserted into the query's execute method and subsequently invoked in the main method. At run-time, this invocation produces a network call to the DBMS to send the query and receive its results. Again, we stress that all of this processing is entirely invisible to the end user. Listing 5.2 is a re-written version of the select OLAP query example described earlier. The select() method is not anymore included in the re-written version of the execute() method as it was defined in the first place just to include the OLAP query. Instead, an execute() method will replace the select() method containing the XML query string of the original OLAP query defined in the select() method. The execute() method returns an empty cube is returned, as it is only visualized as an object in memory by the programmer. It actually contains no data as the real result data cube is sent from the server after the OLAP query is resolved.

To complete this first example, Figure 28 of Chapter 4 shows the relevant portion of the parse tree that is constructed by the pre-processor. This tree corresponds to the query given in Lisiting 5.1.

```
class SimpleQuery extends OlapQuery {

   SimpleQuery(String cubeName) {
      super(cubeName);
  }

  public Cube execute () {
     String xmlQuery =
   ``<?xml version=``1.0'' encoding=``UTF-8'' standalone=``no''?>
        <!DOCTYPE QUERY SYSTEM ``dtd/ClientQuery.dtd''><QUERY>
        <DATA_QUERY> <CUBE_NAME> sample </CUBE_NAME>
        <OPERATION_LIST> <OPERATION> <SELECTION> <DIMENSION_LIST>
        <DIMENSION> <DIMENSION_NAME> Date </DIMENSION_NAME>
      <EXPRESSION> <RELATIONAL_EXP> <SIMPLE_EXP> <EXP_VALUE>
      <ATTRIBUTE> year </ATTRIBUTE> </EXP_VALUE> </SIMPLE_EXP>
      <COND_OP> <EQUALITY_OP> EQUALS </EQUALITY_OP> </COND_OP>
      <SIMPLE_EXP> <EXP_VALUE> <CONSTANT> 2001 </CONSTANT>
      </EXP_VALUE> </SIMPLE_EXP> </RELATIONAL_EXP> </EXPRESSION>
      </DIMENSION> </DIMENSION_LIST> </SELECTION>
      </OPERATION> </OPERATION_LIST> </DATA_QUERY> </QUERY>'';

     Communicator comm = new Communicator();
     comm.sendQuery(xmlQuery);
     return new Cube();
  }
}
```

Listing 5.2: Re-written version of Listing 5.1 that contains the XML string and sends it to the server

While the trees can become fairly complex for larger queries, in this simple case we can see the special "Extends OLAP Query And Body" node that has been inserted by JJTree, as well the long branch of selection criteria nodes that identifies the programmer's query logic.

By "walking the tree", the pre-processor — in conjunction with the DOM facilities — is able to produce the final query XML string depicted in Listing 5.3 that is actually sent to the server. A DOM tree representation of the XML string is given in Figure 30.

5.2.3 A More Sophisticated `SELECTION` Query

An example of a more complex query is given in Listing 5.4. This query performs a "slicing" and "dicing" operation where the sales values are returned for customers whose age is more than 40 years old, where the months are between May and October of 2007, where the supplier's balance $/100 < 45623$ and the products are either the "interior" parts of "automotive" vehicles or the "lights" of the "exterior" parts of the "automotive" vehicles. The "Product Hierarchy" of the "Product" dimension is used to specify OLAP paths to specific levels in the hierarchy. This is explained in detail in Chapter 6. Complex queries made of `SELECTION` and `PROJECTION` operations can be formulated by having **select** method and **project** method in the OLAP query. Similarly, set operations are formulated by query **union**, **intersection** and **difference** methods in the OLAP query.

```
<QUERY>
 <DATA_QUERY>
  <CUBE_NAME>sample</CUBE_NAME>
  <OPERATION_LIST>
   <OPERATION>
    <SELECTION>
     <DIMENSION_LIST>
      <DIMENSION>
       <DIMENSION_NAME>date</DIMENSION_NAME>
       <EXPRESSION>
        <RELATIONAL_EXP>
         <SIMPLE_EXP>
          <EXP_VALUE>
           <ATTRIBUTE>year</ATTRIBUTE>
          </EXP_VALUE>
         </SIMPLE_EXP>
         <COND_OP>
          <EQUALITY_OP>EQUALS</EQUALITY_OP>
         </COND_OP>
         <SIMPLE_EXP>
          <EXP_VALUE>
           <CONSTANT>2001</CONSTANT>
          </EXP_VALUE>
         </SIMPLE_EXP>
        </RELATIONAL_EXP>
       </EXPRESSION>
      </DIMENSION>
     </DIMENSION_LIST>
    </SELECTION>
   </OPERATION>
  </OPERATION_LIST>
 </DATA_QUERY>
</QUERY>
```

Listing 5.3: Simple query XML string

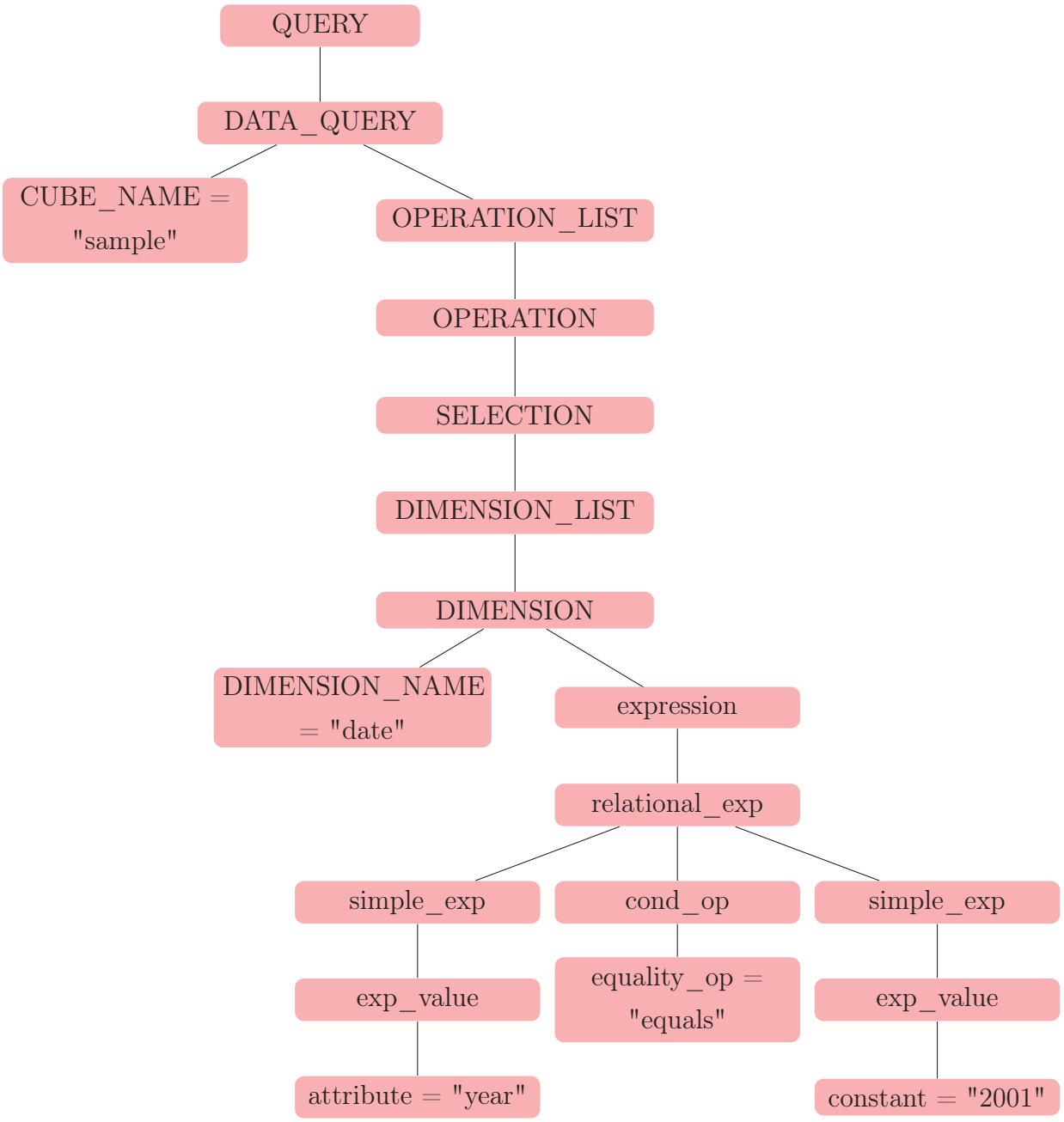


Figure 30: DOM tree representation of the XML string in Listing 5.3

```
class ComplexQuery extends OlapQuery  {

 public boolean select() {

  DateDimension date = new DateDimension();

  Customer customer = new Customer();
  OlapProperty dateMonth = new OlapProperty(date.getMonth());
  Supplier supplier = new Supplier();
  Product1 product = new Product();
  ProductHierarchy proHierarchy = product.getProductHierarchy();

        return ((customer.getAge() > 40) &&
        ((date.getYear() == 2007) && (dateMonth.inRange(5,10)))
        &&
        ((supplier.getBalance() / 100) <  45623.00)   &&
        (proHierarchy.includes(new
           OlapPath(``automotive'',``exterior'',``lights''),
         new OlapPath(``automotive'',``interior'')));
 }
}
```

Listing 5.4: A more complex OLAP query

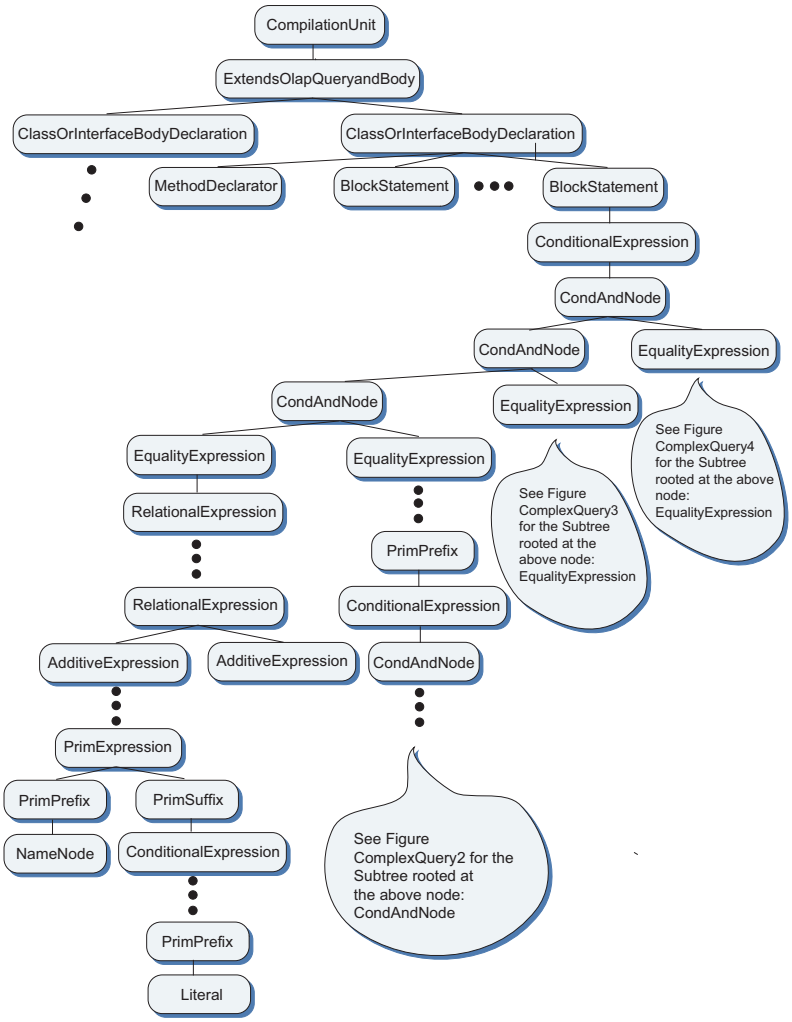


Figure 31: A subtree of the more complex query parse tree

A subtree of the query parse tree generated by the pre-processor is depicted in Figure 31. Traversing the tree, in an in-order way, the reader will note three subtrees that are given in Figure 32, Figure 33 and Figure 34 respectively. Special tokens are located by the NOX pre-processor while traversing the nodes of the OLAP query parse tree and the XML string of the query is constructed accordingly.

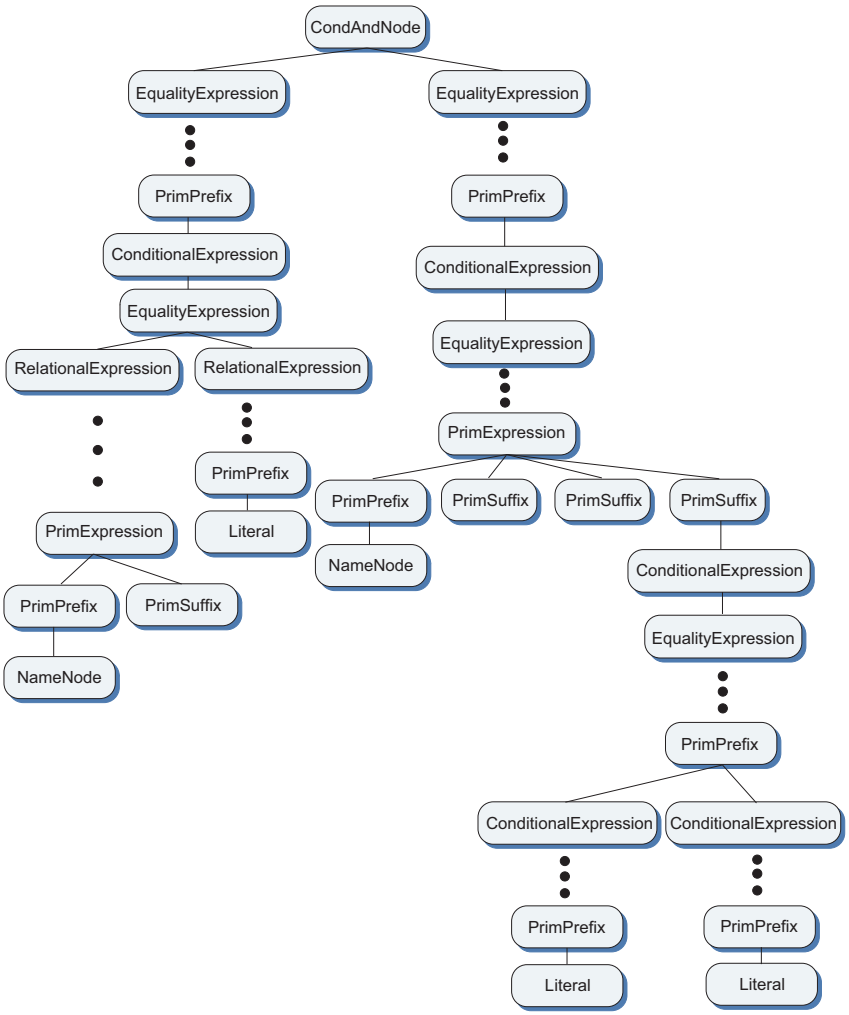


Figure 32: ComplexQuery2: Subtree rooted at "CondAndNode" node of Figure 31

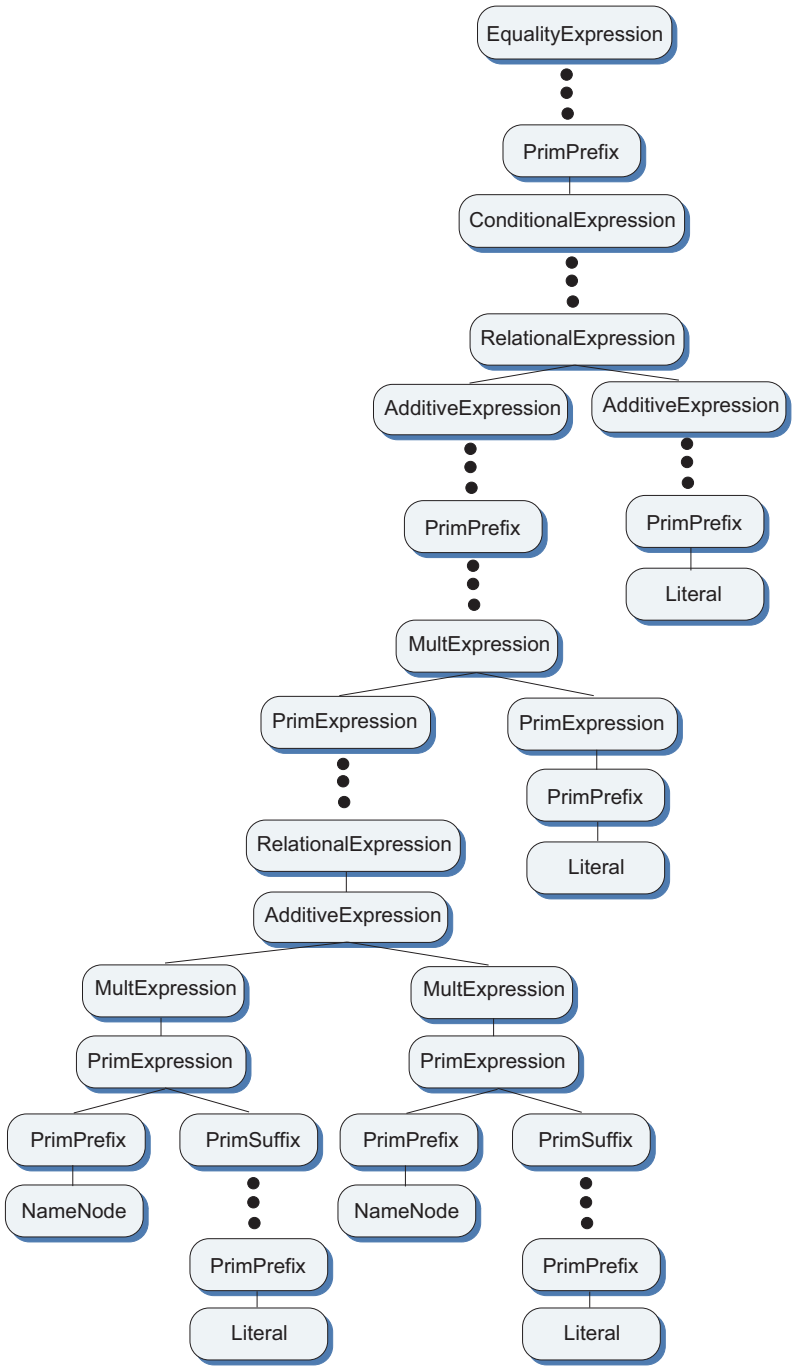


Figure 33: ComplexQuery3: Subtree rooted at tne first “EqualityExpression” node of Figure 31

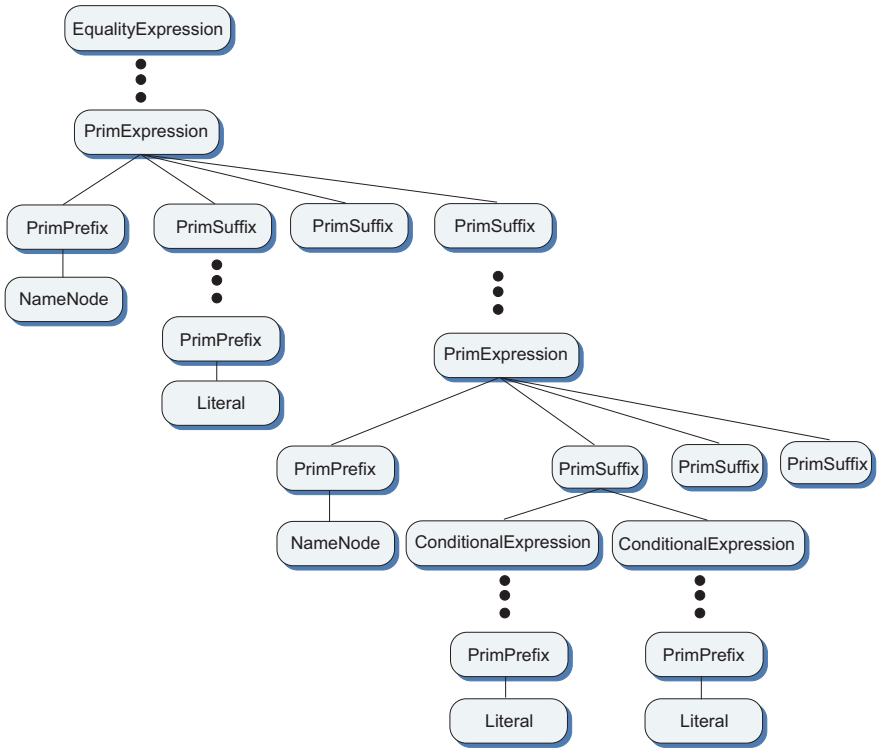


Figure 34: ComplexQuery4: Subtree rooted at the second "EqualityExpression" node of Figure 31

NOX parses the query in Listing 5.4 by walking its parse tree and then sends the resulting XML string, given in Listing C.1 in Appendix C, to the server.

As we will be comparing NOX queries with MDX queries, we give a brief explanation of the grammatical structure of MDX. Listing 5.5 depicts the canonical MDX query format. The `SELECT-FROM-WHERE` is syntactically similar to SQL but definitely its functionality is different. In the `SELECT` clause, the axis_specification defines the data cube axis where features/measures are displayed/returned. The FROM clause specifies the cube name and the WHERE clause specifies the cube cells selection constraints. MDX calls it a *slicer_specification* but it is not only concerned with slicing but dicing as well.

```
<select_statement> ::= [WITH <formula_specification>]

                       SELECT [<axis_specification>
                       [, <axis_specification>]*]

                       FROM [<cube_specification>]

                       [WHERE [<slicer_specification>]]
                       [<cell_props>]
```

Listing 5.5: MDX `SELECT` statement

The MDX equivalence of the query is shown in Listing 5.6. The "slicing" and "dicing" condition with the comparison operators "<" and ">" is expressed in MDX using the FILTER statement in the *axis_specification* (responsible for display in concept)

part of the query. On the other hand, the "slicing" and "dicing" condition with the equality comparison operators can be expressed in either the FILTER statement in the *axis specification* part or the WHERE part of the query. At the very least, this can be considered confusing since the building constructs of the query specification should be data specific, which is not the case in MDX. As such, MDX does not provide the concept of separation of concerns which, as much as possible partitions the program into distinct non-overlapping features or behaviors. Thus, the modularity of programming and encapsulation of data is not achieved in MDX. Though the size of the MDX is smaller in this case, other codes tend to scale more in size as queries get more complicated.

5.3 PROJECTION

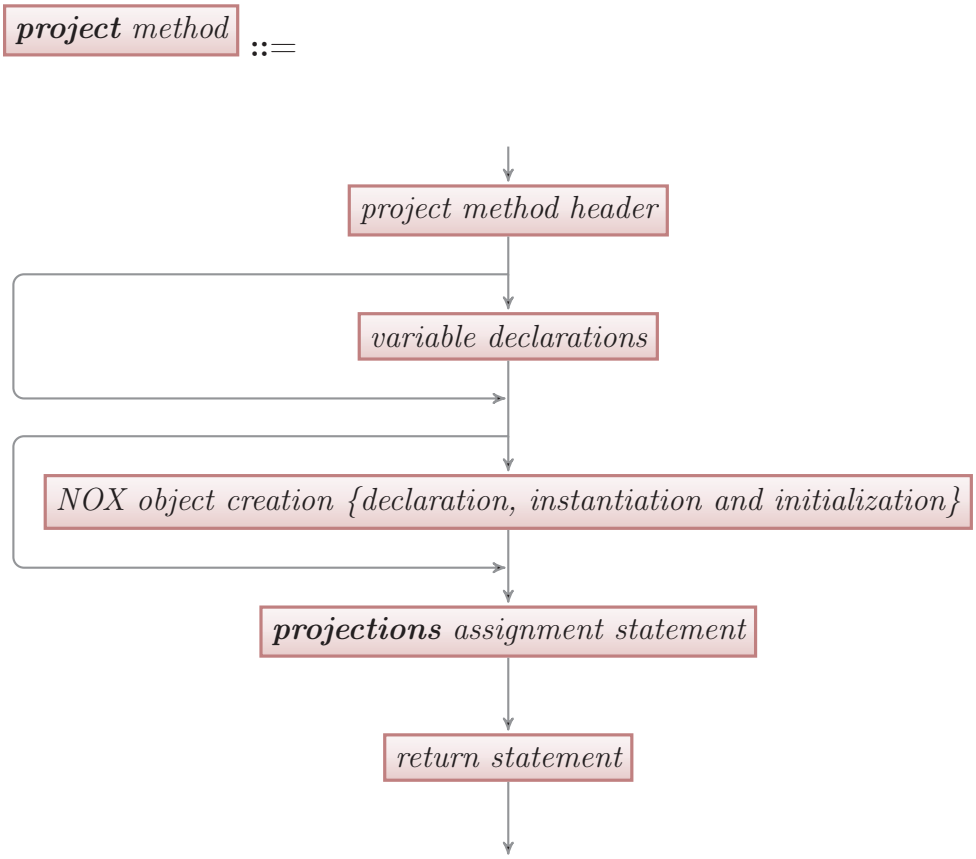
The **project()** method in the OlapQuery class is the method responsible for the `PROJECTION` algebraic operator ($\pi_{attribute_1,...,attribute_n} cube$). `PROJECTION` identifies presentation attributes, including *both* the measure attributes and dimension members (features).

```
SELECT

FILTER({}, (([Suppliers].[Balance])
                / 100 < 45623.00 )
            AND ([Customer].[Age] > 40))

ON ROWS

FROM SampleCube;

WHERE ([Date].[Year].&[2007),
       [Date].[Month].&[5]:[Date].[Month].&[10],
       {[Product].[ProductHierarchy].[automotive].
        [exterior].[lights],
       [Product].[ProductHierarchy].[automotive].[interior]})
```

Listing 5.6: A more complex MDX query corresponding to the query in Listing 5.4

5.3.1 PROJECTION Syntax in NOX

The syntax diagram for the **project** method in NOX is given as follows:



The **project method** syntax starts with the **project method header**, followed by some optional **variable declarations** and **NOX object creation** that are used in the OLAP query. The NOX objects are declared and instantiated and possibly initialized. The OLAP hierarchies would be one such example. The main display statement in the **project method** is the **projections** assignment statement which lists the display criteria, whether features or measures.

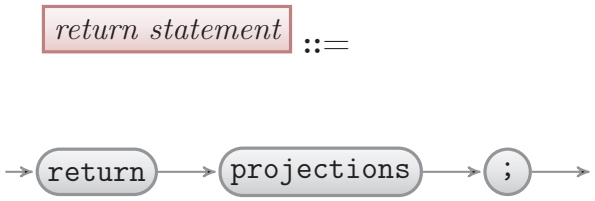
Next, we give the syntax diagrams for the **project method header**, the **projections assignment statement** and the **return statement**. **NOX object creation {declaration, instantiation and initialization}** is the same as the one given in the previous section for `SELECTION`.

project method header ::=

The **project method** as shown in the syntax diagram above returns type Object[ ]. In Java, this implies an array of Objects, indicative of its purpose within the NOX framework to identify various display attributes (strings, ints, floats, etc.).

projections assignment statement ::=

In the above syntax diagram, the right hand side operand of the **projections** assignment statement list Objects to be displayed. In terms of compilation, any object works. However, in terms of logic errors, some types will not make sense to be displayed, such as a boolean operator.



The last syntax diagram returns the Object[ ] type expression that corresponds to the display criteria of the OLAP query. We demonstrate how the **project** method programming constructs define an OLAP query in NOX by a simple example. More complex examples are given throughout the thesis chapters.

5.3.2 A Simple `PROJECTION`

To illustrate how `PROJECTION` is done in NOX, we begin with a query that specifies a simple `PROJECTION` criterion, namely that we would like to return the cells for the MonthlySales measure and display the names of the customers and their age. Listing 5.7 provides the corresponding OlapQuery definition. The **project** method returns the type Object[ ] which is the type of **projections** that must be used to list the features/measures to be displayed. In this example, the **project** method instantiates a Customer dimension and invokes its getName() and getAge() methods. It also instantiates a Measure and invokes its getMonthlySales() to return the MONTHLY SALES measure. Here the types of the getters, when assigned to the **projections** set, are flexible since they are casted to type Object. Features from the same dimension are bound to one axis of the result cube. An example is returning the features Age and Name on one dimension/axis because they belong to the same dimension,

namely Customer. As for the input cube to the OLAP query, in this example, it is given as the parameter to the query constructor, namely "SalesByDate". We note here (similar to what was explained earlier in the select() example of Listing 5.1 that the project() method is not called as it is only defined to contain the query code and then translated to XML query string. It is the execute() method that contains the corresponding XML query string that will be called. The execute() method is defined in Listing 4.5 and called in Listing 5.7. The project() method is defined in Listing 4.5 and over-ridden in Listing 5.7.

The equivalence in MDX of the NOX query given in Listing 5.7 is depicted in Listing 5.8. The measure `MONTHLY SALES` is displayed on the COLUMNS axis. Customer's `Name` and `Age` are displayed on the ROWS axis.

5.4 Set Operations

Previously, we suggested that *set* operations are defined quite simply in the NOX grammar. As it turns out, their specification in the native language is just as straightforward. Listing 5.9 provides a simple illustration. In this case, the programmer defines the "outer" query using the standard **select** method (and possibly others). In the `INTERSECTION` method, the "inner" query is specified merely by returning the query object that defines that query. No additional syntax is required. Using this information, the NOX pre-processor can combine both queries into a single XML string corresponding to the nested style of the grammar. The re-written execute() method will then include the XML string. This process is transparent to the programmer.

```
class SimpleQueryProject extends OlapQuery {

   public SimpleQueryProject(String cubename) {
     super(cubeName);
   }

 public Object[] project() {
  Customer customer = new Customer();
  Measure measure =  new Measure();

  Object[] projections = {measure.getMonthlySales(),
     customer.getName(), customer.getAge()};
  return projections;
 }
   //...selection excluded
}

public class Demo1 {
    public static void main(String[] args) {
       //...DBMS network connection
       SimpleQueryProject myQuery = new
          SimpleQueryProject(``SalesByDate'');
       ResultSet result = myQuery.execute();
       // ...manipulate result set
    }
}
```

Listing 5.7: Simple OLAP query projection

```
SELECT {[Measures].[Monthly Sales]} ON COLUMNS,
       [Customer].[Name], [Customer].[Age] ON ROWS

FROM SampleCube;
```

Listing 5.8: Simple MDX query projection corresponding to the query in Listing 5.7

This is why the select() method does not need to be called neither in the constructor method of the InnerQuery nor in the intersection method of the OuterQuery. The "inner" query class is given in Listing 5.10.

In general, set operations are syntactically modeled on an OOP paradigm. As mentioned in the previous chapter, just similar to a String equality check in Java, where we would write myString.equals("Joe"), rather than something like myString == "joe". In the intersection example given in Listing 5.9 and Listing 5.10, this same approach is expressed as outerQuery.intersection( ).

The MDX query corresponding to the set `INTERSECTION` query of Listing 5.9 is shown in Listing 5.11, where the FILTER statement is used to return sets with the "slicing" and "dicing" condition satisfied, and then these sets are used as arguments to the INTERSECT statement used to do set `INTERSECTION` in MDX. The INTERSECT operation in this example takes two arguments. The first argument has two conditions, namely FILTER ({}, [Customer].[age] $<$ 35) and FILTER ({}, [Customer].[age] $>$ 18). The second argument has one condition, namely FILTER({}, [Product].[weight] $>$ 10). These correspond to the conditions given in the select()

```
class OuterQuery extends OlapQuery{
 public OuterQuery (String cubename) {
   super(cubeName);
 }

 public boolean select(){
  CustomerDimension customer = new CustomerDimension();
  ProductDimension product = new ProductDimension();
  return ( (customer.getAge() < 30 ) && (product.getWeight() >
     10.0) );
 }

 public OlapQuery intersection(){
  return new InnerQuery();
 }
}
```

Listing 5.9: Set `INTERSECTION` operation using the **select** method in NOX

```
class InnerQuery extends OlapQuery {
 public InnerQuery (String cubename) {
   super(cubeName);
 }

 public OlapQuery InnerQuery () {
   return null;
 }

 public boolean select(){
  CustomerDimension customer = new CustomerDimension();
  return ( (customer.getAge() > 18 ) );
 }
}
```

Listing 5.10: The "Inner" query used in the `INTERSECTION` operation of Listing 5.9

```
SELECT

INTERSECT (
     ( FILTER({}, [Customer].[age] < 35),
       FILTER({}, [Customer].[age] > 18)   ) ,
     FILTER({}, [Product].[weight] > 10)
     )

ON COLUMNS

FROM SampleCube
```

Listing 5.11: MDX set `INTERSECTION` query corresponding to the query in Listing 5.9

methods of Listing 5.9, where the method intersection() does the intersection between these two conditions. One condition is the compound condition (customer.getAge() < 30) && (product.getWeight() > 10.0). The other condition is (customer.getAge() > 18).

Another example of the NOX set `INTERSECTION` is given in Listing 5.12, where the programmer now defines the "outer" query using the **project** method. Its "inner" query class is provided in Listing 5.13. Here, we use `Object includesList(OlapPath ...path)` method which identifies and ultimately displays values for members of some hierarchical paths in the dimensions hierarchies. This method is used in the **project** method, hence displaying values for members that belong to the requested hierarchical paths that are passed as arguments to the method. More details about

```
class OuterQueryProject extends OlapQuery {
 public Object[] project() {
   DateDimension date = new DateDimension();
   CalendarHierarchy calendarHierarchy =
      date.getCalendarHierachy();
   Object[] projections = {calendarHierarchy.includesList(new
      OlapPath(``2001'').new OlapPath(``2002'').new
      OlapPath(``2003''))} ;
   return projections;
 }

 public OlapQuery intersection(){
   return new InnerQueryProject();
 }
}
```

Listing 5.12: Set `INTERSECTION` operation using the **project** method in NOX

OLAP hierarchies and paths will be presented in the next chapter.

Listing 5.14 shows the MDX query corresponding to the set `INTERSECTION` query of Listing 5.12, where two sets are used as arguments of the INTERSECT statement, each set containing 3 different years. Hence, the query results in displaying the `INTERSECTION` of the two sets on the ROWS axis.

Similar to the `INTERSECTION` method in NOX that supports the functionality of the intersection of sets in OLAP, NOX's `UNION` and `DIFFERENCE` methods depict the `UNION` and `DIFFERENCE` of sets in OLAP, respectively.

```
class InnerQueryProject extends OlapQuery{
  public OlapQuery InnerQueryProject() {
    return void;
 }

 public Object[] project(){
   DateDimension date = new DateDimension();
   CalendarHierarchy calendarHierarchy =
      date.getCalendarHierachy();
   Object[] projections = {calendarHierarchy.includesList(new
      OlapPath(``2002''),
                          new OlapPath(``2003''),new
                              OlapPath(``2004''))} ;
   return projections;
  }
}
```

Listing 5.13: The "Inner" Query used in the `INTERSECTION` operation of Listing 5.12

```
SELECT

INTERSECT (
  {[Date].[Calendar Year].&[2001],
   [Date].[Calendar Year].&[2002],
   [Date].[Calendar Year].&[2003]}
   , {[Date].[Calendar Year].&[2002],
      [Date].[Calendar Year].&[2003],
      [Date].[Calendar Year].&[2004]} )

ON ROWS

FROM SampleCube
```

Listing 5.14: MDX set `INTERSECTION` query corresponding to the query in Listing 5.12

5.5 Query Inheritance

NOX queries are easily extendable due to the fact that inheritance is well-supported in NOX, as will be illustrated in this section. One of the reasons that we represent algebraic operations in separate methods is simply because most operations are semantically unique, making it very hard to combine operations into a single native language method (with a single return type). However, a second rationale is just as important. Namely, we feel that it is extremely valuable to allow the re-use of previous, often very complex, queries. We saw a simple example of this with the "inner" query in the previous section. A more powerful opportunity would be to allow programmers to re-use portions of already defined queries. Perhaps the most obvious example would be to re-define the **project** method to simply identify a different measure or display attribute. With virtually all current approaches(e.g, MDX) this would involve cutting and pasting a previous chunk of source code, a process that is both inefficient and error prone.

With NOX's distinct query methods, we now have a great deal more latitude in this regard. Listing 5.15 demonstrates how a "new" query extends an "old" one, in this case providing a new `PROJECTION` method. Because NOX obeys inheritance chaining, it sees that a new `PROJECTION` has been specified, and creates a new query with the `SELECTION` method of the "old" query and the `PROJECTION` method of the "new" query. This is because in inheritance chaining, either inherited methods can be used directly as they are or inherited methods can be overrideen by creating new instance methods

in the subclass that has the same signature as the one in the superclass. Any subsequent changes to the source of OlapQuery will be automatically integrated into the new query upon re-compilation. The method `inRange(OlapPath ... path)` accepts a variable length sequence of **OlapPath**'s as its arguments that are essentially used to match programmer-defined values against members of a dimension hierarchical path(s). This method is used in the **select** method, hence aggregating values for members that belong to the matched hierarchical paths. More details about the `inRange(OlapPath ... path)` method will be given in the next chapter.

```
class ThreeYears extends OlapQuery {

 public boolean select() {

   CustomerDimension customer = new CustomerDimension();
   DateDimension date = new DateDimension();
      TimeHierarchy timeHierarchy = date.getTimeHierachy();
   OlapPath fromYear = new OlapPath("1996");
         OlapPath toYear = new OlapPath("2001");

   return (timeHierarchy.inRange(fromYear, toYear) &&
      customer.getAge() == 35);
 }

 public Object[] project() {

         CustomerDimension customer = new CustomerDimension();
         SalesMeasure measure =  new SalesMeasure();

         Object[] projections = {measure.getCount(),
            customer.getName()};
```

```
        return projections;
      }
}

class ExtendsThreeYears extends ThreeYears {

 public Object[] project() {

        ProductDimension product = new ProductDimension();
        SalesMeasure measure =  new SalesMeasure();

        Object[] projections = {measure.getSales(),
           product.getLabel()};

        return projections;
      }
}
```

Listing 5.15: Example 1: Over-riding a query class

With MDX, in contrast, inheritance is not supported. Re-use of sets is permitted only by using the WITH SET statement as illustrated in Listing 5.16. This is very limited in the context of OLAP querying!

Another example of query inheritance in NOX is depicted in Listing 5.17. Here, the base query provides a `SELECTION` method, and two queries extend this base query, where each includes `PROJECTION` method. When this query inheritance scenario is expressed in MDX, as shown in Listing 5.18, it is a big drawback that the slicing criteria have to be repeated, as different members are displayed on the resulting cube axes. This is demonstrated in the FILTER expression in the "axis specification",

```
WITH

SET [3 Years] AS '[Time].[1996]:[Time].[2001]'

SELECT

{[Customer].[name]} on COLUMNS
{[Measures].[count]} on ROWS

FROM InventoryCube

WHERE ([3 Years], [Customer].[age].[35]);


SELECT

{[Product].[label]} on COLUMNS
{[Measures].[sales]} on ROWS

FROM InventoryCube

WHERE ([3 Years], [Customer].[age].[35]);
```

Listing 5.16: MDX query corresponding to the NOX query of Listing 5.15

where different sets are filtered according to the same slicing condition.

5.6 Result Sets

We come now to the representation of the query results. One of the great advantages of ORM systems is that they allow data to be more or less transparently mapped back into client applications. NOX offers the same functionality in the context of multi-dimensional cube results. Specifically, the framework retrieves results from the server and transforms them into a multi-dimensional array object that can be directly accessed via the OlapResultSet reference.

To understand how result sets are represented, it is first necessary to see how they are constructed. Once the analytics server has resolved the query, it packages the result into an XML message. A DTD is again used to define the OlapResultSet format. A listing of the DTD is provided in Listing 5.19. In short, the OlapResultSet is structured as a combination of meta data and cell data. The meta data 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.

Listing 5.20 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 actual fact, these ID values are cube index coordinates and will be used by the NOX client libraries to efficiently construct the OlapResultSet object. In the Raw Data section of the file, we can see how each cell value is associated with the

```
class SelectQuery extends OlapQuery {

 public boolean select() {

  CustomerDimension customer = new CustomerDimension();

  return (customer.getAge() > 35 && customer.getAge() < 65);
 }

}

class ExtendToProjectQuery1 extends SelectQuery {

 public Object[] project() {
  CustomerDimension customer = new CustomerDimension();
  SalesMeasure measure =  new SalesMeasure();

  Object[] projections = {measure.getCount(), customer.getName()};
  return projections;
 }

}

class ExtendToProjectQuery2 extends SelectQuery {

 public Object[] project() {
  ProductDimension product = new ProductDimension();
  SalesMeasure measure =  new SalesMeasure();

  Object[] projections = {measure.getSales(), product.getLabel()};
  return projections;
 }

}
```

Listing 5.17: Example 2: Over-riding query classes

```
SELECT

  FILTER({}, [Customer].[age] > 35
             AND [Customer].[age] < 65) on COLUMNS

FROM InventoryCube;


SELECT

  {[Measures].[count]} on COLUMNS,

  FILTER({[Customer].[name]}, [Customer].[age] > 35
           AND [Customer].[age] < 65) on ROWS

FROM InventoryCube;


SELECT

  {[Measures].[sales]} on COLUMNS,

  FILTER({[Product].[label]}, [Customer].[age] > 35
           AND [Customer].[age] < 65) on rows

FROM InventoryCube;
```

Listing 5.18: MDX query corresponding to the NOX query of Listing 5.17

```
<!ELEMENT RESULT_CUBE (META_DATA, RAW_DATA)>

<!ELEMENT META_DATA (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 DIM_NAME (#PCDATA)>

<!ELEMENT MEMBER_LIST (MEMBER+)>
<!ELEMENT MEMBER (MEMBER_NAME, MEMBER_ID)>
<!ELEMENT MEMBER_NAME (#PCDATA)>
<!ELEMENT MEMBER_ID (#PCDATA)>

<!ELEMENT RAW_DATA (ROW+)>
<!ELEMENT ROW (ID_LIST, VALUE)>
<!ELEMENT ID_LIST (MEMBER_ID+)>
<!ELEMENT VALUE (#PCDATA)>
```

Listing 5.19: Simplified version of OlapResultSet grammar

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. 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.

```
<RESULT_CUBE>
 <META_DATA>
  <CUBE_NAME>Sales</CUBE_NAME>
  <DIM_COUNT>3</DIM_COUNT>
  <DIMENSION_LIST>
   <DIMENSION>
    <DIM_NAME>Customer</DIM_NAME>
    <MEMBER_LIST>
     <MEMBER>
      <MEMBER_NAME>Joe</MEMBER_NAME>
      <MEMBER_ID>0</MEMBER_ID>
     </MEMBER>
     <MEMBER>
      <MEMBER_NAME>Mark</MEMBER_NAME>
      <MEMBER_ID>1</MEMBER_ID>
     </MEMBER>
     <!-- ... additional members -->
    </MEMBER_LIST>
   </DIMENSION>
   <DIMENSION>
    <DIM_NAME>Date</DIM_NAME>
    <MEMBER_LIST>
     <MEMBER>
      <MEMBER_NAME>2001</MEMBER_NAME>
      <MEMBER_ID>0</MEMBER_ID>
     </MEMBER>
```

```
      <MEMBER>
        <MEMBER_NAME>2002</MEMBER_NAME>
        <MEMBER_ID>1</MEMBER_ID>
      </MEMBER>
      <!-- ... additional members -->
    </MEMBER_LIST>
  </DIMENSION>
  <!-- ... additional dimensions -->
 </DIMENSION_LIST>
</META_DATA>
<RAW_DATA>
 <ROW>
  <ID_LIST>
   <MEMBER_ID>0</MEMBER_ID>
   <MEMBER_ID>1</MEMBER_ID>
   <MEMBER_ID>2</MEMBER_ID>
   <!-- ... additional members IDs -->
  </ID_LIST>
  <VALUE>345.24</VALUE>
 </ROW>
 <ROW>
  <ID_LIST>
   <MEMBER_ID>0</MEMBER_ID>
   <MEMBER_ID>1</MEMBER_ID>
   <MEMBER_ID>2</MEMBER_ID>
   <!-- ... additional members IDs -->
  </ID_LIST>
  <VALUE>96.78</VALUE>
 </ROW>
 <!-- ... additional rows/cells -->
</RAW_DATA>
</RESULT_CUBE>
```

Listing 5.20: Partial listing of Result Set

Once the XML result is received at the client, it is immediately transformed into a multi-dimensional object. In case the result set is too big to fit in memory, paging might be necessary. The XML is parsed using the same DOM facilities used to create the original query (of course with a different DTD). Meta data is inserted into a series of lookup data structures (i.e., maps and dictionaries) that not only allow efficient searches, but also permit transparent mapping between "user friendly" member names and the server generated member IDs that are virtually meaningless to the end user. Of course, these same Member IDs are critical to the module that builds the physical Result Set object. Specifically, the Result Set Builder begins by constructing an initially empty multi-dimensional array conforming to the specifications (i.e., dimension and member count) of the meta data. We note that this array must be dynamically generated as the number of dimensions in the result cannot be known in advance. Once this "shell" has been generated, a simple linear pass through the Raw Data section of the XML file allows direct insertion of cell values as per the associated member ID coordinates.

The Result Set API exposes a series of methods that allow for the simple manipulation of the cube results. Individual cell values can be retrieved merely by specifying the appropriate coordinates, either by axis value or member value. More sophisticated access can also be layered on top of the simpler access primitives. For example, Listing 5.21 shows how one might produce a simple report of all cells in the cube. One merely has to retrieve the member values for each dimension and then, with a set of nested FOR loops, combines the relevant coordinates for each cell.

```
String dimension0
// ... members retrieved
for (int member_id_0:= members_id0_dimension0; member_id_0
   <=members_id0n_dimension0; member_id_0++){
    for (int member_id_1:= members_id1_dimension1; member_id_1
       <=members_id1n_dimension1; member_id_1++){
     coordinates = new LinkedList<CubeCoordinate>();
     coordinates.add(new CubeCoordinate(dimension0, member_id_0));
     coordinates.add(new CubeCoordinate(dimension1, member_id_1));
     System.out.println( result.getCellValue(coordinates));
   }
 }
}
```

Listing 5.21: Trivial report method

Figure 35 shows the UML class diagram for the NOX API Result Set classes, where three classes OlapResultCube class, OlapResultDimension class and CubeCoordinate class are given. The CubeCoordinate class describe coordinates. Each coordinate is declared as a string value that belongs to some dimension. The OlapResultDimension defines coordinates that belong to some dimension. It does this by declaring a dimension name along with its TreeMap that contains different coordinate member names and their axis offset. OlapResultCube describes the cube that contains the different dimensions. It includes the cube name, the number of dimensions in the cube, cube data, the TreeMap that contains the differnt dimension information, and the dimensions order.

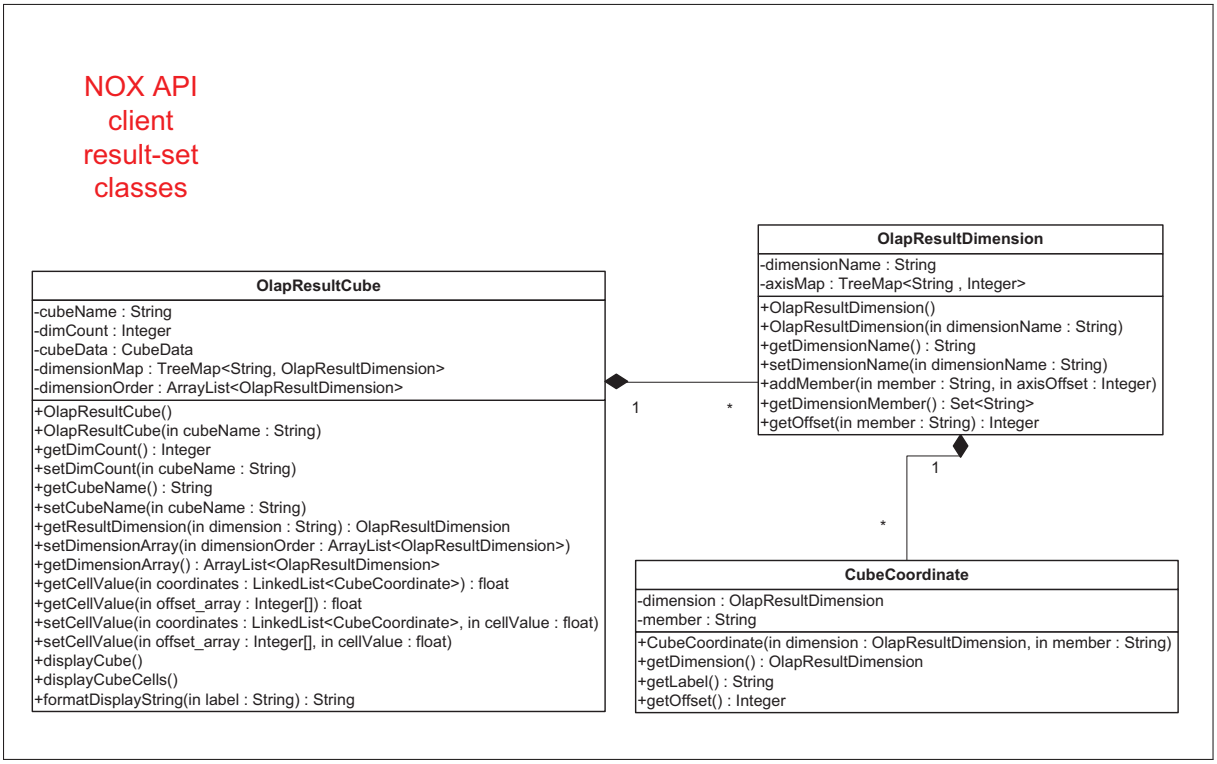


Figure 35: UML class diagram for the NOX API Result Set classes

5.7 Evaluation of the NOX Language

Table 1 compares the NOX OLAP query model to the MDX OLAP query model and illustrates how NOX is superior to MDX in the context of OLAP querying. Advantages of the NOX model over MDX are evident in OLAP query design, implementation, functionality, data encapsulation, separation of concerns, reusability and its high level native language representation.

To illustrate how obscure an MDX query can be, relative to a NOX query, we present in the coming subsection an MDX query and its equivalent NOX query.

5.7.1 Extension of the Project Method

With its flexible Object Oriented features, the NOX model can be easily extended to accommodate all functionalities of MDX. Listing 5.22 depicts the MDX version of a more "sophisticated" query, where measure VariantPercentage is created and defined as a formula in terms of RunningTotalSubs and some hierarchical attributes. Tuples are used here to indicate that RunningTotalSubs of a hierarchy path in 2004 in the time hierarchy is subtracted from that of 2005 in the time hierarchy, then divided by the total RunningTotalSubs of the "all" element in the time hierarchy. The "SELECT" on COLUMNS displays the RunningTotalSubs and the VariantPercentage. However, "SELECT" on ROWS is more complex in this example. There is a CROSSJOIN of TopCount ([DMA] .children , 5000 ,([RunningTotalSubs])) and [Time].[2004].& [1].[1].[1] ,[Time]. [2005].&[1].[1].[1] ,[Time]). This means that the top 5000 RunningTotalSubs of the children of [DMA] are crossjoined with the hierarchy

NOX Queries	**MDX Queries**
NOX is encoded in the same native query language as the implementation programming language.	The MDX language is distinct, and very different from, the implementation language.
NOX queries are checked at compile-time.	MDX queries are validated only at run-time because they are string-based.
NOX queries can easily be refactored.	MDX queries cannot be refactored.
It is relatively simple to re-use the object-oriented OLAP hierarchies as will be illustrated in Chapter 6.	More challenging reuse of hierarchies due to their low-level representation
Passing parameters to queries is flexible and intuitive as will be explained in Chapter 7.	Passing parameters to queries is basic and primitive
NOX queries are easily extendable as Inheritance is well-supported.	MDX query language is not easily extendable (if at all).

NOX Queries	**MDX Queries**
NOX result sets are returned as a cube object and are flexible in their manipulation and display.	Results are dependent on the embedding language that is responsible for their display.
NOX queries have an intuitive nature of set operations.	MDX queries have limited use of set operations
NOX supports the concept of separation of concerns, where slicing and dicing operations are focused in the **select** method and the display requirements in the **project** method	Slicing and Dicing operations are divided between two totally different parts of the **select** query namely the FILTER function in the "slicer specification" part and the WHERE part of the MDX query.
NOX supports programming modularity and encapsulation of data.	MDX queries are not well separated into modules where each module accomplishes one feature. Queries do not support data hiding.
As NOX queries get larger, object oriented features become even more valuable. The code becomes more scalable	As MDX code becomes more obscure, the development cycle increases significantly due to difficulty of writing the code, debugging and testing issues

Table 1: OLAP Queries Comparison between NOX and MDX

path of 2004 of the time hierarchy, the hierarchy path of 2005 of the time hierarchy, and the "all" element. These are the values that will be displayed on ROWS and COLUMNS. Finally, we note that the Selection is from "consumers" cube and the *slicer_spefication* (selection) is that "Zone Id" is equal to 14.

```
WITH
MEMBER [ Measures ] . [ VarientPercentage ] AS

'(([Time].[2004].\&[1].[1].[1],
[RunningTotalSubs]) -
([Time].[2005].\&[1].[1].[1],
[RunningTotalSubs])) /
([Time].[All Time], [RunningTotalSubs] )'

SELECT {
[ RunningTotalSubs ] ,
[ Measures ] . [VarientPercentage] }
ON Columns ,

CrossJoin(
TopCount (
{ [DMA] .children}  , 5000 ,
( [ RunningTotalSubs ] ) ) ,

{[ Time ] . [2004 ] . \& [1].[1].[1] ,
[ Time ] . [ 2005 ] . \&[1].[1].[1] ,
[ Time ] } )
ON ROWS

FROM [ consumers ]
WHERE ( [ Zone Id ] .\&[14] )
```

Listing 5.22: A more complex MDX query

To write the same OLAP query in NOX, we can extend the basic model with a number of relevant methods and programming constructs. Note that we did not implement these in the prototype but included them as part of the future work. Our motivation in this section is simply to demonstrate how flexible and easy to extend the NOX model in general and its specific prototype for this thesis in particular in order to support the OLAP required functionality. Listing 5.23 depicts the NOX version of the query, where the following two new programming constructs are presented.

1. The MainQuery that extends OlapQuery has **project** and **select** methods. The **select** method returns the cube cells that match the criterion that zone id = 14, whereas the **project** method includes a method call to the **project** method of the VariantPercentage. This is a new method that can be addded easily to the API. However, its implementation is not trivial.

2. Another new addition to the NOX API is the measureOperators class that contains operators which can be applied to measures. In this example, the operator is TopCount and takes three arguments: the first is the members to select from, the second is the number of the members, starting at the top, that are selected, and the third is the measure corresponding to the selected members.

```
Class VariantPercentage extends OlapQuery {

  public Object project( )  {
    Measure measures = new Measure();
```

```
    TimeDimension time = new TimeDimension ();
    TimeHierarchy timeHierarchy =  time.getTimeHierarchy();
    OlapPath pathTime1 = new OlapPath(``1 January 2004'');
    OlapPath pathTime2 = new OlapPath(``1 January 2005'');
    Object projections[] = { measures.getRunningTotalSubs
       (timeHierarchy.includesList(pathTime2)) -
        measures.getRunningTotalSubs
           (timeHierarchy.includesList(pathTime1))) /
        measures.getRunningTotalSubs (timeHierarchy.getAll()) };
   return projections;
  }
}

class MainQuery extends OlapQuery {
  public Object project( )  {
    VariantPercentage variantPercentage;
    OlapPath pathTime1 = new OlapPath(``1 January 2004'');
    OlapPath pathTime2 = new OlapPath(``1 January 2005'');
    Measure measures = new Measure();
    measureOperator topCount = Measures.getOperator(''TopCount'');
    DMA dma = new DMA();
    Object projections[] = {Measures.getRunningTotalSubs(),
      variantPercentage.project(),timeHierarchy.includesList
         (pathTime1,pathTime2), timeHierarchy.getAll(),
      measureOperators.topCount(dma.getChildren(),5000,
         Measures.getRunningTotalSubs() ) }
    return projections;
  }

  public boolean select( )  {
     Zone zone = new Zone();
     return ( zone.getId() == 14)
  }
}
```

Listing 5.23: **project** method extended in NOX and equivalent to MDX Listing 5.22

5.8 Conclusion

In this chapter, we described the API side of the NOX model. We demonstrated its object-oriented programming libraries through data encapsulation and inheritance. We illustrated the NOX API by presenting some UML diagrams pertaining to NOX interface/abstract classes. We also presented examples of using the **select** and **project** methods showing how easy and intuitive it is to build OLAP queries in NOX. The set operations, `INTERSECTION`, `UNION` and `DIFFERENCE` are implemented using NOX in an intuitive way. In addition, query inheritance provides the programmer with an intuitive way of reusing queries by the inheritance feature in Java in general, and NOX in particular. Result sets were simply returned from the server and embedded in a cube whose cells are accessed by the programmer. Finally, to show how powerful NOX is, we compare it to the de-facto OLAP language, MDX.

Chapter 6

Manipulating Hierarchies

As previously noted, hierarchical queries are extremely common in OLAP environments. Such roll up (or drill-down) processing is perhaps the most fundamental and thus important of all OLAP operations. NOX provides an intuitive way of specifying hierarchies simply because of its contemporary object-oriented features. Moreover, these OOP facilities give rise to additional query functionality, namely the ability to extend and reuse OLAP hierarchical queries.

In this chapter, we start in Section 6.1 by presenting the components of the framework that are responsible for building the OLAP hierarchies. Then, we give examples in Section 6.2 that demonstrate the flexibility and ease of expressing hierarchical queries using the NOX model.

6.1 Supplemental Hierarchy Classes

Recall that the NOX framework provides a series of classes that define various components of the object-oriented data model, such as dimensions and measures. In the

case of hierarchies, we extend the original design with a pair of new classes:

1. **OlapHierarchy** class

2. **OlapPath** class

The **OlapHierarchy** class provides the stub methods that are inherited by its subclasses. The **OlapPath** is essentially just a wrapper for a String Array that lists textual members of a full or partial hierarchy path. Listing 6.1 depicts the extendable **OlapHierarchy** class. The two methods, `includes(OlapPath ... path)` and `inRange(OlapPath ... path)`, each accepting a variable length sequence of **OlapPaths** as an argument, are essentially used to match programmer-defined values against members of a dimension hierarchical path(s). They are used in the **select** method, hence aggregating values for members that belong to the matched hierarchical paths. The **includes** method is used to request aggregated values of cube cells for members that belong exactly to the hierarchical paths passed as arguments. The **inRange** method requests aggregated values of cube cells whose hierarchical paths match the two arguments and all the hierarchical paths in between. The type returned by these two methods is *boolean* which matches the type returned by the **select** method. The type of the parameters passed is *OlapPath* which is described in Listing 6.2. Conversely, the two methods, `Object includesList(OlapPath ...path)` and `Object rangeList(OlapPath ...path)`, are used to identify and ultimately display values for members of some hierarchical paths in the dimensions hierarchies.

They are used in the **project** method, hence displaying values for members that belong to the requested hierarchical paths. The **includesList** method identifies values for members that belong to the hierarchical paths passed as arguments to the method. The **rangeList** method identifies values for members that belong to the hierarchical paths passed as its two arguments and all the hierarchical paths in between. The type returned by these two methods is *Object* which matches the type returned by the **project** method. The type of the parameters passed is *OlapPath* that is described in Listing 6.2.

The second main class used in building OLAP hierarchies queries is the **OlapPath** class whose implementation is given in Listing 6.2. The class has a private field called **path** which is an array of strings that holds values of a path in some hierarchy. The constructor method `OlapPath(String ... path)` and the public method `setOlapPath(String ... path)` are used to build a path in a given hierarchy with string values passed as parameters. The order of the values given is important. Each value corresponds to a member of a level in the hierarchy, and the values are given in the order of the levels in that hierarchy.

6.2 Hierarchies Examples

In this section, we present a series of examples that demonstrate how OLAP hierarchy queries may be implemented in practice (with an emphasis on the core `SELECTION` and `PROJECTION` operations). Because we now have an arbitrarily defined dimension to restrict (as opposed to the built-in Date dimension), we need a mechanism to

```
public class OlapHierarchy {

  public OlapHierarchy() {
  }

  public boolean includes(OlapPath ... path) {
    return false;
  }

  public Object includesList(OlapPath ...path) {
    return null;
  }

  public boolean inRange(OlapPath path1, OlapPath path2) {
    return false;
  }

  public Object rangeList(OlapPath path1, OlapPath path2) {
    return null;
  }

}
```

Listing 6.1: Class OlapHierarchy

```
public class OlapPath {

  private String[] path;

  public OlapPath() {}

  public OlapPath(String ... path) {
    int i = 0;
    for (String s:path) {
      this.path[i] = s;
      i++;
    }
  }

  public void setOlapPath(String ... path) {
    int i = 0;
    for (String s:path) {
      this.path[i] = s;
      i++;
    }
  }

  public String[] getOlapPath() {
    return path;
  }
}
```

Listing 6.2: Class OlapPath

statically type-check the relevant dimension attributes (Again, we do NOT want to use embedded strings to identify meta data elements). To do this, the programmer simply creates subclasses that inherit the library-provided OlapDimension class and adds the relevant attributes and *getter* methods (NOX can strip the "get" from the getters to obtain case insensitive attribute names). Both dimension attributes and hierarchies can be specified in this manner. Listing 6.3 illustrates this simple approach. GeographicHierarchy is a simple extension of the NOX OlapHierarchy class as shown in Listing 6.4. Given this simple Customer class, and a geographic hierarchy corresponding to that of Figure 18 of Chapter 4, we can now discuss the hierarchical query in the following example.

6.2.1 Hierarchy Example 1

In this example, we want to find data for older customers from California cities who purchased products in the first half of 2007.

In NOX

The hierarchical query in NOX is depicted in Listing 6.5, where the `SELECTION` conditions are expressed on both Date and Customer. We can see how the NOX path object is used to identify the elements of a partial hierarchy path. (Note that the path strings refer to *raw* cube data, NOT typed-checked meta data). Furthermore, we see the use of the built-in **includes** method to constrain the hierarchy condition. How does one interpret the expression `hierarchy.includes(path)`? Again, all selection criterion are defined relative to the current cube cell. Logically, this condition

```
class Customer extends Dimension{

  private int age;
  private String name;
  private int ID;
  private GeographicHierarchy geoHierarchy;

  Customer() {
    super();
  }

  Customer(String name) {
    super(name);
  }

  public int getAge() {
    return age;
  }

  public String getName() {
    return name;
  }

  public GeographicHierarchy getGeographicHierarchy() {
    return geoHierarchy;
  }

  public int getID() {
    return ID;
  }
}
```

Listing 6.3: Simple OLAP dimension

```
public class GeographicHierarchy extends OlapHierarchy {

  public GeographicHierarchy() {}

}
```

Listing 6.4: Class GeographicHierarchy

simply asks "Is this partial path consistent with the hierarchy members of this cell?", a representation that is indeed synonymous with the original query. We note that while there are many variations on hierarchy traversal, the NOX model always uses this same simple logical approach.

In MDX

The hierarchical query is given in MDX in Listing 6.6, where the geographic hierarchy path to California is declared as a member in the Customer dimension using the WITH MEMBER statement in MDX. We can see from the example how intuitive and object oriented hierarchies are expressed in NOX compared to MDX.

6.2.2 Hierarchy Example 2

In this example, we want to display values for the **sales** and **costs** measures for the two periods June-2001 and June-2002.

```
public boolean select(){
  DateDimension date = new DateDimension();
  Customer customer = new Customer();

  GeographicHierarchy hierarchy =
     customer.getGeographicHierarchy();

  OlapPath path= new OlapPath(``USA'', ``California'');

  return ( customer.getAge() > 65   &&
           hierarchy.includes(path) &&
           (date.getYear() == 2007  && date.getMonth() <= 6));
}
```

Listing 6.5: Manipulating hierarchies: example 1

```
WITH MEMBER [Customer].[California] AS
      [Customer].[Geographic Hierarchy].[USA].[California]

SELECT

FILTER({}, ( ([Date].[Month] <= 6) AND ([Customer].[Age] > 65)) )
    on ROWS

FROM SampleCube;

WHERE ( [Date].[Year].&[2007), [Customer].[California] )
```

Listing 6.6: MDX query corresponding to the query in Listing 6.5

In NOX

In the previous example given in Listing 6.5, the path object was declared and the constructor method called in the declaration statement:

`OlapPath path= new OlapPath(‘‘USA’’, ‘‘California’’);`.

Another way to construct OLAP paths is given in the example in Listing 6.7, where the paths are built with the constructor method while passed as parameters in the parameter list of the **includesList** method. In this example, it is a **project** method that is used for the OLAP query. The result of this `PROJECTION` is that the two measures **sales** and **costs** are displayed for members that belong to the "June-2001" partial path (of one level this time) of the hierarchy **TimeHierarchy** and to the "July-2001" partial path of the same hierarchy. The two partial paths are displayed with the measure values as the output of the projection.

In MDX

The query of the example in this subsection is given in MDX in Listing 6.8. Each of the two OLAP paths in the hierarchy is given as "name of the dimension" followed by "name of the hierarchy", followed by members' values at different levels of the hierarchy. In this case, it is one member value which is at the first level of the hierarchy. "June-2001" is given in this MDX example as an OLAP path `[Date].[Time Hierarchy].[June-2001]`.

```
public Object[] project() {
  DateDimension date = new DateDimension();
  Measure measure = new Measure();
  TimeHierarchy timeHierarchy = date.getTimeHierachy();

  Object[] projections = {measure.getSales(), measure.getCosts(),
                          timeHierarchy.includesList(
                            new OlapPath(``June-2001''),
                            new OlapPath(``July-2001'')
                          )
                         };

  return projections;
}
```

Listing 6.7: Manipulating hierarchies: example 2

```
SELECT

  {[Measures].[sales], [Meausres].[costs]} on COLUMNS,
  {[Date].[Time Hierarchy].[June-2001], [Date].[Time
     Hierarchy].[July-2001]} on ROWS

FROM SampleCube;
```

Listing 6.8: MDX query corresponding to the query in Listing 6.7

6.2.3 Hierarchy Example 3

In this example, we want to find data for older customers (older than 65 years) from the three California cities San Diego, Los Angeles, and San Francisco who purchased products in the first half of 2007.

In NOX

The NOX version of the query is given in Listing 6.9, where three path objects are created corresponding to the three cities **San Diego**, **San Francisco** and **Los Angeles** (at the third level of the geographic hierarchy) that are members of the state of **California** . **California** (at the second level of the hierarchy) is a member of the **USA** which is a member at the first level of the hierarchy.

In MDX

The MDX version of the query is given in Listing 6.10. The three geographic hierarchy paths to California cities "San Diego", "San Francisco" and "Los Angeles" are declared as calculated members in the Customer dimension using the WITH MEMBER statement in MDX.

6.2.4 Hierarchy Example 4

In this example, we want to display the values of the **costs** measure for each of the six years from 1996 to 2001.

```
public boolean select(){
  DateDimension date = new DateDimension();
  Customer customer = new Customer();

  GeographicHierarchy hierarchy =
     customer.getGeographicHierarchy();

  OlapPath pathSanDiego =
           new OlapPath(``USA'',``California'',``San Diego'');
  OlapPath pathSanFran =
           new OlapPath(``USA'',``California'',``San
              Francisco'');
  OlapPath pathLosAng =
           new OlapPath(``USA'',``California'',``Los Angeles'');

  return ( customer.getAge() > 65    &&
           hierarchy.includes(pathSanDiego, pathSanFran ,
              pathLosAng)
           &&
           (date.getYear() == 2007   &&  date.getMonth() <= 6)
         );
}
```

Listing 6.9: Manipulating hierarchies: example 3

```
WITH MEMBER [Customer].[USA San Diego] AS
      [Customer].[Geographic Hierarchy].[USA].[California].[San
         Diego]

      MEMBER [Customer].[USA San Francisco] AS
      [Customer].[Geographic Hierarchy].[USA].[California].[San
         Francisco]

      MEMBER [Customer].[USA Los Angeles] AS
      [Customer].[Geographic Hierarchy].[USA].[California].[Los
         Angeles]

SELECT

FILTER({}, (([Date].[Month] <= 6) AND ([Customer].[Age] > 40)) )
    on ROWS

FROM SampleCube;

WHERE ( [Date].[Year].&[2007), [Customer].[USA San Diego],
    [Customer].[USA San Francisco], [Customer].[USA Los Angeles] )
```

Listing 6.10: MDX query corresponding to the query in Listing 6.9

```
public Object[] project() {

  DateDimension date = new DateDimension();
  Measure measure = new Measure();
  TimeHierarchy timeHierarchy = date.getTimeHierachy();

  OlapPath fromYear = new OlapPath("1996"),
           toYear = new OlapPath("2001");

  Object[] projections = {measure.getCosts(),
                          timeHierarchy.rangeList(fromYear,toYear)
                         };

  return projections;

}
```

Listing 6.11: Manipulating hierarchies: example 4

In NOX

Listing 6.11 depicts the OLAP query as a **project** method in NOX. Two path objects are used that specify years 1996 and 2001 which are members of the first level of the time hierarchy. The method **rangeList** of the time hierarchy (inherited from OlapHierarchy) is then used with the two path values passed as its parameters. In this case, values are aggregated and displayed for each of the year members in the range between and including 1996 and 2001 (inclusive).

```
WITH MEMBER [Date].[1996] AS
        [Date].[Time Hierarchy].[1996]

      MEMBER [Date].[2001] AS
        [Date].[Time Hierarchy].[2001]

SELECT

    [Date].[1996]:[Date].[2001] on ROWS
    [Measures].[cost] on COLUMNS

FROM SampleCube;
```

Listing 6.12: MDX query corresponding to the query in Listing 6.11

In MDX

Listing 6.12 depicts the OLAP query in MDX. The two time hierarchy paths for members of the years 1996 and 2001 are declared as calculated members in the Date dimension using the WITH MEMBER statement in MDX. The **:** (colon) operator is used in MDX to specify a range. So the range of years between 1996 and 2001 (inclusive) are expressed as **[Date].[1996]:[Date].[2001]** and displayed on the ROWS axis. The **cost** measure values are displayed on the COLUMNS axis.

6.3 Conclusion

In this chapter, we discussed an environment for defining OLAP queries that directly exploits the dimension hierarchies. Supplemental hierarchy classes were added to the NOX library to permit the implementation of hierarchies in the framework. Examples, given in the last section of this chapter, both in NOX and MDX show how practical and intuitive NOX is, compared to MDX, in defining hierarchy queries. In short, we attempted to emphasize in this chapter the importance of object-oriented facilities of NOX. In particular, extending and reusing (inheriting) existing hierarchy queries are made possible through the OOP features of NOX.

Chapter 7

Parameterization in NOX

Passing parameters to queries at run-time is crucial to relational databases in general. In OLAP, particularly, it is of significant importance to users, where parametrized OLAP queries provide a generic feature that adds to the flexibility of data analysis done on a data warehouse.

In the NOX framework, since the API data model is object-oriented, the dynamic behaviour of its parametrized queries is well-integrated and flexible. Parametrization in NOX provides a channel of communication with the outside world in an intuitive way. Some of the key characteristics of passing parameters in NOX are:

- Values of parameters can be passed in a variety of ways , namely:
  - they can be read from standard input,
  - they can come from a GUI or menu,
  - they can be accepted from other programs in different formats,
  - they are open to more innovative interface methods.

- A query that is written in NOX is designed to accept parameters of different types.
- The order of passing parameters in NOX is not important, adding flexibility to the framework.
- Parameters can be partially specified, so some of the parameters can be assigned statically while others can be assigned dynamically in different invocations of the query class.
- Parameter variable names are not fixed and programmers can define their own naming conventions, making the NOX framework agile.

Having flexible order of passing parameters is important in NOX. The developer can pass the parameters in any order and then the parameters are matched by NOX using the names of the parameters and not their order. Having both static and dynamic invocations of parameters of the query class is also important as it allows the developer to have different invocations combination of parameters. Hence, it allows for more freedom of assigning values to parameters. In this chapter, we present the parametrization of OLAP queries in the NOX framework. Section 7.1 illustrates, using a simple example, passing of parameters in NOX whereas Section 7.2 depicts the algorithm of how parameters are parsed in NOX. Section 7.3 describes The DOM utility used to insert parameters, at run-time, in the XML-string corresponding to the OLAP query. NOX run-time handling of parameters is explained in Section 7.4. In Section 7.5, we present examples of parametrized OLAP queries that demonstrate

NOX's intuitive and flexible usage of parameters. Finally, Section 7.6 compares the functionality of parametrized NOX queries with respect to that of parametrized MDX queries.

7.1 Parameter Parsing in NOX

In a nutshell, a parametrized OLAP query class in NOX is instantiated in the main method of the program by calling its constructor with the values of parameters at run-time. It is easy and practical to pass parameters to queries in NOX this way. In its simplest form, a parametrized query invocation might look like the example depicted in Listing 7.1.

```
myquery = new MainQuery(``Joe'');

myquery.execute();
```

Listing 7.1: Parametrized query invocation

where "Joe" is the parameter value that is passed to the NOX engine at run-time.

We begin with a simple example that illustrates the usage of parameters in NOX. The invocation of the MainQuery using the **execute** method was given in Listing 7.1, where MainQuery is assumed to have a **select** method. The OLAP query class MainQuery describing this `SELECTION` is depicted in Listing 7.2.

```
class MainQuery extends OlapQuery {

  private double parm1;

  public MainQuery(String cubeName1) {
    super(cubeName1);
  }

  public MainQuery(String cubeName1, double parm1) {
    super(cubeName1);
   this.parm1 = parm1;
  }

  public setParm1(double parm1) {
   this.parm1 = parm1;
  }

  boolean select() {

    Customer customer = new Customer();

    return (customer.getAge() > parm1) ;
  }
}
```

Listing 7.2: class MainQuery with parameter parm1

The **select** method returns values for customers whose age is greater than **parm1**, a parameter passed to the query at run-time. A parameter is detected by the NOX parser when a NameNode (which is one kind of node in the parse tree) is matched to a variable that does not have a value. To illustrate this idea, recall from Section 5.2.2

the leaf with type NameNode found in the parse tree depicted in Figure 28. In this case, NameNode is assigned the variable name **parm1** where **parm1** is not assigned a value until run-time. Assigning a value at compile-time does not generate an error since the standard parsing happens as in Section 5.2.2. Of course, the programmer is responsible for any compile-time and run-time errors that are produced with the program. In NOX, we are not adding any new compile-time or run-time parameter checking. The programmer is responsible for error catching and the resolution of errors.

In the NOX framework, parameters should be declared as private fields in the OLAP query class, and then may be used in any of its OLAP operation methods. Of course, other fields can be declared as private by the programmer. Field names are defined by the programmer; no specific naming conventions are imposed by NOX. This adds flexibility but at the same time adds responsibility on the programmer side to have readable well-defined queries. In the example, **parm1** is a parameter declared as a private field in the **MainQuery** class and then used in the **select** method of the class. Note the constructor **public MainQuery(double parm1)** that is used to assign a value to **parm1** while creating a **MainQuery** OLAP query instance. Having this kind of data encapsulation feature is an example of the strength of expressing the OLAP query in an object-oriented manner.

A second way to pass parameters is by using the parameter(s) `setter` method(s), in this example, the **public setParm1(double parm1)** method, defined by the programmer.

7.2 Parameter Parsing Pseudocode

As explained in Chapter 4, when an input Java file is parsed with the NOX parser, an XML string is generated that corresponds to each query method in the file. In the case of parameters, each parameter variable is read and detected as a parameter by the parser. The parameter is then added to the XML query with a leading special flag "?". Pseudocode for parameter parsing is presented in Listing 7.3.

```
1. Input Java source file
2. Detect the OLAP query by NOX parser
3. Parse the OLAP query using NOX parser and
   create parse tree.
4. Detect each parameter by:
   a. Finding NameNode leaf
   b. Checking if the NameNode variable name (parameter name in
      case of parameters) is not assigned a value.
5. Create the XML string corresponding to the query.
6. Add each detected parameter to the XML string with a leading
   special flag ``?''.
```

Listing 7.3: Parameters parsing pseudocode

To illustrate this idea, an example is presented in Listing 7.4, where the XML string that corresponds to the query in Listing 7.2 includes the parameter **parm1**. The leading flag "?" is added by the parser before the parameter **parm1** in the XML string. Note here that any parameter variable name can be used by the programmer and the parser will detect it, which adds to the agility of the NOX framework.

```
<QUERY>
 <DATA_QUERY>
  <CUBE_NAME> SampleCube </CUBE_NAME>
  <OPERATION_LIST>
    <OPERATION>
     <SELECTION>
      <DIMENSION_MEASURE_LIST>
       <DIMENSION>
        <DIMENSION_NAME>
            Customer
        </DIMENSION_NAME>
        <EXPRESSION>
         <RELATIONAL_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>
                    ?parm1
               </CONSTANT>
            </EXP_VALUE>
          </SIMPLE_EXP>
         </RELATIONAL_EXP>
        </EXPRESSION>
       </DIMENSION>
```

```
      </DIMENSION_MEASURE_LIST>
    </SELECTION>
  </OPERATION>
 </OPERATION_LIST>
</DATA_QUERY>
</QUERY>
```

Listing 7.4: XML corresponding to the query with parameter parm1

7.3 Parameter Insertion DOM Utility

In this section, we describe a utility function, named **XMLparametersInsert**, that is responsible for inserting parameter values in an XML document. This utility is part of the NOX DOM utilities library that provides additional methods for manipulating the DOM XML objects. Pseudocode for the algorithm of the **XMLparametersInsert** utility is depicted in Listing 7.5. Here, **XMLparametersInsert** is defined as a function that inputs an XML string with some flagged parameters and outputs the XML string with the parameters replaced by their values. The details of generating the intermediate DOM tree representation of the XML string are not included to simplify the readability of the pseudocode.

The algorithm of **XMLparametersInsert** scans the input XML string for parameters. For each located parameter in the XML string, it will match this parameter to the corresponding field in the class. It will do that by comparing the parameter name sequentially to each field name in the list of class field names until the parameter is found. Then, it will replace the parameter with the value of the field found.

The last step of the algorithm is to return the updated XML string. Note that the names of the OLAP Query object at run-time can be extracted directly by run-time methods that already exist in Java. Using these methods, the fields are returned one by one and their names compared to the parameters names the parser is looking for.

```
Function XMLparametersInsert
Input:  XML string with flagged parameters , OlapQuery object
   with its declared fields
Output: XML string with parameters replaced by their values

REPEAT
 Check if parameter p exists in XMLstring by locating the special
    leading character '?'
 IF (p exists) THEN
  FOR each field f in the current OlapQuery object
     IF (f and p are the same variable name) THEN
       Replace ?p in XMLstring with the value of f
     END IF
  END FOR
 ELSE  // no more parameters in XMLstring
  Exit REPEAT loop
UNTIL no more parameters in XMLstring

RETURN the updated XMLstring
```

Listing 7.5: XMLparametersInsert pseudocode

7.4 Run-time Parameter Handling

As described in Section 7.1, an XML string is constructed for each query class during the parsing phase. If the query contains some parameters, these parameters will be marked by a leading flag "?" in the XML string. Parameters that are input to the program at run-time will also be passed to the OLAP query object at run-time, hence the XML query string needs to be updated before it is sent to the server. For this purpose, the DOM utility function **XMLparametersInsert** presented in Section 7.3 is used, a method that takes the XML query string (with parameters) as input and outputs the XML query string with the parameters replaced with their associated values.

From Chapter 4, we know that during parse-time, an intermediate output Java file, that is transparent to the programmer, is produced. Moreover, the "special" methods in each OLAP query class, with reserved names for OLAP querying operations such as **project()** and **select()**, are replaced with the **execute()** method that contains the corresponding XML string. We add a call to the **XMLparametersInsert** utility function to the body of the **execute()** method in order to detect if parameters exist. And, if they do exist, the NOX parser will replace them at run-time with their values. The call to **XMLparametersInsert** will always be part of the body of **execute()**. Remember that the **execute()** method also contains function calls to pass the resulting XML string to the server. Listing 7.6 shows the intermediate Java file that is produced by the NOX parser when processing the OLAP query (with parameters) of

Listing 7.2. The **execute()** method returns an empty cube, as it is only visualized as an object in memory by the programmer. It actually contains no data as the real result data cube is sent from the server after the OLAP query is resolved.

```
class MainQuery extends OlapQuery {

  private double parm1;

  public MainQuery(String cubeName1) {
    super(cubeName1);
  }

  public MainQuery(Strign cubeName1, double parm1) {
    super(cubeName1)
   this.parm1 = parm1;
  }

  public setParm1(double parm1) {
   this.parm1 = parm1;
  }

  double getParm1() {
    return this.parm1;
  }

  public Cube execute () {
    String xmlQuery =
      ''<QUERY> <DATA_QUERY> <CUBE_NAME> sample </CUBE_NAME>'' +
      ''<OPERATION_LIST> <OPERATION> <SELECTION>'' +
      ''<DIMENSION_MEASURE_LIST> <DIMENSION> <DIMENSION_NAME>'' +
      ''Customer </DIMENSION_NAME> <EXPRESSION>'' +
      ''<RELATIONAL_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> ?parm1 </CONSTANT>'' +
              .
              .
              .

     DOMutilities dom;
     xmlQuery = dom.XMLparametersInsert(xmlQuery);

     Communicator comm = new Communicator();
     comm.sendQuery(xmlQuery);
     return new Cube();
   }
}
```

Listing 7.6: Intermediate Java file with execute() method

7.5 NOX Parametrization in Practice

In this section, we provide parametrized OLAP query examples to demonstrate the flexible and simple usage of parameters in the NOX language. In the OLAP query in Listing 7.7, two parameters **parm1** and **parm2** are passed to the program while instantiating the query object. The parameters' values can be statically or dynamically set. Observe the usage of parameters in this example, where one parameter value is compared to the count measure, while the other parameter value is used as a lower bound on the range of IDs of customers' values returned. As noted earlier, error-checking does not occur in the query itself as it would be quite difficult to analyze arbitrary code. Programmers are responsible to do error-checking themselves in

the main program or the calling function. In this example, if **parm2** is passed with the value 10, some kind of error should be generated by the programmer.

```
class ExampleQuery2 extends OlapQuery {

  private double parm1;
  private int parm2;

  public MainQuery(String cubeName1) {
    super(cubeName1);
  }

  public MainQuery(Strign cubeName1, double parm1, int parm2) {
    super(cubeName1)
   this.parm1 = parm1;
   this.parm2 = parm2;
  }

  public setParm1(double parm1) {
   this.parm1 = parm1;
  }

  double getParm1() {
    return this.parm1;
  }

  public setParm2(int parm2) {
   this.parm2 = parm2;
  }

  int getParm2() {
    return this.parm2;
  }
```

```
 boolean select() {
  Customer customer = new Customer();
  Measure measure = new Measure();
  OlapProperty custID = new OlapProperty(customer.getID());
   return (measure.getCount() > parm1 && custID.inRange(parm2,9))
       ;
  }
}
```

Listing 7.7: class ExampleQuery2 with two parameters

Another example is given in Listing 7.8, where in addition to the two parameters of the previous example, two other parameters **parm3** and **parm4** are used. In the query, **parm3** is compared to the age of customers so that values for customers older than **parm3** are returned, with **parm4** being used as a parameter to hold the member value of an OLAP path level, namely the country member value of the geographic hierarchy of the customer dimension.

```
class ExampleQuery3 extends OlapQuery {

  private String parm1;
  private double parm2,parm3;
  private String parm4;

  public MainQuery(String cubeName1) {
    super(cubeName1);
  }

  public MainQuery(String cubeName1, String parm1, double parm2,
     double parm3, String parm4) {
    super(cubeName1)
```

```
    this.parm1 = parm1;
    this.parm2 = parm2;
    this.parm3 = parm3;
    this.parm4 = parm4;
  }

  public setParm1(String parm1) {
   this.parm1 = parm1;
  }

  String getParm1() {
    return this.parm1;
  }

  public setParm2(double parm2) {
   this.parm2 = parm2;
  }

  double getParm2() {
    return this.parm2;
  }

  public setParm3(double parm3) {
   this.parm3 = parm3;
  }

  double getParm3() {
    return this.parm3;
  }

  public setParm4(String parm4) {
   this.parm4 = parm4;
  }

  String getParm4() {
    return this.parm4;
```

```
  }

  boolean select() {
  Customer customer = new Customer();
  DateDimension date = new DateDimension();

  GeographicHierarchy hierarchy =
     customer.getGeographicHierarchy() ;
  Measure measure = new Measure();

  OlapProperty custID = new OlapProperty(customer.getID());
  OlapProperty dateMonth = new OlapProperty(date.getMonth());

  return (measure.getCount() > parm2 && custID.inRange(parm1,9)
           && customer.getAge() > parm3



           && hierarchy.includes(new OlapPath("USA"), new
              OlapPath(parm4))
           && ( date.getYear()  ==  2007 &&
              dateMonth.inRange(1,5))) ;
 }
}
```

Listing 7.8: class ExampleQuery3 with four parameters

7.6 Parametrized NOX Queries versus Parametrized MDX Queries

In this section, we compare parametrized NOX queries to parametrized MDX queries. We first discuss how MDX passes parameters to its queries. Then, we compare the

parametrization used in MDX queries to that used in NOX queries.

MDX distinguishes parameters from other constructs in its queries by prefixing each parameter name with the at sign (@). An example of a parametrized MDX query in XML for Analysis (XMLA) is presented in Listing 7.9, where the @CountryName is a parameter whose value will be retrieved at run-time [MSD]. Note the awkward way of passing parameters through the use of XML.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <Execute xmlns="urn:schemas-microsoft-com:xml-analysis">
      <Command>
        <Statement>
      select [Measures].members on 0,
          Filter(Customer.[Customer Geography].Country.members,
                Customer.[Customer Geography].CurrentMember.Name =
                @CountryName) on 1
      from [Adventure Works]
    </Statement>
      </Command>
      <Properties />
      <Parameters>
        <Parameter>
          <Name>CountryName</Name>
          <Value>'United Kingdom'</Value>
        </Parameter>
      </Parameters>
    </Execute>
  </Body>
</Envelope>
```

Listing 7.9: Parametrized MDX query example [MSD]

Since its creation, MDX has been augmented and/or modified in an attempt to keep pace with the expanding OLAP domain. Nevertheless, its inherently rigid string-based nature makes it hard to adapt to the continuously evolving programming languages in the industry. Table 2 compares the quality of OLAP query representation in NOX to the quality of their representation in MDX. In terms of the first point in the table, for example, we have already presented a parametrized MDX query in Listing 7.9 in which the parameter is named @CountryName. By contrast, the simple naming policy in NOX allows allows pragrammers to use the variable name that best suits the application.

With respect to the third point, we note that MDX syntax is dependent on its embedding application. Two examples depict this point, as follows:

- When parametrized MDX queries are used with OLE DB, the ICommandWithParameters interface should be utilized.
- When parametrized MDX queries are used with ADOMD.NET, the AdomdCommand.Parameters collection should be employed.

For ADOMD.NET, the parameter is assigned as in Listing 7.10, where conn is the ADOMD connection.

```
AdomdCommand cmd = new AdomdCommand(MDX, conn);
cmd.Parameters.Add("Param1", "abcde");
```

Listing 7.10: Parameter assignment using ADOMD

Parametrization in NOX Queries	**Parametrization in MDX Queries**
Same representation of parameters as any other Java variable name	Awkward representation of parameters by prefixing the name with the at sign (@)
Parameters are type-checked at compile-time	Error-prone as most errors are discovered at run-time
No additional libraries are needed to pass parameters to NOX queries	Dependant on the application where the MDX query string is embedded.
NOX is object-oriented and using parameters is straightforward	In practice, real world programmers report that parametrization is not well-developed in MDX
NOX has a good foundation for parametrization which makes extending parameters relatively simple	The awkwardness of MDX makes parametrization hard to extend and difficult to maintain

Table 2: Parametrized NOX Queries versus Parametrized MDX Queries

Listing 7.11 depicts how parameters are passed using the ADOMD client. It should be relatively obvious that this model is not programmer friendly. As for the OLE DB, programmers often complain that OLE DB does not work at all with parameters in MDX [Mic]. They suggest that Microsoft is not maintaining MDX parametrization well.

```
//using Microsoft.AnalysisServices.AdomdClient;

string MDX = "with member [Measures].[Test] as Str(@Param1) "
+ "SELECT [Measures].[Test] on 0, "
+ "[Product].[Category].[Category].Members on 1 "
+ "from [Adventure Works]";

AdomdConnection conn = new
    AdomdConnection("Provider=MSOLAP.3;Data Source=localhost;
            Initial Catalog=Adventure Works DW;
            Integrated Security=SSPI;Persist Security Info=false;");

conn.Open();
AdomdCommand cmd = new AdomdCommand(MDX, conn);
cmd.Parameters.Add("Param1", "abcde");

System.Data.DataSet ds = new System.Data.DataSet();
AdomdDataAdapter adp = new AdomdDataAdapter(cmd);
adp.Fill(ds);
Console.WriteLine(ds.Tables[0].Rows[0][1]);
conn.Close();
```

Listing 7.11: Parametrized MDX query using ADOMD [Mic]

7.7 Conclusion

Parametrization in NOX facilitates the dynamic customization of OLAP queries at run-time. These parametrized queries provide programmer-friendly interaction with different entities in the system. As we have illustrated in this chapter, passing parameters to NOX queries is done in a simple and straightforward way. While passing parameters to string-based queries in the MDX language is awkward and error-prone, passing parameters to the object-oriented queries in the NOX language is high-level and flexible.

Chapter 8

The NOX Language Expressiveness

We have demonstrated extensively in the previous chapters of this thesis, using the Java prototype developed for this research, the practicality of the NOX model. It should be clear that this model can be extended by DBMS developers to further develop the prototype into a native language OLAP tool. That being said, the advantages are of little practical value if one cannot demonstrate that the proposed approach is capable of representing the range of query patterns developers have come to expect in the OLAP domain.

Being a descendant of SQL, MDX suffers from the limitations of the underlying SQL-like `SELECT-FROM-WHERE` format. Moreover, MDX, as an industrial language supported by Microsoft, did not receive the formal research focus that typically leads to more powerful and flexible programming languages. As a result, although MDX has a grammar, it does not have a formal algebra. On the other hand, NOX has a well-structured algebra that supports operations done in OLAP. As such, in the context of multi-dimensional systems, NOX has the potential to be more intuitive

compared to MDX.

In this chapter, we examine the NOX model from an algebraic perspective, and compare its expressiveness to that of MDX, the de-facto standard query language in this domain. We do so as follows:

- Demonstrating the correspondence between MDX and NOX and analyzing the associated grammars in terms of the core `SELECTION` and `PROJECTION` operations
- Identifying a small set of query forms representative of the two operations and providing concrete instantiations in both MDX and NOX

This approach grounds the research and shows how NOX provides intuitive query functionality while concurrently minimizing the constraints of MDX.

Because most operations in OLAP are a combination of selections and projections, we will focus exclusively on these operations and compare their algebraic formal representation in MDX and NOX. In future work, we hope to extend the same logic of thinking to other OLAP operations like Change Level and Change Base, which are actually executed against result sets that are typically much smaller compared to the disk-based warehouse database. Drill-Across is much less common than is the case in OLTP settings and will also be part of future work.

In this chapter, the grammatical structure of MDX and NOX are depicted in Section 8.1. Section 8.2 illustrates the correspondence of OLAP `SELECTION` between the MDX language and the NOX language, whereas Section 8.3 illustrates the correspondence for the OLAP `PROJECTION` operation. We note at the outset that only the

relevant parts of the grammars are given in order to explain the logic. For complete listings of the MDX and NOX grammar production rules, refer to Appendix D and Appendix E respectively.

8.1 Grammatical Structure

As presented in Chapter 4, the NOX framework is implemented in Java. The intermediate XML-based representation of its queries, based upon the NOX algebra, is generated as part of the re-writing process. The MDX and NOX syntax grammars are quite different syntactically. Listing 8.1 depicts the canonical MDX query format. As explained in Chapter 5, the `SELECT` clause includes the axis_specification that defines the data cube axis where features/measures are displayed/returned. The cube name is specified in the FROM clause. The cube cells selection constraints are given in the WHERE clause known as *slicer_specification*. We will only address the use of axis and slicer specifications in this chapter. We note the use of the optional *cell_props* that are attributes that may be useful for the presentation of data. While they might be useful in that specific context, cell properties are not directly associated with the algebraic operations and will be ignored as other display-related MDX language extensions in this chapter.

Listing 8.2 provides an abbreviated representation of the grammar associated with NOX processing. The complete NOX Grammar is given in Appendix E. Here, we can see that a query is formulated as a cube name, which is the equivalent of the MDX FROM clause. `SELECTION` and `PROJECTION` are two of the algebraic operations that

are listed in the <operation> tag. Note that while MDX queries are actually written in the syntax of Listing 5.5, the NOX grammar is purely an internal representation and is never encoded by the programmer.

```
<select_statement> ::= [WITH <formula_specification>]

                       SELECT [<axis_specification>
                       [, <axis_specification>]*]

                       FROM [<cube_specification>]

                       [WHERE [<slicer_specification>]]
                       [<cell_props>]
```

Listing 8.1: MDX `SELECT` statement

```
<query> ::= <data_query>
          | <meta_query>

<data_query> ::= <cube_name>
                 [, <operation_list>] [, <function_list>]

<operation_list> ::= <operation> [, <operation>]*

<operation> ::= <selection>
              | <projection> | ...
```

Listing 8.2: Top level NOX grammar

In demonstrating the correspondence between the MDX production rules and the

NOX production rules, three primary data types of MDX are mapped to NOX: members of dimensions/hierarchies, tuples, and sets [Nol99]. We focus on these types because the other data types of MDX, namely the scalar, dimension/hierarchy and level, are transparently mapped to NOX during the process of mapping the three primary types between MDX and NOX.

8.2 OLAP SELECTION

The OLAP SELECTION operation refers to the specification of values for some or all of the dimensions of the multi-dimensional data cube. It results in a subcube. In other words, it provides rules and constraints against the cube that restrict and isolate the values requested in the final result. In the industry, it is mapped to slicing and dicing operations. In this section, we demonstrate the correspondence of the production rules of the OLAP SELECTION operation between MDX and NOX. We start with presenting the MDX production rules and their main programming constructs in Subsection 8.2.1. Then, we map NOX production rules and their programming constructs to those of MDX in Subsection 8.2.2. Subsection 8.2.3 includes three parts that evaluate the grammars by identifying three sets of core query forms that represent common query patterns.

8.2.1 SELECTION Production Rules in MDX

As is the case with SQL queries, SELECTION is one of the basic operations in OLAP. MDX supports slicing and dicing through the syntax of the WHERE clause in the

<**select_statement**> production rule. Listing 8.3 provides the producrion rules for the MDX <**slicer_specification**>.

```
[WHERE [<slicer_specification>]]

<slicer_specification> ::= {<set> | <tuple>}

<tuple> ::= <member>
          | (<member> [, <member>]*)
          | <tuple_value_expression>
Note: Each member must be from a different dimension
or from a different hierarchy

<set> ::= <member>:<member>
        | <set_value_expression>
        | <open_brace>[<set>|<tuple>
          [, <set>|<tuple>]*]<close_brace>
        | (<set>)
Note: Each member must be from the same hierarchy
and the same level.

<tuple_value_expression> ::= <set>.CURRENTMEMBER
            | <set>[.ITEM]({<string_value_expression>
              [,<string_value_expression>]*}
              | <index>)

<set_value_expression> ::= <dim_hier>.MEMBERS
                         | <level>.MEMBERS
                         | <member>.CHILDREN
                         | ...

<cube_name> ::= [ [ [ <data_source>.] <catalog_name>.]
                      [<schema_name>.]<identifier>
```

```
<data_source> ::= <identifier>

<catalog_name> ::= <identifier>

<schema_name> ::= <identifier>

<dim_hier> ::= [<cube_name>.]<dimension_name>
             | [[<cube_name>.]< dimension_name>.]<hierarchy_name>

<dimension_name> ::= <identifier>
                   | <member>.DIMENSION
                   | <level>.DIMENSION
                   | <hierarchy>.DIMENSION

<dimension> ::= <dimension_name>

<hierarchy> ::= <hierarchy_name>

<hierarchy_name> ::= <identifier>
                   | < member>.HIERARCHY
                   | <level>.HIERARCHY

<level> ::= [<dim_hier>.]< identifier>
          | <dim_hier>.LEVELS(<index>)
          | <member>.LEVEL

<member> ::= [<level>.]<identifier>
           | <dim_hier>.<identifier>
           | <member>.<identifier>
           | <member_value_expression>


<member_value_expression> ::= <member>.{PARENT | FIRSTCHILD
                                        | LASTCHILD | PREVMEMBER
                                        | NEXTMEMBER    | ... }
```

```
<open_brace> ::= {
<close_brace> ::= }

<open_bracket> ::= [
<close_bracket> ::= ]

<underscore> ::= _

<alpha_char> ::= a | b | c | ...| z | A | B | C | ... | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Listing 8.3: Production rules for the MDX WHERE clause

Adding a "where <slicer_specification>" in MDX does not change what is returned on Rows or Columns in the query; it changes the values returned for each cell. A <slicer_specification> is either a set or a tuple, as shown in the production rules in Listing 8.3. NOX constructs are mapped to MDX constructs in their basic forms of the most important operations, namely `SELECTION` and `PROJECTION`. This is demonstrated in the next subsection. NOX can be easily extended to include the other operations of MDX. The following are the basic constructs in MDX (that will be mapped to NOX):

- A tuple can be a member of a dimension or a hierarchy. It can also be a number of members from different dimensions or hierarchies specified between parenthesis, "(" and ")", and separated by commas. The <tuple_value_expression> grammar rule contains different kinds of productions for building expressions

that result in a tuple value. The <tuple_value_expression> expressions are not implemented in NOX yet and will be added to future work.

- A set is a collection of tuples. It can be a range of members from the same hierarchy and the same level specified by the range's first member and its last member. It can also be a number of members from the same hierarchy and the same level specified between braces, "{" and "}", and separated by commas. Production rules corresponding to the <set_value_expression> MDX production rule will be added to NOX as future work.

- A member is specified, in its basic form, as an identifier in a dimension or an identifier in a level in a hierarchy of a dimension. This identifier corresponds to an attribute name. The other production rules for the <member_value_expression> build expressions that result in a member value. The <member_value_expression> form of expressions are not implemented in the NOX language yet and will be part of the future work.

8.2.2 Mapping the `SELECTION` Production Rules between MDX and NOX

NOX supports the `SELECTION` operation through the syntax of the `SELECT` method defined in the NOX language by the <selection> production rule in the NOX grammar. Listing 4.1 (from Chapter 4) illustrates the corresponding grammar for the NOX `SELECTION` operation.

In the NOX grammar, the <selection> production rule depicts the same functionality as the <slicer_specification> rule of the MDX grammar. From a high level, one can see that a `SELECTION` is a list of dimensions specification, each consisting of a combination of expressions (relational, arithmetic, etc.) and optionally including hierarchical elements or attributes. More specifically, the <selection> is a <dimension_measure_list>, where the <dimension_measure_list> has at least one condition on some member in a dimension or in a hierarchy of a dimension and can be combined by logical operators with other conditions on members or on measures. The translation from MDX to NOX of the three basic constructs described in Subsection 8.2.1 goes as follows:

- The logical operator <logical_op> "AND" in the NOX grammar, that is used to aggregate values corresponding to multiple members from different hierarchies or dimensions, translates to a tuple in the MDX grammar.

- The logical operator <logical_op> "OR" in the NOX grammar, that is used to aggregate values corresponding to multiple members from the same hierarchy and the same level, translates to a set in the MDX grammar.

- As for a member in the NOX grammar, it is defined by its dimension name and an expression. The dimension name is simply an identifier that describes the name of the dimension. An expression is at least one conditional expression where two simple expressions are compared to each other. The conditional

expression can be combined by logical operators with other conditional expressions. The logical operators have the same functionality as in the <dimension_measure_list> production rule described earlier. A simple expression can be a mathematical expression. It can also be an identifier that corresponds to the name of the attribute in the dimension (specified earlier in the <dimension_name> tag). In addition, it can be a hierarchy list <hierarchy_list> or a function list <function_list>.

The grammar rules for the <hierarchy_list> are shown in Listing 8.4. We can have conditions on one or more hierarchies that belong to the dimension specified by the value of the <dimension_name> tag. Each hierarchy is defined by its name, its operator and one or more OLAP hierarchy paths, where:

1. A hierarchy OLAP path is specified by listing, at each level in the hierarchy, the member name. For example, we have a geographic hierarchy where the first level is country and the second level is state. To define the hierarchy OLAP path that corresponds to California, the member at the first level will be "United States" and the member at the second level will be "California". Its hierarchy diagram is the same as the diagram in Figure 18 of Chapter 6. Its XML representation as described by the NOX grammar is the following:

```
<olap_path_list>
   <olap_path>
     <value> United States </value>
```

```
      <value> California </value>
    </olap_path>
  </olap_path_list>
```

2. A hierarchy name is an identifier that describes the hierarchy.

3. A hierarchy operator can be "inRange" or "inList". When "inRange" is used, we have to specify two OLAP paths that correspond to two members in the same level in the hierarchy. Then, it implicitly aggregates values for all members between, and inclusive of, the two members specified. This is like a set in the MDX grammar where the production rule is <set> ::= <member>:<member>. When "inList" is used as the hierarchy operator, we have to specify one or more OLAP path values that correspond(s) to one or more members. These members belong to the same level in the hierarchy and the values returned are aggregated together.

The combination of the above mappings is similar to a production rule for a set in the MDX grammar,

```
<set> ::= <open_brace>[<set>|<tuple> [, <set>|<tuple>...]]<close_brace>,
```

where <set> and <tuple> are already matched to NOX and arbitrary combinations of them may be used.

```
<hierarchy_list> ::= <hierarchy>+

<hierarchy> ::= <hierarchy_name> , <hierarchy_op> ,
   <olap_path_list>

<hierarchy_name> ::= #PCDATA

<hierarchy_op> ::= #PCDATA

<olap_path_list> ::= <olap_path>

<olap_path> ::= <value>+

<value> ::= #PCDATA
```

Listing 8.4: Grammar rules for the "Hierarchy List"

8.2.3 `SELECTION` Constraints

As mentioned previously, the OLAP query `SELECTION` operation provides constraints that restrict the values requested. Given the grammars described above, we now turn to the evaluation itself. Because database queries are by definition open ended, there are specific patterns for the most common queries performed. For `SELECTION`, we can identify and categorize three types of constraints that cover different levels of complexity. Combinations of the three can be used, of course, to produce queries of arbitrary complexity.

1. Single dimension constraint

2. Multiple dimension constraint (open-ended)

3. Multiple members from a single dimension hierarchy

In the next three sections, we will analyze each category of constraints and represent the associated queries in MDX and NOX forms.

Single Dimension Constraint

We begin with a single constraint on a single dimension or hierarchy. In this example, we are interested in returning the value of the "Internet Sales Amount" measure for all calendar years but only for customers who live in the United States. The `SELECTION` then is specified as:

$\sigma_{(Country='UnitedStates')}(\text{Sales})$.

The MDX version of the query is given in Listing 8.5. Here, the query is sliced so that aggregated values for the member "United States" of the fully qualified path name in the "Customer Geography" hierarchy are returned. Figure 18 of Chapter 6 shows the path to "United States" in the hierarchy.

```
SELECT {[Measures].[Internet Sales Amount]} ON COLUMNS,
       [Date].[Calendar Year].MEMBERS ON ROWS
FROM   [Adventure Works]
WHERE ([Customer].[Customer Geography].[Country].[United States])
```

Listing 8.5: MDX query returning values for customers living in the United States

Listing 8.6 illustrates the NOX version of the query. More specifically, it shows

the corresponsing `SELECT` method. The developer has instantiated a new Customer and its corresponding GeographicHierarchy. After getting the hierarchy object, its method "includes(OlapPath ... path)" is called with the argument that is an object of type "OlapPath("United States")" for customers who live in the United States. Hence, the return statement identifies those cells whose hierarchy paths include the "United States". Note that the display attributed `Date` and `Internet Sales` are not part of the `SELECTION` specification. They are associated with the `PROJECTION` that will be discussed in the next section.

```
class selectQuery1 extends OlapQuery {

boolean select() {

  Customer customer = new Customer();
  CustomerHierarchy geohierarchy =
     customer.getGeographicHierarchy();

  return (geohierarchy.includes(new OlapPath("United States")));
}

}
```

Listing 8.6: NOX query returning values for customers living in the United States

For completeness, the NOX XML description of the hierarchy used in the return statement of the `SELECT` method of the query in Listing 8.6 is depicted in Listing 8.7.

```
<DIMENSION>
<DIMENSION_NAME>
Customer
</DIMENSION_NAME><EXPRESSION>
<SIMPLE_EXP>
<EXP_VALUE>
<HIERARCHY_LIST>
<HIERARCHY>
<HIERARCHY_NAME>
GeographicHierarchy
</HIERARCHY_NAME><HIERARCHY_OP>
includes
</HIERARCHY_OP><OLAP_PATH_LIST>
<OLAP_PATH>
<VALUE>
United States
</VALUE>
</OLAP_PATH>
</OLAP_PATH_LIST>
</HIERARCHY>
</HIERARCHY_LIST>
</EXP_VALUE>
</SIMPLE_EXP>
</EXPRESSION>
</DIMENSION>
```

Listing 8.7: XML description of the hierarchy used in the return statement of the **select** method of the query in Listing 8.6

Multiple Dimension Constraints

We turn now to the case in which multiple dimension constraints are defined. We extend our previous query by ensuring that only totals associated with the Auto

Product category are included. Formally, the query is defined as:

$\sigma_{(Country=`UnitedStates' \ \&\&Category=`1')}(\text{Sales})$.

Listing 8.8 gives the MDX version of the query. In the WHERE clause of this query, in addition to the values returned for "United States" as explained in the previous query, aggregated values for "Category" key 1 of the "Product" dimension are also returned. Here, MDX uses the operator "&" to refer to the member that is the key in the "Product" dimension. The values aggregated together are then returned by the query.

```
SELECT {[Measures].[Internet Sales Amount]} ON COLUMNS,
       [Date].[Calendar Year].MEMBERS ON ROWS
FROM   [Adventure Works]
WHERE ([Customer].[Customer Geography].[Country].[United States],
       [Product].[Category].&[1])
```

Listing 8.8: MDX query returning values for customers living in the United States and who bought products in "Category"with key 1

Listing 8.9 depicts the NOX version of the `SELECTION` operation of the query. The tuple in MDX is expressed as an **&&** (AND operator) of two conditions in NOX. So, the first condition of returning values for customers in the United States is conjuncted with the condition of having product category equals to one. "product" is instantiated from the class "Product", its method "getCategory()" is called and the value returned is compared to one. In both cases of MDX and NOX, there is no limit on the number

of dimensions used in the specification.

```
class selectQuery2 extends OlapQuery {

boolean select() {

  Customer customer = new Customer();
  Product product = new Product();
  CustomerHierarchy geohierarchy =
     customer.getGeographicHierarchy();

  return (geohierarchy.includes(new OlapPath("United States"))
          && product.getCategory() == '1');
}

}
```

Listing 8.9: NOX query returning values for customers living in the United States and who bought products in "Category" with key 1

Multiple members from a single dimension hierarchy

Finally, we address the somewhat more complex case in which different members of the same hierarchy are required. We want to show the value of the "Internet Sales" for all calendar years for customers who bought products in the Auto category and live in either the United States or the United Kingdom. Formally, we have:

$$\sigma_{((Country='UnitedStates' \;||\; Country='UnitedKingdom')\&\&\; Category='1')}(\text{Sales}).$$

Listing 8.10 shows how this would be done with MDX. Here, we need to include a set in the WHERE clause to return the logical disjunction of its members. The WHERE

clause implicitly aggregates values for all members in the set. For example, the above query shows aggregated values for the United States and the United Kingdom in each cell.

```
SELECT {[Measures].[Internet Sales Amount]} ON COLUMNS,
       [Date].[Calendar Year].MEMBERS ON ROWS
FROM [Adventure Works]
WHERE ({[Customer].[Customer Geography].[Country].[United States],
       [Customer].[Customer Geography].[Country].[United
           Kingdom]},
       [Product].[Category].&[1])
```

Listing 8.10: MDX query returning values for customers living in the United States or the United Kingdom and who bought products in "Category" with key 1

Listing 8.11 shows the equivalence in NOX of the WHERE clause of the MDX query. Obviously, it is simpler than the MDX query, at least from a readability perspective. In addition to the conditions in the previous examples, another condition on the value of the first level of the geographic hierarchy of the customer is given. So, the values returned are aggregated for customers living in the United States and customers living in the United Kingdom. This is translated to an OR operator (||) in NOX or to the method "includes(OlapPath ... path)" when more than one hierarchy paths are included and values belonging to customers in any of the hierarchies are aggregated and returned.

```
class selectQuery3 extends OlapQuery {

boolean select() {

  Customer customer = new Customer();
  Product product = new Product();
  CustomerHierarchy geohierarchy =
     customer.getGeographicHierarchy();

  return (geohierarchy.includes(new OlapPath("United States"),
                                new OlapPath("United Kingdom"))
          && (product.getCategory() == '1')) ;
}

}
```

Listing 8.11: NOX query returning values for customers living in the United States or the United Kingdom and who bought products in "Category" with key 1

Note that in our NOX Java prototype, we use **&&** (AND operator) to aggregate values corresponding to multiple members from different hierarchies or dimensions. We use || (OR operator), or we use a keyword (includes, inRange) that translates to an OR operator, to aggregate values corresponding to multiple members from the same hierarchy and the same level.

8.3 OLAP PROJECTION

While SELECTION provides dimension constraints, the purpose of an OLAP PROJECTION is to identify display attributes, including measures and features. In this section,

we show that there are production rules in the NOX grammar of the Java OLAP querying language that are equivalent to some of the core MDX grammar production rules that correspond to the <axis_specification> part of the <select_statement> in MDX, hence, demonstrating the correspondence of OLAP `PROJECTION` between MDX and NOX. Subsection 8.3.1 explains the `PROJECTION` production rules in MDX, then Subsection 8.3.2 provides the mapping of `PROJECTION` production rules between MDX and NOX and illustrates by example the main constructs used in the grammar. The last subsection 8.3.3 includes three parts that cover the most common `projection classes` based on OLAP query patterns, similar to what was presented for OLAP `SELECTION` previously.

8.3.1 `PROJECTION` Production Rules in MDX

We show again the syntax of the MDX **select** statement in Listing 8.12.

```
<select_statement> ::= [WITH <formula_specification>]
                       SELECT [<axis_specification>
                       [, <axis_specification>...]]
                       FROM [<cube_specification>]
                       [WHERE [<slicer_specification>]]
                       [<cell_props>]
```

Listing 8.12: MDX `SELECT-FROM-WHERE` syntax

As was the case with the slicer, MDX expects projection criteria to be expressed in one of the three forms of a member, tuple or set. In the case of `PROJECTION` though,

MDX supports these constructs through the syntax of the <axis_specification> in the <**select_statement**>, where the formal syntax of <axis_specification> is as follows:

```
<axis_specification> ::= [NON EMPTY] <set> [<dim_props>] ON <axis_name>
```

Note here that though the <axis_specification> is expressed in terms of a <set> in MDX, it is only one of the production rules of the MDX grammar. Moreover, there is some level of recursion in the MDX Grammar, that makes the MDX grammar itself vague and hard to understand. Listing 8.13 depicts the more detailed production rules of the internal MDX grammar and its recursive style for the <axis_specification>.

Query axes specify the edges of a cellset returned by a Multidimensional Expressions (MDX) SELECT statement. Specifying the edges of a cellset allows the restriction of the returned data that is visible to the client. In MDX, an edge is a set assigned to an axis. To specify query axes, we use the <axis_specification> to assign a set to a particular query axis. Each <axis_specification> value defines one query axis. The number of axes in the dataset is equal to the number of <axis_specification> values in the SELECT statement. Each query axis has a number: zero (0) for the x-axis, 1 for the y-axis, 2 for the z-axis, and so on. In the syntax for the <axis_name> which is part of the right hand side of the <axis_specification> production rule, the <index> value specifies the axis number. An MDX query can support up to 128 specified axes, but very few MDX queries will use more than 5 axes. For the first 5 axes, the aliases COLUMNS, ROWS, PAGES, SECTIONS, and CHAPTERS can be used instead. In MDX, the <axis_specification> in the **select** statement is the

mechanism for displaying a data cube. An <axis_specification> consists of a set and one or more axis keywords.

The production rules of the grammar of the <axis_specification> clause are given in Listing 8.13.

```
<axis_specification> ::= [NON EMPTY] <set> [<dim_props>] ON
                         <axis_name>

<axis_name> ::= COLUMNS
              | ROWS
              | PAGES
              | CHAPTERS
              | SECTIONS
              | AXIS(<index>)

<dim_props> ::= [DIMENSION] PROPERTIES <property> [,
   <property>...]

<tuple> ::= <member>
          | (<member> [, <member>...])
          | <tuple_value_expression>
Note: Each member must be from a different dimension
or from a different hierarchy

<set> ::= <member>:<member>
        | <set_value_expression>
        | <open_brace>[<set>|<tuple>
           [, <set>|<tuple>...]]<close_brace>
        | (<set>)
Note: Each member must be from the same hierarchy
and the same level.
```

```
<tuple_value_expression> ::= <set>.CURRENTMEMBER
            | <set>[.ITEM]({<string_value_expression>
               [,<string_value_expression>...]}
               | <index>)

<set_value_expression> ::= <dim_hier>.MEMBERS
                         | <level>.MEMBERS
                         | <member>.CHILDREN
                         | ...

<cube_name> ::= [ [ [ <data_source>.] <catalog_name>.]
                        [<schema_name>.]<identifier>

<data_source> ::= <identifier>

<catalog_name> ::= <identifier>

<schema_name> ::= <identifier>

<dim_hier> ::= [<cube_name>.]<dimension_name>
             | [[<cube_name>.]< dimension_name>.]<hierarchy_name>

<dimension_name> ::= <identifier>
                   | <member>.DIMENSION
                   | <level>.DIMENSION
                   | <hierarchy>.DIMENSION

<dimension> ::= <dimension_name>

<hierarchy> ::= <hierarchy_name>

<hierarchy_name> ::= <identifier>
                   | < member>.HIERARCHY
                   | <level>.HIERARCHY
```

```
<level> ::= [<dim_hier>.]< identifier>
          | <dim_hier>.LEVELS(<index>)
          | <member>.LEVEL

<member> ::= [<level>.]<identifier>
           | <dim_hier>.<identifier>
           | <member>.<identifier>
           | <member_value_expression>


<member_value_expression> ::= <member>.{PARENT | FIRSTCHILD
                                        | LASTCHILD | PREVMEMBER
                                        | NEXTMEMBER   | ... }

<open_brace> ::= {
<close_brace> ::= }

<open_bracket> ::= [
<close_bracket> ::= ]

<underscore> ::= _

<alpha_char> ::= a | b | c | ...| z | A | B | C | ... | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Listing 8.13: Production rules for the MDX <axis_specification>

8.3.2 Mapping of `PROJECTION` Production Rules between MDX and NOX

NOX supports `PROJECTION` through the syntax of the `PROJECT` method defined in the NOX grammar by the <projection> production rule. The NOX grammar production

rules that describe the `PROJECT` method are given in Listing 4.1 (from Chapter 4).

The <projection> statement of the NOX grammar in Listing 4.1 depicts the same functionality as the <axis_specification> in the <select_statement> of the MDX grammar in Listing 8.13. The <projection> then is a <measure_dimension_list> as shown in the NOX grammar production rules specific for `PROJECTION` in Listing 4.1. The <measure_dimension_list> has at least one condition on some measure and can be combined by logical operators with other conditions on measures or on members in a dimension or a hierarchy of a dimension. As described in the previous subsection, the <axis_specification> is a specification of a set over an axis. The main restriction on a set is that all its elements have to be of the same structure. By structure we mean the following:

- If the set is a set of members, all members have to come from the same dimension or the same hierarchy (even though they can be from different levels).

- If the set is a set of tuples, then the dimensionality should be the same and the corresponding members of the tuples have to be from the same dimension or the same hierarchy.

So, these two cases correspond to NOX as follows:

- Case 1: The set in the <axis_specification> production rule is a set of members. In this case, the members have to be from the same dimension or from the same hierarchy. In the NOX grammar, the <projection> production rules translate to this behaviour by including members of a dimension or a hierarchy in the

<measure_dimension_list>.

- Case 2: The set in the <axis_specification> production rule is a set of tuples. The corresponding members of tuples in a set have to match in dimensionality, meaning from the same dimension or from the same hierarchy. Of course, the number of members in tuples in a set is the same and the order is important. Can we say that a set of tuples is the crossjoin of two sets? To answer this question, let's examine the crossjoin function another time. The Crossjoin function returns the cross product of two or more specified sets. The order of tuples in the resulting set depends on the order of the sets to be joined and the order of their members. Consider two sets:

  1. $S_1 = x_1, x_2, ..., x_n$, and
  2. $S_2 = y_1, y_2, ..., y_n$,

  the cross product of these sets is:
  $S_1 \times S_2 = \{(x_1, y_1), (x_1, y_2), ..., (x_1, y_n), (x_2, y_1), (x_2, y_2), ..., (x_2, y_n), ..., (x_n, y_1), (x_n, y_2), ..., (x_n, y_n)\}$. For any $S_k \subseteq (S_1 \times S_2)$, we assign two sets:

  1. S_{k1} to be the set of the members in the first positions of the tuples in S_k, and
  2. S_{k2} to be the set of the members in the second positions of the tuples in S_k.

  Calculating the cross join $S_{k1} \times S_{k2}$, we have $(S_{k1} \times S_{k2}) \subseteq (S_1 \times S_2)$. Hence,

if we have in MDX in the <axis_specification> a set of tuples S_k on some axis A_m, we can replace this <axis_specification> with two <axis_specification>, one with the set of members S_{k1} on A_m and another with the set of members S_{k2} on the next axis available for use. S_{k1} and S_{k2} are as described above. Now, we're back to case 1 in this proof and we do the translation of sets the same way, namely, we have members from the same dimension or from the same hierarchy in each set. In NOX grammar, the <projection> production rules translate to this behaviour by including members of a dimension or a hierarchy in the <measure_dimension_list>.

The above restrictions of having a set of members such that all members come from the same dimension or the same hierarchy and having a set of tuples with the same dimensionality are not forced in the NOX prototype. This is an important feature to be added in future work. We described earlier how to translate a set from MDX to NOX in Subsection 8.2.2. The same applies here with the addition that all members from each dimension will fall on one axis of the result cube and all measures on some other axis of the result cube. Hence, the result cube will have an axes count that is equivalent to the number of dimensions used in the `PROJECT` method in the query, plus one for the measures axis.

It is important to note that in NOX the query logic is separate from the display requirements. While the grammar in NOX supports the identification of display attributes, it provides no means to specify the actual layout of the results. Therefore, it is expected that other applications (reports, GUI, etc.) used by the client will

take care of the layout. This simplifies the job for programmers and depicts clear accountabilities between applications, especially when it comes to complex queries where displaying the results involves data from multiple axis, which will be shown in the examples provided.

8.3.3 `PROJECTION` Constraints

As was the case with the `SELECTION`, we focus the evaluation process on a small set of `Projection classes` indicative of common OLAP query patterns. We identify the following three possibilities:

1. Display a single dimension and measure
2. Display multiple attributes from a single hierarchy
3. Nested attribute display

Display a single dimension and measure

We begin with the basic case involving the requirement to display a single dimension and measure. In this example, we are interested in displaying the "Internet Sales Amount" measure value, along with all members of the Calendar Year.

We can formalize the `PROJECTION` as:

$\pi_{InternetSalesAmount,[CalendarYear].MEMBERS}(\text{Sales})$.

Listing 8.14 illustrates how this might be done using MDX. In this case, no slicer is required. The end result will be the aggregation of all cube cells into a simple table.

```
SELECT {[Measures].[Internet Sales Amount]} ON COLUMNS,
       [Date].[Calendar Year].MEMBERS ON ROWS
FROM   [Adventure Works]
```

Listing 8.14: MDX query returning a subcube with sales measure on one axis and calendar year members on another axis

The query in Listing 8.15 is the equivalence in NOX of the <axis_specification> of the MDX query. The NOX query is only slightly more verbose. The "Calendar Year" hierarchy object is instantiated by the method "getCalendarYearHierarchy()" in the **Date** class/dimension. After returning the hierarchy object, the method "members()", included within the base Hierarchy class, is used to identify all members at the Year level of the Date hierarchy. The "getInternetSales()" method in the **Measure** class is called to instantiate the **InternetSales** measure. Hence, the subcube returned by the query has the internet sales for customers on one axis and the calendar year members on another axis. Note as well that the `return` type of the `PROJECTION` method is listed as an `Object` array, rather than the boolean used for `SELECTION`.

```
class projectQuery1 extends OlapQuery {

public Object[] project() {

  Measure measures = new Measure();
  Date date = new Date();
  CalendarYearHierarchy calendarYearHierarchy =
                          date.getCalendarYearHierarchy();
```

```
  Object[] projections = {measures.getInternetSales(),
                          calendarYearHierarchy.members()};
  return projections;
}

}
```

Listing 8.15: NOX query returning a subcube with sales measure on one axis and calendar year members on another axis

8.3.4 Display Multiple Attributes from a Single Hierarchy

It is often necessary to select multiple members from a hierarchy for display on a given axis. Let's assume that we want to display the "Internet Sales Amount" measure and provide labels for the year 2005 and the date range 2008 to 2011 inclusive. Formally, we could specify the `PROJECTION` as follows:

$$\pi_{InternetSalesAmount,\ CalendarYear=2005,\ (CalendarYear >= 2008\ \&\&\ CalendarYear <= 2011)}(\text{Sales}).$$

Listing 8.16 depicts the query in MDX. In this case, the date members are listed as a set, with the date range defined using MDX's colon notation. Again, no slicer is required for this simple query.

```
SELECT {[Measures].[Internet Sales Amount]} ON COLUMNS,
    {[Date].[Calendar Year].[2005],
     [Date].[Calendar Year].[2008]:[Date].[Calendar Year].[2011]}
    ON ROWS
FROM  [Adventure Works]
```

Listing 8.16: MDX query returning a subcube with sales measure on one axis and some specified calendar years on another axis

The NOX version of the query is illustrated in Listing 8.17. The "Calendar Year" hierarchy **calendarYearHierarchy** object is instantiated by the method "getCalendarYearHierarchy()" in the **Date** class/dimension. After returning the hierarchy object, the method "includesList(OlapPath ... path)", included within the base Hierarchy class, is called with the argument "OlapPath("2005")", that has the value "2005" for the first level of the hierarchy. Another method of the **calendarYearHierarchy** "includesRange(OlapPath ... path)" is called to identify the ordered list of years in the Date hierarchy. In this case, it is called with the two arguments "OlapPath("2008")" and "OlapPath("2011")" respectively. Note that the DBMS schema designer is expected to identify sort orders for hierarchy levels.

```
class projectQuery2 extends OlapQuery {

public Object[] project() {

  Measure measures = new Measure();
  Date date = new Date();
```

```
  CalendarYearHierarchy calendarYearHierarchy =
                                date.getCalendarYearHierarchy();

  Object[] projections =
     {calendarYearHierarchy.includesList(new OlapPath("2005")),
      calendarYearHierarchy.includesRange(new OlapPath("2008"),
                                          new OlapPath("2011")),
      measures.getInternetSales()};

  return projections;
}

}
```

Listing 8.17: NOX query returning a subcube with sales measure on one axis and some specified calendar years on another axis

8.3.5 Nested Attribute Display

Finally, we turn to the case in which one or more attributes are to be nested within a single display axis. In the language of MDX, this what is known as a crossjoin operation and it is extremely common in the MDX domain. Let's assume in addition to the "Internet Sales Amount" measure on one axis, we want to display the combination of products in the Category range 1 to 5 and the members of the Calendar Year. More formally, we say the following:

$$\pi_{(InternetSalesAmount)\ CrossJoin\ (Category >= `1' \ \&\&\ Category <= `5')CalendarYear.MEMBERS}(\text{Sales}).$$

Listing 8.18 depicts the MDX version of the query. The set that is returned on the Rows axis is the crossjoin of two sets: One is the set of members that range

from Category 1 products to Category 5 products in the "Category" hierarchy of the "Product" dimension, and the other is the set of the members of the calendar years in the "Calendar Year" hierarchy of the "Date" dimension.

```
SELECT
    {[Measures].[Internet Sales Amount]} ON COLUMNS,
    CrossJoin (
          {[Product].[Category].&[1]:[Product].[Category].&[5]},
          { [Date].[Calendar Year].MEMBERS } ) ON ROWS
FROM  [Adventure Works]
```

Listing 8.18: MDX query returning a subcube with sales measure on one axis and the crossjoin of two sets on another axis

The query in Listing 8.19 depicts the NOX version of the query.

In the MDX version of the query, the crossjoin of two sets is projected on one axis of the result cube. In the NOX version of the query, this crossjoin is translated to two different sets implicitly projected on two axes of the result cube. One of the axis contains the members at the first level of the "Calendar Year" hierarchy of the "Date" dimension. Another axis contains the categories in the range between 1 and 5 that are returned as members at the first level of the "Category" hierarchy of the "Product" Dimension. The last axis contains the **InternetSales** measure. Note that the NOX model does not actually provide display functionality, leaving that instead to the external application. As such, true crossjoin functionality is not provided by NOX. Instead, the query is really defined as a multi-dimensional variation on the first

query defined in this subsection.

```
class projectQuery3 extends OlapQuery {

public Object[] project() {

  Measure measures = new Measure();
  Date date = new Date();
  CalendarYearHierarchy calendarYearHierarchy =
                              date.getCalendarYearHierarchy();
  Product product = new Product();
  CategoryHierarchy categoryHierarchy =
                              product.getCategoryHierarchy();

  Object[] projections = {calendarYearHierarchy.members(),
   categoryHierarchy.includesRange(new OlapPath(1),new
       OlapPath(5)),
   measures.getInternetSales()};

  return projections;
}

}
```

Listing 8.19: NOX query returning a subcube with sales measure on one axis and two sets on two other axes

8.4 Conclusion

In this chapter, we demonstrated how the NOX model, which offers a query language that is both native and OLAP-specific, is capable of representing the range of query

patterns developers have come to expect in the OLAP domain. We accomplished this through examining the NOX model from an algebraic perspective and comparing its expressiveness to MDX, the de-facto standard query language in this domain. We focused on `SELECTION` and `PROJECTION` due to the fact that these are by far the most prominent OLAP operations.

The concept of native language OLAP querying has been discussed in the industry but no active development or analysis has been made to advance and publicize such a model. The main advantages of using native languages are that they provide elegant scaling, improved development cycles, compile time checking, ease of testing and better debugging tools. By taking a practical approach and presenting examples with a well-structured algebra that satisfies operations done in OLAP, we demonstrated that NOX and intuitive and it minimizes the constraints of existing BI languages. This, in turn, should attract DBMS developers.

Chapter 9

Conclusion

With the popularity of data warehousing and OLAP techniques in the business intelligence world, having a query language that is both native and OLAP-specific is a significant advantage for developers working in the Business Intelligence domain. Much development effort has been spent on building OOP interfaces for general purpose relational database management systems. However, no domain-specific native language facility has focused on OLAP querying. Given the awkward, almost completely unstandardized nature of the current OLAP application marketplace, we believe that NOX offers exciting possibilities for those building and utilizing products and services in this extremely important area [ETT10] [TET11].

The main objective of this thesis, identified in the introduction, is to present the Native language OLAP query eXecution (NOX) framework specifically tailored to the BI/OLAP domain. The current version of NOX represents a comprehensive implementation of the native language query model. All the examples along with their related concepts mentioned in this thesis are fully implemented and tested. They

are integrated into the Sidera system and executed there. In building a consistent OLAP conceptual model, we have been able to provide transparent cube persistence functionality that allows the programmer to view remote, possibly very large, analytical repositories merely as local objects. In addition to the ability to program against the conceptual model, our framework also provides compile-time type checking, clean refactoring opportunities, and direct Object-Oriented manipulation of the OLAP queries and their result sets. While we chose to target Java in this implementation, the fundamental concepts are language agnostic and could easily be applied to other modern OOP languages. In meeting the main objective, we:

1. Designed a grammar that presents the developer with an Object Oriented representation of the primary OLAP operations pertaining to the OLAP-specific algebra.

2. Built the NOX parser for Java application OLAP programming and demonstrated how developers write queries in NOX to interact with remote data cubes using standard OOP principles and practices.

3. Incorporated parameter passing in NOX in a simple and intuitive manner.

4. Evaluated NOX by comparing and contrasting it to MDX, the de-facto "string-based" OLAP query language.

5. Demonstrated the flexibility of OLAP hierarchies in NOX.

6. Made code re-use possible as afforded by OOP concepts such as inheritance.

7. Encapsulated direct Object-Oriented manipulation of Results Sets by allowing data to be transparently mapped back into the client applications as objects in the NOX API.

8. Demonstrated the formal validation of NOX in the OLAP context by mapping its `SELECTION` and `PROJECTION` grammar production rules to those of the MDX grammar.

To summarize the accomplishments of the thesis, Table 3 provides the mapping between the objectives given in the introduction and the corresponding chapters/sections that address them.

Objective	**Corresponding Sections/Chapters**
1	Sections 4.2, 4.3, and 4.6
2	Sections 4.4 and 4.5
3	Chapter 5
4	Chapter 7
5	Chapters 5 and 6
6	Chapter 6
7	Section 5.5
8	Section 5.6
9	Chapter 8

Table 3: Objectives and the Chapters/Sections where they were implemented

9.1 Research Methodology and Contribution

Computer science research methodologies can be divided into three distinct methods: theoretical, experimental and simulation [DC02]. Our research was mainly done using the experimental research methodology and then validated with relevant theoretical methods. Experimental methods were used to build the model of the OLAP-specific NOX framework and compare its empirical results with those of the de-facto language for OLAP querying, MDX. Comparison of the two grammars of NOX and MDX then allowed us to demonstate the validity of the NOX model.

Ultimately, the main contribution of this thesis is to help programmers write their OLAP queries in the native language itself. While the underlying compilation and translation mechanism is somewhat complex, all of the framework's sophistication is virtually hidden from the developer. As stated at the outset, the focus of the NOX model is clearly on the BI/OLAP domain. In fact, NOX is intended to specifically support higher level analytics servers. It is not expected to resolve all possible queries that might be executed against an underlying relational data warehouse. The primary motivation for this approach is the rejection of the "be all things to all people" mantra that tends to plague systems that must maintain a fully generic, lowest common denominator profile [SMA+07]. Conventional RDBMSs, conceptual mapping frameworks such as JOLAP suffer from this same "curse of generality". JOLAP was introduced in Chapter 3. In the current context, the targeting of a specific application domain ultimately relieves the designer from having to manually construct a

comprehensive data model, along with its constituent processing constructs.

In addition, it is important to note that a second contribution is the construction of the NOX prototype itself. Specifically, we demonstrated the practical viability of a language model that is easily extendable and portable and provides a fully-implementable OLAP native language system.

9.2 Future Work

In this thesis, we have modeled the primary components of the native OLAP language execution framework and tested them extensively by running OLAP queries of different kinds. The model flexibility and object-oriented nature offer various research opportunities for future work such as the following:

- Mapping between a NOX query and its corresponding XML string, generated by NOX framework, is a language problem to be addressed in two steps, namely:
  - The parsing problem: Given the grammar G and a string s the parsing problem answers the question whether or not s $\in$ L(G). If s $\in$ L(G), the answer to this question may be either a parse tree or a derivation [JS].
  - The correspondence between the parse tree, produced by the NOX preprocesser, and the DOM tree (representative of the query's xml string produced).
- Extending the grammar to include additional operations relevant to the data warehouse context, possibly including:

  - Hierarchical navigation functions such as ancestor, first child, sibling, etc.
  - Numeric functions such as correlation, covariance, etc.

- Query optimization would be interesting for future work.
- Adding more programming constructs to the NOX API to increase the expressiveness of the language. Extending the usage of parameters would be a possibility in this context. For example, having different types of parameters such as arrays of parameters will increase the expressiveness of the language.
- As the input Java file is re-written (by the NOX parser) before it is compiled by the regular compiler, debugging using debugging tools becomes a problem. Very important future work will be to tackle this problem.
- Adding additional programming constructs to the NOX API to complete the implementation of the grammar of the language
  - Change level algebraic operator
  - Change base algebraic operator
  - Drill across algebraic operator
- Limitation of not having the whole object-oriented paradigm is in the present version of NOX. Adding object-oriented functionalities such as providing interfaces in NOX will add to the powerfulness of NOX.
- Another limitation is that the basic constructs of MDX are not fully implemented in NOX. Constructs in MDX such as <tuple_value_expression,

<set_value_expression> and <member_value_expression> need to be matched to constructs with similar functionality in NOX.

- Restrictions of having a set of members such that all members come from the same dimension or the same hierarchy and having a set of tuples with the same dimensionality are not enforced in the NOX prototype. This is an important feature to be done in future work.

- Developing an interactive, real-time interface to the data warehouse. While this can be accomplished with, for example, an interactive Java shell, a more interesting option would be to port NOX to a full fledged, interpreted OOP language like Python.

- Testing of the prototype has been done by using ad hoc "case by case" method. Employing more formal testing mechanisms might be an interesting problem.

- The NOX language may be implemented in other languages that are in popular use, with possibilities including languages such as C++ and Delphi.

Bibliography

[AB87] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–170, 1987.

[AGS97] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *Proceedings of the 13th International Conference on Data Engineering (ICDE 1997)*, pages 232–243, Washington, DC, USA, 1997. IEEE Computer Society Press.

[AMM07] A. Adya, J. A. Blakeley , S. Melnik, and S. Muralidhar. Anatomy of the ADO.NET entity framework. In *ACM SIGMOD International conference on Management of Data*, pages 877–888, New York, NY, USA, 2007. ACM.

[AR] A. Abello and O. Romero. On-line analytical processing. Technical report, Universitat Politecnica de Catalunya.

[ASS01] A. Abello, J. Samos, and F. Saltor. A framework for the classification and description of multidimensional data models. In *DEXA 2001*, volume 2113, pages 668–677, 2001.

[ASS03] A. Abello, J. Samos, and F. Saltor. Implementing operations to navigate semantic star schemas. In *Proceedings of DOLAP'2003. ACM*, 2003.

[ASS05] A. Abello, J. Samos, and F. Saltor. Yam^2 (yet another multidimensional model): An extension of uml. In *Information Systems, Elsevier*, 2005.

[BK06] C. Bauer and G. King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[BRK+08] 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, New York, NY, USA, 2008. ACM.

[CCS92] 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.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26:65–74, 1997.

[CR05] W. Cook and S. Rai. Safe query objects: Statically typed objects as remotely executable queries. *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 97–106, 2005.

[CR06] W. Cook and C. Rosenberger. Native queries for persistent objects. A Design White Paper, 2006.

[CT98] L. Cabibbo and R. Torlone. From a procedural to a visual query language for OLAP. In *Proceedings of the 10th Internationla Conference on Scientific and Statistical Database Management (SSDBM)*, pages 74–83, 1998.

[CW00] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 1–10, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[CWM03] CWM, Common Warehouse Metamodel, 2003. `http://www.cwmforum.org`.

[DB4] db4objects. `http://www.db4o.com`.

[DC02] G. Dodig-Crnkovic. Scientific methods in computer science. In *Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, April 2002.

[DKK05] J. P. Dittrich, D. Kossmann, and A. Kreutz. Bridging the gap between OLAP and SQL. In *International conference on Very Large Data Bases (VLDB)*, pages 1031–1042, 2005.

[DTDa] Document Type Definition, Wikepedia. `http://en.wikipedia.org/wiki/Document_Type_Definition`.

[DTDb] Definition of the XML document type declaration from Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/2008/REC-xml-20081126/.

[EDD+07] T. Eavis, G. Dimitrov, I. Dimitrov, D. Cueva, A. Lopez, and A. Taleb. Sidera: a cluster-based server for online analytical processing. In *Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS - Volume Part II*, OTM'07, pages 1453–1472, Berlin, Heidelberg, 2007. Springer-Verlag.

[EJB] EJB 3.0 (Enterprise JavaBeans). http://java.sun.com/products/ejb/.

[ETT10] T. Eavis, H. Tabbara, and A. Taleb. The nox framework: Native language queries for business intelligence applications. In Torben Bach Pedersen, Mukesh K. Mohania, and A Min Tjoa, editors, *DaWak*, volume 6263 of *Lecture Notes in Computer Science*, pages 172–189. Springer, 2010.

[FBV00] E. Franconi, F. Baader, and P. Vassiliadis. Multidimensional data models and aggregation. In M. Jarke, M. Lenzerini, Y. Vassilious, and P. Vassiliadis, editors, *Fundamentals of Data Warehousing*. Springer, Heidelberg, 2000.

[FK04] E. Franconi and A. Kamble. The GMD data model and algebra for multidimensional information. In A. Persson and J. Stirna, editors, *CAiSE 2004. LNCS*, volume 3084, pages 446–462. Springer, Heidelberg, 2004.

[FS00] E. Franconi and U. Sattler. In *A Data Warehouse Conceptual Data Model for Multidimensional Aggregation: A Preliminary Report*, 2000.

[GBB+00] R. Gattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann, 2000.

[GL97] M. Gyssens and L. Lakshmanan. A foundation for multi-dimensional databases. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB 1997)*, pages 106–115, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[GL07] J. Gil and K. Lenz. Eliminating impedance mismatch in C++. In *International conference on Very Large Data Bases (VLDB)*, pages 1386–1389. VLDB Endowment, 2007.

[GMR98] M. Golfarelli, D. Maio, and S. Rizzi. The dimensional fact model: A conceptual model for data warehouses. In *International Journal of Cooperative Information Systems (IJCIS)*, volume 7(2-3), pages 215–247, 1998.

[Has] HaskellDB. `http://www.haskell.org/haskellDB/`.

[Hay04] C. Hays. What wal-mart knows about customers' habits. *The New York Times*, 2004.

[HC97] G. Hamilton and R. Cattell. Jdbc: A java sql api. *Sun Microsystems*, 1997.

[Hil10] I. Hilgefort. *Reporting and Analytics Using SAP BusinessObjects*. Galileo Press, 2010.

[HK06] J. Han and M. Kamber, editors. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[HS98] M.-S. Hacid and U. Sattler. Modeling multidimensional database: A formal object-centered approach. In *Proceedings of the 6th European Conference on Information Systems (ECIS)*, pages 83–102, 1998.

[Jav] Javacc, the java compiler compiler. `https://javacc.dev.java.net/`.

[JDO] JDO (Java Database Objects). `http://java.sun.com/products/jdo/`.

[JJT] Java.net, JJTree reference documentation. `https://javacc.dev.java.net/doc/JJTree.html`.

[JOL03] JSR-69 JavaTM OLAP Interface (JOLAP), JSR-69 (JOLAP) Expert Group, 2003. `http://jcp.org/aboutJava/communityprocess/first/jsr069/index.html`.

[JS] J. Jeuring and D. Swierstra, editors. *Grammars and Parsing*.

[KR02] R. Kimball and M. Ross, editors. *The Data Warehouse Toolkit*. Addison-Wesley, 2002.

[LIN] LINQ: .NET Language Integrated Query. `http://msdn.microsoft.com/en-us/library/bb308959.aspx`.

[LS97] H. Lenz and A. Shoshani. Summarizability in olap and statistical data bases. pages 132–143. IEEE Computer Society, 1997.

[LW96] C. Li and X.S. Wang. A data model for supporting on-line analytical processing. In *Proceedings of 5th International Conference on Information and Knowledge Management (CIKM 1996)*, pages 81–88. ACM Press, 1996.

[Mel02] J. Melton. *Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features*. Elsevier Science Inc., New York, NY, USA, 2002.

[Mic] Microsoft Connect: Your feedback improving Microsoft products. `http://connect.microsoft.com/SQLServer/feedback/details/251601/run-mdx-query-with-parameters-using-oledb`.

[MSD] MSDN Library: Using Parameters in MDX. `http://msdn.microsoft.com/en-us/library`.

[MZ04] E. Malinowski and E. ZimĞnyi. Olap hierarchies: A conceptual perspective. *CAiSE*, 3084:477–491, 2004.

[MZ06] E. Malinowski and E. Zimanyi. Hierarchies in a multidimensional model: From conceptual modeling to logical representation. *Data Knowl. Eng.*, 59(2):348–377, 2006.

[NGD+08] M. C. Norrie, M. Grossniklaus, C. Decurtins, A. de Spindler, A. Vancea, and S. Leone. Semantic data management for db4o. In *ICOODB'08*, pages 21–38, 2008.

[Nol99] C. Nolan. Manipulate and query OLAP data using adomd and multidimensional expressions. *Microsoft Systems Journal*, pages 97–106, 1999.

[ODM] ODMG web site. `http://www.odmg.org`.

[Pow99] D. Power. Decision Support Systems Glossary. `http://DSSResources.COM/glossary/`, 1999.

[RA05] O. Romero and A. Abello. Improving automatic SQL translation for rolap tools. In *Proceedings of 9th Jornadas en Ingenieris del Software y Bases de Datos (JISB 2005)*, volume 3284(5), pages 123–130, 2005.

[RA07] O. Romero and A. Abello. On the need of a reference algebra for OLAP. In *Proceedings of the 9th International Conference on Data Warehousing and Knowledge Discovery (DAWAK 2007)*, pages 99–110, 2007.

[Ran05] J. Ranjan. Applications of data mining techniques in pharmaceutical industry. *Journal of Theoretical and Applied Information*, 2005.

[Rub] Ruby programming language. `http://www.ruby-lang.org/en/`.

[Rus03] C. Russell. Java data objects (jdo) specification jsr-12. *Sun Microsystems*, 2003.

[SBSR08] A. Simitsis, A. Baid, Y. Sismanis, and B. Reinwald. Multidimensional content exploration. In *In VLDB*, 2008.

[SC05] M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[Sel08] M. Seltzer. Beyond relational databases. *Communications of the ACM*, 51:52–58, July 2008.

[SH98] M. Stonebraker and J. Hellerstein, editors. *Readings in Database Systems, third edition.* Morgan Kaufmann, 1998.

[SHW+06] G. Spofford, S. Harinath, C. Webb, D. Hai Huang, and F. Civardi, editors. *MDX Solutions with Microsoft SQL Server Analysis Services 2005 and Hyperion Essbase, Second Edition.* John Wiley and Sons, 2006.

[SMA+07] M .Stonebraker, S. Madden, D. J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In *International conference on Very Large Data Bases (VLDB)*, pages 1150–1160, 2007.

[TCIK99] M. Tatsubori, S. Chiba, K. Itano, and M. Killijian. Openjava: A class-based macro system for java. *OORaSE'99, ACM Workshop on Object-Oriented Reflection and Software Engineeringon*, pages 117–133, 1999.

[TD97] H. Thomas and A. Datta. A conceptual model and algebra for on-line analytical processing in data warehouses. In *Proceedings of 23rd the 7th Workshop on Information Technologies and Systems (WITS 1997)*, pages 91–100, 1997.

[TD01] H. Thomas and A. Datta. A conceptual model and algebra for on-line analytical processing in decision support databases. In *Information Systems 12(1)*, pages 83–102, 2001.

[TET11] A. Taleb, T. Eavis, and H. Tabbara. The NOX OLAP query model: From algebra to execution. In Alfredo Cuzzocrea and Umeshwar Dayal, editors, *DaWaK*, volume 6862 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2011.

[VS99] P. Vassiliadis and T. Sellis. A survey of logical models for OLAP databases. In *SIGMOD Record 28*, volume 4, pages 64–69, 1999.

[WZP05] M. Whitehorn, R. Zare, and M. Pasumansky. *Fast Track to MDX*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

Appendices

Appendix A

Abbreviations

ADOMD: ActiveX Data Objects Multidimensional

API: Application Programming Interface

AST: Abstract Syntax Tree

BI: Business Intelligence

CWM: Common Warehouse Metamodel

DB4O: Database for Objects

DSS: Decision Support Systems

DTD: Document Type Definition

EJB: Enterprise JavaBeans

ETL: Extract, Transform, Load

EM: Entity Mapping

FASMI: Fast Analysis Shared Multidimensional Information

HOLAP: Hybrid OnLine Analytical Processing

HQL: Hibernate Query Language

IDE: Integrated Development Environment

IT: Information Technology

JavaCC: Java Compiler Compiler

JDBC: Java Database Connectivity

JDO: Java Database Objects

JDOQL: Java Database Objects Query Language

JOLAP: Java OLAP Interface

JPA: Java Persistence API

JPQL: Java Persistence Query Language

LINQ: Language Integrated Query

MD: Multi Dimensional

MDX: Multi Dimensional eXpressions OLAP query language

MOLAP: Multidimensional OnLine Analytical Processing

NOX: Native language OLAP query eXecution

ODBC: Open Database Connectivity

OLAP: OnLine Analytical Processing

OLTP: OnLine Transaction Processing

OLEDB: Object Linking and Embedding, Database

OOP: Object Oriented Programming

OQL: Object Query Language

ORM: Object Relational Mapping

#PCDATA: Parsed Charater Data POJO: Plain Old Java Object

PSI: Parallel Service Interface

RDBMS: Relational Database Management Systems

ROLAP: Relational OnLine Analytical Processing

SQL: Structured Query Language

XML: Extensible Markup Language

XMLA: Extensible Markup Language for Analysis

Appendix B

DTD Schema

In this appendix, we go into more details what a DTD schema and a DTD markup are.

In a DTD markup, declarations are used to declare which elements types, attribute lists, entities and notations are allowed in the structure of the corresponding class of XML documents. An Element Type Declaration defines an element and its possible content. A valid XML document only contains elements that are defined in the DTD. An element's content is specified by some key words and characters [DTDa]:

- **EMPTY** for no content
- **FOR** for any content
- **,** for orders
- | for alternatives ("either ... or")
- **(** **)** for groups
- ***** for any number (zero or more)

- + for at least once (one or more)
- **?** for optional (zero or one)
- If there is no *, + or ?, the element must occur exactly one time

An example of a DTD is depicted in Listing B.1. To illustrate, we report the following:

- #PCDATA stands for Parsed Character Data and is the keyword to specify mixed content, meaning an element may contain character data as well as child elements in arbitrary order and number of occurrences.
- The QUERY element contains a DATA_QUERY or a META_QUERY .
- The DATA_QUERY element contains either a CUBE_NAME, an optional OPERATION_LIST, and an optional FUNCTION_LIST.
- The CUBE_NAME element contains plain text.
- The OPERATION_LIST element contains at least one OPERATION.
- The OPERATION element contain one of the following elements: SELECTION or PROJECTION or CHANGE_LEVEL or CHANGE_BASE or DRILL_ACROSS or UNION or INTERSECTION or DIFFERENCE)>.

This DTD example is not complete, as we just wanted to demonstrate how a DTD is defined.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT QUERY (DATA_QUERY | META_QUERY)>
```

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

Listing B.1: DTD example

Appendix C

Complex Query in XML

```
<QUERY>
<DATA_QUERY>
<CUBE_NAME>
SampleCube
</CUBE_NAME><OPERATION_LIST>
<OPERATION>
<SELECTION>
<DIMENSION_MEASURE_LIST>
<DIMENSION>
<DIMENSION_NAME>
Customer
</DIMENSION_NAME><EXPRESSION>
<RELATIONAL_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>
</RELATIONAL_EXP>
</EXPRESSION>
</DIMENSION><LOGICAL_OP>
AND
</LOGICAL_OP><DIMENSION>
<DIMENSION_NAME>
DateDimension
</DIMENSION_NAME><EXPRESSION>
<RELATIONAL_EXP>
<SIMPLE_EXP>
<EXP_VALUE>
<ATTRIBUTE>
year
</ATTRIBUTE>
</EXP_VALUE>
</SIMPLE_EXP><COND_OP>
<EQUALITY_OP>
EQUALS
</EQUALITY_OP>
</COND_OP><SIMPLE_EXP>
<EXP_VALUE>
<CONSTANT>
2007
</CONSTANT>
</EXP_VALUE>
</SIMPLE_EXP>
</RELATIONAL_EXP>
```

```
</EXPRESSION>
</DIMENSION><LOGICAL_OP>
AND
</LOGICAL_OP><DIMENSION>
<DIMENSION_NAME>
DateDimension
</DIMENSION_NAME><EXPRESSION>
<SIMPLE_EXP>
<EXP_VALUE>
<FUNCTION_LIST>
<FUNCTION>
<PARENT>
month
</PARENT><FUNCTION_NAME>
inRange
</FUNCTION_NAME><ARGUMENT_LIST>
<ARGUMENT>
5
</ARGUMENT><ARGUMENT>
10
</ARGUMENT>
</ARGUMENT_LIST>
</FUNCTION>
</FUNCTION_LIST>
</EXP_VALUE>
</SIMPLE_EXP>
</EXPRESSION>
</DIMENSION><LOGICAL_OP>
AND
</LOGICAL_OP><DIMENSION>
<DIMENSION_NAME>
Supplier
</DIMENSION_NAME><EXPRESSION>
<RELATIONAL_EXP>
<SIMPLE_EXP>
<ARITHMETIC_EXP>
```

```
<SIMPLE_EXP>
<EXP_VALUE>
<ATTRIBUTE>
balance
</ATTRIBUTE>
</EXP_VALUE>
</SIMPLE_EXP><ARITHMETIC_OP>
DIVIDE
</ARITHMETIC_OP><SIMPLE_EXP>
<EXP_VALUE>
<CONSTANT>
100
</CONSTANT>
</EXP_VALUE>
</SIMPLE_EXP>
</ARITHMETIC_EXP>
</SIMPLE_EXP>
<COND_OP>
<RELATIONAL_OP>
LT
</RELATIONAL_OP>
</COND_OP><SIMPLE_EXP>
<EXP_VALUE>
<CONSTANT>
45623.00
</CONSTANT>
</EXP_VALUE>
</SIMPLE_EXP>
</RELATIONAL_EXP>
</EXPRESSION>
</DIMENSION><LOGICAL_OP>
AND
</LOGICAL_OP><DIMENSION>
<DIMENSION_NAME>
Product1
</DIMENSION_NAME><EXPRESSION>
```

```
<SIMPLE_EXP>
<EXP_VALUE>
<HIERARCHY_LIST>
<HIERARCHY>
<HIERARCHY_NAME>
ProductHierarchy
</HIERARCHY_NAME><HIERARCHY_OP>
includes
</HIERARCHY_OP><OLAP_PATH_LIST>
<OLAP_PATH>
<VALUE>
"automotive"
</VALUE><VALUE>
"exterior"
</VALUE><VALUE>
"lights"
</VALUE>
</OLAP_PATH>
<OLAP_PATH>
<VALUE>
"automotive"
</VALUE><VALUE>
"interior"
</VALUE>
</OLAP_PATH>
</OLAP_PATH_LIST>
</HIERARCHY>
</HIERARCHY_LIST>
</EXP_VALUE>
</SIMPLE_EXP>
</EXPRESSION>
</DIMENSION>
</DIMENSION_MEASURE_LIST>
</SELECTION><PROJECTION>
<MEASURE_DIMENSION_LIST>
<DIMENSION>
```

```
<DIMENSION_NAME>
DateDimension
</DIMENSION_NAME><EXPRESSION>
<SIMPLE_EXP>
<EXP_VALUE>
<HIERARCHY_LIST>
<HIERARCHY>
<HIERARCHY_NAME>
TimeHierarchy
</HIERARCHY_NAME><HIERARCHY_OP>
rangeList
</HIERARCHY_OP><OLAP_PATH_LIST>
<OLAP_PATH>
<VALUE>
"1996"
</VALUE>
</OLAP_PATH><OLAP_PATH>
<VALUE>
"2001"
</VALUE>
</OLAP_PATH>
</OLAP_PATH_LIST>
</HIERARCHY>
</HIERARCHY_LIST>
</EXP_VALUE>
</SIMPLE_EXP>
</EXPRESSION>
</DIMENSION>
</MEASURE_DIMENSION_LIST>
</PROJECTION>
</OPERATION>
</OPERATION_LIST>
</DATA_QUERY>
</QUERY>
```

Listing C.1: XML string corresponding to the query in Listing 5.4

Appendix D

MDX Grammar Production Rules

In this appendix, we show the grammar that describes the MDX language. Listing D.1 shows the productions rules for this grammar.

Listing D.1: MDX grammar

```
<MDX_statement> ::= <select_statement>
                  | <create_formula_statement>
                  | <drop_formula_statement>

<select_statement> ::= [WITH <formula_specification>]
                       SELECT [<axis_specification>
                       [, <axis_specification>...]]
                       FROM [<cube_specification>]
                       [WHERE [<slicer_specification>]]
                       [<cell_props>]

<axis_specification> ::= [NON EMPTY] <set> [<dim_props>] ON
   <axis_name>

<axis_name> ::= COLUMNS
              | ROWS
```

```
                | PAGES
                | CHAPTERS
                | SECTIONS
                | AXIS(<index>)

<dim_props> ::= [DIMENSION] PROPERTIES <property> [,
   <property>...]


<slicer_specification> ::= {<set> | <tuple>}

<tuple> ::= <member>
          | (<member> [, <member>...])
          | <tuple_value_expression>
Note: Each member must be from a different dimension
or from a different hierarchy

<set> ::= <member>:<member>
        | <set_value_expression>
        | <open_brace>[<set>|<tuple>
           [, <set>|<tuple>...]]<close_brace>
        | (<set>)
Note: Each member must be from the same hierarchy
and the same level.

<tuple_value_expression> ::= <set>.CURRENTMEMBER
            | <set>[.ITEM]({<string_value_expression>
               [,<string_value_expression>...]}
               | <index>)

<set_value_expression> ::= <dim_hier>.MEMBERS
                         | <level>.MEMBERS
                         | <member>.CHILDREN
                         | ...
```

```
<cube_name> ::= [ [ [ <data_source>.] <catalog_name>.]
                          [<schema_name>.]<identifier>

<data_source> ::= <identifier>

<catalog_name> ::= <identifier>

<schema_name> ::= <identifier>

<dim_hier> ::= [<cube_name>.]<dimension_name>
             | [[<cube_name>.]< dimension_name>.]<hierarchy_name>

<dimension_name> ::= <identifier>
                   | <member>.DIMENSION
                   | <level>.DIMENSION
                   | <hierarchy>.DIMENSION

<dimension> ::= <dimension_name>

<hierarchy> ::= <hierarchy_name>

<hierarchy_name> ::= <identifier>
                   | < member>.HIERARCHY
                   | <level>.HIERARCHY

<level> ::= [<dim_hier>.]< identifier>
          | <dim_hier>.LEVELS(<index>)
          | <member>.LEVEL

<member> ::= [<level>.]<identifier>
           | <dim_hier>.<identifier>
           | <member>.<identifier>
           | <member_value_expression>


<member_value_expression> ::= <member>.{PARENT | FIRSTCHILD
```

```
                                        | LASTCHILD | PREVMEMBER
                                        | NEXTMEMBER   | ... }

<open_brace> ::= {
<close_brace> ::= }

<open_bracket> ::= [
<close_bracket> ::= ]

<underscore> ::= _

<alpha_char> ::= a | b | c | ...| z | A | B | C | ... | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Appendix E

NOX Grammar Production Rules

In this appendix, we show the grammar that describes the NOX language. Listing E.1 shows the productions rules for this grammar.

Listing E.1: NOX grammar

```
<query> ::= <data_query>
          | <meta_query>

<data_query> ::= <cube_name>
                 [, <operation_list>] [, <function_list>]

<operation_list> ::= <operation> [, <operation>, ...]

<operation> ::= <selection>
              | <projection>
              | ...

<selection> ::= <dimension_measure_list>

<dimension_measure_list> ::=
```

```
            <dimension> (, <logical_op>, (<dimension> |
               <measure>))*
          | <measure> , <logical_op> , <dimension>
               (, <logical_op>, (<dimension> | <measure>))*
          | <measure> (, <logical_op>, (<dimension> |
             <measure>))*,
               <logical_op> , <dimension>

<projection> ::= <measure_dimension_list>

<measure_dimension_list> ::=
            <measure> (, <logical_op>, (<dimension> | <measure>))*
          | <dimension> , <logical_op> , <measure>
               (, <logical_op>, (<dimension> | <measure>))*
          | <dimension> (, <logical_op>, (<dimension> |
             <measure>))*,
               <logical_op> , <measure>

<measure> ::= <measure_name> [, <cond_op>, <simple_exp>]
<measure_name> ::= #PCDATA

<dimension> ::= <dimension_name> , <expression>

<dimension_name> ::= #PCDATA

<expression> ::= <relational_exp>
               | <compound_exp>

<compound_exp> ::= <expression>, <logical_op>, <expression>

<relational_exp> ::= <simple_exp> , <cond_op> , <simple_exp>

<simple_exp> ::= <exp_value>
               | <airthmetic_exp>

<arithmetic_exp> ::= <simple_exp> , <arithmetic_op> , <simple_exp>
```

```
<exp_value> ::= <constant>
              | <attribute>
              | <hierarchy_list>
              | <function_list>

<constant> ::= #PCDATA

<attribute> ::= #PCDATA

<logical_op> ::= #PCDATA

<cond_op> ::= <relational_op>
            | <equality_op>
            | <olap_op>

<relational_op> ::= #PCDATA

<equality_op> ::= #PCDATA

<olap_op> ::= #PCDATA

<arithmetic_op> ::= #PCDATA

<function_list> ::= <function>+

<function> ::= <parent> , <function_name> [, argument_list]

<parent> ::= #PCDATA

<function_name> ::= #PCDATA

<argument_list> ::= <argument>+

<argument> ::= #PCDATA
```

```
<hierarchy_list> ::= <hierarchy>+

<hierarchy> ::= <hierarchy_name> , <hierarchy_op> ,
   <olap_path_list>

<hierarchy_name> ::= #PCDATA

<hierarchy_op> ::= #PCDATA

<olap_path_list> ::= <olap_path>

<olap_path> ::= <value>+

<value> ::= #PCDATA
```