Minimizing the MOLAP/ROLAP Divide: You Can Have Your Performance and Scale It Too

Todd Eavis*
Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
eavis@cs.concordia.ca

Ahmad Taleb
College of Computer Science and Information Systems, Najran University, Najran, Saudi Arabia
amtaleb@nu.edu.sa

Abstract
Over the past generation, data warehousing and online analytical processing (OLAP) applications have become the cornerstone of contemporary decision support environments. Typically, OLAP servers are implemented on top of either proprietary array-based storage engines (MOLAP) or as extensions to conventional relational DBMSs (ROLAP). While MOLAP systems do indeed provide impressive performance on common analytics queries, they tend to have limited scalability. Conversely, ROLAP's table oriented model scales quite nicely, but offers mediocre performance at best relative to the MOLAP systems. In this paper, we describe a storage and indexing framework that aims to provide both MOLAP like performance and ROLAP like scalability by essentially combining some of the best features from both. Based upon a combination of R-trees and bitmap indexes, the storage engine has been integrated with a robust OLAP query engine prototype that is able to fully exploit the efficiency of the proposed storage model. Specifically, it utilizes an OLAP algebra coupled with a domain specific query optimizer, to map user queries directly to the storage and indexing framework. Experimental results demonstrate that not only does the design improve upon more naive approaches, but that it does indeed offer the potential to optimize both query performance and scalability.

Category: Convergence computing

Keywords: Analytics; OLAP; Data warehousing

I. INTRODUCTION

Data warehousing and online analytical processing (OLAP) have been popular targets for researchers over the past 10–15 years, with papers published on a wide variety of related topics. In the OLAP domain, early work often focused on the development of algorithms for the efficient computation of the data cube. Later, the cube methods were expanded to include mechanisms for the computation or representation of hierarchies derived from the cube’s dimensions. In general, academics built upon table-based models, as the associated relational systems were well understood. On the positive side, scalability for relational OLAP (ROLAP) was very impressive and was generally only limited by the underlying hardware. Unfortunately, such systems often provided poor
query performance as they were ill suited to OLAP’s complex, multi-dimensional data model.

For this reason, commercial vendors often developed proprietary array-based server products that were meant to resemble the hyper-cubic nature of the data cube more closely. Performance on these multi-dimensional OLAP (MOLAP) servers was/is indeed impressive as the direct indexing provided by arrays often leads to a much improved query response time. Of course, everything comes at a price and, in the case of MOLAP, scalability remains a concern. Specifically, the sparsity of high cardinality OLAP spaces significantly limits the size of the cube structures in enterprise environments.

In this paper, we discuss the storage architecture for an OLAP-specific server designed from the ground up as a high performance analytics engine. The system is capable of efficiently generating full or partial cubes and subsequently provides complex processing on common cube queries (slice/dice, drill down/roll up, etc.). Until recently, the database management system (DBMS) essentially relied on the file system for storage services. Some indexing was available but was quite limited in nature. Our recent work, as presented in the current paper, has significantly extended the original model to include both R-tree and bitmap indexing facilities. Specifically, we have integrated the open source Berkeley Database (DB) libraries into the server so as to encapsulate both indexes and cube data within a single data store. Cube dimensions are also efficiently stored as Berkeley DB and are, as expected, hierarchy aware. Non-hierarchical attributes, in turn, are stored as a set of FastBit bitmap indexes. Ultimately, the integrated architecture represents a very efficient OLAP storage engine that provides the kind of query performance that one would expect from a MOLAP system, with the scalability typically associated with table-oriented relational servers.

We note that the current paper significantly extends upon an earlier publication [1]. In particular, we have expanded upon our discussion of the storage engine itself, providing a much more thorough presentation of the individual elements of the system. We also extend the presentation of the associated algebra and its relationship to the storage structures. As a result, the motivation for the underlying research should be clearer. Additional experimental results have been included as well. Finally, supplemental material has been added as a series of appendices.

This paper is organized as follows. In Section II, we discuss a number of related research projects. In Section III, we introduce the conceptual model upon which the storage engine is based. A detailed look at how data is encoded is provided in Section IV, including both its abstract and physical representation. The integration with the Berkeley DB and FastBit bitmap libraries is discussed in Section V. Then, in Section VI, we present an overview of the query processing logic that actually utilizes the relevant storage components. Finally, we round out the paper with some conclusions in Section VIII.

II. RELATED WORK

Subsequent to the initial definition of the data cube operator [2], a number of researchers proposed techniques for the compact representation of the cube. Both the Dwarf cube [3] and QC-trees [4], for example, define compact non-relational tree-based structures that provide efficient data access. However, their complex models were never integrated into practical systems, academic or otherwise. Conversely, the CURE cube [5] supports the representation of cubes and dimension hierarchies and does so with relatively compact table storage. Still, the CUBE model lacks the native multi-dimensional indexing schemes that are essential for high performance query functionality.

In terms of OLAP indexing, a number of researchers have proposed methods that would improve warehouse access. In the simplest case, clusters of B-trees have been proposed, though such an approach is neither scalable nor efficient in higher dimensions [6]. A more interesting proposal was perhaps the CUBE tree, a warehouse-specific mechanism based upon the R-tree [7]. The CUBE tree not only demonstrated that the R-tree was well suited to OLAP access patterns, but also provided an efficient update mechanism.

Recently, column store databases have been investigated as a means to minimize IO costs on aggregation queries [8]. It is important to note, however, that column stores are best suited to general purpose warehouses that perform ad hoc, real-time querying involving massive amounts of raw data. True OLAP servers, often working in conjunction with a supporting warehouse (possibly a column store), typically use a combination of pre-aggregation, specialized indexing, and query optimization to target the most common query forms. In practice, OLAP servers and column store DBMSs can be seen as complementary rather than competitive.

A second recent theme has been the exploitation of the increasingly popular MapReduce framework [9]–and its open source implementation Hadoop—as a kind of parallel DBMS subsystem. Integration of Hadoop and traditional relational DBMSs (with their storage and indexing architectures) has also been suggested [10]. While this work is indeed interesting, there remains considerable doubt as to whether such systems can compete directly with the performance offered by purpose-built DW/OLAP servers [11]. At present, MapReduce applications may be best suited to extract-transform-load (ETL) processes or those with large but limited analysis requirements.

Finally, we note that non-academic DBMSs are certainly available. The open source Java-based Mondrian server, for example, provides OLAP query functionality.
Mondrian, however, is primarily an OLAP query application programming interface (API) and actually piggy-backs existing database servers. Commercially, leading vendors such as Microsoft [13] and Oracle [14] also provide warehousing and OLAP applications, often with very rich functionality. Even here, however, users generally have a choice between the scalability of ROLAP or the performance of MOLAP. It is true that hybrid OLAP (HOLAP) promises the “best of both worlds” by incorporating relational tables and array-storage into the same repository but, in practice, this is an awkward, complex configuration at best.

III. THE DATA CUBE MODEL

Before examining the storage framework itself, we briefly discuss the model the DBMS is meant to represent. 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 or count) of the associated members. Fig. 1a provides an illustration of a very simple three-dimensional cube depicting sales totals for various combinations of Employees, Customers, and Products. Here, each unique combination of dimension members represents an aggregation on the measure. For example, we can see that $220$ of Product HJ45 was purchased by Customer CA45 from Employee E105D1 (assuming a Sum measure). Note, as well, that each dimension is associated with a distinct aggregation hierarchy. Customers, for instance, are organized in Country → Province → City groupings. There are in fact many variations on the form of OLAP hierarchies [15] (e.g., symmetric, ragged, non-strict). Regardless of the form, however, traversal of these aggregation paths—typically associated with roll up and drill down operations—is perhaps the single most common query pattern in the OLAP domain.

In practice, the cube is typically modeled as a Star Schema, essentially a central Fact table surrounded by one or more Dimension tables. Fig. 1b demonstrates how the cube would be logically represented in a relational system. Note that the primary keys of the Dimension tables form a composite primary key in the Fact table. Moreover, it is important to keep in mind that the Fact table typically dwarfs the Dimension tables in size. As a result, it is imperative that we minimize both its size and processing costs.

IV. ENCODING THE DATABASE

We now examine how our DBMS actually encodes the contents of the database. For illustrative purposes, we will continue to build on the schema depicted in Fig. 1b. We will assume as well that the actual data associated with the warehouse is consistent with that illustrated in Fig. 2. Note that the ProductNumber, CustomerNumber, and EmployeeNumber values in the Fact table are taken from the primary key columns in the associated Dimension tables. Also, note as well that the Fact table would in practice be hundreds or even thousands of times larger than the Dimension tables.

We begin with an overview of Dimension encoding (i.e., Customers, Employees, and Products). We note that Dimension attributes can be described as either hierarchical or non-hierarchical, with each requiring a distinct representation. In short, hierarchical attributes are those found within a dimension aggregation pathway (e.g., Country → Province → City), while non-hierarchical attributes are descriptive elements typically used to restrict a user query (e.g., ProductName). Encoding of a non-hierarchical dimension—a dimension that does not contain
any hierarchy—is relatively straight-forward and is accomplished with a linear pass through the native data set that simply assigns an incremental surrogate key to the table (i.e., an artificial, integer-base primary key). If we assume, for example, that no hierarchy is defined on the Employee dimension, then the associated Dimension table would be represented by Fig. 3a. Here, we need only add the integer surrogate key, EmployeeID.

Encoding of hierarchical values is more involved and is based upon the notion of hierarchy linearity [16]. Given a hierarchy $A$, we define the hierarchy levels $A_1$, $A_2$, ..., $A_k$, for hierarchy depth $k$, in terms of decreasing granularity. So, for example, $A_1 = \text{city}$ is more granular than $A_2 = \text{province}$. Briefly, we say that a hierarchy on an attribute $A$ is linear if for all direct descendants $A_{j}$ of $A_{i}$ there are $|A_{j}|+1$ values, $x_1 < x_2 < ... < x_{|A_{j}|}$ in the range $1 ... |A_{j}|$ such that:

$$A_{j}[k] = \sum_{l=1}^{k} A_{i}[l]$$

where the array index notation [ ] indicates a specific value within a given hierarchy level. Informally, we can say that if a hierarchy is linear, there is a contiguous range of values $R_{A_{j}}$ on $A_{j}$ that may be aggregated into a contiguous range $R_{A_{i}}$ on $A_{i}$. For example, aggregate totals for the first 93 days of the calendar year would be equivalent to those of the first three months.

The DBMS exploits hierarchy linearity by using mapping tables that represent a sorting of the hierarchical column values of the associated dimension. In effect, values are ordered as $A_1, A_2, ..., A_k$, where $A_1$ is the base attribute in the hierarchical dimension.

For each hierarchical attribute level $L$ in the dimension, a sibling column $L_{ID}$ is added. Values of $L_{ID}$ are created as consecutive integer IDs and are used to delineate the hierarchical group-by levels. Fig. 3b illustrates the mapping table for the Product dimension with the three-level hierarchy ProductNumber $\rightarrow$ Type $\rightarrow$ Category. In the next section, we will see that these mapping tables are used to load hierarchy-aware data structures that provide real-time query value transformations.

Once the Dimension tables have been processed, a pre-
liminary encoding of the Fact data can be undertaken. As with any Fact table, its schema essentially consists of one or more measure attributes, and a set of feature attributes. The feature attributes are, of course, nothing more than the dimensional surrogate keys previously defined. We note at this point that data in the Fact table is represented at the most granular or detailed level of each dimension. As we will see, the DBMS exploits this feature to allow efficient run-time transformations to arbitrarily defined aggregation levels. As well, one must bear in mind that this is still an abstract depiction of the Fact table. Its concrete representation will be discussed below.

A simple example of a Fact structure is shown in Fig. 4a, assuming a single “total sales” measure. For the sake of clarity, the records are listed in sorted order, using a Product-Employee-Customer scheme. In practice, of course, partial aggregates or group-by scan also be computed so as to minimize processing time on common user queries. Fig. 4b depicts the associated Product-Employee cuboid, illustrating how the Customer information is integrated into the more compact summary view.

A. Dimension Table Storage

We must now consider how these Dimension tables are physically stored on a disk. We address hierarchical attributes first. Once the mapping tables have been defined, the encoded values are stored to a disk using the open source Berkeley DB libraries [17]. Specifically, for each sub-attribute in the hierarchy, we create a simple Berkeley database using the Recno access method. The Recno access method is backed with a flat-text file in (key, data) form and provides fast sequential read access. The key in this case is the encoded attribute value (e.g., values of attribute TypeID in dimension Product), while the data has two elements: a native attribute representation (e.g., values of attribute Type from dimension Product) and an integer value that represents the corresponding maximum encoded value in the primary attribute. Fig. 5a illustrates how the hierarchical attributes Type of dimension Product is stored as a Berkeley Recno database. One can see, for example, that the third record in the table represents Interior products and, further, that the maximum ProductID for this product Type is 7 (as per the mapping table of Fig. 3b).

At the DBMS startup, the Berkeley Recno databases are used to initialize a data structure called a mapGraph [16]. Fig. 5b presents a mapGraph sub-structure called a hMap that is used to model our simple Product hierarchy. In effect, the mapGraph is a two-way hashing structure for the hierarchy levels above the base. Specifically, for a sub-attribute (A_j), j ≥ 2, the associated map is made up of
the maximum encoded value from the range on \( A_i \), corresponding to the current encoded value of \( A_i \). We add the native values of \( A_i \) to allow conversion of any encoded value of \( A_i \) to its native value. For example, we can use the Category component to retrieve the encoded value \( l \) of the Automotive category in \( O(1) \) time. Then, we can use the map associated with the Category to find all \( \text{Product IDs} \ (1 \rightarrow 7) \) that are Automotive. Conversely, mapping base level IDs to coarser levels (i.e., base value \( 7 \rightarrow \text{Interior} \) at the Type level) can be done as a \( O(\log n) \) binary search, where \( n \) is the cardinality of the given level (typically quite small relative to the base level). Since data in the associated Fact structure is physically stored at the base level (i.e., the dimension record’s surrogate key), the mapGraph allows extremely efficient run-time mapping in response to the user’s query specification. Specifically, constraints can be transformed between levels at run-time, with the final results aggregated as required.

**Non-Hierarchical Attributes:** While the transparent mapping of hierarchical attribute values is crucial for optimal query performance, non-hierarchical attribute processing must also be efficiently supported. In particular, if a non-hierarchical attribute is used in the restriction of an OLAP query or displayed in an OLAP report (e.g., “All customers older than 40”), then joins between the appropriate group-bys and Dimension tables are required. This process can be very expensive thus its costs must be minimized.

Our DBMS utilizes bitmap indexes for this purpose. For each indexed non-hierarchical attribute we provide a one bit string for each distinct value on the dimension. For \( k \)-non-hierarchical attributes, with each attribute having \( m \) distinct values, we would therefore have \((k \times m)\) bit strings. In practice, compression techniques (typically some variation of Run Length Encoding) significantly minimize storage requirements. Ultimately, the advantage of bitmap indexes for non-hierarchical attributes is that they allow us to identify the surrogate key values (e.g., \( \text{Product ID} \)) matching multiple non-hierarchical column constraints, typically without retrieving any records from the Dimension table itself. In the current context, the DBMS uses the open source FastBit bitmap indexing libraries [18] as its bitmap subsystem.

**B. Fact Structures**

While the Dimension tables, and their indexes in particular, are involved in the resolution of virtually every query, the bulk of both the raw IO and post-processing is associated with the enormous Fact Structures. Earlier, we saw how the DBMS associates each Dimension table record with a surrogate key. It is these integer values, along with the associated measures (e.g., a total sales summation), that are housed within the Fact Structure. Physically, the DBMS stores and indexes data using what is known as a packed R-tree [7], a structure used to cluster spatially related points into common disk blocks, thereby reducing IO overhead for the range queries so common in OLAP. Specifically, the underlying data records are ordered as per the Hilbert space filling curve [19]. Recall that for a \( d \)-dimensional space of side-length \( s \), the \( s^d \)-length Hilbert curve identifies a unique, strictly increasing order on the \( s^d \) point positions. We refer to the numeric representation of a point position as a Hilbert ordinal.

Fig. 6 depicts a simple 2-dimensional space, with each of the 256 points representing potential feature values of two distinct dimensions (e.g., think Product on the vertical axis and Customer on the horizontal axis). Moreover, a Hilbert ordered view can be stored in compressed form, using a technique known as Hilbert tuple differential compression [19]. Here, data records are first sorted in terms of their relative position in the \( H^2 \) space (for side length \( s = 2^d \)). Pairs of adjacent points \( i \leq j \) along the curve may then be represented in integer form as the difference value \( \text{ordinal}_{i} – \text{ordinal}_{j}, i < j \). For example, in Fig. 1b the shaded box (B8) holds two points located at ordinal positions 226 (square) and 247 (diamond) from the \( H^2 \) origin (note that point 226 would reference <Customer 12, Product 3>, while point 247 represents <Customer 14, Product 4>).

With the first point serving as the anchor, the second point is stored as 247 – 226 = 21, or 10101 in binary form. This is 59 bits less that the default encoding (assuming 64-bit integers). When coupled with compaction techniques that strip away leading zeros, differential compression can reduce the need for storage by up to 90%.

It should be clear that the physical representation of the Fact data, while conceptually encoded as a table of records (i.e., ROLAP), bears little resemblance to a traditional table. In essence, it is a block-based collection of compacted bit strings, each representing a specific point in the multi-dimensional data space. Moreover, its supporting Hilbert R-tree index provides rapid retrieval of

![Fig. 6. Hilbert data representation.](image-url)
points in contiguous ranges. In short, it blurs the line between ROLAP and MOLAP by providing some of the best features of both.

V. CUBE CONSOLIDATION

In practice, the DBMS allows the administrator to either fully or partially materialize the $O(2^d)$ summary views or group-bys in the $d$-dimensional space. This can be done to reflect disk space availability or performance constraints. Each group-by effectively represents a subset of the Fact Structure and is physically represented as a pair of files—one that houses the data in Hilbert sort order and one that defines the R-tree index metadata and bounding boxes. Even for a partially materialized cube, this can represent a large number of files that have to be independently managed by the OS and the DB admin. Moreover, these individual files are not databases in any sense of the word and lack even basic mechanisms for caching, locking, ACID compliance, etc.

For this reason, we have chosen to embed the Berkeley DB libraries within the larger DBMS framework. While the Berkeley API offers a number of indexing methods (B-tree, Hash, Recno, Queue), it provides no direct support for R-trees. As such, we have extended the Berkeley C++ interface to allow for the creation and access of Hilbert packed R-trees using standard Berkeley protocols. We note that Berkeley supports the storage of multiple Database Objects in one physical file known as an environment. In the current context, the Berkeley database contains a master B-tree database that, in turn, points to all related group-by metadata, indexes, and tuple compressed data. The extended API transparently routes data access requests as required. In Fig. 7a we see how the DBMS stores, in one physical file, the seven materialized group-bys for the 3-dimensional cube $ABC$ (here, individual letters represent dimension names). For each indexed group-by, the following blocks are required: one block to store the metadata, consecutive blocks to store the data blocks in their Hilbert ordered form, and consecutive blocks to store the Hilbert R-tree index. In this case, 56 contiguous blocks are used in total.

Fig. 7b, on the other hand, provides a more complete illustration of the core components for the storage engine, albeit for a very simple example with just two dimensions. One can see how the Dimension tables are accompanied by a (memory-resident) hMap structure to support hierarchical attributes, multiple bitmaps for non-hierarchical attributes, and a system generated surrogate key. During query resolution (discussed below), strings of surrogate keys pass in/through the query engine to the storage backend. At that point, records can be matched against Hilbert compressed data, using the Berkeley Master B-tree to locate the required view/block combinations.

A. Supporting DBMS Components

While the table storage and indexing components are the focus of the current paper, we note that the DBMS as a whole provides a relatively comprehensive processing stack. Fig. 8a illustrates the full architecture, including the native components as well as the Berkeley extensions. Note that the View Manager is responsible for the identification of the most cost effective group-by—and is initialized by scanning the primary Master B-tree database that contains references to all indexed group-bys—while the Hierarchy Manager builds and maintains the in-memory mapGraph structures. Furthermore, note that OLAP Caching has nothing to do with the Berkeley caching component that stores recently accessed disk blocks, but refers instead to a native, multi-dimensional OLAP query cache.
Finally, we note that the DBMS is in fact a fully parallelized architecture. Specifically, the server has been constructed from the ground up as a “shared nothing” cluster data management system. The processing stack depicted in Fig. 8a, in fact, depicts the software components residing on each node of the cluster. Referred to as sibling servers, each physical node operates more or less independently on a distinct subset of Hilbert striped data. In turn a global Parallel Service Interface (PSI) integrates the physical elements into a single logical DBMS—striping, merging, and aggregating data as required. The full architecture is illustrated in Fig. 8b. In this paper, our focus is purely upon the functionality associated with the individual sibling servers.

VI. QUERY PROCESSING LOGIC

The DBMS provides full query processing functionality. In other words, it must not only parse incoming queries, but it must also decompose those queries into their constituent elements and optimize these operations relative to the supporting indexes and data structures. While the full details of the optimization process are beyond the scope of this paper (and are more fully discussed in [20]), it is nonetheless important in the current context to understand how the storage and indexing facilities are integrating into the query engine. In this section, we examine the logic of the query resolution process as it relates to the work presented above.

A. The Data Model

To begin, we note that the data cube depicted in Fig. 1a can, in fact, be interpreted as a conceptual data model for the OLAP domain. In other words, both its structure (cells, dimensions, and hierarchies) and the operations associated with them (slicing and dicing dimensions, as well as rolling up and drilling down on hierarchies) are directly representative of the intuitive query environment envisioned by end users. Due to the consistency of this model across the OLAP domain, a number of researchers have identified the core operations of a supporting OLAP algebra, including formal analysis showing the algebra to be both closed and complete [21] (note that these operations represent a read-only query algebra, as updates are expected to be performed via system-controlled ETL processes). Below, we list the primary operations of the algebra.

- **SELECTION** \( (\sigma_{\text{cube}}) \): the identification of one or more cells from within the full \( d \)-dimensional search space, providing basic slice and dice functionality.
- **PROJECTION** \( (\pi_{\text{attribute}_1,\ldots,\text{attribute}_n}) \): the identification of presentation attributes, including both measure attributes and feature attributes.
- **DRILL ACROSS** \( (\text{cube}_1 \bowtie \text{cube}_2) \): the integration of two independent cubes, where each cube possesses common dimensional axes (effectively a cube “join”).
- **CHANGE LEVEL** \( (\chi_{\text{base}_1\rightarrow\text{base}_2}) \): the modification of the granularity of aggregation. This process is typically referred to as “drill down” and “roll up”.
- **CHANGE BASE** \( (\phi_{\text{base}_1\rightarrow\text{base}_2}) \): addition or deletion of one or more dimensions from the current result set. Aggregated cell values must be recalculated accordingly.
- **PIVOT** \( (\phi_{\text{base}}) \): rotation of the cube axes to provide an alternate perspective of the cube. No recalculation of cell values is required.
- **UNION** \( (\text{cube}_1 \cup \text{cube}_2) \): union of two cubes sharing common dimensional axes.
- **INTERSECTION** \( (\text{cube}_1 \cap \text{cube}_2) \): intersection of two cubes sharing common dimensional axes.
- **DIFFERENCE** \( (\text{cube}_1 - \text{cube}_2) \): difference of two cubes sharing common dimensional axes.
to accept and optimize native OLAP operations (as described in the next section). This implies, of course, that queries are delivered in a form that is amenable to algebraic representation. The “obvious” way to accomplish this would be to translate SQL/MDX to the core operations of the algebra, as is done by conventional servers (i.e., from SQL to the operations of the relational algebra). Sidera, however, uses an entirely different approach, one that is based upon the notion of native language queries. In other words, queries are defined in the application language (e.g., Java) rather than an embedded query language such as SQL. Ultimately, such a mechanism allows for 1) compile time syntactic and semantic checking, 2) rich refactoring opportunities, 3) the use of Object Oriented features such as query inheritance, and 4) the utilization of a single programming API for both the application and the database backend.

The details of the native language model—known as native language OLAP query eXecution (NOX)—have been described in a previous publication [22]. However, it is important in the current context to at least understand the relationship between the query representation, the optimizer, and the storage engine. We therefore begin with a (very) concise depiction of the NOX query specification and resolution process. Programmers define OLAP queries via a series of Query Object classes provided as part of the NOX API. The OlapQuery class, in particular, identifies a series of Operator Stub methods, one for each of the core algebraic operations (selection, projection, drill across, change level, pivot, etc.). Within these methods, developers may utilize additional Query Objects (e.g., Dimensions, Cells) to constrain the query in arbitrary ways. At compile time, a Java compliant pre-processor (built with JavaCC and JJTree) examines the source code, builds a parse tree, identifies API objects, extracts the query logic, and rewrites the OLAP query as per the XML-encoded grammar natively understood by the DBMS (the grammar is the concrete representation of the algebra). The XML string is then wrapped in a DBMS network call and inserted into the OlapQuery’s execute method. The standard Java compiler translates this final source code into bytecode. At run-time, the DBMS call is invoked and the pre-compiled query is processed, with its results inserted back into a local ResultSet Object. It is important to understand that all of this functionality is entirely transparent to the programmer, who only sees the classes of the NOX API itself.

Listing 1 provides a partial representation of atypical OLAP query—the quantity of items ordered by customers over the age of 40, between May and October of 2007, with the result grouped by Product, Type, and Province—along with a small main method that demonstrates how the query’s execute method would be invoked. We can see that the select method instantiates a built-in DateDimension and invokes its getYear() method. In terms of the selection criterion, note how it is specified simply via a boolean-generating return statement. In fact, the query logic for each operation method utilizes this same mechanism, making it quite trivial to determine the query’s meaning. Note as well that 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 test evaluates to true, the cell is included in the result; if not, it is ignored. In actual fact, the backend DBMS is free to optimize and resolve the query however it likes.

C. The Query Optimizer

Whether the query is generated using the NOX API or is constructed as a translation of a traditional SQL or MDX query, it will eventually be sent to the query optimizer as a sequence of algebraic operations. Listing 2 in Appendix A depicts the translated query (in XML format) associated with Listing 1. Once the query arrives in this form, it is ready to be optimized relative to the structures provided by the storage engine. Algorithm 1 is a somewhat simplified representation of the core logic.

```java
class SimpleQuery extends OlapQuery{
    public boolean select()
    { DateDimension date = new DateDimension();
        TimeHierarchy timeHierarchy = date.getTimeHierarchy();
        Customer customer = new Customer();
        return (customer.getAge() > 40) &&
            ((date.getYear() == 2007 &&
                timeHierarchy.inRange(5, 10));
    }
    public Object[] project()
    { Customer customer = new Customer();
        Product product = new Product();
        Measure measure = new Measure();
        Object[] projections = {
            product.getType(),
            customer.getProvince(),
            measure.getQuantity().ordered();
        }
    }
    public class Demo {
        public static void main(String[] args) {
            DBMS boilerplate connection
            SimpleQuery myQuery = new SimpleQuery("SalesByDate’");
            ResultSet result = myQuery.execute();
            // ... manipulate result set
        }
    }
}

Listing 1. Simple OLAP query.
implemented by the query engine. Queries are transmitted to/from the end user and verified syntactically against the grammar (i.e., XML Schema). Valid queries are then evaluated for semantic correctness to ensure that they comply with the database schema. If so, an algebraic plan is generated, optimized via a set of transformation rules, and then converted into a series of function calls that carry out the plan. It is these execution algorithms that will actually manipulate the indexes and storage structures described in the paper. The View Manager, Hierarchy Manager, and Bitmap Manager are then updated, if necessary, as per the current query parameters (note that they will already have been initialized at the DBMS startup with the schema information associated with the structures described in Section IV). Finally, once the query has been resolved (using buffer pipelining where appropriate), the encoded integer values are converted back into the text-based column values expected by the end user.

D. The Selection Operator

We now turn to the logic implemented by the second FOR loop; that is, the actual data access methods. While each algebraic operator is associated with a distinct set of implementation functions, we will focus here on SELECTION as it is arguably the most important and expensive of the core operations. Given the underlying indexes and storage structures, it is the SELECTION algorithms job to map the user’s query constraints to the Dimension and Fact structures. This happens in two stages. First, hierarchical and non-hierarchical query attributes are converted as required into the base level attributes found in the Fact table. Algorithm 2 describes this process. Using either the mapGraph Hierarchy Manager or the FastBit bitmap indexes, ranges of contiguous base level IDs are extracted. Logical AND or OR operations are applied as required. The end result is an ordered list of Dimension record IDs (i.e., surrogate keys) that may be passed as input to the Fact Structure search algorithm.

Once the DimensionID lists have been generated, they are passed to the cube storage engine to be matched against the Hilbert ordinals of the Fact Structure (note that the View Manager transparently selects the most cost effective group within the Berkeley DB). Given an ordered list of $O(d)$ range sets, the search algorithm traverses the nodes in the selected R-tree based on a breadth first traversal strategy, visiting each node in a level-by-level, left-to-right fashion. Algorithm 3 describes the process. For a level $i$ of the tree, the algorithm identifies at level $i – 1$ the $f$ nodes (block numbers) that intersect the user query. It places these block numbers into a page list $W$. Using the block numbers in $W$, the algorithm traverses the blocks at level $i – 1$ and replaces $W$ with a new list $W'$. This procedure is repeated until the leaf level has been reached. At this point, the algorithm identifies and returns the $d$-dimensional records encapsulated by the user query. Fig. 9 illustrates the traversal logic of the Fact Structure search. Since the Fact table stores dimension attributes as base level record IDs, and because the input to the search algorithm is a set of base level IDs sorted in ascending order,
Minimizing the MOLAP/ROLAP Divide: You Can Have Your Performance and Scale It Too

You can have your cake and eat it too, but why settle for just one? When it comes to data warehousing, the challenge is to have both high performance and scalability. Todd Eavis and Ahmad Taleb explore how to achieve this balance in their paper.

A breadth-first search is able to make a single pass through the table, incrementally adding relevant block IDs to the result list (while it is not obvious in the illustration, the levels of the R-tree index are physically ordered on a disk in this same root-to-leaf fashion). Moreover, because of the explicit Hilbert ordering of data, target records tend to be clustered into a small number of disk blocks. In fact, even when selectivity is very high, the combination of Hilbert ordering and breadth first search implies that, in the worst case, Fact Structure access can be no worse than a sequential scan (and is typically much better).

E. Cost of SELECTION

We now turn to the cost of the SELECTION operation, in terms of its exploitation of the underlying data and storage structures.

Theorem 1. The cost of the SELECTION operation is bounded as the cost of sequentially scanning \( B(V) \) and \( D(V) \), where \( V \) is the appropriate packed R-tree index to answer the SELECTION, \( B(V) \) is the number of index blocks, and \( D(V) \) is the number of disk blocks. Cost = \( B(V) + D(V) \) I/O.

Proof. SELECTION uses the linear breadth first search strategy to retrieve records that satisfy its condition. Linear breadth first search uses a top-to-bottom/left-to-right search pattern for the packed R-tree indexed cube. As was discussed in the previous section, the indexed cube is stored physically on a disk per consecutive disk IDs, using the same top-to-bottom/left-to-right fashion. Also, the data blocks follow this ordering. The worst case is to scan all index blocks and data blocks sequentially. The number of disk I/Os is therefore \( B(V) + D(V) \) blocks.

Of course, processor time should also be considered.

Theorem 2. The worst case processor running time of the SELECTION operator has a bound of \( O(m \times \log(n)) \).

Proof. In the worst case, we scan all index blocks and data blocks of view \( V \) sequentially. For each index block \( b \), we perform a binary search to check if it intersects the selection condition that is stored as a set of sorted arrays. The worst case processor running time for the index scan is \( k \times \log(n) \times B(V) \). Also, in the worst case, for each record (cell) of \( V \) we have to perform a binary search to check if it intersects the selection condition. The worst case running time for the data scan is \( k \times \log(n) \times m \). We therefore have \( k \times \log(n) \times B(V) + k \times \log(n) \times m = k \times \log(n) \times (B(V) + m) \). We note, however, that the number of records \( m \) dominates the number of index blocks, and \( k \) represents a small number of feature attributes. Consequently, in practice, the worst case running time can be bounded as \( O(m \times \log(n)) \).

The full cost of the SELECTION algorithm can be represented as the sum of (a) the disk I/O and (b) the processor running time. We observe, however, that for most queries...
the number of disk I/O dominates the processor time.

VII. EXPERIMENTAL RESULTS

We now turn to the effectiveness of the integrated storage engine. To begin, we note that all evaluations, unless otherwise indicated, are conducted on a Linux-based workstation running a standard copy of the 2.6.x kernel, with 8 GB of main memory and a 3.2 GHz CPU. Disks are 160 GB SATA drives operating at 7200 RPM. The Berkeley DB components are taken from version db4.7.25. Data sets are generated using a custom data generator developed specifically for this environment. We first generate a multi-dimensional Fact table (the dimension count varies with the particular test), with cardinalities arbitrarily chosen in the range of 2–10000. Depending on the test involved, row counts typically vary from 100,000 to 10 million records. The primary Fact tables are then used to compute fully materialized data cubes containing hundreds of additional views or cuboids. For example, a 10-dimensional input set of 1,000,000 records produced a data cube of 1024 views and approximately 120 million total records. Once the cubes are materialized, we index the data using the R-tree and bitmap mechanisms.

Since individual millisecond-scale queries cannot be accurately timed, we use the standard approach of timing queries in the batch mode. In the succeeding tests, five batches of queries are generated and the average run-time is computed for each plotted point. Since query benchmarks are not well standardized for OLAP (the OLAP APB benchmark is effectively dead and TPC-H is better suited to long running, ad hoc warehouse queries), we define our own query classes (Appendix A–C). The queries themselves are typically written in SQL and then translated to a comparable XML representation as required. Finally, we note that when evaluating query performance, we use the “drop caches” option available in the newer Linux kernels to delete the OS page cache between runs.

A. Non-Hierarchical Attributes: FastBit Bitmap versus Standard B-tree

We begin with a comparison of the FastBit indexing subsystem for non-hierarchical attributes versus clusters of standard B-trees (implemented by Berkeley DB). We create a dimension (called Customer) with five non-hierarchical attributes (Age, FirstName, LastName, Balance, and Nationality) and 1,000,000 records (i.e., the cardinality of the primary key CustomerID). The cardinalities of the non-hierarchical attributes were arbitrarily chosen in the range of 100–1000.

We constructed 3 sets of queries against the Customer dimension, with each set containing five queries. The SQL format of two sample queries from each category is given in Fig. 10a. We can see, for example, that Set 1 contains only look-up queries on a single non-hierarchical attribute. Set 2 includes multi-column constraints, while Set 3 consists of range queries with one or more attributes.

Fig. 10b shows a comparison of the running time using the two indexing implementations. For the first set (simple look-up on one attribute), we can see that the running times are actually quite similar. However, when we move to the more complex queries in Set 2 and Set 3, there is a two to three increase in running time for the B-tree indexing method. The difference is primarily due to the efficient bitwise logical operations (AND and OR) directly supported on the compressed FastBit bitmap indexes. (For a higher count of non-hierarchical attributes, the performance of B-trees is quite poor.) We note as well that the physical size of the B-tree indexes in this case is four times greater than the size of the compressed FastBit bitmap indexes—13.8 MB for the B-trees versus 3.5 MB for the bitmaps. Finally, we stress that while B-trees provide
important indexing functionality in many database environments—particularly those requiring single record searches and updates—many of these advantages are not particularly relevant in OLAP environments. Consequently, bitmap performance benefits for complex searches/range queries are a significant comparative strength.

B. Cube Construction

As previously noted, one of the advantages regarding the use of the Berkeley libraries is that its environment construct allows us to encapsulate all views and indexes into a single table space, thereby reducing the burden on the OS (and administrator). We therefore compared cube construction times (indexes and datasets) for the Berkeley environment versus the multi-file approach, as a function of both Fact table size and Dimension count.

In the first test, the full cube ($2^d$ views) was generated from 9-dimensional input sets (i.e., Fact tables) ranging in size from 10,000 to 1,000,000 records. Fig. 11a shows the running time for index cube construction before and after Berkeley DB integration. On average, the integration of Berkeley into our server reduces the index cube construction time by 40% to 60%. The primary reason for this reduction in time is that because the new method uses a single, integrated DB repository, its contiguous block layout allows for very efficient IO, even on larger R-trees. Conversely, use of multiple OS files leads to considerable disk thrashing.

An increase in Dimension count has a similar impact in that each additional dimension effectively doubles the number of views to be computed and stored. In Fig. 11b, we see the results for a dataset of 1 million records and dimension counts of 5, 7, and 9 (common dimension counts in many OLAP environments). Again, we observe that the running time when using Berkeley DB drops by 40% to 60% due to the fact that we are storing the indexed cube in one contiguous physical file.

C. Berkeley Query Resolution

In this case, we create a cube from an input set of 1 million records, 9 non-hierarchical dimensions, and mixed cardinalities in the range of 100–10000, with the full cube representing over 200 million records and 12 Gigabytes of total data. We generate the Hilbert R-tree indexed cube in the following two ways: 1) the Berkeley supported Sidera DBMS in one physical file, and 2) the “standard” Sidera server in 1,024 files (2 files per view). We then use our query generator to generate batches of non-hierarchical queries. By non-hierarchical queries, we mean those queries whose ranges have been restricted to the base attribute.

Fig. 12 demonstrates the total response time for non-hierarchical queries in the Berkeley supported Sidera query engine versus the original Sidera query engine. Results are shown for batches of 100, 500, and 1000 non-hierarchical OLAP queries. The graph shows the improve-
ment from the Berkeley DB integration. Specifically, in all three cases, the integration of the Berkeley code into our Sidera DBMS reduces the OLAP query resolution time by 15% to 20%.

D. Hierarchical Query Performance

As previously noted, the mapGraph module supports various hierarchical dimension patterns (e.g., symmetric and ragged hierarchies) [16]. It is therefore important to compare the performance of the Berkeley supported Sidera DBMS against the current Sidera engine in resolving hierarchical OLAP queries. In this case, we create 9 dimension hierarchies made up of a mixture of symmetric strict, ragged strict and non-strict forms (note that the primary feature of ragged and non-strict hierarchies is that they include optional nodes within the hierarchy graph). We employ batches of 1000 OLAP queries, this time in hierarchical form only. Fig. 13 shows the running time for data sets ranging in size from 10,000 to 1,000,000 records. There are two points that must be made with respect to a direct interpretation of the results. First, answering OLAP hierarchical queries in the Berkeley supported Sidera query engine is faster than the old Sidera query engine by an average of 15%. Second, the total overhead is less than 25% relative to the non-hierarchical case described in the previous section. In short, the integration of the mapGraph manager into Berkeley DB allows it to answer hierarchical OLAP queries with a modest overhead compared to the non-hierarchical case. We note that this overhead is more than acceptable given the power and flexibility that the graph manager provides.

E. R-tree Index Scanning

While indexes are generally quite beneficial when selectivity is low (i.e., a small percentages of records is accessed), performance typically deteriorates quickly as selectivity rises due to the effects of disk thrashing. In fact, selectivity levels beyond 5% of the base dataset typically result in a simple scan out-performing conventional indexes. In practice, this is quite important in OLAP settings as query selectivity can often be quite high for aggregate-based requests. As noted in Section VI-E, however, Sidera attempts to minimize the effects of high selectivity with the combination of a linear breadth first search and a corresponding disk layout strategy. In this test, we therefore look at the running time of queries running against the R-tree indexed data sets versus those resolved via a direct sequential scan. Specifically, we use 12 OLAP queries (equivalent to those queries in Appendix A) to compare the two resolution strategies in the case of both 4-dimensional and 6-dimensional cubes generated from a Fact table of 10 million records. We can see in Fig. 14 that the sequential scanning approach takes roughly four times longer than the R-tree variation. We note, of course, that there is a point beyond which no index can improve upon a sequential scan. In our server this happens when the result of the query exceeds 20% to 25% of the records in the view that is used to answer the query. However the penalty associated with pathologically large queries is small—less the 5%—because of the indexing/search mechanism employed by the DBMS. In practice, this is important not only from a pure performance perspective but also because it significantly minimizes the complexity of query execution planning (i.e., the indexes can be used in all cases, regardless of selectivity concerns).

F. Selection Optimization

The query optimization process builds on a series of algebraic transformation rules that have been optimized for the data and storage structures discussed in this paper. Specifically, the optimizer module manipulates the original query parse tree so as to minimize the cost of query resolution. While the details of the process are discussed in an earlier paper [20], we note that the primary SELEC-
TION transformations include pushing selection operations down the parse tree (i.e., towards the source data sets) and the integration of independent selection predicates. To evaluate the benefit of the optimization process, we execute our query test suite against both the optimized and unoptimized versions of the DBMS, for input data sets of one million and 10 million records. Fig. 15a provides the test results. Here, the elapsed time represents the cumulative time for the full test set. In short, execution time for the un-optimized backend, which can be interpreted as a DBMS that is not able to exploit the physical structures presented in the paper, is roughly an order of magnitude longer than the case for the fully optimized DBMS.

G. Projection Optimization

As noted earlier in the paper, the OLAP algebra consists of a series of core operators, each of which can be optimized independently. For the sake of completeness, we present optimization results for the Projection operator as well, arguably the next most significant OLAP operation. Again, transformation rules (as per [20]) are used to minimize the processing cost of the physical projection operation. Fig. 15b illustrates the performance of the query engine as the number of records in the Fact table varies from one million to 10 million records. The results are in fact quite similar to those obtained when evaluating the Selection operator. Specifically, for un-optimized processing, we see a performance penalty that approaches a factor of ten relative to the fully optimized model.

H. Competitive Query Performance

Of course, the ultimate purpose of a DBMS server is to provide impressive query performance. While it would be possible to only test the system against an artificially defined baseline, a more meaningful comparison can be

![Selection optimization](image)

![Projection optimization](image)

![ROLAP versus sibling](image)

![ROLAP versus sibling](image)
made against existing DBMS servers. Therefore, we have also evaluated the DBMS relative to systems often used in industrial database environments, namely the open source MySQL server (the “lower end”) and Microsoft Analysis Services (the “higher end”). In this case, we generate a 6-dimensional, 10-million record database (using the dimension characteristics described previously) and load it into both DBMS platforms in the standard Star Schema format (the Microsoft server was installed on a Windows partition on the same workstation). Queries were prepared in both SQL format and the native XML form of our own OLAP DBMS. Fig. 16 shows comparative results for both platforms and demonstrates that the MySQL server takes approximately 10–15 times as long to resolve the same queries, while Microsoft Analysis Services—running in the ROLAP mode—is three to six times slower. Note that the term “Sibling Server” refers to a single node of our parallel DBMS.

One can argue, of course, that MOLAP offers superior performance to ROLAP configurations, at least for data sets of this size. So we loaded the same Star Schema data using the MOLAP mode of Microsoft Analysis Services. Fig. 17a shows that MOLAP does indeed outperform our OLAP DBMS by a factor of approximately 5 to 1. However, we note that in this test, our DBMS was not permitted to materialize any additional data; it was essentially just an efficient Star Schema. In Fig. 17b, we see the result once aggregate materialization is added to the Fact Structure (note that production systems would typically use partial cube materialization consisting of the Fact data and a set of low-dimensional group-bys. In practice, this produces a compressed cube database that is not much bigger than the original Fact and Dimension tables). While Microsoft MOLAP server still has a slight advantage, we note that 1) the Microsoft DBMS benefits from years of optimization, and 2) MOLAP is ideally suited to the scale of the current test (i.e., 1–10 million records). Given that our DBMS framework is not constrained by the limits of array-based storage [23], these preliminary results suggest that the current DBMS—and the architecture it represents—does indeed have the potential to provide MOLAP-style performance with ROLAP-style scalability (we note that a number of legacy components in the codebase currently prevent true Terabyte scale testing. However, an ongoing software re-engineering effort is expected to remove these limitations in the coming year).

VIII. CONCLUSIONS

In this paper, we described the storage and indexing architecture of a high performance OLAP DBMS. Current OLAP DBMS platforms generally take an “either/or” (MOLAP/ROLAP) approach to data representation and query processing, with the result being a very clear trade-off between the scalability of relational systems and the performance of array-based platforms. The DBMS described in this paper attempts to build on the best features of both. Specifically, it uses a Fact Structure storage model that is constrained primarily by the disk space available, rather than the sparsity of the cube space. At the same time, the use of compressed Hilbert ordered R-tree indexes, mapGraph mapping tables for hierarchical attributes, and bitmap indexes on non-hierarchical attributes, coupled with a linearized Fact Structure search strategy, produces query performance beyond what one would expect with relational systems. In addition, the DBMS includes a robust query optimization engine that integrates the core storage and indexing facilities with an algebra designed specifically for OLAP environments. Furthermore, a series of experiments confirmed that not only are the storage structures compact (and easily administered), but that query performance is actually comparable to commercial, and far less scalable, MOLAP servers. Given the enormous size of both existing and projected warehouses, we believe that the principles presented in the current paper offer great potential for the OLAP servers of the future.
APPENDIX

A. XML Query Format

The DBMS prototype has been integrated with a native language query interface. Queries written in the client side language (e.g., Java) are translated at compile time into an OLAP-specific grammar and passed to the DBMS at run-time. The grammar is encoded in XML. Listing 2 shows a simple XML query corresponding to the client side query of Listing 1.

B. Evaluation Schema

The test environment utilizes the familiar Star Schema format, in which multiple dimensions represent the primary entities of the organization. The specification for six common dimensions is listed below.

1. Customer
   • Schema:
     (a) C_ID
     (b) Name
     (c) Age
     (d) Country
     (e) Region
   • Size: 1,000,000 records
   • Hierarchy: C_ID → Region → Country

2. Product
   • Schema:
     (a) P_ID
     (b) ProdDesc
     (c) Quantity
     (d) Category
     (e) Type
   • Size: 200,000 records
   • Hierarchy: P_ID → Type → Category

3. Time
   • Schema:
     (a) T_ID
     (b) DayName
     (c) DayOfWeek
     (d) Year
     (e) Quarter
     (f) Month
   • Size: 3,650 records
   • Hierarchy: DayName → Month → Quarter → Year

4. Store
   • Schema:
     (a) S_ID
     (b) StoreName
     (c) StoreState
     (d) StoreCity
     (e) StoreCountry
   • Size: 655 records
   • Hierarchy: StoreName → StoreCity → StoreState → StoreCountry

5. Vendor
   • Schema:
     (a) VendorNumber
     (b) VendorName
     (c) Phone
     (d) CountryName
     (e) StateName
     (f) City
   • Size: 416 records
   • Hierarchy: VendorNumber → City → StateName → CountryName

6. Employee
   • Schema:
     (a) SIN
     (b) FirstName
     (c) LastName
     (d) Phone
     (e) Email
   • Size: 300 records
   • Hierarchy: No hierarchy

C. Sample Queries

Evaluation queries have been designed so as to represent common OLAP query forms. Specifically, database access emphasizes grouping operations, slice and dice analysis, and hierarchical navigation. A series of test samples are included below.

1. SELECT c.Region, SUM(s.Total_Sales)
   FROM customer as c, sales as s
   WHERE s.C_ID = c.C_ID and c.Age = 50 and c.Region = ‘Quebec’
   GROUP BY c.Region

2. SELECT t.Month, SUM(s.Total_Sales)
   FROM time as t, sales as s
   WHERE s.T_ID = t.t_ID and t.Year = 2005 and DayName = ‘Monday’ and t.Quarter = ‘Q1’
   GROUP BY t.Month

3. SELECT p.Type, SUM(s.Total_Sales)
   FROM product as p, sales as s
   GROUP BY p.Type

4. SELECT s.StoreCity, t.Month, SUM(ss.Total_Sales)
   FROM time as t, store as s, sales as ss
   WHERE t.T_ID = ss.T_ID and ss.S_ID = s.S_ID and ((t.Year = 2005 and t.DayName = ‘Monday’) and (t.Quarter = ‘Q1’ or t.Quarter = ‘Q2’)) and s.StoreState = ‘Ontario’
   GROUP BY s.StoreCity, t.Month

5. SELECT c.Region, p.Category, SUM(ss.Total_Sales)
   FROM customer as c, product as p, sales as ss
   WHERE ss.S_ID = c.S_ID and ss.P_ID = p.P_ID and c.Age = 40 and c.Country = ‘Canada’ and...
p.Quantity = 200 and p.Category = 'Automotive'
GROUP BY c.Region, p.Category
6. SELECT c.country, t.Month, SUM(ss.Total Sales)
FROM customer as c, time as t, sales as ss
WHERE ss.C_ID = c.C_ID and ss.T_ID = t.T_ID and
((t.year = 2005 and t.DayName = 'Monday') and
(t.Month = 'May' or t.Month = 'June')) and (c.Age = 40 and c.Region = 'Quebec'))
GROUP BY t.Month, c.country
7. SELECT c.Region, p.Type, s.StoreCity, SUM(ss.Tot.al Sales)
FROM customer as c, product as p, store as s, sales as ss
WHERE ss.C_ID = c.C_ID and ss.S_ID = s.S_ID and
ss.P_ID = p.P_ID and c.Region = 'Ontario' and
s.StoreState = 'Ontario' and p.Category = 'Household'
GROUP BY c.Region, p.Type, s.StoreCity
8. SELECT s.StoreCity, t.Month, SUM(ss.Total Sales)
FROM time as t, customer as c, store as s, sales as ss
WHERE ss.C_ID = c.C_ID and ss.T_ID = t.T_ID and
and ss.S_ID = s.S_ID and t.Month = 'May' or t.Month = 'June') and (c.Age = 40 and c.Region = 'Quebec') and s.StoreState = 'Ontario'
GROUP BY c.StoreCity, t.Month
9. SELECT t.Quarter, p.Type, s.StoreCity, SUM(ss.Total Sales)
FROM time as t, customer as c, store as s, sales as ss
WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and
GROUP BY t.Quarter, p.Type, s.StoreCity
10. SELECT t.Quarter, c.Region, p.Type, s.StoreCity, SUM(ss.Total Sales)
FROM time as t, customer as c, product as p, store as s, sales as ss
WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and
GROUP BY c.Region, p.Type, s.StoreCity, t.Quarter
11. SELECT p.Type, s.StoreState, SUM(ss.Total Sales)
FROM time as t, customer as c, product as p, store as s, sales as ss
WHERE t.T_ID = ss.T_ID and ss.C_ID = c.C_ID and
and ss.P_ID = p.P_ID and t.Year = 2005 and s.StoreState = 'Ontario' and p.Type = 'Engine'
GROUP BY t.Quarter

Listing 2. XML query.
REFERENCES


Todd Eavis

Todd Eavis received his Ph.D. in Computer Science from the Faculty of Computer Science, Dalhousie University, Halifax, Canada, in 2003. Following an NSERC postdoctoral fellowship at Carleton University, he moved to Concordia University in Montreal, where he is currently an Associate Professor in the Department of Computer Science and Software Engineering. His current research interests include data warehousing architectures, parallel algorithms, cloud and cluster computing, and multi-dimensional indexing.

Ahmad Taleb

Ahmad Taleb obtained a B.Sc. in Computer Science in 2004 from Ajman University of Science and Technology in Ajman, United Arab Emirates (UAE). He then completed his Master's and Ph.D.'s degrees in 2007 and 2011, respectively, from Concordia University in Montreal, Canada, under the supervision of Dr. Todd Eavis. Currently, Dr. Taleb is an Assistant Professor in the College of Computer Science and Information Systems at Najran University in Saudi Arabia. His primary research interests include the design, implementation, and evaluation of data structures and algorithms that can be used to dramatically reduce processing times for complex real world queries in business Intelligence environments.