The NOX Framework: Native Language Queries for Business Intelligence Applications

Todd Eavis, Hiba Tabbara, and Ahmad Taleb

Concordia University, Montreal, Canada

Abstract. Over the past ten to fifteen years, Business Intelligence applications have become increasingly important and visible components of enterprize computing environments. While relational database management systems often form the backbone of the BI software stack, the unique modeling and processing requirements of BI applications often make for a relatively awkward fit with RDBMS platforms in general, and their SQL query interfaces in particular. In this paper, we present a new framework for BI/OLAP applications that directly exploits a domain specific conceptual data model. In turn, the new paradigm allows us to support native, client-side OOP querying without the need to embed an intermediate, non-OOP language such as SQL or MDX. A pre-processor essentially translates standard OOP source code into a query grammar developed specifically for BI analysis. The end result is a query facility that is far more intuitive to use, as well as being more amenable to contemporary code development tools. We provide numerous examples to illustrate the flexibility and convenience of the new framework.

1 Introduction

Over the past three decades, relational DBM systems have secured their place as the cornerstone of contemporary data management environments. 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. With respect to operational databases, 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, opportunities to explore new data models, as well as the languages that might exploit them, have emerged.

One particular area of interest is the Business Intelligence/Online Analytical Processing (OLAP) context. 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 enterprize data. Beyond trivial exploitation of BI GUI facilities, however, meaningful analysis can become quite complex and can necessitate a considerable investment of the developer's time and energy.

Of particular significance in this regard is the awkward relationship between the development language and the data itself. Given the relational model of the underlying DBMS, BI querying typically relies upon non-procedural SQL or one of its proprietary derivatives. Unlike transactional databases, however, which are often cleanly modeled by a set-based representation, the nature of BI/OLAP environments argues against the use of such languages. In particular, concepts such as cubes, dimensions, aggregation hierarchies, granularity levels, and drill down relationships map poorly at best to the standard logical model of relational systems.

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:

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

In this paper, we present a comprehensive new data access framework called NOX (Native language OLAP query eXecution) that is specifically tailored to the BI/OLAP domain. Beginning with the specification of an OLAP algebra, we develop a robust query grammar that presents the developer with an Object Oriented representation of the primary OLAP structural elements. The grammar, in turn, is the foundation of a native language query interface that eliminates the reliance on an intermediate, string based embedded language. We illustrate the new design via the Java programming language, and demonstrate how developers can transparently interact with massive, remote data cubes using standard OOP principles and practices. While the underlying compilation and translation mechanism is somewhat complex, virtually all of the framework's sophistication is hidden from the developer. In short, NOX represents a significant step towards "making the OLAP DBMS disappear".

The paper is organized as follows. In Section 2, we present an overview of related work. Section 3 introduces the primary NOX components, while Section 4 discusses the underlying conceptual model. The full details of the client architecture are then presented in Section 5. Future work and final conclusions are provided in Sections 6 and 7 respectively.

2 Related Work

For more than 30 years, Structured Query Language (SQL) has been the defacto standard for data access within the relational DBMS world. Because of its *relative age*, however, numerous attempts have been made to modernize database access mechanisms. Two themes in particular are noteworthy in the current context. In the first case, Object Relational Mapping (ORM) frameworks have been used to define type-safe mappings between the DBMS and the native objects of the client applications. With respect to the Java language, industry standards such as JDO (Java Data Objects) [4], as well as the open source Hibernate framework [10] have emerged. In all cases, however, it is important to note that while the ORM frameworks do provide *transparent persistence* for individual objects, additional string-based query languages such as JDOQL (JDO), or HQL (Hibernate) are required in order to execute joins, complex selections, subqueries, etc. The result is a development environment that often seems as complex as the model it was meant to replace.

More recently, Safe Query Objects (SQO) [12] have been introduced. Rather than explicit mappings, safe queries are defined by a class containing, in its simplest form, a *filter* and *execute* method. Within the filter method, the developer encodes query logic (e.g., selection criteria) using the syntax of the native language. The compiler checks the validity of query types, relative to the objects defined in the filter. The *execute* method is then rewritten as a JDO call to the remote database. The approach is quite elegant, though it can be difficult to accurately model completely arbitrary SQL statements.

In contrast to the ORM models, a second approach extends the development languages themselves. The Ruby language [7], for example, employs *ActiveRecords* to dynamically examine method invocations against the database schema. HaskellDB [5], on the other hand, "decomposes" queries into a series of distinct algebraic operations (e.g., restrict, project) . Microsoft's LINQ extensions (C# and VisualBasic) [11] are also quite interesting in that they essentially integrate the mapping facilities of the ORM frameworks into the language itself (via the ubiquitous SELECT-FROM-WHERE format). It should be noted, however, that none of these language extensions are in any way OLAP-aware.

In terms of OLAP and BI specific design themes, most contemporary research builds in some way upon the OLAP *data cube* operator [15]. In addition to various algorithms for cube construction, including those with direct support for dimension hierarchies [21,19], researchers have identified a number of new OLAP *operators* [9,13], each designed to minimize in some way the relative difficulty of implementing core operations in "raw SQL". There has also been considerable interest in the design of supporting algebras [8,16,20]. The primary focus of this work has been to define an API that would ultimately lead to transparent, intuitive support for the underlying data cube. In a more general sense, these algebras have identified the core elements of the OLAP conceptual data model.

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 [18], though vendor support for these operations in DBMS platforms is inconsistent at best [14]. In addition to SQL, many commercial applications support Microsoft's MDX query language [23]. While syntactically reminiscent of SQL, MDX provides direct support for both multi-level dimension hierarchies and a *crosstab* data model. Still, MDX remains an embedded string based language with an irregular structure and is plagued by the same limitations as those discussed in Section 1.

Finally, we note that query languages such as SQL and MDX are typically encapsulated within a programmatic API that exposes methods for connection configuration, query transfer, and result handling. While relational systems utilize mature standards (e.g., JDBC, ODBC), no definitive API has emerged for OLAP. A recent attempt to do so was the ill-fated JOLAP specification, JSR-69 [3], an industry-backed initiative to define an enterprize-ready, Java-oriented meta data and query framework based upon the Common Warehouse Metamodel [2]. JOLAP proved to be exceedingly complex and, consequently, no viable JOLAP-aware applications were ever developed. At present, the most widely supported API is arguably XML for Analysis (XMLA) [1], a low-level XML/-SOAP mechanism running across HTTP. In practice, XMLA is effectively just a wrapper for MDX, though XMLA result sets are structured in an OLAP-aware format.

3 NOX: Native Language OLAP Query eXecution

To begin, we note that a fundamental design objective for any new query framework or API must be the minimization of the complexity associated with transparent persistence, as the introduction of obscure and non-intuitive design and programming patterns severely limits the likelihood of adoption. We therefore state at the outset that the NOX focus is explicitly on the OLAP/BI domain. In fact, NOX is intended to specifically support higher level analytics servers and is not expected to resolve every "ad hoc" query 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 [22]. Conventional RDMSs, conceptual mapping frameworks, and even JOLAP suffer from this same "curse of generality". 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.

3.1 The NOX Components

Given the preceding objective, NOX has been constructed from the ground up so as to emphasize the *transparency* in the term "transparent persistence". Doing so, of course, requires considerable infrastructure. In the remainder of the paper, we discuss the design, implementation, and use of the NOX framework, using a number of programming examples to illustrate its practical value. Before digging in to the details, however, it is useful to first provide a brief overview of the primary physical and logical elements of the framework. Keep in the mind that the following list includes elements that are both visible and invisible to the developer.

- **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.
- **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 short, the developer's view of the OLAP environment consists solely of the API and the Result Set. More to the point, from the developer's perspective, all OLAP data is housed in a series of cube objects housed in local memory. The fact that these repositories are not only remote, but possibly Gigabytes or even Terabytes in size, is largely irrelevant.

4 Conceptual Model

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

Briefly, we consider analytical environments to consist of one or more *data cubes*. Each cube is composed of a series of d dimensions (sometimes called *feature* attributes) and one or more *measures*. The dimensions can be visualized as delimiting a d-dimensional hyper-cube, with each axis identifying the *members*

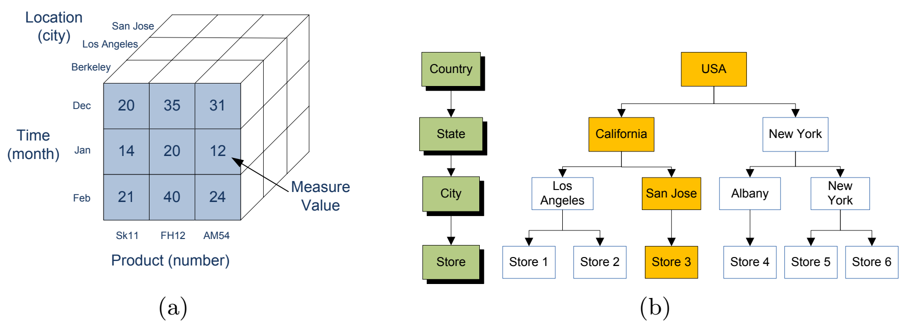

Fig. 1. (a) NOX conceptual query model (b) A simple symmetric hierarchy

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 1(a) provides an illustration of a very simple three dimensional cube. We can see, for example, that 12 units of Product AM54 were sold in the Berkeley location during the month of January (assuming a Count measure).

Beyond the basic cube, however, the conceptual OLAP model relies extensively on aggregation hierarchies provided by the dimensions themselves. In fact, hierarchy traversal is one of the more common and important elements of analytical queries. In practice, there are many variations on the form of OLAP hierarchies [17] (e.g., symmetric, ragged, non-strict). NOX supports virtually all of these, and does so by augmenting the conceptual model with the notion of an arbitrary graph-based hierarchy that may be used to *decorate* one or more cube dimensions. Figure 1(b) illustrates a simple geographic hierarchy that an organization might use to identify intuitive customer groupings.

4.1 OLAP 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. As noted in Section 2, a number of researchers have identified the core operations of such an algebra. 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 common to virtually all proposed OLAP algebras has been identified. Below, we list and briefly describe these operations. Note that we do not provide a formal analysis of the semantics of the algebraic operations, nor their equivalence to the components of the relational algebra, as these issues have been extensively discussed in the original publications.

- **selection:** the identification of one or more cells from within the full *d*-dimensional search space.
- **projection:** the identification of presentation attributes, including *both* the measure attribute(s) and dimension members.

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

Listing 1.1. Core operations of the NOX algebra

- **drill across:** the integration of two independent cubes, each possessing common dimensional axes. In effect, this is a cube "join".
- **union/intersection/difference:** basic set operations performed on two cubes sharing common dimensional axes.
- **change level:** modification of the granularity of aggregation, typically referred to as "drill down" and "roll up".
- **change base:** the addition or deletion of one or more dimensions from the current result.
- **pivot:** rotation of the cube axes to provide an alternate *perspective.*

Several explanatory notes are in order at this stage. First, the **selection** is the driving operation behind most analytical queries. In fact, if suitable defaults are available for the **projection**, many queries can be expressed with nothing more than a selection. Second, the final three operations — **change level**, **change base**, and **pivot** — are distinct from the first six in that each is only relevant as a query against an existing result set. 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. 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 are performed via distinct ETL processes.

4.2 The NOX Grammar

NOX encapsulates the algebra in a formal schema encoded by a Document Type Definition (DTD). The DTD is relatively complex as it effectively represents the foundation for an expressive, XML-based analytics language. Due to space limitations, we do not present the full schema specification here. However, key elements are presented below.

Listing 1.1 defines the core structure of a NOX query. Each query is associated with a single cube (though references to other cubes are possible), as well as a Function List and an Operations List. We do not discuss cube functions extensively in this paper but, for the sake of completeness, we can informally define a cube function as one that is logically associated with a result set, rather than a specific cell or dimension member. The common *top10* function is a simple example.

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

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

Listing 1.2. Selection elements

```
<!ELEMENT UNION (DATA_QUERY)>
<!ELEMENT INTERSECTION (DATA_QUERY)>
<!ELEMENT DIFFERENCE (DATA_QUERY)>
```

Listing 1.3. Set Operations

The Operations List contains the algebraic elements of the query, and each may occur exactly zero or one time in a single query. Given the significance of the `selection` operation, we will look at it in greater detail. Listing 1.2 demonstrates that a `selection` is defined as a listing of one or more dimensions, each associated with an expression. In effect, the expression represents a query restriction on the associated dimension (this will become more clear in Section 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.

Finally, in Listing 1.3, we illustrate the remarkable simplicity of the *set operation* specifications. In effect, set operations are syntactically modeled on an OOP paradigm. 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.

5 Client Side API

Within the NOX framework, the conceptual model and its associated grammar are intended to provide an abstract development environment for expressive analytical programming. In order to provide such an interface, however, supporting client side functionality is required. In a nutshell, NOX provides persistent transparency via a source code re-writing mechanism that interprets the developer's OOP query specification and decomposes it into the core operations of the OLAP algebra. These operations are given concrete form within the NOX grammar and then transparently delivered (via standard socket calls) at run-time to the back-end analytics server for processing. Results are again transparently injected back into the running application and made available through a standard OOP API.

We note at this point that we have chosen to implement the API functionality using external libraries 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. However, it is also due to the fact that while OLAP/BI is an immensely important commercial domain (thereby justifying this work in the first place), OLAP-specific language extensions would have virtually no relevance to the vast majority of developers working in arbitrary domains.

5.1 The NOX Preprocessor

As should be obvious, source code augmentation of this form is non-trivial. In short, NOX must identify query-specific elements of the source code and transform them as required before passing the output to the standard Java compiler. The pre-preprocessor is produced with the JavaCC parser generator and its JJTree Tree builder plug-in [6]. 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 re-written. The output of JJTree is then used by JavaCC to construct a Java parser that actually "walks the parse tree" in order to locate and transform these constructs. We note that although NOX utilizes a complete Java 1.5 grammar for its parser, the pre-processor only examines and/or processes parse 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 1.4 provides a partial listing of its contents. We make note of the following points. First, method names correspond directly to the operations of the algebra/grammar (Note: We currently do not include the **change base**, **change level**, and **pivot** methods in the OlapQuery class as we consider these operations to be manipulations of the Result Set. Their exact implementation is the subject of ongoing research). Second, method bodies have no meaningful implementation, other than a nominal return value (required for successful compilation). In fact, this is true of most client library methods, a fact that makes sense when one realizes that the only code that will actually be executed is the code eventually inserted by the pre-processor. Third, each query method has a return type unique to its own semantic abstraction (the upcoming examples will make this more clear). Fourth, the `execute` method serves as the link between the programmer's conceptual view and NOX's algebraic view. More to the point, it is the `execute` method that will be re-written to include an XML statement corresponding to the specifications of the other methods. The XML string is then "wrapped" in a message that is sent to the server when execute() is invoked in the application. Finally, the OlapQuery is declared *abstract*, though

```
public abstract class OlapQuery {
 public boolean select() {return false;}
 public Object[] project() {return null;}
 public OlapQuery drillAcross() {return null;}
 public OlapQuery union() {return null;}
 public OlapQuery intersection() {return null;}
 public OlapQuery difference() {return null;}

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

Listing 1.4. The OLAP Query class

none of its methods are abstract, a model reminiscent of Java's Adapter classes. 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.

Figure 2 graphically illustrates the process described thus far. In the box at the 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. In terms of the compilation process, the pre-processor take as input the original source file and then, using the parse tree constructed from this source, converts the relevant source elements into an XML decomposition of the OlapQuery. Throughout this process, various DOM utilities and services are exploited in order to generate and verify the XML. Finally, once the source has been transformed, it is run through a standard Java compiler and converted into an executable class file. We note that, in practice, the NOX translation step would be integrated into a build task (ANT, makefile, IDE script, etc.) and would be completely transparent to the programmer.

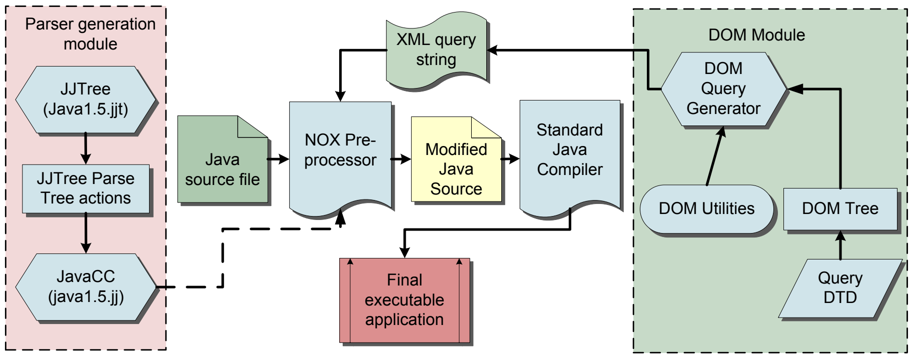

Fig. 2. The client compilation model

```
class SimpleQuery extends OlapQuery {
 public boolean select(){
  DateDimension date = new DateDimension();
  return date.getYear() == 2001;
 }
 //...projection excluded
}

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

Listing 1.5. Simple OLAP Query

5.2 Application Programming

While novel algebras, grammars, and parsing methods are interesting for their own sake, they provide little benefit unless they ultimately lead to a clean, intuitive programming experience for the developer. In this section, we provide a number of examples that demonstrate the practical use of the NOX model.

A Simple Selection. We begin with a query that specifies a simple `selection` criteria, namely that we would like to list total sales for 2001. Listing 1.5 provides the corresponding OlapQuery definition, along with a small `main` method that demonstrates how the query's `execute` method would be invoked. (For simplicity, we will ignore the projection method that would specify the measure and display attributes, as well as the "boilerplate" 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.

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 and every* cell. If the `selection` test evaluates to true, the cell's content is included in the result; if not, it is ignored. In actual fact, of course, the server would almost certainly not resolve a query in this manner. However, that is irrelevant here as our goal is simply to allow the developer to program against an intuitive conceptual model. Once the query is decomposed and sent to the server, the backend DBMS is free to do what it likes.

```
public class CustomerDimension extends OlapDimension {
 private String name;
 private int age;
 CustHierarchy geographicHierarchy;

 public String getName() { return name; }
 public int getAge() { return age; }
 public CustHierarchy getGeographicHierarchy(){
  return geographicHierarchy;
}}
```

Listing 1.6. Simple OLAP dimension

In terms of the decomposition itself, it is of course represented in an XML string generated by the pre-processor (Due to space limitations, we do not reproduce the associated XML here; we simply note that it corresponds directly to the DTD depicted in Listing 1.2). This string is inserted — by the pre-processor — 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 and receive the query and its results. Again, we stress that all of this processing is entirely invisible to the end user.

Manipulating Hierarchies. As previously noted, hierarchical queries are extremely common in OLAP environments. For this reason, much of the current NOX research focuses on extending the expressive capabilities of the framework in this context. With the example below, we give the reader a sense of the NOX philosophy with respect to hierarchical navigation.

Let us assume that we would like to find sales data for older customers from California cities who purchased products in the first half of 2007. Because we now have an arbitrarily defined dimension to restrict (as opposed to the built-in Date dimension), we need a mechanism to statically type-check the relevant dimension attributes so that we can ensure at compile time that all query element are being used appropriately (e.g, integers compared with integers). (Again, we do NOT want to rely on embedded strings like SQL/MDX since type validity could then only be assessed at run-time.) To do this, the programmer simply sub-classes the library-provided OlapDimension class and adds the relevant attributes/types 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 1.6 illustrates this simple approach. Note that CustHierarchy is a simple extension of the NOX OlapHierarchy class.

Now, given this simple Customer class, and a geographic hierarchy corresponding to that of Figure 1(b), we can now discuss the hierarchical query of Listing 1.7. Here, 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

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

   CustHierarchy hierarchy = customer.getGeographicHierarchy();
   OlapPath path= new OlapPath(``USA'', ``California'');

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

Listing 1.7. Manipulating hierarchies

```
class OuterQuery extends OlapQuery{
 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 1.8. Set operations

the expression `hierarchy.includes(path)`? Again, all selection criterion are defined relative to the current cube cell. Logically, this condition simply asks "Is this partial path consistent with the hierarchy members of this cell?" We note that while there are many variations on hierarchy traversal, NOX always uses this same simple approach.

Several additional points are worth noting. First, our NOX objects are fully amenable to standard IDE refactoring methods. For example, should the DB administrator modify the customer `name` field to `cname`, we can directly refactor the attribute name/schema without relying on a tedious and error-prone "find and replace". Second, pre-computation of the query verifies its sematic validity. In other words, while we cannot guarantee that the user's specific selection criteria will actually match any cells in the remote database, we *can* guarantee at compile-time that the query is structurally sound in terms of its use of dimensions, hierarchies, members, etc. Finally, by decomposing the query into its constituent algebraic elements at compile time, we relieve the server of the computational overhead that would normally be done at run-time. Embedded query string APIs — while superficially appealing for trivial queries — simply cannot provide this functionality.

Set Operations. Previously, we showed 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 1.8 provides a simple illustration.

```
class OldQuery extends OlapQuery{
 // ... select method definition
 // ... project method definition
}

class NewQuery extends OldQuery {
 public Object[] project() {
  CustomerDimension customer = new CustomerDimension();
  ProductDimension product = new ProductDimension();
  SalesMeasure measure = new SalesMeasure();

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

Listing 1.9. Over-riding query classes

Here, the programmer defines the "outer" query using the standard `selection` method (and possibly others). In the *intersection* method, the "inner" query (previously defined) is specified merely by returning a reference to the relevant query object. Using this info, the NOX pre-parser can combine both queries into a single XML string corresponding to the nested style of the grammar.

Query Inheritance. One of the reasons that we represent algebraic operations in separate methods is simply because most operations are semantically unique, making it very difficult to combine them 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 for the re-use of previous, often very complex, queries. We saw a simple example of this with the "inner" query above. 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 `projection` method to simply identify a different measure or display attribute. With virtually all current approaches, this would involve cutting and pasting previous chunks of source code, each of which would have to be independently located and updated in the future.

With NOX's distinct query methods, we now have a great deal more latitude in this regard. Listing 1.9 demonstrates how a "new" query extends an "old" one by providing a new `projection` method. Because NOX obeys *inheritance chaining*, it sees that a new projection has been specified, and creates a new query that consists of the `selection` method of the "old" query and the projection method of the "new" query. Any subsequent changes to the source of OldQuery will be automatically integrated into the NewQuery upon re-compilation.

As a final point, this listing also demonstrates the use of the `projection` method itself. Note that its `return` argument is an array of Objects, indicative of its purpose to identify *measure* and *display* attributes (strings, ints, floats, etc). Measures extend the OlapMeasure class and are defined in a manner similar to dimension classes; that is, a list of measures and their associated getters.

5.3 Result Sets

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. The format of the result is again defined by a DTD and is essentially 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 row format that maps cell values to the corresponding axis coordinates. For example, a meta data element defined in the DTD as `(MEMBER_NAME, MEMBER_ID)` would associate a member — say the customer John Smith — with an integer representing the axis offset – say 4. In the *cell data* section of the XML document, this ID would then be embedded within a record of the form `<4,1,2,345.24>`. Assuming a Sales measure and a Customer–Product–Location cube, the row `<4,1,2,345.24>` would indicate that John Smith has purchased $345.24 of Product 1 at Location 2.

Once the XML result is received at the client, it is immediately transformed into a multi-dimensional object. The XML is parsed using the same DOM facilities used to create the original query (albeit with a different DTD). The aforementioned MEMBER_ID values are directly utilized as cube axes coordinates, thereby allowing a linear time population of the Result Set object. 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 meaningless to the end user.

The Result Set API then 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 1.10 shows how one might produce a simple report of all cells in a simple Customer-Product cube, assuming that the `execute` method has already been invoked and an OlapResultSet created. One merely has to retrieve the member values for each dimension and then, with a set of nested `FOR` loops, combine the relevant coordinates for each cell. It should be clear that this is really quite trivial relative to the alternatives (e.g., a JDBC ResultSet model).

```
// ... retrieve lists of dimension members from result object
for (String CustMember: CustList){
 for (String ProdMember: ProdList){
    coordinates = new LinkedList<CubeCoordinate>();
    coordinates.add(new CubeCoordinate(CustDimension,
        CustMember));
    coordinates.add(new CubeCoordinate(ProdDimension,
        ProdMember));
    System.out.println( result.getCellValue(coordinates));
}}}
```

Listing 1.10. Trivial report method

6 Future Work

NOX is already a very large system and is currently the subject of a great deal of ongoing research. Of particular importance at the present time are the following challenges:

– The expansion of the facilities for hierarchical navigation to include more flexible and varied traversal options.
– The enhancement of the ResultSet model to include transparent **change base**, **change level**, and **pivot** operations, as well as paged retrieval of result sets that are either too big or too sparse to be fully encapsulated inside a local array.
– Support for run-time parameterization of query values (i.e., user-defined query parameters). This will likely be done via query constructors, with "stubs" identifying the location for run-time XML augmentation.
– Full integration with the OLAP DBMS. While NOX includes a simple server that validates and parses the final XML query (including all examples in this paper), a parallel project is currently developing an optimized OLAP server that natively understands the algebra of the NOX model. However, even in the absence of such a server, we note that it is entirely possible to convert the NOX output to MDX and deliver it to an XMLA-compliant server.

7 Conclusions

In this paper we have provided a relatively thorough presentation of NOX, the Native language OLAP query eXecution framework. The current version of NOX represents a comprehensive implementation of the native language query model. In building upon the notion of a consistent OLAP conceptual model, we have been able to provide almost fully 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 re-factoring opportunities, and direct Object-Oriented manipulation of Results

Sets. While we chose to target Java in this initial implementation, the fundamental concepts are language agnostic and could easily be applied to other modern OOP languages. Given the awkward, loosely standardized 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.

References

1. XML for Analysis Specification v1.1. (2002), http://www.xmla.org/index.htm
2. CWM, Common Warehouse Metamodel (2003), http://www.cwmforum.org/
3. JSR-69 JavaTM OLAP Interface (JOLAP), JSR-69 (JOLAP) Expert Group (2003), http://jcp.org/aboutJava/communityprocess/first/jsr069/index.html
4. JSR 243: Java Data Objects 2.0 - An Extension to the JDO specification (2008), http://java.sun.com/products/jdo/
5. HaskellDB (2010), http://www.haskell.org/haskellDB/
6. JavaCC, the Java Compiler Compiler (2010), https://javacc.dev.java.net/
7. Ruby programming language (2010), http://www.ruby-lang.org/en/
8. Agrawal, R., Gupta, A., Sarawagi, S.: Modeling multidimensional databases. In: International Conference on Data Engineering (ICDE), Washington, DC, USA, pp. 232–243. IEEE Computer Society, Los Alamitos (1997)
9. Akinde, M.O., Bohlen, M.H.: Efficient computation of subqueries in complex OLAP. In: International Conference on Data Engineering (ICDE), pp. 163–174 (2003)
10. Bauer, C., King, G.: Java Persistence with Hibernate. Manning Publications Co., Greenwich (2006)
11. Blakeley, J.A., Rao, V., Kunen, I., Prout, A., Henaire, M., Kleinerman, C.: .NET database programmability and extensibility in Microsoft SQL Server. In: ACM SIGMOD International Conference on Management of Data, pp. 1087–1098. ACM, New York (2008)
12. Cook, W.R., Rai, S.: Safe query objects: statically typed objects as remotely executable queries. In: International Conference on Software Engineering (ICSE), pp. 97–106 (2005)
13. Cunningham, C., Graefe, G., Galindo-Legaria, C.A.: PIVOT and UNPIVOT: Optimization and execution strategies in an RDBMS. In: International Conference on Very Large Data Bases (VLDB), pp. 998–1009 (2004)
14. Dittrich, J.-P., Kossmann, D., Kreutz, A.: Bridging the gap between OLAP and SQL. In: International Conference on Very Large Data Bases (VLDB), pp. 1031–1042 (2005)
15. Gray, J., Bosworth, A., Layman, A., Pirahesh, H.: Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In: International Conference on Data Engineering (ICDE), Washington, DC, USA, pp. 152–159. IEEE Computer Society, Los Alamitos (1996)
16. Gyssens, M., Lakshmanan, L.V.S.: A foundation for multi-dimensional databases. In: International Conference on Very Large Data Bases (VLDB), pp. 106–115. Morgan Kaufmann Publishers Inc., San Francisco (1997)
17. Malinowski, E., Zimanyi, E.: Hierarchies in a multidimensional model: From conceptual modeling to logical representation. Data Knowl. Eng. 59(2), 348–377 (2006)

18. Melton, J.: Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features. Elsevier Science Inc., New York (2002)
19. Morfonios, K., Ioannidis, Y.: CURE for cubes: cubing using a ROLAP engine. In: International Conference on Very Large Data Bases (VLDB), pp. 379–390. VLDB Endowment (2006)
20. Romero, O., Abelló, A.: On the need of a reference algebra for OLAP. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654, pp. 99–110. Springer, Heidelberg (2007)
21. Sismanis, Y., Deligiannakis, A., Kotidis, Y., Roussopoulos, N.: Hierarchical dwarfs for the rollup cube. In: International Workshop on Data Warehousing and OLAP (DOLAP), pp. 17–24. ACM, New York (2003)
22. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era (it's time for a complete rewrite). In: International Conference on Very Large Data Bases (VLDB), pp. 1150–1160 (2007)
23. Whitehorn, M., Zare, R., Pasumansky, M.: Fast Track to MDX. Springer, New York (2005)