

# BEYOND SQL: MULTI-DIMENSIONAL MDX AND XML XQUERY

*William A. Wimsatt, Wells Landers Group*

Oracle V2 from Relational Software, Inc (now Oracle Corp.), introduced commercial SQL in 1979. SQL has served us well over the years. Oracle has offered extensions for multi-dimensional structures, extensions for XML, but there are other languages that are specialized for different storage structures. Two such languages are XQuery for XML structure and MDX for multi-dimensional structures. There are many resources available on the Internet and books on store shelves that can provide in-depth understanding. This paper will provide some basic information to get the reader started on an exploration of data query languages beyond SQL.

An XML document is essentially an outline in which order and hierarchy are the two main structural units. XQuery is based on the structure of XML and leverages this structure to provide query capabilities for the same range of data that XML stores. To be more precise, XQuery is defined in terms of the XQuery 1.0 and XPath 2.0 Data Model [XQ-DM], which represents the parsed structure of an XML document as an ordered, labeled tree in which nodes have identity and may be associated with simple or complex types. Note that the data model used by XQuery is quite different from the classical relational model, which has no hierarchy, treats order as insignificant, and does not support identity. XQuery is a functional language—instead of executing commands as procedural languages do, every query is an expression to be evaluated, and expressions can be combined quite flexibly with other expressions to create new expressions. XQuery 1.0 became a W3C Recommendation on January 23, 2007.

The Multidimensional Expressions (MDX) emerged circa 1998, when it first began to appear in commercial applications. MDX was created to query OLAP databases, and has become adopted within the realm of analytical applications. MDX forms the language component of OLE DB for OLAP, and was designed by Microsoft as a standard for issuing queries to, and exchanging data with, multidimensional data sources.

## MDX

MDX is principally associated with Microsoft Analysis Services; however, there are many platforms that use MDX such as:

- SAP BW (BAPI)
- Essbase
- Cognos ReportNet
- BusinessObjects MDX Connect

Interestingly, none of these are Oracle environments. So, does MDX have any play in and Oracle environment? The answer is definitively – Yes! MDX in this paper is used to query Oracle via an open source project called Mondrian. Mondrian has recently entered the portfolio of Pentaho and has been renamed Pentaho Analysis Services.

MDX syntax appears, at first glance, to be remarkably similar to the syntax of Structured Query Language (SQL). In many ways, the functionality supplied by MDX is also similar to that of SQL; with effort, you can even duplicate some of the functionality provided by MDX in SQL. This paper only scratches the surface of MDX cube declaration and MDX queries. There is a lot of information available on the Internet for more information.

OLAP is a way of looking at data through rotating the data as if it were a n-sided rubik's cube. MDX is a query language designed for querying data in multi-dimensional data structures. Star schema facts and dimensions are natural pivot points for MDX.

Suppose you were logging sales in Oracle, you could have a table SALES which had VALUE, CUSTOMER, PRODUCT, TIMEOFSALE, SALESREP, STORE, COUNTRY etc. If you wanted to know how many sales each sales rep had made you might use:

```
select s.salesrep, count (*) as 'total'
from sales s
group by s.salesrep
```

A fairly typical query. But, suppose you wanted to report on something else at the same time? You might want to see a cross-tab report of SALESREP by PRODUCT. This is messy to write in SQL as you have to use case or decode statements for each value of whichever field you want on the columns. Typically, you end up joining to the same table multiple times which is not very efficient.

```
SELECT *
FROM (SELECT job,
             sum(decode(deptno,10,sal)) DEPT10,
             sum(decode(deptno,20,sal)) DEPT20,
             sum(decode(deptno,30,sal)) DEPT30,
             sum(decode(deptno,40,sal)) DEPT40
      FROM scott.emp
      GROUP BY job)
ORDER BY 1;
```

Using Oracle's CUBE function provides more functionality:

```
SELECT job,
       NVL(sum(decode(deptno,10,sal)),0) DEPT10,
       NVL(sum(decode(deptno,20,sal)),0) DEPT20,
       NVL(sum(decode(deptno,30,sal)),0) DEPT30,
       NVL(sum(decode(deptno,40,sal)),0) DEPT40,
       SUM(SAL) TOTAL
      FROM emp
      GROUP BY CUBE(job)
ORDER BY 1
```

## Comparing SQL and MDX

SQL:

- Is Inherently 2 -Dimensional
- Has Semantics Strongly Tied to Normalized Data Model
- Requires Explicit Grouping to Aggregate
- Has limited Concept of Relativity, Hierarchies or Time as Special
- Has No Concept of Expressions Independent of Queries

MDX:

- Is Inherently N -Dimensional
- Has Semantics Strongly Tied to Navigating in N -Dimensional Space
- Aggregates Implicitly
- Provides Relativity, Hierarchies and Time as Core Concepts
- Allows Expressions Independent of Queries
- Uses the Terms "Tuple" and "Set" Differently Than SQL

The principal difference between SQL and MDX is the ability of MDX to reference multiple dimensions. Although it is possible to use SQL exclusively to query dimensional database models, MDX provides commands that are designed specifically to retrieve data as multidimensional data structures with almost any number of dimensions.

SQL refers to only two dimensions, columns and rows, when processing queries. Because SQL was designed to handle only two-dimensional tabular data, the terms "column" and "row" have meaning in SQL syntax.

MDX, in comparison, can process one, two, three, or more dimensions in queries. Because multiple dimensions can be used in MDX, each dimension is referred to as an axis. The terms "column" and "row" in MDX are simply used as aliases for the first two axis dimensions in an MDX query; there are other dimensions that are also aliased, but the alias itself holds no real meaning to MDX. MDX supports such aliases for display purposes; many OLAP tools are incapable of displaying a result set with more than two dimensions.

MDX for Oracle requires Pentaho Analysis Services (formerly Mondrian). Mondrian is a Java application that takes the MDX query generates the appropriate SQL and then returns the data. Mondrian is typically fronted with a display product such as JPivot. JPivot provides a visual interaction environment for multidimensional queries.

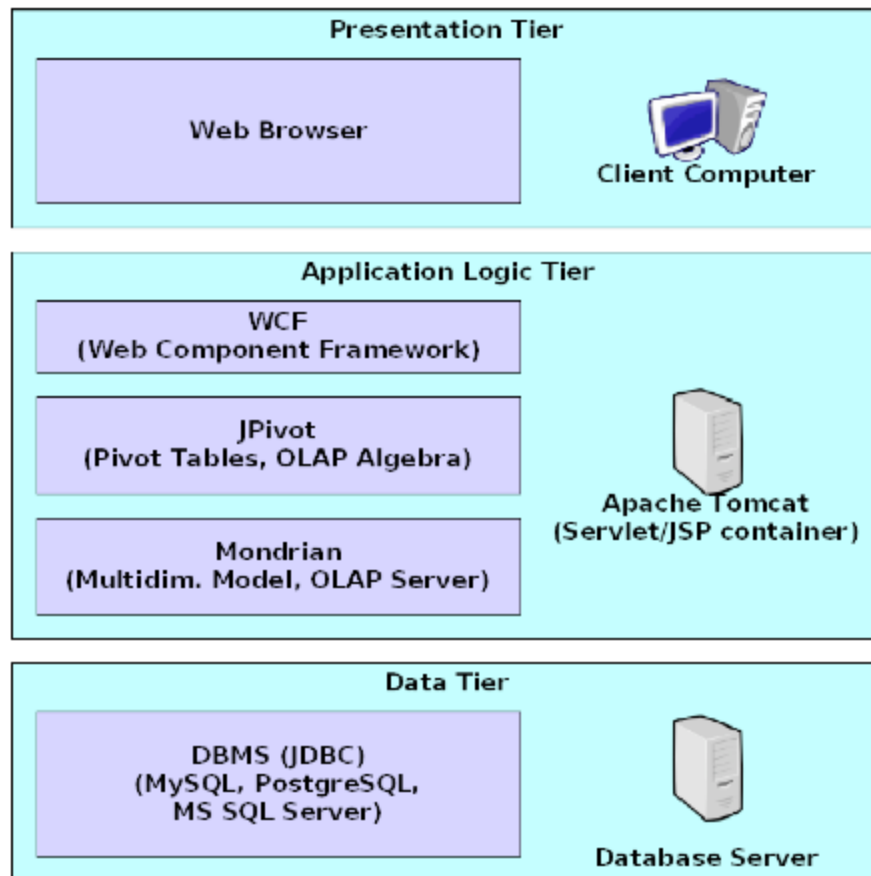


Figure 1 Mondrian Deployment Architecture

## Terminology

An introduction to the rest of the terminology may be useful at this point:

Dimension

A dimension is one of the fields that you want to report on. Dimensions have a tree structure, allowing complicated data to be reported on at different levels. For instance TIMEOFSALE could be a dimension if you wanted to report on it

Measure

What we are actually reporting on, be it sum, count or average.

Level

A level is a step along a dimension. In TIMEOFSALE we could have YEAR, MONTH, DAY and HOUR allowing us to report on sales per hour or products year on year.

Member

A member is a value of a level. In [timeofsale].[Year] we might have [2001], [2002], [2003], or in [timeofsale].[Month] we have [March], [April], [May] etc

Hierarchy: groups members of a dimension

Member-Properties: additional information

Slice

We may want to cross tab by two fields for some specific value of a third, for instance [timeofsale] by [product] for a particular [salesrep].

When we picture the report as a cube we think of this filter as a slice into the cube, to show the values on a new face.

## MDX Queries

So that's the lexicon done, now how do we use it? Well here is the structure of a statement

```
select
{set 0} on axis(0) , /* this would be a block comment */
{set 1} on axis(1) , // this is a line comment
...
{set n} on axis(n)
from [cube]
where (tuple)
```

So if we wanted a report of [product] on columns by [salesrep] on rows we would execute:

```
select
( ( [product].[productname].[product1] ) ,
 ( [product].[productname].[product2] ) ) on columns ,
{ [salesrep].[repname]. members } on rows
from [sales]
```

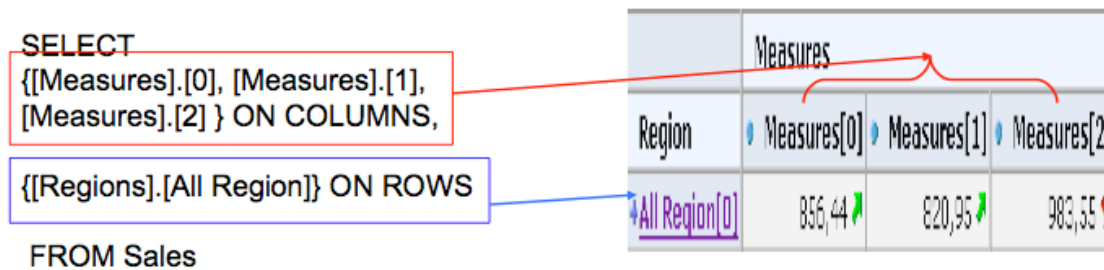


Figure 2 MDX Query Components

On the columns I have a set with two tuples from the same dimension. The () brackets are not required in this case because each tuple contains just one member. The {} are required. The rows has a function . members , which returns a set with all the member of that level it. As . members returns a set we don't need the {} brackets but again I've put them in.

Here is another one:

```
select
{ [product].[productname].[product1] : [product].[productname].[product20] } on columns ,
{ [timeofsale].[Year].[2002]. children } on rows
from [sales]
```

In this example I've used a **range** to give me a set of all the products inclusive between [product1] and [product20] on columns. On rows I've used another function called . children to give me all the months in [timeofsale].[Year].[2002]

.members works on a level to give all the members at that level. .children works on a member to give all the members below it (assuming there are any).

Two more useful features before we look at slices:

```
select
non empty { [product].[productname]. members } on columns ,
{ { [timeofsale].[Year].[2002]. children }
*
{ [salesrep].[repname]. members } } on rows
from [sales]
```

First of all the keyword “non empty” excludes any values from that axis where no values are returned. The \* operator does a cross join between the two sets, and works in a similar way to a cross join in sql. The final set will be made up of every possible combination of the tuples in the two sets.

Now we will add a slice:

```
select
{ [product].[productname]. members } on columns ,
{ [timeofsale].[Year].[2002]. children } on rows
from [sales]
where ( [salesrep].[repname].[Mr Sales Rep1] )
```

Note that the where criteria requires a tuple rather than a slice and that tuple cannot contain any of the same dimensions as the sets on the axis.

## MDX Cube Declaration

The following declaration is based on Mondrian (Pentaho Analysis Services) MDX. Tools that generate MDX for Microsoft do not always produce code that will work with Pentaho Analysis Services.

A cube definition is described in an .xml file. It has a first tag <Schema>. This is the logical schema definition for the multi-dimensional structure. Within this tag is a <Cube> which is a named collection of measures, dimensions and hierarchies. Note that the <cube> tag is a logical name and not related to a physical table.

Fact tables are defined using the <Table> tag. More advanced structures can be created using the <View> and <Join> tags. The following MDX cube definition is a fairly simple declaration of a multi-dimensional structure using three physical tables: sales\_fact\_1997, customer, time\_by\_day. Note the reference to the relational table physical structure.

```
<Schema>
  <Cube name="Sales">
    <Table name="sales_fact_1997"/>
    <Dimension name="Gender" foreignKey="customer_id">
      <Hierarchy hasAll="true" allMemberName="All Genders" primaryKey="customer_id">
        <Table name="customer"/>
        <Level name="Gender" column="gender" uniqueMembers="true"/>
      </Hierarchy>
    </Dimension>
    <Dimension name="Time" foreignKey="time_id">
      <Hierarchy hasAll="false" primaryKey="time_id">
        <Table name="time_by_day"/>
      </Hierarchy>
    </Dimension>
  </Cube>
</Schema>
```

```

        <Level name="Year" column="the_year" type="Numeric" uniqueMembers="true"/>
        <Level name="Quarter" column="quarter" uniqueMembers="false"/>
        <Level name="Month" column="month_of_year" type="Numeric" uniqueMembers="false"/>
    </Hierarchy>
</Dimension>
<Measure name="Unit Sales" column="unit_sales" aggregator="sum" formatString="#,###"/>
<Measure name="Store Sales" column="store_sales" aggregator="sum" formatString="#,###,###"/>
<CalculatedMember name="Profit" dimension="Measures" formula="[Measures].
[Store Sales]-[Measures].[Store Cost]"/>
    <CalculatedMemberProperty name="FORMAT_STRING" value="$#,###0.00"/>
</CalculatedMember>
</Cube>
</Schema>

```

This schema contains a single cube, called "Sales". The Sales cube has two dimensions, "Time", and "Gender", and two measures, "Unit Sales" and "Store Sales".

## Dimension Definitions

Dimensions are the qualitative factors of the model. MDX provides for direct column mappings, hierarchies (single and multiple), user declared dimension tables, time dimensions, and shared dimension definitions. The constructs can get quite complex and Pentaho Analysis Services is a stickler for syntax and case sensitivity. There is a cube modeler that can be downloaded, but it is only beta and it does not work well (as of version 0.7).

Here is a very simple dimension definition:

```

<Dimension name="Gender" foreignKey="customer_id">
  <Hierarchy hasAll="true" primaryKey="customer_id">
    <Table name="customer"/>
    <Level name="Gender" column="gender" uniqueMembers="true"/>
  </Hierarchy>
</Dimension>

```

The <dimension> element identifies the name of the dimension and the foreign that is used to join the fact table. You can have flat tables without FK declarations, but that will be discussed a bit later. This dimension is based on the table CUSTOMER. This is declared in the <table> element. Finally, this dimension consists of a single hierarchy, which consists of a single level called Gender.

The values for the dimension come from the gender column in the CUSTOMER table. The gender column contains two values, 'F' and 'M', so the Gender dimension contains the members [Gender].[F] and [Gender].[M].

In this case, the gender dimension is the gender of the customer who made a particular purchase. Given the MDX cube definition above, this is expressed by joining from the fact table "sales\_fact\_1997.customer\_id" to the dimension table "customer.customer\_id".

## Hierarchies

A dimension is joined to a cube by means of a pair of columns, one in the fact table, the other in the dimension table. The <Dimension> element has a foreignKey attribute, which is the name of a column in the fact table; the <Hierarchy> element has primaryKey attribute.

If the hierarchy has more than one table, you can disambiguate using the primaryKeyTable attribute.

The uniqueMembers attribute is used to optimize SQL generation. If you know that the values of a given level column in the dimension table are unique across all the other values in that column across the parent levels, then set uniqueMembers="true", otherwise, set to "false". For example, a time dimension like [Year].[Month] will have uniqueMembers="false" at the Month level, as the same month appears in different years. On the other hand, if you had a [Product Class].[Product Name] hierarchy, and you were sure that [Product Name] was unique, then you can set uniqueMembers="true". If you are not sure, then always set uniqueMembers="false". At the top level, this will always be uniqueMembers="true", as there is no parent level.

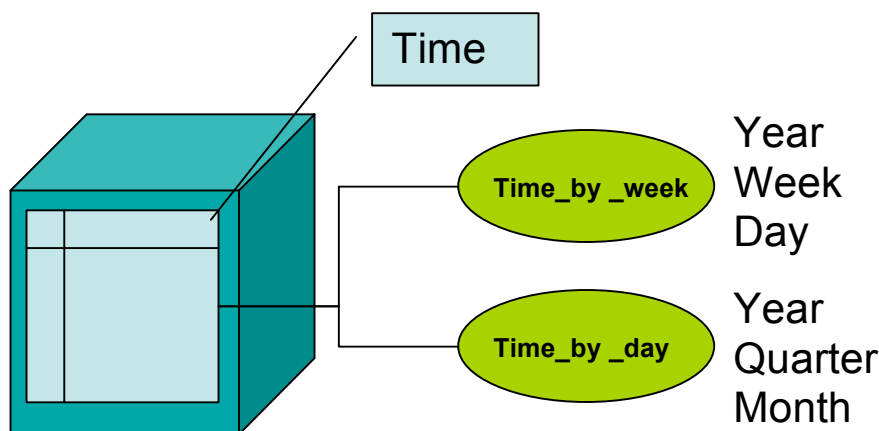
By default, every hierarchy contains a top level called '(All)', which contains a single member called '(All {hierarchyName})'. This member is parent of all other members of the hierarchy, and thus represents a grand total. It is also the default member of the hierarchy; that is, the member which is used for calculating cell values when the hierarchy is not included on an axis or in the slicer. The allMemberName and allLevelName attributes override the default names of the all level and all member.

If the <Hierarchy> element declares hasAll="false", the 'all' level is suppressed. The default member of that dimension will now be the first member of the first level; for example, in a Time hierarchy, it will be the first year in the hierarchy. Changing the default member can be confusing, so you should generally use hasAll="true".

**MULTIPLE HIERARCHIES**

A dimension can contain more than one hierarchy:

```
<Dimension name="Time" foreignKey="time_id">
  <Hierarchy hasAll="false" primaryKey="time_id">
    <Table name="time_by_day"/>
    <Level name="Year" column="the_year" type="Numeric" uniqueMembers="true"/>
    <Level name="Quarter" column="quarter" uniqueMembers="false"/>
    <Level name="Month" column="month_of_year" type="Numeric" uniqueMembers="false"/>
  </Hierarchy>
  <Hierarchy name="Time Weekly" hasAll="false" primaryKey="time_id">
    <Table name="time_by_week"/>
    <Level name="Year" column="the_year" type="Numeric" uniqueMembers="true"/>
    <Level name="Week" column="week" uniqueMembers="false"/>
    <Level name="Day" column="day_of_week" type="String" uniqueMembers="false"/>
  </Hierarchy>
</Dimension>
```



**Figure 3 Multiple Time Hierarchies**

Notice that the first hierarchy doesn't have a name. By default, a hierarchy has the same name as its dimension, so the first hierarchy is called "Time".

These hierarchies don't have much in common — they don't even have the same table! — except that they are joined from the same column in the fact table, "time\_id". The main reason to put two hierarchies in the same dimension is because it makes more sense to the end-user: end-users know that it makes no sense to have the "Time" hierarchy on one axis and the "Time Weekly" hierarchy on another axis. If two hierarchies are the same dimension, the MDX language enforces common sense, and does not allow you to use them both in the same query.

**Time dimensions**

Time dimensions have type="TimeDimension". The role of a level in a time dimension is indicated by the level's levelType attribute, whose allowable values are as follows:

levelType value	Meaning
TimeYears	Level is a year
TimeQuarters	Level is a quarter
TimeMonths	Level is a month
TimeDays	Level represents days

Here is an example of a time dimension:

```
<Dimension name="Time" type="TimeDimension">
  <Hierarchy hasAll="true" allMemberName="All Periods" primaryKey="dateid">
    <Table name="datehierarchy"/>
    <Level name="Year" column="year" uniqueMembers="true" levelType="TimeYears" type="Numeric"/>
    <Level name="Quarter" column="quarter" uniqueMembers="false" levelType="TimeQuarters" />
    <Level name="Month" column="month" uniqueMembers="false" ordinalColumn="month" nameColumn="month_name"
levelType="TimeMonths" type="Numeric"/>
    <Level name="Week" column="week_in_month" uniqueMembers="false" levelType="TimeWeeks" />
    <Level name="Day" column="day_in_month" uniqueMembers="false" ordinalColumn="day_in_month" nameColumn="day_name"
levelType="TimeDays" type="Numeric"/>
  </Hierarchy>
</Dimension>
```

Notice that in the time hierarchy example above the <Level> element contains ordinalColumn and nameColumn attributes. These effect how levels are displayed in a result. The ordinalColumn attribute specifies a column in the Hierarchy table that provides the order of the members in a given Level, while the nameColumn specifies a column that will be displayed.

For example, in the Month Level above, the datehierarchy table has month (1 .. 12) and month\_name (January, February, ...) columns. The column value that will be used internally within MDX is the month column, so valid member specifications will be of the form: [Time].[2005].[Q1].[1]. Members of the [Month] level will displayed in the order January, February, etc.

Levels contain a type attribute, which can have values "String", "Integer", "Numeric", "Boolean", "Date", "Time", and "Timestamp". The default value is "Numeric" because key columns generally have a numeric type. If it is a different type, the MDX server needs to know this so it can generate SQL statements correctly; for example, string values will be generated enclosed in single quotes:

```
WHERE productSku = '123-455-AA'
```

Time dimensions based on year/month/week/day are coded differently in the MDX schema due to the MDX query time related functions such as:

ParallelPeriod	Returns a member from a prior period in the same relative position as a specified
----------------	---

	<p>member. This function is handy because it offers a quick way to compare an indicator across two identical time series. Check that the arguments you are passing it really are a level (not a member), numeric expression and member.</p> <p>Syntax</p> <p>&lt;Member&gt; ParallelPeriod()                  &lt;Member&gt; ParallelPeriod(&lt;Level&gt;)                  &lt;Member&gt; ParallelPeriod(&lt;Level&gt;, &lt;Numeric Expression&gt;)                  &lt;Member&gt; ParallelPeriod(&lt;Level&gt;, &lt;Numeric Expression&gt;, &lt;Member&gt;)</p>
PeriodsToDate	<p>Returns a set of periods (members) from a specified level starting with the first period and ending with a specified member.</p> <p>Syntax</p> <p>&lt;Set&gt; PeriodsToDate()                  &lt;Set&gt; PeriodsToDate(&lt;Level&gt;)                  &lt;Set&gt; PeriodsToDate(&lt;Level&gt;, &lt;Member&gt;)</p>
WTD	<p>A shortcut function for the PeriodsToDate function that specifies the level to be Quarter.</p> <p>Syntax</p> <p>&lt;Set&gt; Qtd()                  &lt;Set&gt; Qtd(&lt;Member&gt;)</p>
MTD	<p>A shortcut function for the PeriodsToDate function that specifies the level to be Quarter.</p> <p>Syntax</p> <p>&lt;Set&gt; Qtd()                  &lt;Set&gt; Qtd(&lt;Member&gt;)</p>
QTD	<p>A shortcut function for the PeriodsToDate function that specifies the level to be Quarter.</p> <p>Syntax</p> <p>&lt;Set&gt; Qtd()                  &lt;Set&gt; Qtd(&lt;Member&gt;)</p>
YTD	<p>A shortcut function for the PeriodsToDate function that specifies the level to be Quarter.</p> <p>Syntax</p> <p>&lt;Set&gt; Qtd()                  &lt;Set&gt; Qtd(&lt;Member&gt;)</p>
LastPeriod	<p>Returns a set of members prior to and including a specified member.</p> <p>Syntax</p> <p>&lt;Set&gt; LastPeriods(&lt;Numeric Expression&gt;)                  &lt;Set&gt; LastPeriods(&lt;Numeric Expression&gt;, &lt;Member&gt;)</p>

**Degenerate dimensions**

A degenerate dimension is a dimension which is so simple that it isn't worth creating its own dimension table. For example, consider following the fact table:

product_id	time_id	payment_method	customer_id	store_id	item_count	dollars
55	20040106	Credit	123	22	3	\$3.54
78	20040106	Cash	89	22	1	\$20.00
199	20040107	ATM	3	22	2	\$2.99
55	20040106	Cash	122	22	1	\$1.18

and suppose we created a dimension table for the values in the payment\_method column:

payment_method
Credit
Cash
ATM

This dimension table is fairly pointless. It only has 3 values, adds no additional information, and incurs the cost of an extra join.

Instead, you can create a degenerate dimension. To do this, declare a dimension without a table, and Mondrian will assume that the columns come from the fact table.

```
<Cube name="Checkout">
  <!-- The fact table is always necessary. -->
  <Table name="checkout">
    <Dimension name="Payment method">
      <Hierarchy hasAll="true">
        <!-- No table element here.
        Fact table is assumed. -->
        <Level name="Payment method"
        column="payment_method" uniqueMembers="true" />
      </Hierarchy>
    </Dimension>
  <!-- other dimensions and measures -->
</Cube>
```

Note that because there is no join, the foreignKey attribute of Dimension is not necessary, and the Hierarchy element has no <Table> child element or primaryKey attribute.

## Measure Definitions

Each measure has a name, a column in the fact table, and an aggregator—usually "sum", but "count", "mix", "max", "avg", and "distinct count". Factless facts are generated using "count". An optional formatString attribute specifies how the value is to be printed \$#,##0.00 is a two decimal format request. The datatype attribute specifies how cell values are represented in Mondrian's

cache, and how they are returned via XML for Analysis

We can write an MDX query on this schema:

```
SELECT {[Measures].[Unit Sales], [Measures].[Store Sales]} ON COLUMNS,
      {[Time].[1997].[Q1].descendants} ON ROWS
FROM [Sales]
WHERE [Gender].[F]
```

This query refers to the Sales cube ([Sales]), each of the dimensions [Measures], [Time], [Gender], and various members of those dimensions.

Measures can be more than just column mappings. Measures can contain a SQL expression to calculate its value. The measure "Promotion Sales" is an example of this.

```
<Measure name="Promotion Sales" aggregator="sum" formatString="#,###.00">
  <MeasureExpression>
    <SQL dialect="generic">
      (case when sales_fact_1997.promotion_id =
        0 then 0 else sales_fact_1997.store_sales end)
    </SQL>
  </MeasureExpression>
</Measure>
```

In this case, sales are only included in the summation if they correspond to a promotion sales. Arbitrary SQL expressions can be used, including subqueries. However, the underlying database must be able to support that SQL expression in the context of an aggregate. Variations in syntax between different databases is handled by specifying the dialect in the SQL tag.

## XQuery

XQuery provides the means to extract and manipulate data from XML documents or any data source that can be viewed as XML, such as relational databases or office documents.

XQuery uses XPath expression syntax to address specific parts of an XML document. It supplements this with a SQL-like "FLWOR expression" for performing joins. A FLWOR expression is constructed from the five clauses after which it is named: FOR, LET, WHERE, ORDER BY, RETURN.

The language also provides syntax allowing new XML documents to be constructed. Where the element and attribute names are known in advance, an XML-like syntax can be used; in other cases, expressions referred to as dynamic node constructors are available. All these constructs are defined as expressions within the language, and can be arbitrarily nested.

The language is based on a tree-structured model of the information content of an XML document, containing seven kinds of node: document nodes, elements, attributes, text nodes, comments, processing instructions, and namespaces.

The type system of the language models all values as sequences (a singleton value is considered to be a sequence of length one). The items in a sequence can either be nodes or atomic values. Atomic values may be integers, strings, booleans, and so on: the full list of types is based on the primitive types defined in XML Schema.

XQuery 1.0 does not include features for updating XML documents or databases; it also lacks full text search capability. These features are both under active development for a subsequent version of the language.

XQuery is a programming language that can express arbitrary XML to XML data transformations with the following features:

- Logical/physical data independence
- Declarative
- High level
- Side-effect free
- Strongly typed language

## Examples

```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>

  <book year="1992">
    <title>Advanced Programming in the UNIX Environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>

  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>65.95</price>
  </book>

  <book year="1999">
    <title>The Economics of Technology and Content
```

```

    for Digital TV</title>

    <editor>
    <last>Gerbarg</last>
    <first>Darcy</first>
    <affiliation>CITI</affiliation>

    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>

</book>

</bib>

```

The most commonly used operators in path expressions locate nodes by identifying their location in the hierarchy of the tree. A path expression consists of a series of one or more steps, separated by a slash, /, or double slash, //. Every step evaluates to a sequence of nodes. For instance, consider the following expression:

```
doc("books.xml")/bib/book
```

This expression opens books.xml using the doc() function and returns its document node, uses /bib to select the bib element at the top of the document, and uses /book to select the book elements within the bib element. This path expression contains three steps. The same books could have been found by the following query, which uses the double slash, //, to select all of the book elements contained in the document, regardless of the level at which they are found:

```
doc("books.xml")//book
```

Predicates are Boolean conditions that select a subset of the nodes computed by a step expression. XQuery uses square brackets around predicates. For instance, the following query returns only authors for which last="Stevens" is true:

```
doc("books.xml")/bib/book/author[last="Stevens"]
```

If a predicate contains a single numeric value, it is treated like a subscript. For instance, the following expression returns the first author of each book:

```
doc("books.xml")/bib/book/author[1]
```

Note that the expression author[1] will be evaluated for each book. If you want the first author in the entire document, you can use parentheses to force the desired precedence:

```
(doc("books.xml")/bib/book/author)[1]
```

XQuery is a functional language consisting entirely of expressions. There are no statements, even though some of the keywords appear to suggest statement-like behaviors. To execute a function, the expression within the body gets evaluated and its value returned. Thus to write a function to double an input value, you simply write:

```
declare function local:doubler($x) { $x * 2 }
```

To write a full query that says Hello World you write the expression:

```
"Hello World"
```

## Combining Data via XQuery

FLWOR expressions, pronounced "flower expressions," are one of the most powerful and common expressions in XQuery. They are similar to the SELECT-FROM-WHERE statements in SQL. However, a FLWOR expression is not defined in terms of tables, rows, and columns; instead, a FLWOR expression binds variables to values in for and let clauses, and uses these variable bindings to create new results. A combination of variable bindings created by the for and let clauses of a FLWOR expression is called a tuple.

For instance, here is a simple FLWOR expression that returns the title and price of each book that was published in the year 2000:

```
for $b in doc("books.xml")//book
where $b/@year = "2000"
return $b/title
```

This query binds the variable \$b to each book, one at a time, to create a series of tuples. Each tuple contains one variable binding in which \$b is bound to a single book. The where clause tests each tuple to see if \$b/@year is equal to "2000," and the return clause is evaluated for each tuple that satisfies the conditions expressed in the where clause. In our sample data, only Data on the Web was written in 2000, so the result of this query is

```
<title>Data on the Web</title>
```

A where clause eliminates tuples that do not satisfy a particular condition. A return clause is only evaluated for tuples that survive the where clause. The following query returns only books whose prices are less than \$50.00:

```
for $b in doc("books.xml")//book
where $b/price < 50.00
return $b/title
```

Here is the result of this query:

```
<title>Data on the Web</title>
```

A where clause can contain any expression that evaluates to a Boolean value. In SQL, a WHERE clause can only test single values, but there is no such restriction on where clauses in XQuery. The following query returns the title of books that have more than two authors:

```
for $b in doc("books.xml")//book
let $c := $b//author
where count($c) > 2
return $b/title
```

Here is the result of the above query:

```
<title>Data on the Web</title>
```

## Joins: Combining Data Sources with for and where Clauses

A query may bind multiple variables in a for clause in order to combine information from different expressions. This is often done to bring together information from different data sources. For instance, suppose we have a file named reviews.xml that contains book reviews:

```
<reviews>
<entry>
<title>TCP/IP Illustrated</title>
<rating>5</rating>
```

```
<remarks>Excellent technical content. Not much plot.</remarks>
</entry>
</reviews>
```

A FLWOR expression can bind one variable to our bibliography data and another to the reviews, making it possible to compare data from both files and to create results that combine their information. For instance, a query could return the title of a book and any remarks found in a review.

A Cartesian product of two sets contains all possible combinations of the items in those sets. When a where clause is used to select interesting combinations from the Cartesian product, this is known as a join. The following query performs a join to combine data from a bibliography with data from a set of reviews:

```
for $t in doc("books.xml")//title,
    $e in doc("reviews.xml")//entry
where $t = $e/title
return <review>{ $t, $e/remarks }</review>
```

The result of this query is as follows:

```
<review>
<title> TCP/IP Illustrated</title>
<remarks>Excellent technical content. Not much plot.</remarks>
</review>
```

In this query, the for clauses create tuples from the Cartesian -product of titles and entries, the where clause filters out tuples where the title of the review does not match the title of the book, and the return clause constructs the result from the remaining tuples. Note that only books with reviews are shown. SQL programmers will recognize the preceding query as an inner join, returning combinations of data that satisfy a condition.

## Applications

Below are a few examples of how XQuery can be used:

- Extracting information from a database for a use in web service.
- Generating summary reports on data stored in an XML database.
- Searching textual documents on the Web for relevant information and compiling the results.
- Selecting and transforming XML data to XHTML to be published on the Web.
- Pulling data from databases to be used for the application integration.
- Splitting up an XML document that represents multiple transactions into multiple XML documents.

## XQuery vs SQL

Interestingly XQuery follows relational algebra just as SQL; however, the approach is quite a bit different. The standard functions of relational algebra (abbreviation of Codd's relational algebra) are as follows:

- SELECTION – In set-oriented operations on content, predicates filter millions of records down to a few relevant content items (SQL WHERE clause)
- PROJECTION – Present to the user only the information necessary. This is useful for both efficiency and quality of information eliminating extraneous information. (SQL SELECT clause)
- JOIN – Process two sets as one. (SQL WHERE predicate clause)
- SORT – Order contents of set (SQL ORDER BY clause)

**XQuery Comparison**

- + Already exists and supports this extended relational model
- + Supports XPath
- + Supported by Oracle, IBM and Microsoft databases
- + Existing open source implementations for those who don't have it
- Not yet a standard (but probably will be)
- Not very easy to formulate
- Immature development bindings usually resulting in DOM manipulation (not very efficient)
- Difficult language to optimize which may result in poor performance
- Limited developer community
- Limited developer tools
- It's currently a read-only language and the update additions are not focused on transactional systems

**SQL Comparison**

- + Millions of developers know (and love) it
- + Most repositories already support it
- + Supports PROJECTION, JOIN and UNION
- + Long established standard
- + Efficient language bindings with JDBC and ODBC
- + Lots of development tools
- Need to specify and implement HIERARCHY ENUMERATION
- Need to work around the multi-value problem with PROJECTION and SELECTION functions (non 1<sup>st</sup> Normal Form)

**Conclusion**

XQuery and MDX offer specialized data processing capabilities beyond tried and true SQL. XQuery provides XML data selection, and filtering. Oracle offers XQuery as a part of Oracle 10g release 2. MDX is a language that is only provided by external libraries and tools. MDX is a very powerful multidimensional cube definition and query language. It provides facilities, that although are possible in SQL, designed to optimize the developer and user experience.