DBA 4: Analyzing Data (PDF)

Introduction
DBA 4: Analyzing Data Lesson 1

How to Learn Using O'Reilly Learning Courses

Welcome to DBA 4, the final course in the O'Reilly DBA Series!

In this course you'll learn how to take your MySQL knowledge to the next level. We'll assume you have worked through the first three courses in the series, are familiar with MySQL, and understand data warehousing concepts. Feel free to revisit the previous courses at any time to refresh your memory about any concepts we learned earlier.

In this course we'll cover *MDX*, or **M**ulti**d**imentional E**x**pressions, an industry-standard way to query data warehouses.

But before we dive in, let's go over the OST learning philosophy and format once more. At OST we believe that the best way to learn new technology is to play with suggested code and experiment as much as possible. As you go through the course, we encourage you to experiment with the code given in the lesson examples. At the end of each lesson you'll be assigned a project or quiz to complete. This will be your opportunity to engage with the code, learn new skills, and fulfill the most important course requirement--to have fun!

The more you experiment, the more you learn. Our learning system is designed to encourage experimentation and help you *learn how to learn*. Here are some tips for using our system effectively:

- **Learn in your own voice**

  Work through your unique ideas and trust your instincts in order to learn these new skills. We want you to facilitate your own learning, so we avoid lengthy video or audio streaming, and keep spurious animated demonstrations to a minimum.

- **Take your time**

  Learning takes time. Rushing through the work can actually slow your progress. Take time to try out new things and you'll really learn the material.

- **Create your own examples and demonstrations**

  In order to understand a complex concept, you need to understand its various parts. We'll help you do that, by offering guidance as you create a demonstration project, piece by piece.

- **Experiment with your ideas and questions**

  You're encouraged to wander from the path often to explore possibilities! We can't possibly anticipate all of your questions and ideas, so it's up to you to experiment and create on your own.

- **Accept guidance, but don't depend on it**

  Try to work through any difficulties you run into on your own before seeking help. Grappling with problems and eventually solving them yourself is the best way to learn any new skill. Our goal is for you to use the technology independent of us. Of course, you can always contact your mentor if you need help.

- **Create real projects**

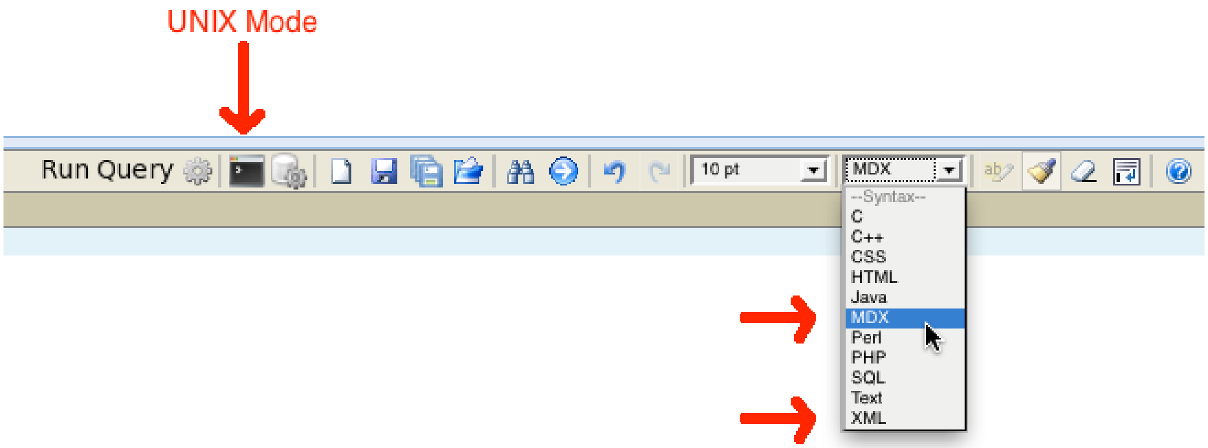
  Real projects are more meaningful and rewarding to complete than simulated projects. Working on real projects will help you to understand what's involved in real world situations. After each lesson you'll be given objectives or quizzes so you can test your new knowledge.

- **Have fun!**

  Relax, keep practicing, and don't be afraid to make mistakes! There are no deadlines in this course, and your instructor will keep you at it until you've mastered each skill. We want you to experience that satisfied *I'm so cool! I did it!* feeling. And when you're finished, you'll have some really cool projects to show off.

Understanding the Learning Sandbox Environment

In this course we'll be using the MDX, Unix, and XML modes in CodeRunner:



CODE TO TYPE:
We'll ask you to type code that you'll see in white boxes like this, into CodeRunner. Every time you see a white box, it's your cue to experiment.

Similarly, when we want you just to observe some code or result, we'll put it in a gray box like this:

OBSERVE:
Gray boxes will be used for observing code. You are not expected to type anything from these "example" boxes.

What is MDX?

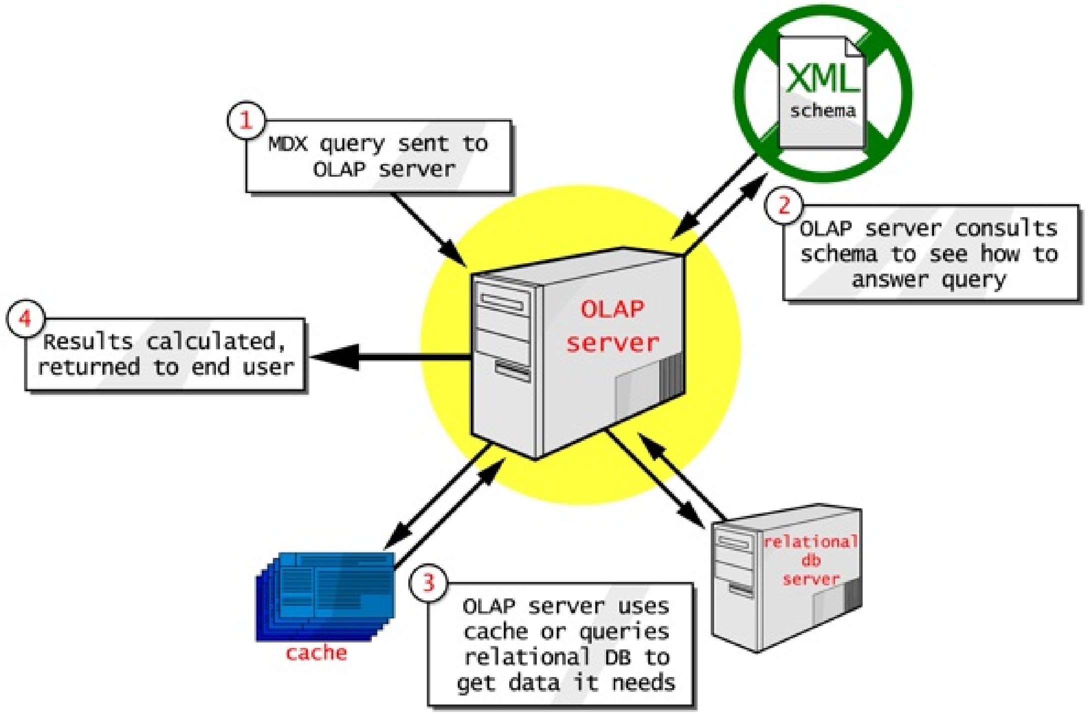
MDX is a query language originally developed by Microsoft for its *Analysis Services* component of SQL Server. Analysis Services is Microsoft's OLAP (online analytical processing) server. Since its release in 1997, MDX has been embraced by many other data warehouse vendors, including the open source OLAP server Mondrian. In this course we will use *Mondrian*, however the core concepts also apply to other OLAP servers such as *Analysis Services*.

At the end of the DBA 3 course, we learned how to query the data warehouse using SQL. So why would we want to learn a different way to query? Well, you may recall that there are several potential pitfalls we could encounter when using SQL. We discussed two of them in DBA 3 -- **Bad Joins** and **Incorrect Filtering**. While no computer language can keep you from making any mistakes, MDX makes it *much* less likely that these problems will occur.

The syntax of MDX makes answering common data warehouse questions such as, "*In 2008, how did the east region's sales compare to those of the west region?*," less complicated. The same query would be lengthy and complex using SQL.

An MDX query runs like this:

1. An MDX query is sent to the OLAP server.
2. The OLAP server consults its *schema* to see how it can answer the query.
3. The OLAP server may answer the query using its own data structures or cache, or it will generate SQL queries in order to retrieve the data it needs to answer the query.
4. Results are computed and returned to you.



MDX consists of *Multi-dimentional Expressions*. The MDX structure incorporates some of the concepts we learned in the last DBA course. It is comprised of a warehouse of multiple *dimensions* (such as date and customer), that surrounds *facts* (such as sales).

We'll learn MDX in the first part of this course. In the second part, we will learn more about the *schema*, the model that is used to map multi-dimensional databases with the underlying SQL database.

Your First MDX Query

For this course we'll use the sample database included with Mondrian, called **foodmart**. At the heart of **foodmart** is a traditional data warehouse that's just like the one we developed in the last course. But **foodmart** contains some extra tables that are used to demonstrate features found in Mondrian. We'll examine those tables in the second half of the course.

For now, don't worry about the specifics of the MDX queries below. We will explain them in detail over the next lessons. Right now we're only interested in comparing SQL to MDX.

Let's get started. Suppose we want to know the total sales from 1997. First, we'll write the SQL query that will return this information. Switch to **Unix** mode in CodeRunner, and connect to the `foodmart` database. At the Unix prompt, run the this command:

CODE TO TYPE:

```
cold:~$ mysql -u anonymous -h sql foodmart
```

In the `foodmart` data warehouse, sales information is kept in a fact table called `sales_fact`, and the date dimension is called `time_by_day`. To find the total sales for 1997, we'll write a short query. At the MySQL prompt, type this code:

CODE TO TYPE:

```
mysql> select sum(store_sales) from sales_fact sf
join time_by_day t on (sf.time_id=t.time_id)
WHERE t.the_year=1997;
```

If you typed everything correctly, you'll see these results:

OBSERVE:

```
mysql> select sum(store_sales) from sales_fact sf
    -> join time_by_day t on (sf.time_id=t.time_id)
    -> WHERE t.the_year=1997;
+------------------+
| sum(store_sales) |
+------------------+
|      565238.1300 |
+------------------+
1 row in set (0.73 sec)
```

```
mysql>
```

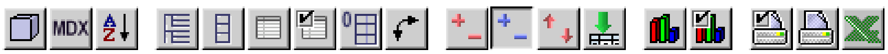
Now let's switch back to the Editor, and to MDX mode. In MDX Mode, type this query:

CODE TO TYPE:

```
select
  { [Measures].[Store Sales] } ON COLUMNS
from Sales
where ([Time].[1997])
```

Note Pay special attention when writing MDX queries. Both parentheses **()** and curly brackets **{}** are used in MDX, and they are not always interchangeable.

When you're done, click on the **Run Query** button. You'll see a new window that looks like this:



Measures
Store Sales
565,238.13

Slicer: [Year=1997]

We got exactly the result we wanted without worrying about underlying tables, joins, or whether `SUM()` was the proper way to get the result.

Now suppose we wanted to see sales for 1998 only. Let's try to retrieve that information using SQL first; switch back to Unix mode. At the MySQL prompt, run this query:

CODE TO TYPE:

```
mysql> select sum(store_sales) from sales_fact sf
join time_by_day t on (sf.time_id=t.time_id)
WHERE t.the_year=1998;
```

Sales were really up in 1998!

OBSERVE:

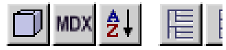
```
mysql> select sum(store_sales) from sales_fact sf
    -> join time_by_day t on (sf.time_id=t.time_id)
    -> WHERE t.the_year=1998;
+-------------------+
| sum(store_sales)  |
+-------------------+
|     1199308.3100  |
+-------------------+
1 row in set (1.54 sec)
```

Now let's try to get that same information using MDX. Switch back to the editor. In MDX Mode, write the following query:

CODE TO TYPE:

```
select
  { [Measures].[Store Sales] } ON COLUMNS
from Sales
where ([Time].[1998])
```

Mondrian responds with your answer:



Measures
Store Sales
1,199,308.31

Slicer: [Year=1998]

Now suppose you want to see two columns of data: Sales for 1997 and Sales for 1998. How would you do this in SQL?

One way would be to use a *CASE* statement. Switch back to Unix mode. At the MySQL prompt, run this query:

CODE TO TYPE:

```
mysql> select sum(case when t.the_year=1997 then store_sales else null end) as `Sales 1997`,
sum(case when t.the_year=1998 then store_sales else null end) as `Sales 1998`
from sales_fact sf
join time_by_day t on (sf.time_id=t.time_id);
```

Though our query is starting to get complex, it does return the answers we want.

OBSERVE:

```
mysql> select sum(case when t.the_year=1997 then store_sales else null end) as `Sales 1997`,
    -> sum(case when t.the_year=1998 then store_sales else null end) as `Sales 1998`
    -> from sales_fact sf
    -> join time_by_day t on (sf.time_id=t.time_id)
    -> ;
+-------------+--------------+
| Sales 1997  | Sales 1998   |
+-------------+--------------+
| 565238.1300 | 1199308.3100 |
+-------------+--------------+
1 row in set (2.48 sec)

mysql>
```

So, how would this work in MDX? I'm glad you asked! In MDX Mode, write this query:

CODE TO TYPE:

```
select
  { ( [Time].[1997]:[Time].[1998] ) * [Measures].[Store Sales] } ON COLUMNS
from Sales
```

Once again, Mondrian answers our query:

Time	
+1997	+1998
Measures	Measures
Store Sales	Store Sales
565,238.13	1,199,308.31

Slicer:

At this point you're probably wondering, "*So what does this all mean? How does this all work?*"

Excellent questions! In the next lesson we'll start looking at MDX from the ground up to begin answering them. See you there!

MDX From the Ground Up
DBA 4: Analyzing Data Lesson 2

The Basics

Welcome back!

In the last lesson we saw our first MDX queries. In this lesson, we'll discuss what makes MDX unique and compare it to SQL.

Note	In MDX the term *measure* is commonly used instead of the term *fact*. In this course we will use the term *measure* instead of *fact*. For our purposes, these terms are synonymous with one another.

What is a Cube?

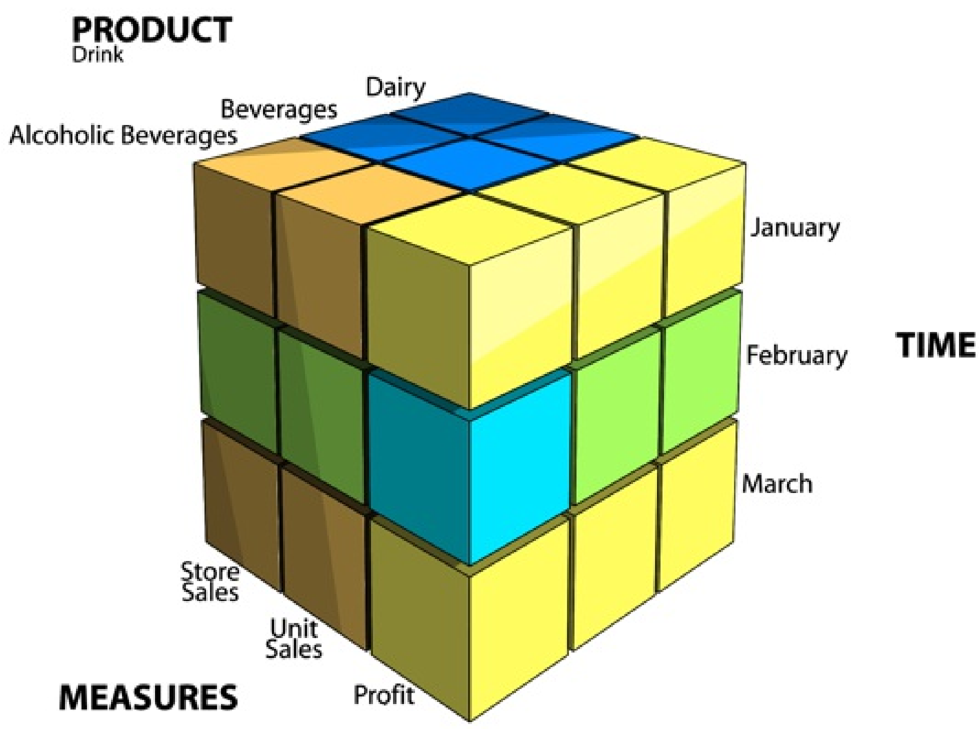
In the DBA 3 course, we learned how to organize our measures and dimensions to create a data warehouse, in order to answer questions like, "*How much profit did we have on alcoholic beverages last month?*" We rewrote that question in a slightly different form to extract the **facts** and **dimensions**: "***How much profit did we have by product and by month****?*"

Note	In the last course we learned that data warehouses usually have a **date** dimension. The warehouse we are using in this course also has a **date** dimension, but it is called **Time**.

We used a *star schema* to relate our measures and dimensions in our relational data warehouse. This structure in the MDX world is known as a *cube*.

Why is it a *cube*? Imagine a [Rubik's cube](#) - a large cube made of of small boxes. The cube is three dimensional, and two physical dimensions map to two of our data warehouse dimensions such as **Product** and **Time**. The last physical dimension maps to our measure - **profit**.

Here's a visual representation of our MDX cube:



If we want to determine the **profit** from **alcoholic beverages** sold in **February**, we would find the box that represents the intersection of **profit**, **alcoholic beverages**, and **February**. This is sometimes called *slicing* or *dicing* the cube - imagine cutting open the Rubik's cube to retrieve the single box you are interested in examining.

In the MDX world, dimensions and measures are not so different. In fact, the collection of measures comprise a special dimension that's called **Measures**.

Switch the editor to MDX mode and write this query:

CODE TO TYPE:

```
select {[Measures].[Store Sales]} ON COLUMNS,
  {[Product].[Drink].[Alcoholic Beverages]} ON ROWS
from [Sales]
 where [Time].[1997].[Q1].[2]
```

If you typed the query correctly, you'll see this result:

	Measures
Product	Store Sales
+Alcoholic Beverages	919.27

Slicer: [Month=2]

Each of the physical dimensions on the cube correspond to an *axis* in MDX. In our query we specified three axes. The first two were displayed to us: **COLUMNS** and **ROWS**. The third is known as the **slicer** axis - it *slices* our cube, but is not displayed in the results. In our query, our **Time** dimension was our **slicer** axis.

MDX doesn't care how you specify your axes. You're free to change your rows, columns, and slicer as needed. Let's see how that works. In MDX Mode, write the following query:

CODE TO TYPE:

```
select {[Measures].[Store Sales]} ON ROWS,
  {[Time].[1997].[Q1].[2] } ON COLUMNS
from [Sales]
 where [Product].[Drink].[Alcoholic Beverages]
```

Our result might look a little strange, but MDX doesn't mind:

	Time
Measures	2
Store Sales	919.27

Slicer: [Product Department=Alcoholic Beverages]

The previous example used two axes. The MDX specification allows you to use more than two axes, however more than two axes are tough to visualize on 2-D computer screens. Many tools (such as ours) throw an error when you try to use more than two axes. Should you find yourself with a tool that can handle many axes, you must use a different syntax to specify which data you want **ON COLUMNS** and which data you want **ON ROWS**.

The alternate syntax is short. Instead of using **ON COLUMNS**, you use **ON 0**. Instead of using **ON ROWS** you use **ON 1**.

Let's try it! In MDX mode, write this query:

CODE TO TYPE:

```
select {[Measures].[Store Sales]} ON 1,
{[Time].[1997].[Q1].[2] } ON 0
from [Sales]
where [Product].[Drink].[Alcoholic Beverages]
```

The result looks exactly the same:

	Time
Measures	2
Store Sales	919.27

Slicer: [Product Department=Alcoholic Beverages]

What would happen if you tried to use *three* axes (by using **ON 0**, **ON 1**, and **ON 2**), and no slicer? Let's try that as well. In MDX Mode, write this query:

CODE TO TYPE:

```
select {[Measures].[Store Sales]} ON 1,
{[Time].[1997].[Q1].[2] } ON 0,
[Product].[Drink].[Alcoholic Beverages]  on 2
from [Sales]
```

Our tool can't handle a three-dimensional (we're talking about three physical dimensions, not data warehouse dimensions) result, so it complains:

Your query contained an error:

```
select {[Measures].[Store Sales]} ON 1,
{[Time].[1997].[Q1].[2] } ON 0,
[Product].[Drink].[Alcoholic Beverages]  on 2
from [Sales]
```

javax.servlet.jsp.JspException: java.lang.IllegalArgumentException: TableRenderer requires 0, 1 or 2 dimensional result

We'll learn ways to get around this limitation in future lessons.

Basic MDX Queries

In the first lesson we wrote a SQL query and an MDX query to determine sales in 1997. Here is the MDX query we wrote:

OBSERVE:

```
select
{ [Measures].[Store Sales] } ON COLUMNS
from Sales
where ([Time].[1997])
```

In the last course we learned that data warehouses usually have a *date* dimension. The warehouse we're using in this course also has a *date* dimension, but it is called **Time**. The brackets [] are just part of the MDX syntax, and are not part of the dimension's name. Periods are used to separate parts of dimensions.

A specific item in a dimension is called a *member*. In the **Time** dimension above, **1997** would be a *member*. Similarly, **Store Sales** is a member of the **Measures** dimension (the collection of measures/facts).

Let's compare our SQL and MDX queries from the last lesson, side by side:

SQL	MDX
select **sum(store_sales)** **from sales_fact sf join time_by_day t on (sf.time_id=t.time_id)** **WHERE t.the_year=1997**	**select** **{ [Measures].[Store Sales] } ON COLUMNS** **from Sales** **where ([Time].[1997])**

These queries look really similar, and for good reason! The design of MDX was heavily influenced by SQL. The major difference is that SQL is *relational* and MDX is *multi-dimentional*. Now let's break down the differences between SQL and MDX:

	SQL	MDX	
select	Tells the sql server we want to retrieve rows and columns - a *two dimensional* data set.	Tells the OLAP server we want to retrieve a data set, which can have zero or more *axes*.	**select**
sum(store_sales)	The list of **columns** to be returned by each row in the data set.	The list of **dimensions** we want to view. Remember, measures form a special dimension called **Measures**.	**{ [Measures].[Store Sales] }**
	In SQL there is no concept of *axes*. All rows have the same structure and data types, as defined by the columns. PIVOTs aside, rows and columns are not symmetrical, meaning you cannot change your query to return data on columns instead of rows.	The list of **dimensions** we want to view on each **axis**. In two dimensions, there are two axes: COLUMNS and ROWS. Instead of specifying **ON COLUMNS**, you could use **ON ROWS** or a number like **ON 2**. **Axes** are symmetric, meaning you can easily change from **COLUMNS** to **ROWS**.	**ON COLUMNS**

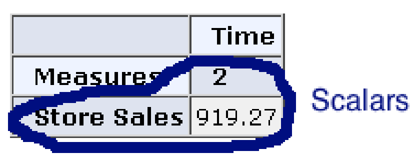
from sales_fact sf join time_by_day t on (sf.time_id=t.time_id)	The tables from which to retrieve data, and the relationships between those tables specified by *joins*.	The *cube* from which to retrieve data. A *cube* represents every combination of dimensions and facts, and is specified in the OLAP server's schema. Remember the star schema from the last DBA course? Conceptually speaking, that diagram represents the cube.	**from Sales**
WHERE t.the_year=1997	Filters the rows returned, so only rows that match the specified criteria are returned.	*Slices* the data returned by a specific dimension to restrict the multidimensional data set returned. *Slicing* is **not** the same as filtering. *Slicing* restricts the data set returned to a dimension. In other words, you cannot *slice* data to return sales greater than $5,000 - you must *filter* the data instead. We'll see more on filtering later.	**where ([Time].[1997])**

While MDX and SQL are similar in some respects, they are quite different in others.

Data Types

SQL databases have several different data types: integers, characters, decimals, and more. MDX has six data types:

Scalar values are numbers or strings, like the number 5 or the words "WEST REGION." Take a look at the results from our previous query:



Slicer: [Product Department=Alcoholic Beverages]

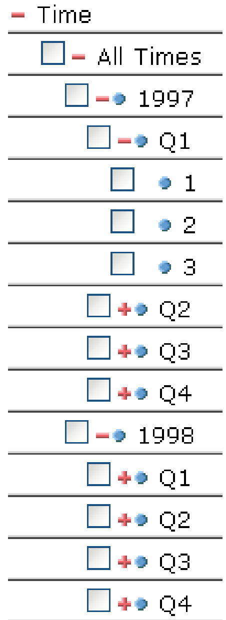
In this result set, the scalar values are **Store Sales**, **2**, and **919.27**

The next data type is **dimension**. Just like the dimensions in our relational data warehouse, they organize and categorize measures. In our example, **Time** and **Measures** are dimensions.

Under the hood, our OLAP server has two types of dimensions: *shared* dimensions and *cube-specific* dimensions. Dimensions such as **Time** are usually needed in in every cube, so that dimension is *shared* whenever it is needed. Other dimensions, such as "Promotion Media" may only be used in one cube, so it is *cube-specific*.

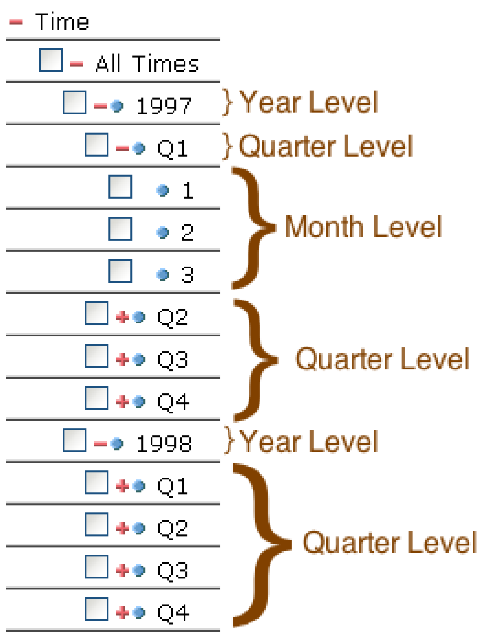
Slicer: [Product Department=Alcoholic Beverages]

All dimensions have a hierarchical structure in MDX, so the next data type is **hierarchy**. The hierarchical structure of dimensions defines the way measures are "rolled up." Dimensions don't have to use a hierarchy if doing so is impractical, but many do. The **Time** dimension has a hierarchy in our data warehouse which consists of **Year -> Quarter -> Month**:

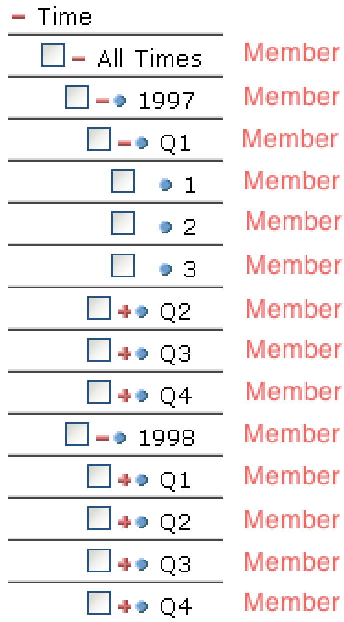


It may appear that the data warehouse has a monthly *grain* (level of detail) instead of a daily *grain*, however that is not necessarily the case. It might be that our business users aren't particularly concerned about daily sales, so the **Time** dimension hides days by default.

A **level** is literally a level in a hierarchical dimension. In our **Time** dimension, *Year* is a level, *Quarter* is a level, and *Month* is a level:



Specific items in dimensions are called **members**. In our **Time** dimension, 1997 is a member, written as `[Time].[1997]`. Q1 is also a member, however, you must specify the exact level in the hierarchy in order to refer to a specific member: `[Time].[1997].[Q1]`:



You can write an MDX query to return a specific member:

CODE TO TYPE:

```
select
{[Time].[1997].[Q1] } ON 0
from [Sales]
```

Click **Run Query**. The results may look a little strange. That's because we didn't select any measures:

Time
+Q1
66,291

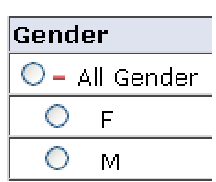
Slicer:

Cubes may have *default* measures defined. For our `Sales` cube, the default measure is **Unit Sales**. This measure is returned by default if we do not specify a measure.

Since dimensions are hierarchical, members have a single parent (except the top or *root* member), and members have one or more *children* (except the bottom or *leaf* member, which has no children).

Note Measures (*facts*) have their own dimension called **Measures**. The **Measures** hierarchy is flat.

Some hierarchies have a special **All** member. This member allows you to calculate percentages verses totals. We'll learn more about this later. In the **Gender** dimension in our data warehouse, the hierarchy looks like this and includes the **All** member:



Members can also have properties. Many dimensions (like customers) usually have ID or key fields that tie data back to source systems. These fields are not useful to most business users, but they may be of interest to some.

The last two data sets are very important - so important we will cover them in detail in the next lesson. They are:

- **Tuple** - an ordered collection of one or more members from different dimensions.
- **Set** - an ordered collection of *tuples* with the same dimensionality.

We've only just begun to cover MDX. In the next lesson we'll learn more about tuples and sets, and start writing our own MDX queries from scratch. See you there shortly!

Tuples and Sets
DBA 4: Analyzing Data Lesson 3

Welcome back! In the last lesson we got our first real look at MDX. We defined some new terms, and checked out a few example queries. The are two important terms we defined at the end of lesson 2 that we will go over in greater detail now: tuples and sets. We'll also take a look at some ways to interact with our query tool to make our work a bit easier.

Tuples and Sets

Tuples

In MDX, a **tuple** is an ordered collection of one or more members from different dimensions. You specify a tuple using **parentheses** and **commas**. Let's try using a tuple by answering this question: *How many unit sales were made in Q1 1997 by males?* In MDX mode, run this query:

CODE TO TYPE:

```
select
(
[Time].[1997].[Q1] , [Gender].[M]
)
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

Here you can see where our **parentheses** and **comma** are used to denote our tuple. Our tuple has one member from the **Time** dimension and one member from the **Gender** dimension.

The result looks good - it shows us exactly what we wanted to see:

		Measures
Time	Gender	Unit Sales
+Q1	M	33,381

Now let's suppose we only want to see *married* folks. In MDX mode, write this query:

CODE TO TYPE:

```
select
(
[Time].[1997].[Q1], [Gender].[M], [Marital Status].[M]
)
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

For this query we added another **comma** and **member** to our existing tuple. Our results are returned quickly:

			Measures
Time	Gender	Marital Status	Unit Sales
+Q1	M	M	16,311

You might be wondering what that little blue plus sign **+** next to **Q1** means. If you hover over the plus, you'll notice a change in your cursor:

			Measures
Time	Gender	Marital Status	Unit Sales
+Q1	M	M	16,311

Slicer:

That **+** is how our query tool (JPivot) tells us that we can *drill down* into that dimension. Click on the **+**. You'll see Q1's *child* members, the months 1, 2, and 3.

			Measures
Time	Gender	Marital Status	Unit Sales
-Q1	M	M	16,311
1	M	M	5,057
2	M	M	5,230
3	M	M	6,024

Drilling down means to go from a summary level of information, like the quarter level, into more specific level of detail, like the month level. Similarly, *drilling up* means to go from a more specific detail level to a summary level.

The structure of the cube combined with the power of the query tool makes this type of data navigation much easier. Try that with SQL!

Sets

A **tuple** is composed of members from different dimensions. What happens if you try to use members from the same dimension? Let's find out. In MDX Mode, write this query:

CODE TO TYPE:

```
select
(
[Time].[1997].[Q1] , [Time].[1997].[Q2]
)
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

In this case Mondrian (our OLAP server) throws an error:

Your query contained an error:

```
select
(
[Time].[1997].[Q1] , [Time].[1997].[Q2]
)
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

javax.servlet.jsp.JspException: Mondrian Error:Tuple contains more than one member of dimension '[Time]'.

A **set** is an ordered collection of *tuples* with the same dimensionality. We can use a **set** to see sales from Q1 and Q2 since a tuple doesn't do the job. Sets use curly brackets {}. It's easy to confuse parentheses and curly brackets. Here is what they look like in a large font:

Tuples: ( ) Parentheses

Sets: { } Braces

To make our query work, we need to use curly brackets instead of parentheses. In MDX Mode, type this query:

CODE TO TYPE:

```
select
{
[Time].[1997].[Q1] , [Time].[1997].[Q2]
}
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

Now we're getting results!

	Measures
Time	Unit Sales
+Q1	66,291
+Q2	62,610

Since there are two blue plus signs **+** in the results, you can drill down on Q1 or Q2. Try drilling down on both Q1 and Q2, and you'll see this:

	Measures
Time	Unit Sales
-Q1	66,291
1	21,628
2	20,957
3	23,706
-Q2	62,610
4	20,179
5	21,081
6	21,350

Did you notice the little blue dot next to **Unit Sales**? Hover over it:

	Measures
Time	Unit Sales
-Q1	66,291
1	21,628

Natural Order

You'll see the words **Natural Order**. JPivot has many tricks; one trick it can perform is **sorting**. By default, measures are sorted in **Natural Order** which means in the order specified by the dimensions in your query. The natural order of our time dimension would show us the oldest data first and the newest data last.

Click on the blue dot, and you'll see something amazing happen. The data is now sorted in ascending order, which means the lowest unit sales are shown first. The amazing part isn't in the ascending order, but in the way the hierarchy of the time dimension was retained. JPivot shows you this by changing the blue dot to a upward-facing triangle:

	Measures
Time	Unit Sales
-Q2	62,610
4	20,179
5	21,081
6	21,350
-Q1	66,291
2	20,957
1	21,628
3	23,706

Click on that triangle. Now you'll see the data sorted in descending order, and the hierarchy is remains intact. Now the triangle points down:

	Measures
Time	Unit Sales
-Q1	66,291
3	23,706
1	21,628
2	20,957
-Q2	62,610
6	21,350
5	21,081
4	20,179

Click the triangle again and you'll be back to the natural order. In a future lesson we'll talk more about sorting.

Earlier we defined a set this way: a set is an ordered collection of *tuples* with the same **dimensionality**. **Dimensionality** refers to the quantity and dimension of each member in the tuple. In other words: **sets must have the same number of members, and each member must be of the same dimension**.

In our prior example, we used two members from the time dimension to construct our set. In reality, we were making a set of two tuples, each tuple having one member. Our query could have been written another way. In MDX Mode, write this query:

CODE TO TYPE:

```
select
{
 ( [Time].[1997].[Q1] ) ,
 ( [Time].[1997].[Q2] )
}
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

See how we added parentheses () around our members to make a set? Run that query, and you'll see the same exact results as before:

	Measures
Time	Unit Sales
+Q1	66,291
+Q2	62,610

Again, tuples are made up of members from one or more dimensions. We can combine tuples into a set as long as each tuple has the same **dimensionality** (same dimension order and number of members).

Let's try an example query that compares "Unit sales in Q1 1997 for males" to "unit sales in 1997 for females." In MDX Mode, write the following query:

CODE TO TYPE:

```
select
{
 ( [Time].[1997].[Q1] , [Gender].[M] ),
 ( [Time].[1997], [Gender].[F] )
}
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

In this query we are building a **set** using **curly brackets {}**, and using two tuples. The **first tuple** combines **Q1 1997** and **males**. The **second tuple** combines **1997** (the entire year) and **females**.

Mondrian returns our answer:

		Measures
Time	Gender	Unit Sales
+Q1	M	33,381
-1997	F	131,558

These two sets have the same **dimensionality** because they both have two members. They have a member from the time dimension and a member from the gender dimension, in that order.

What happens if we try to mix our tuple up? Let's swap out the time dimension in one set with the marital status dimension. In MDX Mode, write this query:

CODE TO TYPE:

```
select
{
 ( [Time].[1997].[Q1] , [Gender].[M] ),
 ( [Marital Status].[M], [Gender].[F] )
}
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

Run this query:

Your query contained an error:

```
select
{
 ( [Time].[1997].[Q1] , [Gender].[M] ),
 ( [Marital Status].[M], [Gender].[F] )
}
 ON ROWS,
[Measures].[Unit Sales] on COLUMNS
from [Sales]
```

javax.servlet.jsp.JspException: Mondrian Error:All arguments to function '{}' must have same hierarchy.

It looks like we made Mondrian cranky! This error tells us that our sets do not have the same dimensionality, and so they cannot be combined into a tuple.

Navigating the Cube

You may be asking yourself, *"how can I get a description of the dimensions and measures available in this cube?"* What a great question!

It's the responsibility of the database administrator to document the data warehouse. This documentation could be a word document, a web page, or pages in the company's wiki. The location isn't important, as long as the documentation is useful to our end-users.

We are using Mondrian as our OLAP server, which uses XML files to store schemas. Using XML is good for us, because it allows us to use xsl to convert the schema into a web page.

Note Don't worry if you aren't familiar with XML or XSL. If you would like more information on XML or XSL, check out the XML course.

Below is the schema for our FoodMart data warehouse. First, it shows the shared dimensions which may be used in one or more cubes. Next, it shows the cubes, and then the *virtual cubes*. In Mondrian, *Virtual cubes* are the concatenation of one or more cubes. Among other things, virtual cubes allow the database administrator to create smaller, interest-specific cubes such as **Warehouse** and **Sales**, which fulfill the needs of most business users. Other users who may need to analyze **Warehouse** *and* **Sales** can do so with the larger virtual cube **Warehouse and Sales**.

Take a look!

Schema: FoodMart

Shared Dimensions

Dimension: Store

Hierarchy: *default* Has All Member: true All Member Name: `[All]`			
Level	**Level In Hierarchy**	**Unique Members**	**Properties**
`Store Country`	`[Store].[Store Country]`	true	
`Store State`	`[Store].[Store Country].[Store State]`	true	
`Store City`	`[Store].[Store Country].[Store State].[Store City]`	false	
`Store Name`	`[Store].[Store Country].[Store State].[Store City].[Store`	true	Store Type, Store Manager, Store Sqft, Grocery Sqft, Frozen Sqft, Meat Sqft, Has

The schema contains lots of stuff we haven't covered yet, like calculated members and format strings. Don't worry, we'll cover them later.

Scroll down to the **Measures** section of the **Sales** cube. Here you'll see each measure included in sales. They are:

- [Measures].[Unit Sales]
- [Measures].[Store Cost]
- [Measures].[Store Sales]
- [Measures].[Sales Count]
- [Measures].[Customer Count]
- [Measures].[Promotion Sales]

Let's use one of these items in a query. In MDX Mode, type this query:

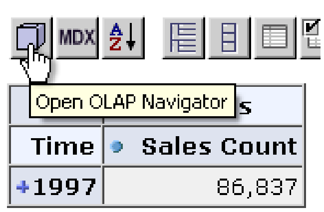
CODE TO TYPE:

```
select
[Time].[1997] ON ROWS,
[Measures].[Sales Count] on COLUMNS
from [Sales]
```

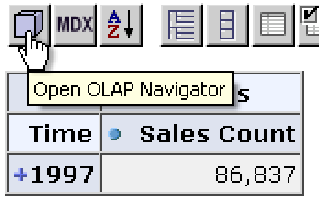
If you typed everything correctly, you'll see these result:

	Measures
Time	Sales Count
1997	86,837

Do you see the little cube  in the upper left hand corner of your window? Hover over that icon:



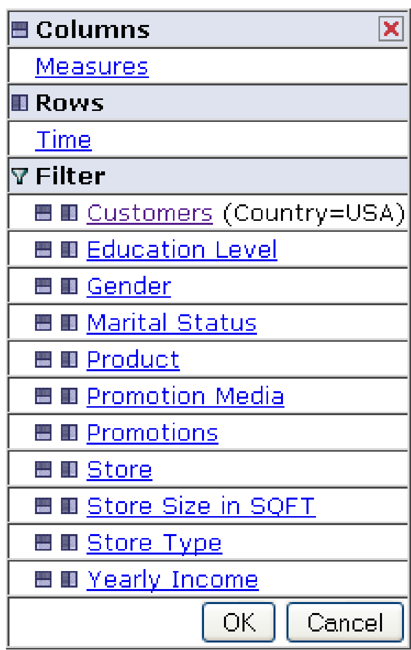
JPivot has a way to browse the query and cube you are working with - it's called the **OLAP Navigator**. Click on the cube to open the navigator:



Here you can move dimensions from axis to axis, add a slicer axis (under the *filter* section), or just take a peek at the structure of the cube.

Click on the **Customers** link, then on the + next to **All Customers**. Next, click the radio button next to **USA** and click **OK**:

JPivot will tell you that you've picked a member for the slicer (filter). Click OK again:



There you have it! The newly sliced data:

Slicer: [Country=USA]

Play around with the navigator. You'll see how easy it is to explore your query.

You've done a lot in this lesson! In the next lesson we'll talk about using calculated members in our queries. See you then!

Calculated Members
DBA 4: Analyzing Data Lesson 4

In the last lesson we took a look at the schema of our data warehouse, and we saw that the cube we've been working with had several *calculated members*. In this lesson we're going to learn all about those calculated members.

Calculated Members

Data warehouses usually contain a lot of information, but it is impossible to calculate and store every possible assembly of data. Sometimes data isn't stored anywhere - *profit*, for example, is usually calculated as costs subtracted from sales. Some users may only want to see profit for 1997, while others might want to see profit according to store. Enumerating, calculating, and storing all combinations of dimensions, sales, and costs would take entirely too much space and effort.

Let's try to calculate profit (sales *minus* costs) in a query. It seems like we should be able to write a query using `[Measures].[Store Sales] - [Measures].[Store Cost]` in order to come up with profit. Let's try it. In MDX Mode, write the following query:

CODE TO TYPE:

```
select
[Time].[1997] ON ROWS,
[Measures].[Store Sales] - [Measures].[Store Cost] on COLUMNS
from [Sales]
```

It looks like Mondrian isn't happy with our query:

Your query contained an error:

```
select
[Time].[1997] ON ROWS,
[Measures].[Store Sales] - [Measures].[Store Cost] on COLUMNS
from [Sales]
```

javax.servlet.jsp.JspException: Mondrian Error:Axis 'COLUMNS' expression is not a set

It is telling us that `[Measures].[Store Sales] - [Measures].[Store Cost]` is not a set. So how do we calculate profit? We need to use a **calculated member**.

The simplest use of **calculated members** allows you to define a calculation and use it in many places within your query. Let's see how a calculated member is defined by calculating profit. In MDX Mode, write this query:

CODE TO TYPE:

```
WITH
 member [Measures].[My Profit]
as [Measures].[Store Sales] - [Measures].[Store Cost]

select
[Time].[1997] ON ROWS,
[Measures].[My Profit] on COLUMNS
from [Sales]
```

Any query that uses calculations must start with the word **WITH**. Next come the **type** and **name** of the calculation, in this case **member [Measures].[My Profit]**. **Type** and **name** tell the MDX server that we are adding a calculated **member** to the **[Measures]** dimension, called **[My Profit]**.

Then we add the word **as** and then the **definition** of the calculation. For *profit*, the definition is sales minus cost, or **[Measures].[Store Sales] - [Measures].[Store Cost]**.

In order to see our new member, we use it on an axis. In our query, our new member is located on the **columns** axis.

Run the query, you'll see these results:

	Measures
Time	My Profit
+1997	339,610.90

Now suppose the Vice President of your company tells you that the profit goal for 1997 was **$325,000**. How would you calculate the difference between the sales goal and the profit? By using another calculated member, of course! Let's start by defining a member with our goal. In the query below we need to use curly brackets {} because we need to form a set with `[My Profit]` and `[Goal]`. In MDX Mode, write this query:

CODE TO TYPE:

```
WITH
 member [Measures].[My Profit]
as [Measures].[Store Sales] - [Measures].[Store Cost]

member [Measures].[Goal] as 325000

select
[Time].[1997] ON ROWS,
  { [Measures].[My Profit], [Measures].[Goal] } on COLUMNS
from [Sales]
```

It looks good so far!

	Measures	
Time	My Profit	Goal
+1997	339,610.90	325,000

Now let's add another calculation to figure out the difference. In MDX Mode, write this query:

CODE TO TYPE:

```
WITH
 member [Measures].[My Profit]
as [Measures].[Store Sales] - [Measures].[Store Cost]

member [Measures].[Goal] as 325000

member [Measures].[Goal Difference] as
  [Measures].[My Profit] - [Measures].[Goal]

select
[Time].[1997] ON ROWS,
{ [Measures].[My Profit], [Measures].[Goal], [Measures].[Goal Difference] } on COLUMNS
from [Sales]
```

If you typed everything correctly (don't forget about the curly brackets {}) you'll see these results:

	Measures		
Time	My Profit	Goal	Goal Difference
+1997	339,610.90	325,000	14,610.90

Fantastic! We were $14,610.90 over our goal!

Format Strings

Now, before we hand these results over to the Vice President, let's improve their appearance a little. To do that, we'll use two components found in MDX: a **FORMAT_STRING** in your query, and a query tool (such as JPivot) that supports parsing and displaying format strings.

Format strings work on string, numeric, or date values. Format strings allow you to convert string text to uppercase, lowercase, and even handle special cases such as NULLs or empty strings. Format strings allow you to format numbers any way you'd like, and handle positive, negative, zero, and empty values differently.

Let's try a format string now. Our "My Profit" calculation is in US Dollars, so we might want to include the dollar sign **$** in the results. And let's suppose that the cents (.90) are not important to our end user, so we'll hide those as well. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
 member [Measures].[My Profit]
as [Measures].[Store Sales] - [Measures].[Store Cost], FORMAT_STRING = "$#,##0"

member [Measures].[Goal] as 325000
```

```
member [Measures].[Goal Difference] as
  [Measures].[My Profit] - [Measures].[Goal]

select
[Time].[1997] ON ROWS,
{ [Measures].[My Profit], [Measures].[Goal], [Measures].[Goal Difference] } on COLUMNS
from [Sales]
```

We specify our format string by setting the **FORMAT_STRING** property on our calculated member. To specify the property, we use a **comma ,** and then **FORMAT_STRING = "*actual format string in quotation marks*"** .

Run the query, and you'll see the newly formated results:

	Measures		
Time	My Profit	Goal	Goal Difference
+1997	$339,611	325,000	$14,611

Everything looks great! The `[Goal Difference]` member inherited the format string from `[My Profit]`. Now let's improve the appearance of the `[Goal]`. In MDX Mode, write the following query:

CODE TO TYPE:

```
WITH
 member [Measures].[My Profit]
as [Measures].[Store Sales] - [Measures].[Store Cost], FORMAT_STRING = "$#,##0"

member [Measures].[Goal] as 325000, FORMAT_STRING = "$#,##0"
member [Measures].[Goal Difference] as
  [Measures].[My Profit] - [Measures].[Goal]

select
[Time].[1997] ON ROWS,
{ [Measures].[My Profit], [Measures].[Goal], [Measures].[Goal Difference] } on COLUMNS
from [Sales]
```

This looks even better!

	Measures		
Time	My Profit	Goal	Goal Difference
+1997	$339,611	$325,000	$14,611

We really want to impress the Vice President by making the report as clear and easy to understand as possible. To accomplish that, let's change the background color for `[Goal Difference]` so it is **green** when it is positive and **red** when it is negative. We'll also surround negative values with parentheses (), similar to the way Microsoft Excel might indicate negative values. We can do all of this because format strings for numbers allow us to specify different strings for positive and negative values (as well as zero and empty values).

Let's give it a try! In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
 member [Measures].[My Profit]
as [Measures].[Store Sales] - [Measures].[Store Cost], FORMAT_STRING = "$#,##0"

member [Measures].[Goal] as 325000, FORMAT_STRING = "$#,##0"
member [Measures].[Goal Difference] as
  [Measures].[My Profit] - [Measures].[Goal], FORMAT_STRING = "|$#,##0|style='green';|($#,##0)|style='

select
[Time].[1997] ON ROWS,
{ [Measures].[My Profit], [Measures].[Goal], [Measures].[Goal Difference] } on COLUMNS
from [Sales]
```

Our format string is now quite a bit larger. It is divided by the **semicolon ;** into two sections. The first section is for positive values; the second section is for negative values.

The positive values section is further divided by **bars |**. The bars surround the specific formatting for positive numbers - in our case `$#,##0`. After the bar, we have the cell style, **style='green'**.

The negative values section is also divided by **bars |**. The bars surround the specific formatting for negative numbers - in our case `($#,##0)`. After the bar, we have the cell style, **style='red'**.

Run the query to see the results:

	Measures		
Time	My Profit	Goal	Goal Difference
1997	$339,611	$325,000	$14,611

Great! But what happens when the goal difference is negative? Suppose our goal was really $395,000. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
 member [Measures].[My Profit]
as [Measures].[Store Sales] - [Measures].[Store Cost], FORMAT_STRING = "$#,##0"

member [Measures].[Goal] as 395000, FORMAT_STRING = "$#,##0"
member [Measures].[Goal Difference] as
  [Measures].[My Profit] - [Measures].[Goal], FORMAT_STRING = "|$#,##0|style='green';|($#,##0)|style='

select
[Time].[1997] ON ROWS,
{ [Measures].[My Profit], [Measures].[Goal], [Measures].[Goal Difference] } on COLUMNS
from [Sales]
```

Hopefully we'll never see red!

	Measures		
Time	My Profit	Goal	Goal Difference
1997	$339,611	$395,000	($55,389)

For more information on format strings, check out Microsoft's SQL Server 2008 web site. Our OLAP server, Mondrian, doesn't support everything that SQL Server 2008 does, but most of the information is relevant.

Named Sets

We've just seen how to add calculated members to our query. MDX has another trick; you are allowed to create *named sets* as well. Suppose we want to see the 1997 unit sales for all 1% and 2% milk products. We could spend a lot of time typing and come up with the following query in MDX:

CODE TO TYPE:

```
select
( [Time].[1997], [Measures].[Unit Sales] ) ON COLUMNS,
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Club].[Club 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Club].[Club 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Even Better].[Even Better 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Even Better].[Even Better 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Gorilla].[Gorilla 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Gorilla].[Gorilla 2% Milk]
}
  ON ROWS
from [Sales]
```

Run the query. You should see some results:

	Time
	1997
	Measures
Product	Unit Sales
Booker 1% Milk	189
Booker 2% Milk	177
Carlson 1% Milk	212
Carlson 2% Milk	131
Club 1% Milk	155
Club 2% Milk	145
Even Better 1% Milk	190
Even Better 2% Milk	177
Gorilla 1% Milk	160
Gorilla 2% Milk	133

This query is a bit bulky, and somewhat difficult to understand. But MDX lets us define a *named set* to get rid of some of the complexity. Let's give that a try.

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Club].[Club 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Club].[Club 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Even Better].[Even Better 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Even Better].[Even Better 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Gorilla].[Gorilla 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Gorilla].[Gorilla 2% Milk]
}

select
( [Time].[1997], [Measures].[Unit Sales] ) ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

Like calculated members, you must use the **WITH** keyword when you use a named set. Then you tell MDX that you're creating a **set** and **give it a name** - in this case we named our set **[1% and 2% Milk]**. This is followed by the required word: **as**.

Finally, you provide your **set definition**. In the select section of your query, you use your set, referring to it by its **name**, **[1% and 2% Milk]**.

Run the query - you'll see the same results as before, but the query is easier to understand:

	Time
	+1997
	Measures
Product	Unit Sales
Booker 1% Milk	189
Booker 2% Milk	177
Carlson 1% Milk	212
Carlson 2% Milk	131
Club 1% Milk	155
Club 2% Milk	145
Even Better 1% Milk	190
Even Better 2% Milk	177
Gorilla 1% Milk	160
Gorilla 2% Milk	133

We'll see more uses for named sets in future lessons.

In the next lesson we'll discuss the functions we can use to work with tuples, sets, levels, hierarchies, and dimensions. See you soon!

MDX Functions, Part I
DBA 4: Analyzing Data Lesson 5

Welcome back! In the last lesson we took a look at tuples and sets. In this lesson we'll take a look at MDX functions we can use to make our queries easier.

Union and Cross Join

At the end of the last lesson we went over named sets. We used a named set to show the unit sales for various milk products in 1997. Now let's remove some of the items to make our set smaller. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}

select
( [Time].[1997], [Measures].[Unit Sales] ) ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

The results look good:

	Time
	1997
	Measures
Product	Unit Sales
Booker 1% Milk	189
Booker 2% Milk	177
Carlson 1% Milk	212
Carlson 2% Milk	131

Now let's say we want to show sales for 1997 *and* 1998? How would we do that?

We use the Time dimension on our COLUMNS axis. The dimension is in a tuple:

OBSERVE:

```
( [Time].[1997], [Measures].[Unit Sales] ) ON COLUMNS
```

Remember that tuples cannot have more than one member from the same dimension. It looks like we'll have to create a set of members in the Time dimension, like this:

OBSERVE:

```
( { [Time].[1997], [Time].[1998] }, [Measures].[Unit Sales] ) ON COLUMNS
```

Let's try to run it. In MDX Mode, write this query:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
```

```
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}

select
( { [Time].[1997], [Time].[1998] }, [Measures].[Unit Sales] ) ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

Mondrian doesn't seem to like our query:

Your query contained an error:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}

select
( { [Time].[1997], [Time].[1998] }, [Measures].[Unit Sales] ) ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

javax.servlet.jsp.JspException: Mondrian Error:No function matches signature '(,)'

Let's take a step back and consider the data we want to see. If we were writing our query in plain English, in sentence form, it would be something like:

For 1997 and 1998, show me the unit sales of 1% and 2% milk products

We could break that sentence into pieces, writing it this way:

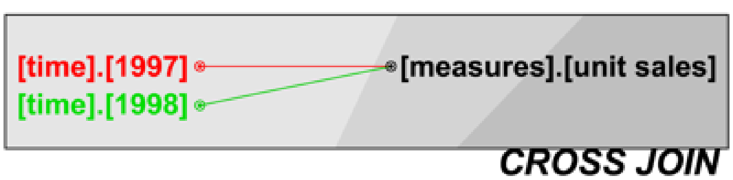
For 1997, show me the unit sales of 1% and 2% milk products.
For 1998, show me the unit sales of 1% and 2% milk products.

This new version of our sentence shows our query as the **cross product** of time (specifically 1997, 1998) and unit sales. A **cross product** (also known as a **cross join** or **cartesian product**) of two sets is a new set made up of every combination of data from the original sets. In our query we have two sets:

1. **{ [Time].[1997], [Time].[1998] }** - a set of two members
2. **{ [Measures].[Unit Sales] }** - a set of one member

There are only two total combinations in the cross product of those sets:

1. **{ ([Time].[1997] , [Measures].[Unit Sales]) }** - a set of one tuple
2. **{ ([Time].[1998] , [Measures].[Unit Sales]) }** - a set of one tuple



We can combine those tuples by using a *union*. As it turns out, we've already seen a *union*, we just didn't know that is was called a union. Remember when we built a set using **curly brackets {}** and **commas**? We made a set then using *unions*. In MDX Mode, write this query:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
```

```
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}

select
{
 { ( [Time].[1997] , [Measures].[Unit Sales] ) }
,
 { ( [Time].[1998] , [Measures].[Unit Sales] ) }
}

ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

In the listing above, **curly brackets {}** and **commas** are used to bring **([Time].[1997] , [Measures].[Unit Sales])** and **([Time].[1998] , [Measures].[Unit Sales])** into a union.

Run the query now, and you'll see the results for 1997 and 1998:

	Time	
	+1997	+1998
	Measures	Measures
Product	Unit Sales	Unit Sales
Booker 1% Milk	189	441
Booker 2% Milk	177	326
Carlson 1% Milk	212	404
Carlson 2% Milk	131	393

Slicer:

MDX also has an alternate syntax for unions. Check it out. In MDX Mode, write this query:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}

select
UNION (
 { ( [Time].[1997] , [Measures].[Unit Sales] ) }
,
 { ( [Time].[1998] , [Measures].[Unit Sales] ) }
)

ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

This query produces the same result as before, but now it's a bit easier to read and understand:

	Time	
	1997	1998
	Measures	Measures
Product	Unit Sales	Unit Sales
Booker 1% Milk	189	441
Booker 2% Milk	177	326
Carlson 1% Milk	212	404
Carlson 2% Milk	131	393

Slicer:

It wasn't too difficult to come up with the cross product of Time and Unit Sales by hand. But what if our sets were larger? Surely Mondrian could calculate the cross product for us. In MDX there are two ways to do a cross product: the asterisk * or the keyword **CROSSJOIN**.

Note Many functions in MDX, like UNION and CROSSJOIN, have two or more syntaxes. It doesn't matter which syntax you pick, as long as your query makes sense to you and others.

So how do we use CROSSJOIN? In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}

select

{ [Time].[1997], [Time].[1998] }
*
{ [Measures].[Unit Sales] }

ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

In the code above, we use the asterisk * to calculate the cross join of two sets: **{ [Time].[1997], [Time].[1998] }** and **{ [Measures].[Unit Sales] }**.

Once again, our results look exactly the same, but this time our query is much shorter:

	Time	
	1997	1998
	Measures	Measures
Product	Unit Sales	Unit Sales
Booker 1% Milk	189	441
Booker 2% Milk	177	326
Carlson 1% Milk	212	404
Carlson 2% Milk	131	393

Slicer:

Other Set Functions

Now suppose you wanted to break these sales down even further, according to *gender*. In previous lessons we used the `[Gender]` dimension in a query, and saw how its members were called `[Gender].[M]` and `[Gender].[F]`. Now we want a way to return the set of all members from a specific level in a dimension. Fortunately, MDX has a special function called **Members** that returns a set of every member of a dimension. **Members** can be used on a dimension, level, or hierarchy. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}

select

{ [Time].[1997], [Time].[1998] }
*
[Gender].Members
*
{ [Measures].[Unit Sales] }

ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

In this query we continue to use asterisks * to cross join our sets. And this time we've added a new set: **[Gender].Members**. This new set includes all members of the `[Gender]` dimension.

Run the query and you'll see the newly augmented results:

	Time					
	+1997			+1998		
	Gender			Gender		
	-All Gender	F	M	-All Gender	F	M
	Measures	Measures	Measures	Measures	Measures	Measures
Product	Unit Sales	Unit Sales	Unit Sales	Unit Sales	Unit Sales	Unit Sales
Booker 1% Milk	189	109	80	441	231	210
Booker 2% Milk	177	84	93	326	150	176
Carlson 1% Milk	212	82	130	404	228	176
Carlson 2% Milk	131	62	69	393	170	223

Slicer:

We've seen the **F** and **M** members of the gender dimension before, but where did the **All Gender** member come from? Mondrian allows schema authors to specify special **All** members for levels. This makes it possible to compare members against totals for percentages.

So, what if we didn't want to see the **All Member** member? MDX has another function we can use, called **Children**. This function returns the set of children of a member. Let's try it! In MDX Mode, write the following query:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}

select

{ [Time].[1997], [Time].[1998] }
*
[Gender].Children
*
{ [Measures].[Unit Sales] }
```

```
ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

This query is almost the same as the last one, but this one uses **[Gender].Children** instead of **[Gender].Members**.

Now we'll only see **F** and **M** under the gender dimension:

	Time			
	+1997		+1998	
	Gender		Gender	
	F	M	F	M
	Measures	Measures	Measures	Measures
Product	Unit Sales	Unit Sales	Unit Sales	Unit Sales
Booker 1% Milk	109	80	231	210
Booker 2% Milk	84	93	150	176
Carlson 1% Milk	82	130	228	176
Carlson 2% Milk	62	69	170	223

We are looking good!

Until now we've used a small set of products for our queries. Let's take a look at the hierarchy that is part of the `[Products]` dimension:

Level	Level In Hierarchy
Product Family	`[Product].[Product Family]`
Product Department	`[Product].[Product Family].[Product Department]`
Product Category	`[Product].[Product Family].[Product Department].[Product Category]`
Product Subcategory	`[Product].[Product Family].[Product Department].[Product Category].[Product Subcategory]`
Brand Name	`[Product].[Product Family].[Product Department].[Product Category].[Product Subcategory].[Brand Name]`
Product Name	`[Product].[Product Family].[Product Department].[Product Category].[Product Subcategory].[Brand Name].[Product Name]`

Here's the set we've used:

OBSERVE:

```
 set [1% and 2% Milk]
as
{
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Booker].[Booker 2% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 1% Milk],
  [Product].[Drink].[Dairy].[Dairy].[Milk].[Carlson].[Carlson 2% Milk]
}
```

All of these members have the same [Product Subcategory] of [Milk]. Each has a different `[Brand Name]` and `[Product Name]`.

Suppose the Vice President wants to see 1997 and 1997 unit sales for all Milk products, according to gender. How would we answer that query? It would take a lot of time to look up each milk brand and product, and our named set would become really large. Then to complicate matters further, each time our stores added a new milk brand, our query would need to be updated. There has to be a better way!

We've already seen the **Children** function, so let's try to use it in our query. Type this in MDX Mode:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as
[Product].[Drink].[Dairy].[Dairy].[Milk].Children
```

```
select

{ [Time].[1997], [Time].[1998] }
*
[Gender].Children
*
{ [Measures].[Unit Sales] }

ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

In this query we've replaced our large set with a single function: **[Product].[Drink].[Dairy].[Dairy].[Milk].Children**.

	Time			
	+1997		+1998	
	Gender		Gender	
	F	M	F	M
	Measures	Measures	Measures	Measures
Product	Unit Sales	Unit Sales	Unit Sales	Unit Sales
+Booker	416	356	934	864
+Carlson	414	513	1,030	1,005
+Club	356	411	866	760
+Even Better	425	534	911	870
+Gorilla	376	385	932	777

The results from this query are not exactly what we want though. MDX did its job - it returned the children of the `[Product Subcategory]` level, which is `[Brand Name]`. We can click on the + to expand `[Brand Name]` to `[Product Name]`, but it would be better to get all product names by default. If we tried to use `.Children` we would have to list out every `[Brand Name]` - something we are trying to avoid. There is a better way. MDX has a function called **Descendants** which can make short work of this problem.

The **Descendants** function returns a set of all members at a given level. At first glance this seems exactly like the `Members` or `Children` function, but there is one major difference.

`Children` returns a set of members below a specific member. In the last example it returned a set of `[Brand Name]`s below the `[Product Subcategory]` level.

Descendants returns a set of members at a specific level, which are descendants (like *children*, *grandchildren*, *great grandchildren*, and so on) of a specified level. Take a look. Type this query in MDX Mode:

CODE TO TYPE:

```
WITH
 set [1% and 2% Milk]
as

Descendants(
[Product].[Drink].[Dairy].[Dairy].[Milk],
[Product].[Product Name]
)

select

{ [Time].[1997], [Time].[1998] }
*
[Gender].Children
*
{ [Measures].[Unit Sales] }

ON COLUMNS,
 [1% and 2% Milk] ON ROWS
from [Sales]
```

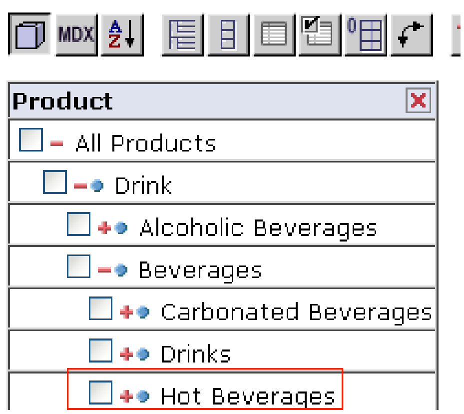
In this query we use the **Descendants** function to get the descendants of the member **[Product].[Drink].[Dairy].[Dairy].[Milk]**, at the **[Product].[Product Name]** level.

When you run the query, you'll see the following results (actually you'll see more - this image only shows the first few items):

	Time			
	+1997		+1998	
	Gender		Gender	
	F	M	F	M
	Measures	Measures	Measures	Measures
Product	Unit Sales	Unit Sales	Unit Sales	Unit Sales
Booker 1% Milk	109	80	231	210
Booker 2% Milk	84	93	150	176
Booker Buttermilk	61	49	209	171
Booker Chocolate Milk	77	56	180	143
Booker Whole Milk	85	78	164	164
Carlson 1% Milk	82	130	228	176
Carlson 2% Milk	62	69	170	223
Carlson Buttermilk	96	79	220	214
Carlson Chocolate Milk	76	99	232	180
Carlson Whole Milk	98	136	180	212
Club 1% Milk	70	85	183	153
Club 2% Milk	66	79	185	133
Club Buttermilk	66	74	198	163
Club Chocolate Milk	78	81	160	177
Club Whole Milk	[illegible]	[illegible]	[illegible]	[illegible]

It looks good. Now what if we changed our mind, and wanted **Hot Beverages** instead?

First you would need to use the **OLAP Navigator** to peek at the dimension structure. You can see **Hot Beverages** here in the hierarchy:



This translates into the following MDX:

OBSERVE:

```
[Product].[Drink].[Beverages].[Hot Beverages]
```

In MDX we can quickly swap this in our previous query:

CODE TO TYPE:

```
WITH
 set [Beverages]
as

Descendants(
[Product].[Drink].[Beverages].[Hot Beverages],
[Product].[Product Name]
)

select
```

```
{ [Time].[1997], [Time].[1998] }
*
[Gender].Children
*
{ [Measures].[Unit Sales] }

ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

Run the query. This time you'll see **Hot Beverages** instead of milk:

	Time			
	1997		1998	
	Gender		Gender	
	F	M	F	M
	Measures	Measures	Measures	Measures
Product	Unit Sales	Unit Sales	Unit Sales	Unit Sales
BBB Best Hot Chocolate	92	79	156	161
CDR Hot Chocolate	81	86	198	170
Landslide Hot Chocolate	67	88	171	165
Plato Hot Chocolate	82	93	153	240
Super Hot Chocolate	61	73	208	180
BBB Best Columbian Coffee	65	93	189	178
BBB Best Decaf Coffee	114	75	201	209
BBB Best French Roast Coffee	128	105	204	206
BBB Best Regular Coffee	121	73	201	164
CDR Columbian Coffee	70	117	158	164
CDR Decaf Coffee	77	117	176	192
CDR French Roast Coffee	77	73	138	139
CDR Regular Coffee	92	91	188	165
Landslide Columbian Coffee	97	62	165	160

Fantastic! We're on the way to becoming MDX experts!

As usual, we've covered a lot in this lesson. To learn more about the resources we've covered visit [Microsoft's MSDN web site](). Some of the information applies to SQL Server Analysis Services only, but much of the information is relevant to Mondrian.

In the next lesson we'll learn about filtering, and some additional MDX functions. Stay tuned!

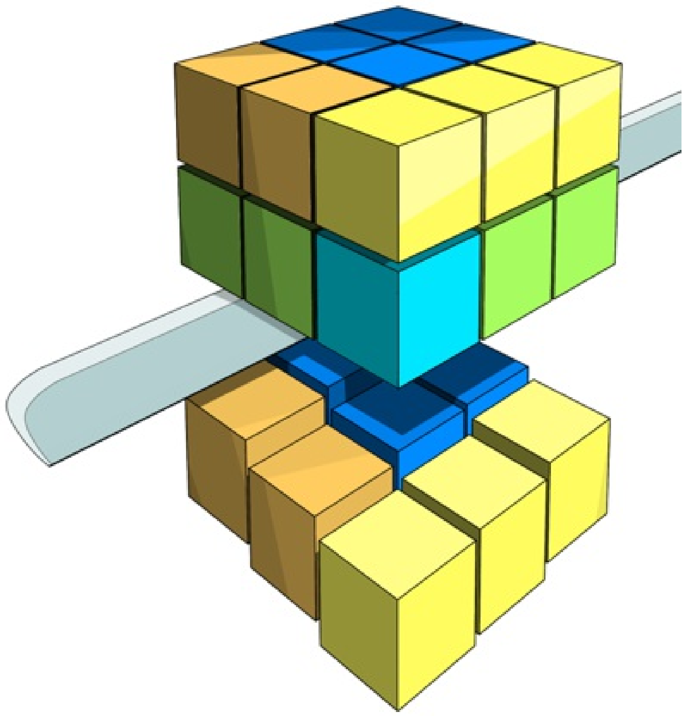
MDX Functions, Part II
DBA 4: Analyzing Data Lesson 6

In the last lesson we learned how to use some powerful MDX functions. In this lesson we'll go over several more MDX functions, including the extremely powerful `Filter` function.

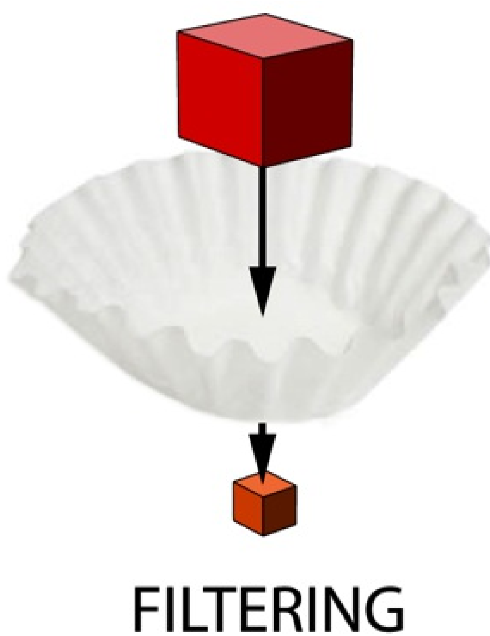
Filtering Data

Earlier in the course we discussed *slicing* data by using the WHERE clause and the slicer axis. We learned that *slicing* data is not the same thing as *filtering* data.

Slicing is essentially a hidden axis - it allows you to restrict the data returned to one or more hidden dimensions:



Filtering takes a set and a logical expression as parameters, and returns a set. It also allows you to restrict data, but it can be used on any axis, including the slicer axis. Because the filter uses a logical expression, you can limit results to things such as **Unit Sales greater than 400**:



Let's try an example. Suppose we're interested in seeing the Unit Sales of all Hot Beverages in 1998 with unit sales greater than 400.

For now, we'll ignore the "greater than 400" requirement, and just write a simple query to get started. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
set [Beverages]
as
{ [Time].[1998] } * Descendants(  [Product].[Drink].[Beverages].[Hot Beverages],
 [Product].[Product Name] )

select
{ [Measures].[Unit Sales] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

The query returns a lot of data, most of it with Unit Sales under 400:

		Measures
Time	Product	Unit Sales
1998	BBB Best Hot Chocolate	317
	CDR Hot Chocolate	368
	Landslide Hot Chocolate	336
	Plato Hot Chocolate	393
	Super Hot Chocolate	388
	BBB Best Columbian Coffee	367
	BBB Best Decaf Coffee	410
	BBB Best French Roast Coffee	410
	BBB Best Regular Coffee	365

Now let's *filter* the data to remove Unit Sales less than 400. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
set [Beverages]
as
FILTER(
{ [Time].[1998] } * Descendants(  [Product].[Drink].[Beverages].[Hot Beverages], [Product].[Product Na
,
[Measures].[Unit Sales] > 400
)
select
{ [Measures].[Unit Sales] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

In this listing, we are using the **FILTER** function on the **blue** set, limiting the set to the logical expression `[Measures].[Unit Sales] > 400`.

That seems to have done the trick!

		Measures
Time	Product	Unit Sales
1998	BBB Best Decaf Coffee	410
	BBB Best French Roast Coffee	410
	Plato French Roast Coffee	407

But why did we filter that particular **set**? You might think that we should have filtered the **COLUMNS** axis instead. Let's try that to see what happens In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
set [Beverages]
as
{ [Time].[1998] } * Descendants(  [Product].[Drink].[Beverages].[Hot Beverages], [Product].[Product Na
select
```

```
FILTER(
{ [Measures].[Unit Sales] }
,
[Measures].[Unit Sales] > 400
)
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

Here we've moved the **FILTER** function to the COLUMNS axis, filtering the set `{ [Measures].[Unit Sales] }` with the logical expression `[Measures].[Unit Sales] > 400`:

		Measures
Time	Product	Unit Sales
1998	BBB Best Hot Chocolate	317
	CDR Hot Chocolate	368
	Landslide Hot Chocolate	336
	Plato Hot Chocolate	393
	Super Hot Chocolate	388
	BBB Best Columbian Coffee	367
	BBB Best Decaf Coffee	410
	BBB Best French Roast Coffee	410
	BBB Best Regular Coffee	365
	CDR Columbian Coffee	322
	CDR Decaf Coffee	368
	CDR French Roast Coffee	277

Wait! What happened?!? **We filtered the wrong data**.

Take a look at that query again:

OBSERVE:

```
WITH
set [Beverages]
as
{ [Time].[1998] } * Descendants(  [Product].[Drink].[Beverages].[Hot Beverages], [Product].[Product Na
select
FILTER(
{ [Measures].[Unit Sales] }
,
[Measures].[Unit Sales] > 400
)
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

If you were to write this query in English, you might wright it this way:

For all Hot Beverages in 1998, show the Unit Sales when the **TOTAL unit sales are greater than 400**.

Compare this to the first filtered query:

OBSERVE:

```
WITH
set [Beverages]
as
FILTER(
{ [Time].[1998] } * Descendants(  [Product].[Drink].[Beverages].[Hot Beverages], [Product].[Product Na
,
[Measures].[Unit Sales] > 400
)
select
{ [Measures].[Unit Sales] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

This query might be written like this in English:

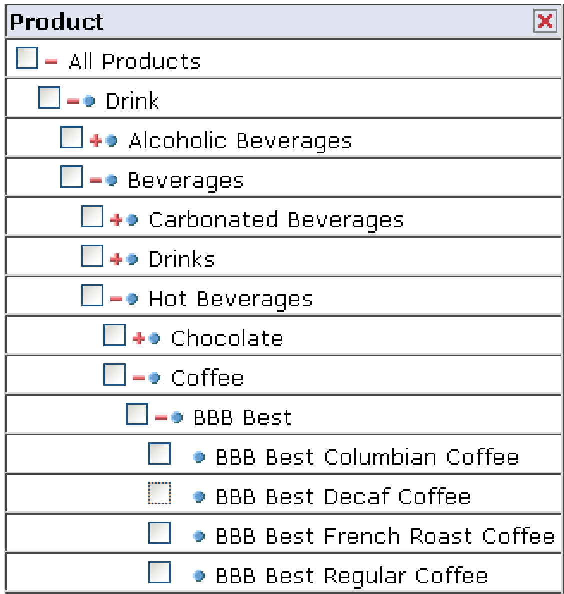
For **Hot Beverages in 1998 with unit sales greater than 400**, show the Unit Sales.

Notice the difference? The first query shows the Hot Beverages in 1998 where the **total sales** for 1998 is greater than 400, whereas the second query shows only Hot Beverages in 1998 that have sales greater than 400.

The difference is subtle, but important. When you filter data, you need to be careful and understand exactly what you are asking.

So, what else can we do with filtering?

Suppose our boss is interested in seeing sales for **Decaf** coffee only. You take a quick look at the OLAP Navigator to see what the Product dimension contains:



There is no "sub category" level under the Brand level, so we can't use a short and simple MDX query to get the data we want. But we can *filter by product name*.

In order to make our filter work, we'll have to use some new functions. First, we need a way to extract the name from a dimension member. To test this, we'll add a calculated member to our MDX query:

CODE TO TYPE:

```
WITH
set [Beverages]
as
{ [Time].[1998] } * Descendants(  [Product].[Drink].[Beverages].[Hot Beverages], [Product].[Product Na
member [Measures].[Test] as
   [Product].[Product Name].CurrentMember.Name

select
{ [Measures].[Unit Sales], [Measures].[Test] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

Run the query, and you'll see we've successfully extracted the product names:

		Measures	
Time	Product	Unit Sales	Test
1998	BBB Best Hot Chocolate	317	BBB Best Hot Chocolate
	CDR Hot Chocolate	368	CDR Hot Chocolate
	Landslide Hot Chocolate	336	Landslide Hot Chocolate
	Plato Hot Chocolate	393	Plato Hot Chocolate
	Super Hot Chocolate	388	Super Hot Chocolate
	BBB Best Columbian Coffee	367	BBB Best Columbian Coffee
	BBB Best Decaf Coffee	410	BBB Best Decaf Coffee
	BBB Best French Roast Coffee	410	BBB Best French Roast Coffee
	BBB Best Regular Coffee	365	BBB Best Regular Coffee

In this query, we add our calculated measure called **[Measures].[Test]**. It returns the **Name** of the **Current Member** in the

[Product] dimension at the **[Product Name]** level.

So why couldn't we use **[Product].[Product Name]** alone? We couldn't because **[Product].[Product Name]** doesn't represent a single member; it is a set. We *cannot* check a set to see if the product name contains the word "Decaf." We *can* check an individual product to see if its name contains the word "Decaf."

We use the function **CurrentMember** to get the specific current member from the **[Product]** dimension at the **[Product Name]** level.

The function **Name** returns a member's name as a string, something we can use in our filter.

Now that we've extracted the product name, let's use another new function to see if the name contains the word "Decaf." To do this we can use the **InStr** function. **InStr** takes two strings as arguments. The first is the string to search, and the second is the string we want to find. It returns the position where the second string was found in the first string, or it returns zero if nothing was found. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
set [Beverages]
as
{ [Time].[1998] } * Descendants(  [Product].[Drink].[Beverages].[Hot Beverages], [Product].[Product Na
member [Measures].[Test] as
InStr(  [Product].[Product Name].CurrentMember.Name,
    "Decaf"
)

select
{ [Measures].[Unit Sales], [Measures].[Test] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

Run the query. We're making progress! As you can see, the decaf products have values greater than 0 in the **Test** column:

		Measures	
Time	Product	Unit Sales	Test
1998	BBB Best Hot Chocolate	317	0
	CDR Hot Chocolate	368	0
	Landslide Hot Chocolate	336	0
	Plato Hot Chocolate	393	0
	Super Hot Chocolate	388	0
	BBB Best Columbian Coffee	367	0
	BBB Best Decaf Coffee	410	10
	BBB Best French Roast Coffee	410	0
	BBB Best Regular Coffee	365	0
	CDR Columbian Coffee	322	0
	CDR Decaf Coffee	368	5
	CDR French Roast Coffee	377	0

Now we can get rid of our calculated member called Test, and restore our FILTER. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
set [Beverages]
as
FILTER(
{ [Time].[1998] } * Descendants(  [Product].[Drink].[Beverages].[Hot Beverages], [Product].[Product Na
,
 InStr(
   [Product].[Product Name].CurrentMember.Name,
   "Decaf")
 > 0
)

select
{ [Measures].[Unit Sales] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

It looks like this query does the job nicely:

		Measures
Time	Product	Unit Sales
1998	BBB Best Decaf Coffee	410
	CDR Decaf Coffee	368
	Landslide Decaf Coffee	386
	Plato Decaf Coffee	336
	Super Decaf Coffee	363

Functions for Hierarchies

Your boss stops by your desk once again, this time with a request for different information. Now she wants to see 1998 Unit sales for the product sub category of Hot Beverages. You quickly come up with the following query:

CODE TO TYPE:

```
WITH
set [Beverages]
as { [Time].[1998] } * { [Product].[Drink].[Beverages].[Hot Beverages].Children}

select
{ [Measures].[Unit Sales] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

You run it, and the results look good so far:

		Measures
Time	Product	Unit Sales
1998	Chocolate	1,802
	Coffee	7,056

Now she wants to know the percentage of the total unit sales for hot beverages that chocolate and coffee each comprise. Because there are only two values, you can calculate the total unit sales to be **8,858**, and the percentages manually, like this:

Product	Unit Sales	Calculation	% Sales
Chocolate	1,802	1,802 / 8,858	20.34%
Coffee	7,056	7,056 / 8,858	70.66%

Calculating this example by hand doesn't take a whole lot of time, but it would be tedious for larger queries. We need a way to write a query to return this information.

Let's add a calculated member to return the total unit sales of hot beverages. In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
set [Beverages]
as { [Time].[1998] } * { [Product].[Drink].[Beverages].[Hot Beverages].Children}
member [Measures].[All Hot Beverages] as ([Product].[Drink].[Beverages].[Hot Beverages] , [Measures].[

select
{ [Measures].[Unit Sales], [Measures].[All Hot Beverages] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

Here we added a calculated member called **All Hot Beverages** which is the **Unit Sales** for **[Product].[Drink].[Beverages].[Hot Beverages]**:

		Measures	
Time	Product	Unit Sales	All Hot Beverages
1998	Chocolate	1,802	8,858
	Coffee	7,056	8,858

We're in good shape now. Let's add another calculated member that returns our percentage. We'll also use a `FORMAT STRING` to make the display look nice. In MDX mode, type this query:

CODE TO TYPE:

```
WITH
set [Beverages]
as { [Time].[1998] } * { [Product].[Drink].[Beverages].[Hot Beverages].Children}
member [Measures].[All Hot Beverages] as ([Product].[Drink].[Beverages].[Hot Beverages] , [Measures].[
member [Measures].[% of All Hot Beverages] as ([Product].CurrentMember , [Measures].[Unit Sales]) / [M
, FORMAT_STRING="Percent"
select
{ [Measures].[Unit Sales], [Measures].[All Hot Beverages], [Measures].[% of All Hot Beverages] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

Here we added our new member called **% of All Hot Beverages**, which is derived from the **current product's unit sales divided by All Hot Beverages**:

		Measures		
Time	Product	Unit Sales	All Hot Beverages	% of All Hot Beverages
+1998	+Chocolate	1,802	8,858	20.34%
	+Coffee	7,056	8,858	79.66%

Fantastic! Our totals match the manual calculations we did earlier. The calculation works even if you click on the + icon to expand the hierarchy. Try it. Expand **Coffee** and then **BBB Best**:

		Measures		
Time	Product	Unit Sales	All Hot Beverages	% of All Hot Beverages
+1998	+Chocolate	1,802	8,858	20.34%
	-Coffee	7,056	8,858	79.66%
	-BBB Best	1,552	8,858	17.52%
	BBB Best Columbian Coffee	367	8,858	4.14%
	BBB Best Decaf Coffee	410	8,858	4.63%
	BBB Best French Roast Coffee	410	8,858	4.63%
	BBB Best Regular Coffee	365	8,858	4.12%
	+CDR	1,320	8,858	14.90%

Our query shows us that **BBB Best Columbian Coffee's** 367 sales represent 4.14% of the total hot beverage sales.

Now suppose we want to know the percentage of **BBB Best Columbian Coffee's** sales compared to all of **BBB Best**, or the percentage of **CDR Columbian Coffee** sales compared to all of **CDR**. How could we find all that information?

We'll use the **Parent** function. In MDX the **Parent** function can be used to navigate to a member's parent in the hierarchy. Let's try it. (We will rename our members so our calculations are clear to the end users.) In MDX Mode, type this query:

CODE TO TYPE:

```
WITH
set [Beverages]
as { [Time].[1998] } * { [Product].[Drink].[Beverages].[Hot Beverages].Children}
member [Measures].[All Parent] as ([Product].CurrentMember.Parent , [Measures].[Unit Sales])
member [Measures].[% of Parent] as ([Product].CurrentMember , [Measures].[Unit Sales]) / [Measures].[A
, FORMAT_STRING="Percent"
select
{ [Measures].[Unit Sales], [Measures].[All Parent], [Measures].[% of Parent] }
ON COLUMNS,
 [Beverages] ON ROWS
from [Sales]
```

At first glance our result is exactly the same as before:

		Measures		
Time	Product	Unit Sales	All Parent	% of Parent
+1998	+Chocolate	1,802	8,858	20.34%
	+Coffee	7,056	8,858	79.66%

To see what the rest of the hierarchy looks like, expand **Coffee** and then **BBB Best**. MDX has calculated percentages for each level in the hierarchy:

		Measures		
Time	Product	Unit Sales	All Parent	% of Parent
+1998	+Chocolate	1,802	8,858	20.34%
	-Coffee	7,056	8,858	79.66%
	-BBB Best	1,552	7,056	22.00%
	BBB Best Columbian Coffee	367	1,552	23.65%
	BBB Best Decaf Coffee	410	1,552	26.42%
	BBB Best French Roast Coffee	410	1,552	26.42%
	BBB Best Regular Coffee	365	1,552	23.52%
	+CDR	1,320	7,056	18.71%
	+Landslide	1,397	7,056	19.80%
	+Plato	1,368	7,056	19.39%
	+Super	1,419	7,056	20.11%

100%

22.00%

100%

+ 22.00 = 100%

You're doing great so far! You've used filtering, member functions, and hierarchy functions. In the next lesson we will talk about special date functions we can use to calculate year to date values, and compare data from one period to another. See you there!

Traveling in Time
DBA 4: Analyzing Data Lesson 7

Welcome back! In the last lesson we learned to filter and to create percentages. In this lesson we'll look at functions that have been created to work with the date dimension.

Common Date Questions

Year to Date

A common question asked in the business world is **"How are we doing so far?"** There are several ways to answer that question. You could calculate how many things you've sold so far, or compare this year's sales to last year's. Or maybe your company's status changes very quickly, so right now you are interested in comparing only this week's sales to last week's. However you approach the question, you'll need to use the date dimension. Fortunately, MDX has an entire set of functions specifically for the date dimension.

Let's start by looking at one specific question: **"How many units have we sold so far this year?"** To answer this question we need to calculate a *cumulative total*, also known as a *running total*.

For any day, a running total is the sum of the previous day's values. Take a look:

Day	Units Sold	Running Total	Calculation
Jan 01	59	59	= 59
Jan 02	43	102	= 59 + 43
Jan 03	62	164	= 59 + 43 + 62
Jan 04	38	202	= 59 + 43 + 62 + 38

In order to get MDX to return these values for us, we need to use two new functions. The first function is called **AGGREGATE**. **AGGREGATE** tells MDX that we want to see the aggregation of our measure `[Measures].[Unit Sales]`. In this case, the aggregation is a **SUM**.

But wait a minute. MDX has a function called **SUM**, so why don't we use that? We have worked with unit sales before, so we know that SUMming the data is the correct aggregate. But this is not true for all measures.

What if we have a measure that is an account balance? Account balances are usually stored as **point in time** values. Take a look:

Date	Description	Payment	Deposit	Balance
01/01	Starting Balance			**592.20**
01/02	Grocery Store	25.90		**566.30**
01/03	Computer Store	19.50		**546.80**
01/04	Consulting Work		1500.00	**2046.80**

The account balance on **01/03** is **546.80**, and the account balance on **01/04** is **2046.80**. If today is January 4th, the account balance for the month of January is **2046.80**, not 592.20 + 566.30 + 546.80 + 2046.80 = 3752.10. Similarly, the account balance for **2008** is also **2046.80**, not 3752.10.

The proper aggregate for an account balance is not `SUM`, it is `LAST`. The **AGGREGATE** function always uses the correct function as defined in the data warehouse's schema, so there's no risk of using the incorrect aggregate.

The next function we'll use is called **YTD**, which stand for **Year To Date**. **YTD** is a shorthand way of telling MDX that it needs to calculate a running total for the Year level.

We will use these functions in a calculated member called **YTD Unit Sales**. Let's try the query in MDX mode:

CODE TO TYPE:

```
WITH
 member [Measures].[YTD Unit Sales] as
   AGGREGATE (
      YTD(),
      [Measures].[Unit Sales] )
select
{ [Measures].[YTD Unit Sales]  } ON COLUMNS,
{ [Time].[1997] } on ROWS

from [Sales]
```

In this query we see the **calculated member**, the **AGGREGATE** function, and the **YTD** function. We are **aggregating** the **Unit Sales** measure.

Run the query, you'll see these results:

	Measures
Time	YTD Unit Sales
+1997	266,773

So far this isn't very impressive. Click on the + icon to expand 1997, and you'll see the real magic:

	Measures
Time	YTD Unit Sales
-1997	266,773
+Q1	66,291
+Q2	128,901
+Q3	194,749
+Q4	266,773

Here we can see the running total of unit sales across each quarter. If you click to expand **Q1** you'll see the running totals broken down by month:

	Measures
Time	YTD Unit Sales
-1997	266,773
-Q1	66,291
1	21,628
2	42,585
3	66,291
+Q2	128,901
+Q3	194,749
+Q4	266,773

Comparing Sales from Month to Month

In some businesses, last year's sales don't matter as much as last month, or even last week. How do you compare one month to another month? We could create calculated members for two months, and use subtraction to calculate the difference. Let's take a look to see how that might work. In MDX, type this query:

CODE TO TYPE:

```
WITH
 member [Measures].[Jan] as ( [Time].[1997].[Q1].[1], [Measures].[Unit Sales] )
 member [Measures].[Feb] as ( [Time].[1997].[Q1].[2], [Measures].[Unit Sales] )
 member [Measures].[Diff] as [Measures].[Feb] - [Measures].[Jan]
select
{ [Measures].[Jan], [Measures].[Feb], [Measures].[Diff]  } ON COLUMNS
from [Sales]
```

In this query we've created calculated members for **January** and **February** which are then used to calculate the **difference** in unit sales for those two months. Run the query, and you'll see the results:

Measures		
Jan	Feb	Diff
21,628	20,957	-671

everything looks good, but it would be great if we could just calculate the difference without having to specify each month. We can make short work of this using a special MDX function called **PrevMember**.

The **PrevMember** function returns the previous member in a level. In other words, February's **PrevMember** is January. Let's try it in our existing query to see how it works. In MDX, type this query:

CODE TO TYPE:

```
WITH
 member [Measures].[Jan] as ( [Time].[1997].[Q1].[2].PrevMember, [Measures].[Unit Sales] )
 member [Measures].[Feb] as ( [Time].[1997].[Q1].[2], [Measures].[Unit Sales] )
 member [Measures].[Diff] as [Measures].[Feb] - [Measures].[Jan]
select
{ [Measures].[Jan], [Measures].[Feb], [Measures].[Diff]  } ON COLUMNS
from [Sales]
```

Run the query and you'll see the same result as before:

Measures		
Jan	Feb	Diff
21,628	20,957	-671

PrevMember works with any dimension, not just date dimensions. Armed with this new function, let's rewrite our query so we can compare the current month to the prior month. In MDX, type this query:

CODE TO TYPE:

```
WITH
 member [Measures].[Current Unit Sales] as ( [Time].CurrentMember, [Measures].[Unit Sales] )
 member [Measures].[Prior Unit Sales] as ( [Time].CurrentMember.PrevMember, [Measures].[Unit Sa
 member [Measures].[Diff] as [Measures].[Current Unit Sales] - [Measures].[Prior Unit Sales]
select
{ [Measures].[Current Unit Sales], [Measures].[Prior Unit Sales], [Measures].[Diff]  } ON COLUM
{ [Time].[1997].[Q1].[1]:[Time].[1998].[Q4].[12] } on ROWS
from [Sales]
```

In this query we've added a calculated member for the **current month**, and another calculated member for the **prior month**, using the **PrevMember** function. We calculate the **difference** the same way as before.

Finally, we use the **colon :** to tell MDX that we want all months between January 1997 and December 1998 on our ROWS axis. The **colon :** is inclusive, so both January 1997 and December 1998 are included in the axis.

Run the query, you'll see these results:

	Measures		
Time	Current Unit Sales	Prior Unit Sales	Diff
1	21,628		21,628
2	20,957	21,628	-671
3	23,706	20,957	2,749
4	20,179	23,706	-3,527
5	21,081	20,179	902
6	21,350	21,081	269
7	23,763	21,350	2,413
8	21,697	23,763	-2,066
9	20,388	21,697	-1,309
10	19,958	20,388	-430

That looks good, but it's hard to tell where 1997 ends and 1998 begins. To remedy this issue, click on the "Show Parents" button:

Now our results are much easier to understand:

Time			Measures		
Year	Quarter	Month	Current Unit Sales	Prior Unit Sales	Diff
1997	Q1	1	21,628		21,628
		2	20,957	21,628	-671
		3	23,706	20,957	2,749
	Q2	4	20,179	23,706	-3,527
		5	21,081	20,179	902
		6	21,350	21,081	269
	Q3	7	23,763	21,350	2,413
		8	21,697	23,763	-2,066
		9	20,388	21,697	-1,309
	Q4	10	19,958	20,388	-430
		11	25,270	19,958	5,312
		12	26,796	25,270	1,526
1998	Q1	1	46,313	26,796	19,517
		2	44,431	46,313	-1,882

So why do you suppose the prior unit sales are blank for January 1997? That's because we don't have any sales data for the prior month, December 1996. Our data starts on January 1st, 1997.

Prior Quarter

The query we just looked at compares the current month to the previous month. Now let's suppose we want to compare the current month to three months ago? To do that, we'll use a new function called **Lag**. You can use **Lag** to go back a specified number of members. For example, if you use **Lag(3)** on April, you'll be in January. **Lag(1)** is exactly the same as PrevMember.

Let's try using **Lag**, replacing PrevMember with **Lag(3)**. In MDX, type this query:

CODE TO TYPE:

```
WITH
 member [Measures].[Current Unit Sales] as ( [Time].CurrentMember, [Measures].[Unit Sales] )
 member [Measures].[Prior Unit Sales] as ( [Time].CurrentMember.Lag(3), [Measures].[Unit Sales]
 member [Measures].[Diff] as [Measures].[Current Unit Sales] - [Measures].[Prior Unit Sales]
select
{ [Measures].[Current Unit Sales], [Measures].[Prior Unit Sales], [Measures].[Diff]  } ON COLUM
{ [Time].[1997].[Q1].[1]:[Time].[1998].[Q4].[12] } on ROWS
from [Sales]
```

Run the query, and you'll see these results:

Time			Measures		
Year	Quarter	Month	Current Unit Sales	Prior Unit Sales	Diff
1997	Q1	1	21,628		21,628
		2	20,957		20,957
		3	23,706		23,706
	Q2	4	20,179	21,628	-1,449
		5	21,081	20,957	124
		6	21,350	23,706	-2,356
	Q3	7	23,763	20,179	3,584
		8	21,697	21,081	616
		9	20,388	21,350	-962
	Q4	10	19,958	23,763	-3,805

This looks great!

Parallel Periods

Suppose your manager wanted to compare 1998 sales to 1997 sales, but also wants the flexibility to compare month to month, quarter to quarter, or year to year. That seems like a tall order, doesn't it?

Lucky for you, MDX has a specific function to solve this problem - **ParallelPeriod**. If your current member is March 1998, **ParallelPeriod** will take you back to March 1997. If your current member is Q2 1998, **ParallelPeriod** will take you back to Q2 1997.

Let's try it! In MDX, type this query:

CODE TO TYPE:

```
WITH
 member [Measures].[Current Unit Sales] as ( [Time].CurrentMember, [Measures].[Unit Sales] )
 member [Measures].[Prior Unit Sales] as (
    ParallelPeriod(
      [Time].[Year] ,
      1 ,
      [Time].CurrentMember)
      , [Measures].[Unit Sales] )
 member [Measures].[Diff] as [Measures].[Current Unit Sales] - [Measures].[Prior Unit Sales]
select
{ [Measures].[Current Unit Sales], [Measures].[Prior Unit Sales], [Measures].[Diff]  } ON COLUM
{ [Time].[1997]:[Time].[1998] } on ROWS
from [Sales]
```

ParallelPeriod takes three arguments or parameters. The first is the level you want to use to go back in time. We want to go back years, so we use **[Time].Year**. We want to go back one year, so we use **1** for the second parameter. The third parameter tells MDX that we want to go back one year from the **CurrentMember** of the time dimension.

We've simplified our **rows axis** by specifying 1997 and 1998, but no specific months.

Run the query, and you'll see the results below:

Time	Measures		
Year	Current Unit Sales	Prior Unit Sales	Diff
+1997	266,773		266,773
+1998	566,716	266,773	299,943

Expand 1998, and you'll see the sales broken down by quarter. The differences are also correctly calculated:

Time		Measures		
Year	Quarter	Current Unit Sales	Prior Unit Sales	Diff
+1997		266,773		266,773
-1998		566,716	266,773	299,943
1998	+Q1	137,078	66,291	70,787
	+Q2	135,745	62,610	73,135
	+Q3	139,412	65,848	73,564
	+Q4	154,481	72,024	82,457

Expand Q1 for 1998 and you'll see sales broken down by month:

Time			Measures		
Year	Quarter	Month	Current Unit Sales	Prior Unit Sales	Diff
+1997			266,773		266,773
-1998			566,716	266,773	299,943
1998	-Q1		137,078	66,291	70,787
	Q1	1	46,313	21,628	24,685
		2	44,431	20,957	23,474
		3	46,334	23,706	22,628
	+Q2		135,745	62,610	73,135
	+Q3		139,412	65,848	73,564
	+Q4		154,481	72,024	82,457

Wow, we received a lot of information, and we didn't even have to change our query!

Good job so far. In the next lesson we'll look at some "top" queries that can be used to answer a variety of questions. See you then!

Top Queries
DBA 4: Analyzing Data Lesson 8

Good to have you back! In the last lesson we learned how to use functions to travel around the time dimension. In this lesson we'll look at functions that help us answer some common "top" questions.

Top

Your boss stops by one morning with a fairly straightforward question: **"What are the top five products by unit sales?"**

To answer that question, first let's write a query to return the unit sales for all products. We'll use the **Descendants** function. In MDX Mode, type this query:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
{ Descendants([Product], [Product].[Product Name]) } on ROWS
from [Sales]
```

Run the query and you'll see *many* results:

	Measures
Product	Unit Sales
Good Imported Beer	606
Good Light Beer	534
Pearl Imported Beer	589
Pearl Light Beer	593
Portsmouth Imported Beer	597
Portsmouth Light Beer	563
Top Measure Imported Beer	459
Top Measure Light Beer	475
Walrus Imported Beer	536
Walrus Light Beer	522

At this point the results are really mixed up. In SQL we use `ORDER BY` to sort results, but in MDX we use the `ORDER` function. Let's try it. In MDX Mode, type this query:

CODE TY TYPE:

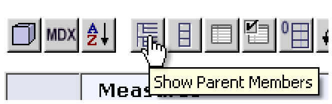
```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
Order(
 Descendants([Product], [Product].[Product Name]) ,
 [Measures].[Unit Sales],
 DESC ) ON ROWS
from [Sales]
```

The **ORDER** command takes three parameters. The first is the **set** to sort, in this case **Descendants([Product], [Product].[Product Name])**. The next parameter is the data to **sort by**. Here we want to sort by **unit sales**. The third and last parameter specifies how we want to sort, either ASCending or **DESC**ending.

At first it seems like the data is completely sorted. But scroll down - you'll see something that looks a bit strange:

Hermanos Mushrooms	525
Hermanos Garlic	514
Hermanos Asparagus	511
Hermanos Dried Mushrooms	502
Tell Tale Fresh Lima Beans	698
Tell Tale Lettuce	630
Tell Tale Sweet Onion	605

The data returned *is* sorted, but the **ORDER** function kept the hierarchy intact. You can confirm this by clicking on the "Show Parents" button:



Your results display will be fairly wide:

Product						Measures
Product Family	Product Department	Product Category	Product Subcategory	Brand Name	Product Name	Unit Sales
Food	Produce	Vegetables	Fresh Vegetables	Hermanos	Hermanos Green Pepper	645
					Hermanos Potatos	643
					Hermanos Elephant Garlic	631
					Hermanos Tomatos	619
					Hermanos Red Pepper	601
					Hermanos Sweet Peas	601

We need to ignore the hierarchy so that we can find the overall products with the most unit sales. To do this, we can use the **ORDER** function with a different third parameter. In MDX Mode, type this query:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
Order(
 Descendants([Product], [Product].[Product Name]),
 [Measures].[Unit Sales],
 BDESC ) ON ROWS
from [Sales]
```

The **BDESC** function means "break hierarchy." When you run it you'll see much different results:

Product						Measures
Product Family	Product Department	Product Category	Product Subcategory	Brand Name	Product Name	Unit Sales
Food	Produce	Vegetables	Fresh Vegetables	Tell Tale	Tell Tale Fresh Lima Beans	698
Non-Consumable	Health and Hygiene	Hygiene	Personal Hygiene	Steady	Steady Whitening Toothpast	684
		Bathroom Products	Mouthwash	Hilltop	Hilltop Mint Mouthwash	676
Food	Deli	Meat	Fresh Chicken	Moms	Moms Roasted Chicken	675
Non-Consumable	Health and Hygiene	Bathroom Products	Conditioner	Hilltop	Hilltop Silky Smooth Hair Conditioner	665

Here we can see that the top item is **Tell Tale Fresh Lima Beans**.

If we were writing a query using MySQL, we could now use the `LIMIT` keyword to restrict our results to the top five products. How do we do this using MDX?

We'll forget about using ORDER, and use a function called **TopCount** instead. **TopCount** is like having MySQL's ORDER and LIMIT, all combined into one function.

To determine the "top" values, **TopCount** implicitly sorts data in descending order, breaking the hierarchy. Let's try it out. In MDX Mode, type this query:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
TopCount(
 Descendants([Product], [Product].[Product Name]),
 5,
 [Measures].[Unit Sales]
 ) ON ROWS
from [Sales]
```

TopCount takes three parameters. The first is the **set** to sort, in this case **Descendants([Product], [Product].[Product Name])**. The second is the number of items you want to return, **5**. The third parameter is the data used to determine the **top** values; for us this is the **unit sales**. And here we have our top five values:

Product						Measures
Product Family	Product Department	Product Category	Product Subcategory	Brand Name	Product Name	Unit Sales
Food	Produce	Vegetables	Fresh Vegetables	Tell Tale	Tell Tale Fresh Lima Beans	698
Non-Consumable	Health and Hygiene	Hygiene	Personal Hygiene	Steady	Steady Whitening Toothpast	684
		Bathroom Products	Mouthwash	Hilltop	Hilltop Mint Mouthwash	676
Food	Deli	Meat	Fresh Chicken	Moms	Moms Roasted Chicken	675
Non-Consumable	Health and Hygiene	Bathroom Products	Conditioner	Hilltop	Hilltop Silky Smooth Hair Conditioner	665

We are looking good.

Now suppose your boss stops by your desk again, this time with a slightly different question: **"What products represent the top 1% of our unit sales?"**

In order to answer this question, we need to know the total unit sales for all products, then sort the products by unit sales. The subset of products whose combined unit sales is at least 1% of the total is returned.

Fortunately, MDX has a built in function to handle this work for us called **TopPercent**. Let's try it. In MDX Mode, type this query:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
TopPercent(
 Descendants([Product], [Product].[Product Name]),
 1,
 [Measures].[Unit Sales]
 ) ON ROWS
from [Sales]
```

We are using the **TopPercent** function on the same set -- **Descendants([Product], [Product].[Product Name])** -- but this time we want the top **1%** of **[Measures].[Unit Sales]**.

Run the query. You'll see the following results:

Product						Measures
Product Family	Product Department	Product Category	Product Subcategory	Brand Name	Product Name	Unit Sales
Food	Produce	Vegetables	Fresh Vegetables	Tell Tale	Tell Tale Fresh Lima Beans	698
Non-Consumable	Health and Hygiene	Hygiene	Personal Hygiene	Steady	Steady Whitening Toothpast	684
		Bathroom Products	Mouthwash	Hilltop	Hilltop Mint Mouthwash	676
Food	Deli	Meat	Fresh Chicken	Moms	Moms Roasted Chicken	675
Non-Consumable	Health and Hygiene	Bathroom Products	Conditioner	Hilltop	Hilltop Silky Smooth Hair Conditioner	665
Food	Baked Goods	Bread	Muffins	Great	Great English Muffins	663
Non-Consumable	Health and Hygiene	Cold Remedies	Cold Remedies	Steady	Steady Childrens Cold Remedy	658
Food	Produce	Specialty	Nuts	Ebony	Ebony Mixed Nuts	653
Non-Consumable	Household	Bathroom Products	Toilet Brushes	Sunset	Sunset Economy Toilet Brush	651
Food	Produce	Vegetables	Fresh Vegetables	Ebony	Ebony Fresh Lima Beans	648
Non-Consumable	Household	Cleaning Supplies	Cleaners	Red Wing	Red Wing Glass Cleaner	647
Food	Breakfast Foods	Breakfast Foods	Cereal	Special	Special Wheat Puffs	645
	Produce	Vegetables	Fresh Vegetables	Hermanos	Hermanos Green Pepper	645

This is *much* easier than returning the same data with SQL!

Generate

You've answered many of your boss's questions lately, and she is very happy with the results. But the questions keep coming! Now she asks, **"What were the top five products in 1997 and the top five products in 1998?"**

You sit at your keyboard, and bring up the query from earlier in the lesson. You change it slightly, adding 1997 and 1998 to the TopCount. In MDX Mode, type this query:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
TopCount(
 ( { [Time].[1997], [Time].[1998] } * Descendants([Product], [Product].[Product Name]) ),
 5,
 [Measures].[Unit Sales]
 ) ON ROWS
from [Sales]
```

The results are not exactly what you were looking for. In fact it seems there is no data from 1997:

Time	Product						Measures
Year	Product Family	Product Department	Product Category	Product Subcategory	Brand Name	Product Name	Unit Sales
1998	Food	Baked Goods	Bread	Muffins	Great	Great English Muffins	499
		Deli	Meat	Fresh Chicken	Moms	Moms Roasted Chicken	479
			Side Dishes	Deli Salads	Moms	Moms Potato Salad	469
		Produce	Vegetables	Fresh Vegetables	Tell Tale	Tell Tale Fresh Lima Beans	467
		Baked Goods	Bread	Sliced Bread	Great	Great Pumpernickel Bread	466

As it turns out, we answered a slightly different question: **"What were the top five products from 1997 *or* 1998?"**

Essentially we made a list of sales from 1997 and 1998, and then picked the top five products from that list. Sales were really high in 1998, so all of the top five products were from 1998.

One way we can get around this problem is to calculate TopCount for 1997 and TopCount for 1998, and then combine the results with a **Union**. In MDX Mode, type this query:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
Union (
TopCount(
 ( { [Time].[1997] } * Descendants([Product], [Product].[Product Name]) ),
 5,
 [Measures].[Unit Sales]
 )
,
TopCount(
 ( { [Time].[1998] } * Descendants([Product], [Product].[Product Name]) ),
 5,
 [Measures].[Unit Sales]
 )
)
 ON ROWS
from [Sales]
```

Here we use **Union** to combine the top sales for **1997** and the top sales for **1998**. These results look much better:

Time	Product						Measures
Year	Product Family	Product Department	Product Category	Product Subcategory	Brand Name	Product Name	Unit Sales
1997	Food	Breakfast Foods	Breakfast Foods	Cereal	Special	Special Wheat Puffs	267
		Snack Foods	Snack Foods	Dried Meat	Fast	Fast Beef Jerky	258
		Produce	Vegetables	Fresh Vegetables	Hermanos	Hermanos Broccoli	257
	Drink	Beverages	Pure Juice Beverages	Juice	Fabulous	Fabulous Apple Juice	252
	Food	Frozen Foods	Pizza	Pizza	Big Time	Big Time Frozen Mushroom Pizza	246
1998	Food	Baked Goods	Bread	Muffins	Great	Great English Muffins	499
		Deli	Meat	Fresh Chicken	Moms	Moms Roasted Chicken	479
			Side Dishes	Deli Salads	Moms	Moms Potato Salad	469
		Produce	Vegetables	Fresh Vegetables	Tell Tale	Tell Tale Fresh Lima Beans	467
		Baked Goods	Bread	Sliced Bread	Great	Great Pumpernickel Bread	466

The last MDX query certainly worked, but it required quite a bit of repetitive typing. If our manager wanted us to return the top 20 products for a period of 5 years, we would have a lot of work to do. There must be a better way. Of course there is. MDX has a function called **Generate** that can make life much easier.

Generate takes two sets as parameters. Tuples in the second set are applied to each tuple in the first set, and the results are *unioned* and returned. In other words, **Generate** does exactly what we just did, but without all that typing.

Let's take a look. In MDX Mode, type this query:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
Generate (
{ [Time].[1997], [Time].[1998] } ,
 TopCount(
  ( { [Time].CurrentMember } * Descendants([Product], [Product].[Product Name]) ),
  5,
  [Measures].[Unit Sales]
  )
 )
 ON ROWS
from [Sales]
```

The first parameter to **Generate** is our set of tuples from the **time** dimension. The second parameter is our `TopCount` function - but this time we use **Time.CurrentMember** instead of actually specifying a member from the time dimension.

When this MDX query runs, the TopCount for 1997 is calculated, then the TopCount for 1998 is calculated. The results are *unioned* and returned:

Time	Product						Measures
Year	Product Family	Product Department	Product Category	Product Subcategory	Brand Name	Product Name	Unit Sales
1997	Food	Breakfast Foods	Breakfast Foods	Cereal	Special	Special Wheat Puffs	267
		Snack Foods	Snack Foods	Dried Meat	Fast	Fast Beef Jerky	258
		Produce	Vegetables	Fresh Vegetables	Hermanos	Hermanos Broccoli	257
	Drink	Beverages	Pure Juice Beverages	Juice	Fabulous	Fabulous Apple Juice	252
	Food	Frozen Foods	Pizza	Pizza	Big Time	Big Time Frozen Mushroom Pizza	246
1998	Food	Baked Goods	Bread	Muffins	Great	Great English Muffins	499
		Deli	Meat	Fresh Chicken	Moms	Moms Roasted Chicken	479
			Side Dishes	Deli Salads	Moms	Moms Potato Salad	469
		Produce	Vegetables	Fresh Vegetables	Tell Tale	Tell Tale Fresh Lima Beans	467
		Baked Goods	Bread	Sliced Bread	Great	Great Pumpernickel Bread	466

Generate saved us a lot of work!

We learned a lot in this lesson. In the next lesson we'll learn one last MDX function that will help us predict the future. Stay tuned!

Predicting the Future
DBA 4: Analyzing Data Lesson 9

How to Predict the Future

People have been trying to predict the future for a long time. We want to know what the weather will be like tomorrow, if our stock will go up in value, or what our sales will be next month. When most people try to predict the future, they are usually refer to historical data as guide for future activity. So how does it work for us as database administrators?

Let's use an example. Suppose you set up a lemonade stand. You carefully record your sales during your first week:

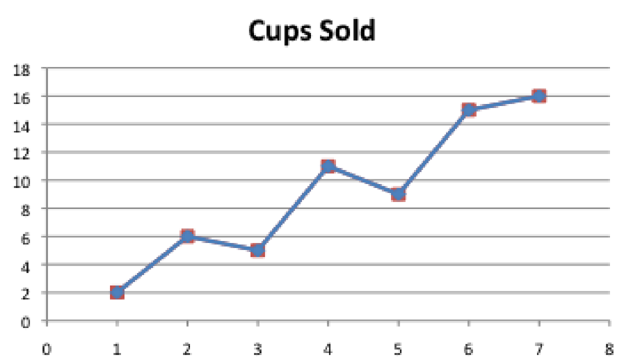
Day	Cups Sold
1	5
2	10
3	15
4	20
5	25
6	30
7	35

Your business is really growing! If this trend continues, you could probably expect to sell **40** cups of lemonade on your eighth day in business. You can make this prediction because on each day you sold five more cups than the day before.

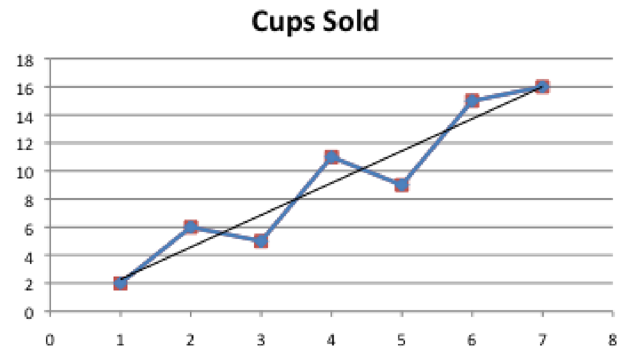
In the real world sales are rarely this nice and orderly. In fact, real world sales might look something like this:

Day	Cups Sold
1	2
2	6
3	5
4	11
5	9
6	15
7	16

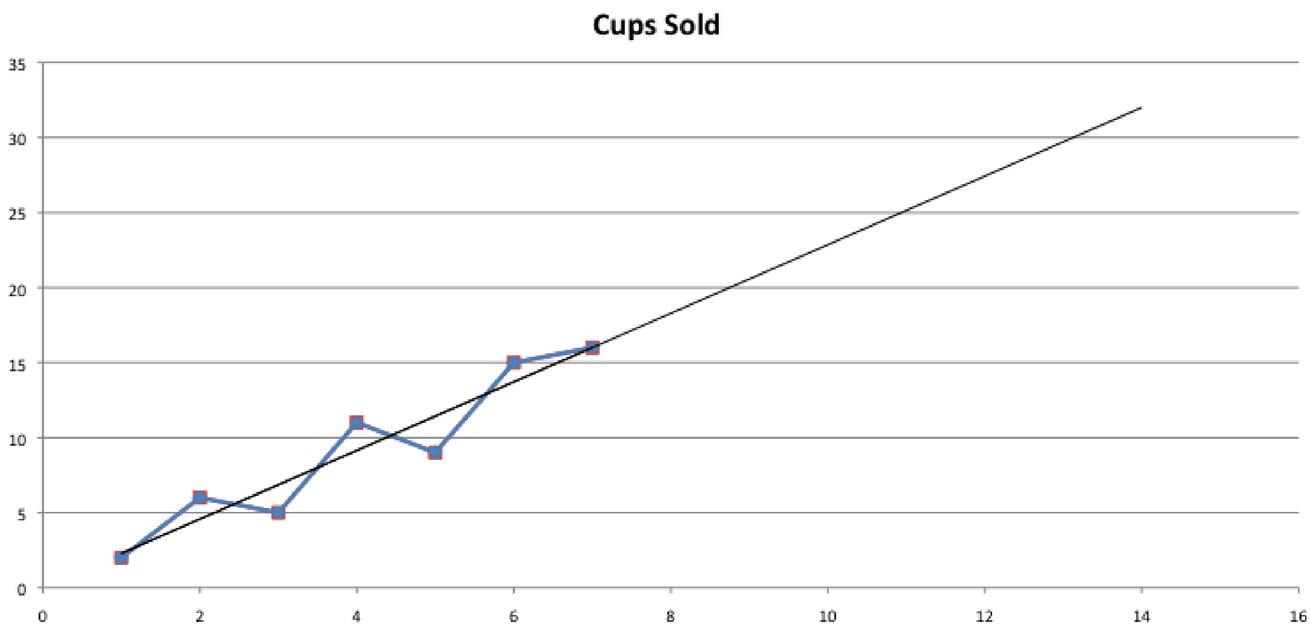
We can graph this data, and come up with this chart:



Using our ruler, we can attempt to draw a nice line through our graph to show how our sales are increasing:



While we have our ruler out, let's extend our line farther:



We can use this line to estimate how many cups of lemonade we will sell on the eighth, ninth and tenth days. Our estimates put the next weeks sales at:

Day	Cups Sold
8	18
9	21
10	23
11	25
12	27
13	30
14	32

These estimates are a lot like weather forecasts. Short-term forecasts are usually more accurate than long-term forecasts. This type of analysis is known as *linear regression*. It's a formal way of saying "given a set of data, try to draw a straight line on a graph to represent the data."

Linear regression isn't the only type of analysis possible, and it may not be the best type of analysis for your data. It just happens to be the algorithm that's been implemented as a function in MDX.

Predicting the Future with MDX

Now that we have a method for predicting the future, let's discuss the MDX function for *linear regression*.

First we need to come up with our data set. We usually want to use the smallest possible grain when we forecast data. That's because we can always aggregate to a higher grain, but we can't split the data into a smaller grain. That's why we should usually use daily data.

To keep things as simple as possible for these examples, we'll use monthly sales for 1998 instead of daily sales. Let's start by typing this query in MDX Mode:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS,
Descendants( [Time].[1998] , [Time].[Month]) ON ROWS
from [Sales]
```

Mondrian returns the daily sales for 1998:

	Measures
Time	Unit Sales
1	46,313
2	44,431
3	46,334
4	45,049
5	45,085
6	45,611
7	46,671
8	44,777
9	47,964
10	43,945
11	53,807
12	56,729

Note Remember - if you want to see the quarters and months, click on the icon to show parent members.

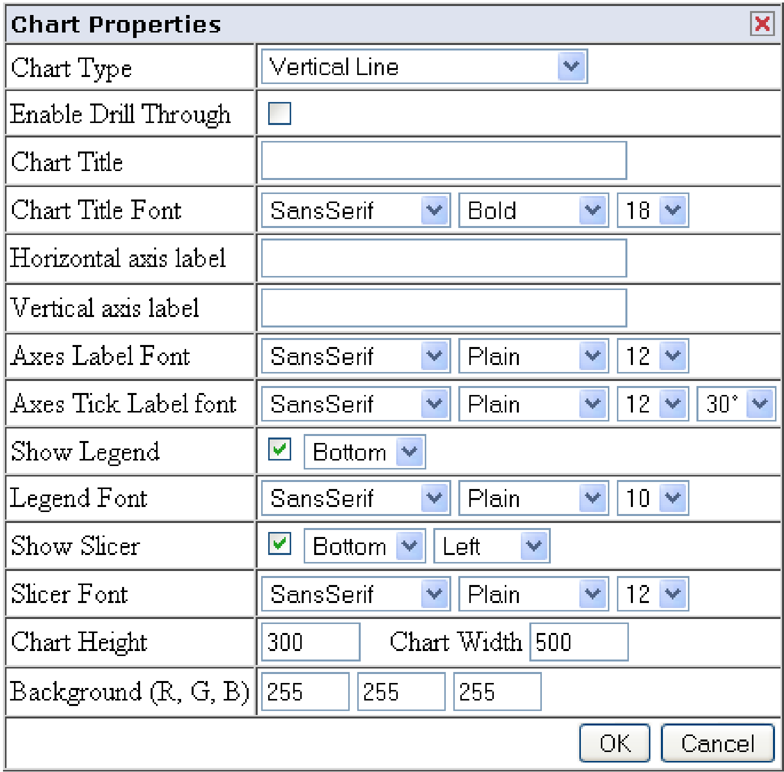
That's really nice, but it would be even nicer to see a graph with this data. Two-dimensional graphs use pairs of data points - an X value and a Y value, usually written as `( X , Y )`. In our case, our data points are `(` *`month`*, *`unit sales`* `)`. We'll write *month* as `Year.Quarter.Month`.

OBSERVE:

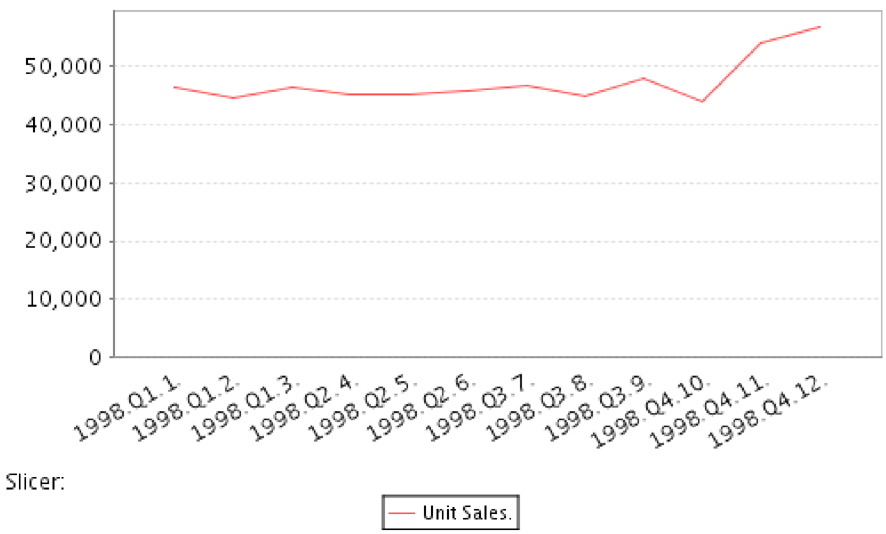
```
( month       , unit sales )
( 1998.Q1.1   , 46313      )
( 1998.Q1.2   , 44431      )
( 1998.Q1.3   , 46334      )
( 1998.Q2.4   , 45049      )
( 1998.Q2.5   , 45085      )
( 1998.Q2.6   , 45611      )
( 1998.Q3.7   , 46671      )
( 1998.Q3.8   , 44777      )
( 1998.Q3.9   , 47964      )
( 1998.Q4.10  , 43945      )
( 1998.Q4.11  , 53807      )
( 1998.Q4.12  , 56729      )
```

Our MDX client JPivot has simple graphing capabilities built into it. JPivot can plot these data points and link them together using a line in a *Vertical Line Chart*. Before we view a chart, we need to set some properties. To do this, click on the **Config Chart** icon on the toolbar:

Next, change the **Chart Type** to **Vertical Line** and click OK:



Finally, click on the **Show Chart** button to see the result. You'll see each data point, such as `(1998.Q1.1 , 46313)`. Here the month is written out with its parent members, year, and quarter:



Excellent. This graph is showing us the *Unit Sales* on the **Y-Axis** as values between 40,000 and 50,000. The *Months* are on the **X-Axis** as values from `1998.Q1.01` to `1999.Q4.12`.

One function MDX has to calculate *linear regression* is called **LinRegPoint**. Conceptually speaking, the **LinRegPoint** function will take two parameters - the first being a **date X-value** and the second being the set of "real" unit sales data. It will return a corresponding forecasted **unit sales Y-value**. In other words, we can use the **LinRegPoint** function to come up with the expected unit sales for months where we have sales data, and for months in the future where we do not have sales data:

OBSERVE:

```
print LinRegPoint('1998.Q1.1', set of 1998 unit sales);
// output is 43,439

print LinRegPoint('1999.Q1.1', set of 1998 unit sales);
// output is 51,702
```

The "real" MDX function **LinRegPoint** takes four parameters instead of two. The first parameter is the **date X-Value** we want to see, and the next three specify the set of real data. Let's see how this function works first, then we'll break down each parameter. In MDX

Mode, type this query:

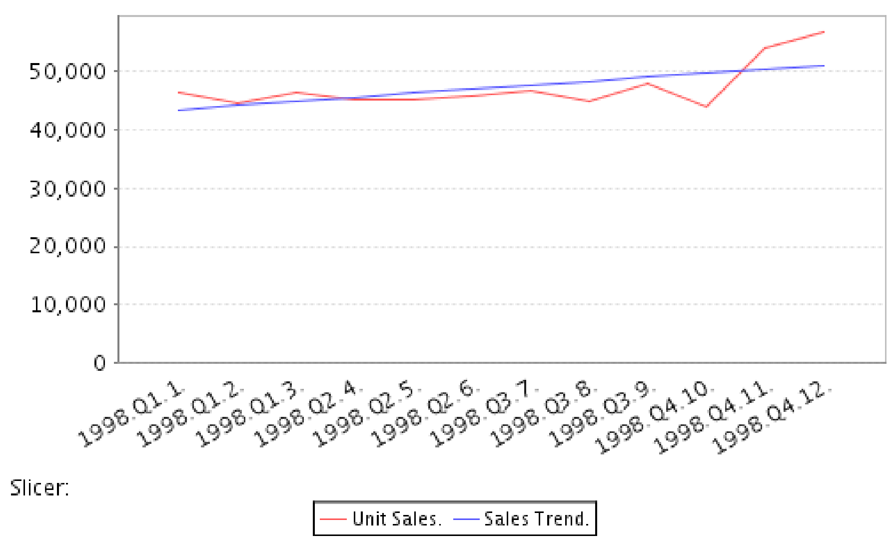
CODE TO TYPE:

```
with member
[Measures].[Sales Trend]
as
LinRegPoint(
 Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS),
 Descendants( [Time].[1998] , [Time].CurrentMember.Level),
 [Measures].[Unit Sales],
 Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS)
)
select
   { [Measures].[Unit Sales], [Measures].[Sales Trend] } ON COLUMNS,
   Descendants( [Time].[1998] , [Time].[Month]) ON ROWS
from [Sales]
```

Run this query, and you'll see these results:

	Measures	
Time	Unit Sales	Sales Trend
1	46,313	43,439
2	44,431	44,128
3	46,334	44,816
4	45,049	45,505
5	45,085	46,193
6	45,611	46,882
7	46,671	47,571
8	44,777	48,259
9	47,964	48,948
10	43,945	49,637
11	53,807	50,325
12	56,729	51,014

The corresponding graph looks like this (our trend/forecast line is in **blue**):



So how does the **LinRegPoint** function work? Let's take a closer look at the first parameter:

OBSERVE:

```
LinRegPoint(
 Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS ),
 Descendants( [Time].[1998] , [Time].CurrentMember.Level),
 [Measures].[Unit Sales],
 Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS)
)
```

The first parameter to **LinRegPoint** is the **date X-Value** we want to forecast, but it needs to be numeric. Our date dimension has values like `1998.Q1.1` to `1998.Q4.12`, but this is not numeric.

In order to come up with a numeric value, we use the MDX function called **Rank**. In English, this means the **Rank** function searches for the occurrence of the **value** called **Time.CurrentMember** in a **set** called **Time.CurrentMember.Level.MEMBERS**. It returns the position of the element within the set. So, the *Rank* of `1998.Q1.1` is **1** because it is the first member in the set, and the *Rank* of `1998.Q3.8` is **8** because it is the eighth member of the set.

Let's take a look at the next three parameters:

OBSERVE:

```
LinRegPoint(
 Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS ),
 Descendants( [Time].[1998] , [Time].CurrentMember.Level),
 [Measures].[Unit Sales],
 Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS)
)
```

The **second** and **third** parameters combined make up the set of **real Y-values**.

The **second** and **fourth** parameters combined make up the set of **real X-values**. We use **Rank** here for exactly the same reason we used Rank for the first parameter.

To put it another way, the four parameters are:

OBSERVE:

```
LinRegPoint(
 What month X-Value do we want to forecast?
  Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS ),
 What is the set of real X and Y Values?
  Descendants( [Time].[1998] , [Time].CurrentMember.Level),
  [Measures].[Unit Sales],
  Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS)
)
```

So how do we extend this into the future, or at least into 1999, where we don't have any sales data? We extend our **Rows** axis by **Unioning** the months from 1998 with the months from 1999. In MDX Mode, type this query:

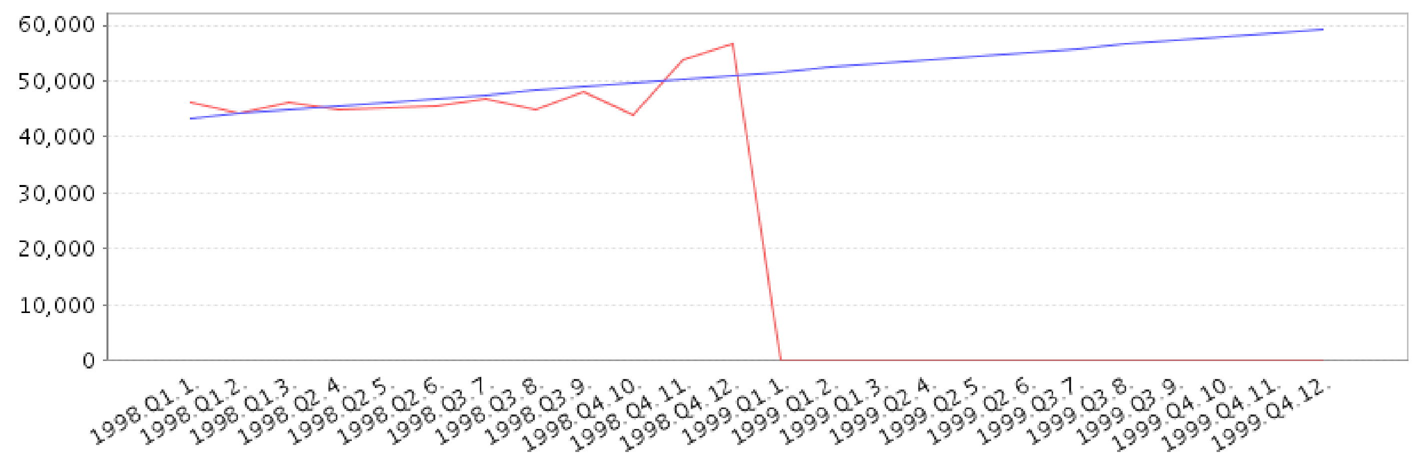
CODE TO TYPE:

```
with member
[Measures].[Sales Trend]
as
LinRegPoint(
 Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS),
 Descendants( [Time].[1998] , [Time].CurrentMember.Level),
 [Measures].[Unit Sales],
 Rank(Time.CurrentMember, Time.CurrentMember.Level.MEMBERS)
)
select
   { [Measures].[Unit Sales], [Measures].[Sales Trend] } ON COLUMNS,
   Union (
   Descendants( [Time].[1998] , [Time].[Month]),
   Descendants( [Time].[1999] , [Time].[Month])
) ON ROWS
from [Sales]
```

The LinRegPoint is able to answer our query:

	Measures	
Time	Unit Sales	Sales Trend
+1	46,313	43,439
+2	44,431	44,128
+3	46,334	44,816
+4	45,049	45,505
+5	45,085	46,193
+6	45,611	46,882
+7	46,671	47,571
+8	44,777	48,259
+9	47,964	48,948
+10	43,945	49,637
+11	53,807	50,325
+12	56,729	51,014
+1		51,702
+2		52,391
+3		53,080
+4		53,768
+5		54,457
+6		55,146
+7		55,834
+8		56,523
+9		57,211
+10		57,900
+11		58,589
+12		59,277

Now change the chart configuration so that the **Chart Width** is 800 and you can view the results for all of 1999. Then take a look at the chart. It should look like this:



Slicer:

You'll see the **real unit sales** drop to zero after 1998 because there are no recorded sales for 1999. The **forecasted unit sales** show good growth in 1999, reaching almost 60,000 in December 1999.

Now let's say you want to forecast sales through 2000, or 2001. For most data warehouses you can do this because the time dimension usually extends far into the future. Our data warehouse has been kept small though, so our time dimension only has values from 1997 to 1999.

Linear regression is just one technique that we can use to forecast the future. Depending on your data, it may not be the most accurate method, but it can help you get close.

In the next lesson we'll switch gears and discuss the way Mondrian connects to the relational data warehouse through its *schema*. Keep going, you're doing great!

Mondrian Schemas
DBA 4: Analyzing Data Lesson 10

We've learned a lot of MDX so far. Now it's time to take a step back, and examine just how the OLAP server Mondrian translates our MDX queries into something the underlying database can interpret.

Writing a Basic Schema

In lesson 1, we described the process for answering MDX queries. It works like this:

1. An MDX query is sent to the OLAP (OnLine Analytical Processing) server.
2. The OLAP server consults its *schema* to see how it can answer the query.
3. The OLAP server may answer the query using its own data structures or cache, or it will generate SQL queries in order to retrieve the data it needs to answer the query.
4. Results are computed, and returned to you.

In the last course we built a relational data warehouse. Now it's time to write a *schema* to allow our OLAP server, Mondrian, to answer MDX queries. We'll use the same pre-built data warehouse we've used throughout this course: `foodmart`.

So, you ask, "why Mondrian can't just read the database tables and structures directly and avoid this whole schema business?' The answer: *flexibility*.

In these courses we presented one way to develop a data warehouse. In real life there are many variations of the core data warehousing principals. Instead of using MySQL, some companies might use PostgreSQL. Instead of using a *date* dimension, some companies may use a *time* dimension.

It's your responsibility as the database administrator to write a schema to link the relational data warehouse to the multidimensional data warehouse.

The Structure of a Schema

If you recall from lesson 2, a multi-dimensional data warehouse is made up of one or more *cubes*. Cubes have dimensions and facts. Some dimensions (such as **Time** or date) may be used across multiple cubes. Those dimensions are known as *shared dimensions*.

A Mondrian schema is stored in an XML file. It has this structure:

OBSERVE:

```
Dimension  (shared dimension)
 Hierarchy
  Level
   Property
Cube
 Dimension Usage
 Dimension
 Measure
 Calculated Measures
Virtual Cube
```

There are three major sections to the schema. The shared dimensions, cube definitions, and then virtual cube definitions.

To develop our schema, we'll use the XML and MDX modes in CodeRunner. You'll write your schema in XML mode, then switch to MDX mode to write a query to test your schema.

XML Documents

If you have never used XML before, don't fear! To get started, switch to XML mode and type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
</Schema>
```

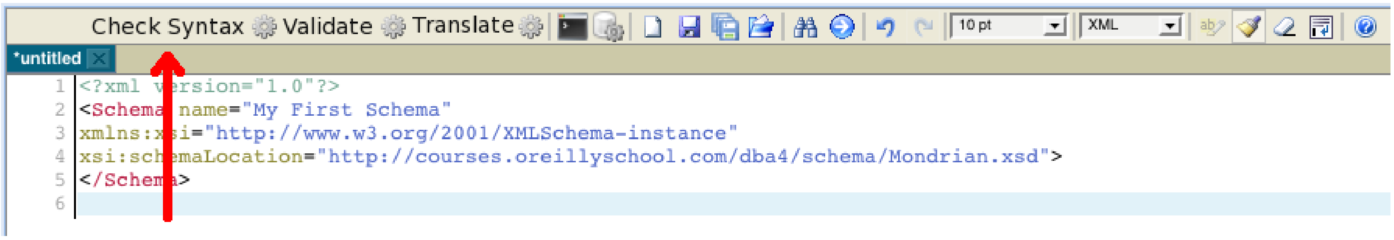
Note When you save XML files, make sure you use **.xml** as the extension.

Now let's go over a few of the basic properties of XML:

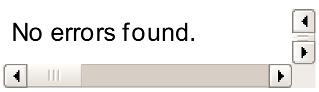
- XML documents always start with the code **<?xml version="1.0"?>**. There cannot be any white space before **<?xml version="1.0"?>**
- The **XMLSchema-instance** and **Mondrian.xsd** lines specify the structure of the XML file. In XML terms, a schema is a set of rules that define the structure and content of an XML file. In Mondrian terms, an schema is a set of rules that connect the relational world to the OLAP world. You will see more on this soon.
- XML documents are composed of *elements*: words surrounded by less-than < and greater-than > symbols. In our code, **Schema** is the opening element; the matching closing element, **</Schema>**, is typed later.
- Elements can have *attributes*. Our **Schema** element has an attribute called **name**, with the value of **My First Schema**. Attributes must be set within quotation marks.
- A XML document has a single "root" element. For our schema, the **<Schema>** element is the root element.

- Even thought hey look similar, XML is not HTML.
- XML is case sensitive. <Schema> and <SCHEMA> are NOT the same thing!
- Elements in XML must be closed. This means our opening element **<Schema>** must have a matching closing element **</Schema>**; **<Schema/>** is an example of an single tag that is also closed.

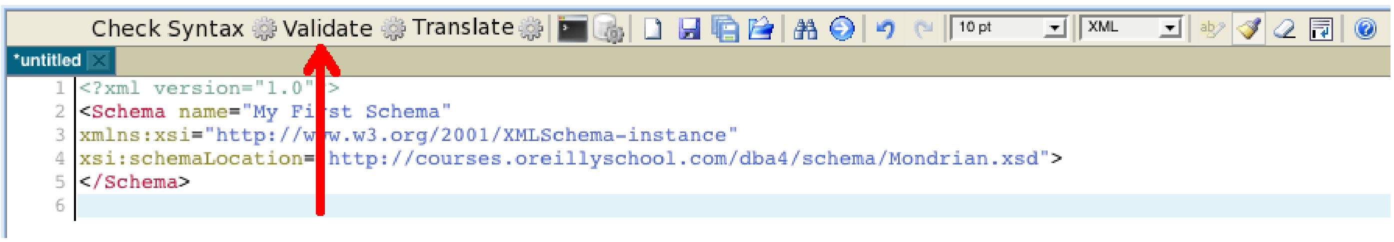
In CodeRunner you can click on the **Check Syntax** button to make sure your XML file follows all of XML's rules. **Check Syntax** won't tell you whether Mondrian can read your schema properly, but it's a good place to start:



Click on **Check Syntax**. If you typed everything correctly, you'll see this:



To check your XML file against the rules defined for the file, click the **Validate** button.



Click on **Validate**. Even if you typed everything correctly, you'll see an error message because our schema is not complete.

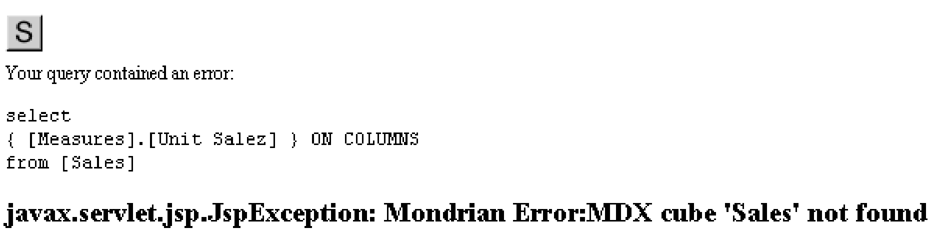
Checking syntax and validating your document can help you track down problems with your schema, however it is still possible to have a syntactically correct and valid schema that gives an error when used with Mondrian, as you will see.

Create a new document, set it to MDX mode and type this short query:

CODE TO TYPE:

```
select
{ [Measures].[Unit Sales] } ON COLUMNS
from [Sales]
```

When you're done, click on **Run Query**. Your XML Schema and your MDX query are sent to the Mondrian server, which responds with this:



This error is the appropriate result, because we haven't defined any cubes in our schema. (We will shortly though.) You can click on the S button to see the documentation of your schema. Since we don't have anything defined right now, when we check our schema documentation, not much is returned:

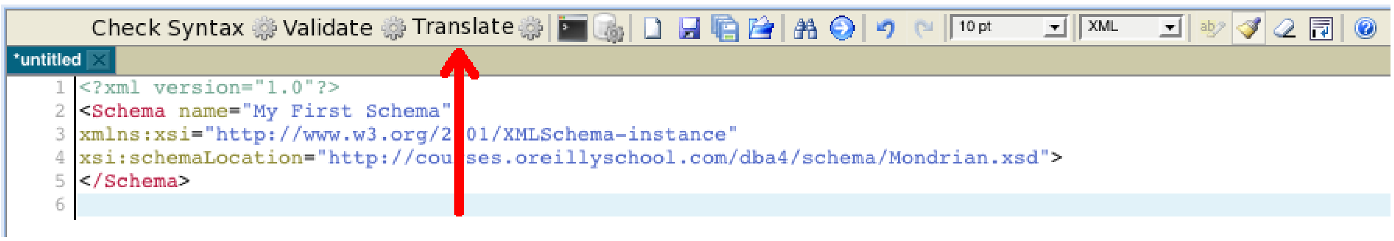
There is another way we can preview our schema. Since the schema is an XML document, we can use XSL to convert our XML document to an HTML document. Don't worry about the specifics of this transformation.

Switch back to your XML file. Update it so it contains the following line:

CODE TO TYPE:

```
<?xml version="1.0"?>
<?xml-stylesheet href="http://courses.oreillyschool.com/dba4/schema/MetaData.xsl" type="text/xs
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
</Schema>
```

Next, click the **Translate** button:



Your schema is reformatted and displayed to you, just like before:

For more information on XML and XSL modes, check out the XML course.

Shared Dimensions

Our first task when authoring a schema is to define the *shared dimensions*. Let's get started!

The first shared dimension that we're going to create is the **Time** dimension. The underlying table for **Time** is called `time_by_day`. Back in lesson 2, we learned that all dimensions in MDX are hierarchical. For example, the time dimension has a hierarchy of **Year -> Quarter -> Month -> Day**.

Let's log into the database to examine its structure.

Note In this course we will tell you which tables to use for facts and dimensions. If you were working on your own data warehousing project, you would consult the documentation for your data warehouse.

Switch to Unix mode, then run this command to connect to the `foodmart` database:

CODE TO TYPE:

```
cold:~$ mysql -h sql -u anonymous foodmart
```

You don't need to enter a password to connect to this database. If you typed everything correctly, you'll see this:

OBSERVE:

```
cold:~$ mysql -h sql -u anonymous foodmart
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 442243
Server version: 5.0.62-log Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Now we can use the *describe* command to see what `time_by_day` looks like. At the MySQL prompt, type this:

CODE TO TYPE:

```
mysql> describe time_by_day;
```

This table isn't exceedingly large:

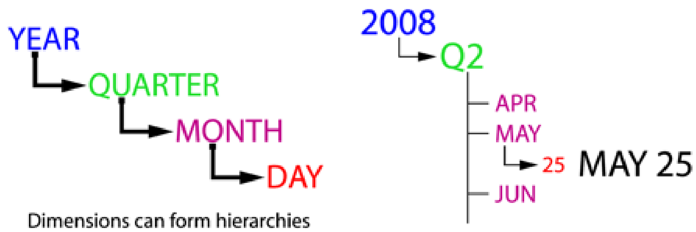
OBSERVE:

```
mysql> describe time_by_day;
+---------------+------------+------+-----+---------+-------+
| Field         | Type       | Null | Key | Default | Extra |
```

```
+---------------+-------------+------+-----+---------+-------+
| time_id       | int(11)     | NO   | PRI | NULL    |       |
| the_date      | datetime    | YES  | UNI | NULL    |       |
| the_day       | varchar(30) | YES  |     | NULL    |       |
| the_month     | varchar(30) | YES  |     | NULL    |       |
| the_year      | smallint(6) | YES  | MUL | NULL    |       |
| day_of_month  | smallint(6) | YES  |     | NULL    |       |
| week_of_year  | int(11)     | YES  |     | NULL    |       |
| month_of_year | smallint(6) | YES  | MUL | NULL    |       |
| quarter       | varchar(30) | YES  | MUL | NULL    |       |
| fiscal_period | varchar(30) | YES  |     | NULL    |       |
+---------------+-------------+------+-----+---------+-------+
10 rows in set (0.00 sec)

mysql>
```

The table's primary key is `time_id` (as shown by the `PRI` under the `Key` column). This data naturally forms a hierarchy of `the_year -> quarter -> the_month -> the_day`.



Dimensions can form hierarchies

This table also contains weekly data, but that data forms a different hierarchy of `the_year -> week_of_year -> the_day`. This hierarchy has some potential problems - weeks can span years, so data may not roll up properly from days to weeks to years. You'll need to ask your business users to help you decide how to document a solution to that problem.

Let's add this dimension to our schema. Switch back to the editor, and to XML mode. In XML mode, add this line to your schema:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
 <Dimension name="Time" type="TimeDimension">

 </Dimension>
</Schema>
```

Here we defined our first shared dimension, called **Time**. Remember there are special MDX functions to deal with the date dimension. We use an attribute called `type` to tell Mondrian that our shared dimension is special - in this case, it's the **TimeDimension**.

Next, we can declare our first **hierarchy** for the time dimension. This will be the *default* hierarchy, with an **All** member. The **All** member is special as well - it tells Mondrian to add a special member to dimension levels automatically. With this special member, Mondrian will let you run queries like this:

OBSERVE:

```
select {[Time].[All Times]} ON COLUMNS
{[Measures].[Unit Sales]} ON ROWS
from [Sales]
```

Here in the query, **All Times** translates to "show me the unit sales for all times." This is handy because it allows us to write queries that compare sales for one particular time period to the *entire* time period, without difficulty.

Let's add this **hierarchy**. We'll need to specify the primary key of `time_id`. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
 <Dimension name="Time" type="TimeDimension">
  <Hierarchy hasAll="true" primaryKey="time_id">

  </Hierarchy>
 </Dimension>
</Schema>
```

Next, we need to define the table that's the source of this hierarchy - the `time_by_day` table. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
 <Dimension name="Time" type="TimeDimension">
  <Hierarchy hasAll="true" primaryKey="time_id">
   <Table name="time_by_day"/>
  </Hierarchy>
 </Dimension>
</Schema>
```

Now we can define the levels within our hierarchy. Let's start with the *Year* level. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
 <Dimension name="Time" type="TimeDimension">
  <Hierarchy hasAll="true" primaryKey="time_id">
   <Table name="time_by_day"/>
   <Level name="Year" column="the_year" type="Numeric"
     uniqueMembers="true" levelType="TimeYears"/>
  </Hierarchy>
 </Dimension>
</Schema>
```

Now we have our new level, called **Year**. The source is the **the_year** column, and we're defining the type as **numeric**. This level has **unique members**, which means, for example, there is only one year called 1997. Finally, we specify that this level is the **year** level in the time dimension.

At this point, it's probably a good idea to hit the **Translate** or Check Syntax button to make sure your schema is well formed. If you hit **Translate**, you'll see this:

Schema: My First Schema

Shared Dimensions

Dimension: Time

Hierarchy: *default* Has All Member: true All Member Name: [All]

Level	Level In Hierarchy	Unique Members	Properties
Year	[Time].[Year]	true	

Now we're getting somewhere! Let's add the remaining levels for Quarter, Month, and Day. Add the following levels. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
 <Dimension name="Time" type="TimeDimension">
  <Hierarchy hasAll="true" primaryKey="time_id">
   <Table name="time_by_day"/>
   <Level name="Year" column="the_year" type="Numeric"
     uniqueMembers="true" levelType="TimeYears"/>
   <Level name="Quarter" column="quarter" uniqueMembers="false" levelType="TimeQuarters"/>
   <Level name="Month" column="month_of_year" uniqueMembers="false" type="Numeric" levelType="T
   <Level name="Day" column="day_of_month" uniqueMembers="false" type="Numeric" levelType="Time
  </Hierarchy>
 </Dimension>
</Schema>
```

Quarter, Month, and Day are not unique. 1997 has a first quarter, as does 1998, and 1999. Every year also has a January, February, and so on.

There we have it, our first shared dimension! Our next shared dimension will be for **Product**, which is stored in the tables called `product` and `product_class`.

First, let's take a look at `product`. At the MySQL prompt, type this code:

CODE TO TYPE:

```
mysql> describe product;
```

This table has lots of columns defined on it:

OBSERVE:

```
mysql> explain product;
+--------------------+---------------+------+-----+---------+-------+
| Field              | Type          | Null | Key | Default | Extra |
+--------------------+---------------+------+-----+---------+-------+
| product_class_id   | int(11)       | NO   | MUL | NULL    |       |
| product_id         | int(11)       | NO   | PRI | NULL    |       |
| brand_name         | varchar(60)   | YES  | MUL | NULL    |       |
| product_name       | varchar(60)   | NO   | MUL | NULL    |       |
| SKU                | bigint(20)    | NO   | MUL | NULL    |       |
| SRP                | decimal(10,4) | YES  |     | NULL    |       |
| gross_weight       | double        | YES  |     | NULL    |       |
| net_weight         | double        | YES  |     | NULL    |       |
| recyclable_package | tinyint(1)    | YES  |     | NULL    |       |
| low_fat            | tinyint(1)    | YES  |     | NULL    |       |
| units_per_case     | smallint(6)   | YES  |     | NULL    |       |
| cases_per_pallet   | smallint(6)   | YES  |     | NULL    |       |
| shelf_width        | double        | YES  |     | NULL    |       |
| shelf_height       | double        | YES  |     | NULL    |       |
| shelf_depth        | double        | YES  |     | NULL    |       |
+--------------------+---------------+------+-----+---------+-------+
15 rows in set (0.00 sec)

mysql>
```

Next, take a look at `product_class`. At the MySQL prompt, type this:

CODE TO TYPE:

```
mysql> describe product;
```

This is where the product categories and sub-categories are defined:

OBSERVE:

```
mysql> explain product_class;
+---------------------+-------------+------+-----+---------+-------+
| Field               | Type        | Null | Key | Default | Extra |
+---------------------+-------------+------+-----+---------+-------+
| product_class_id    | int(11)     | NO   |     | NULL    |       |
| product_subcategory | varchar(30) | YES  |     | NULL    |       |
| product_category    | varchar(30) | YES  |     | NULL    |       |
| product_department  | varchar(30) | YES  |     | NULL    |       |
| product_family      | varchar(30) | YES  |     | NULL    |       |
+---------------------+-------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

mysql>
```

These two tables are joined by the `product_class_id` column. Fortunately, as we'll soon see, Mondrian's schema allows us to specify how these tables are joined together.

To get started, let's add a new shared dimension to our existing schema. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
 <Dimension name="Time" type="TimeDimension">
  <Hierarchy hasAll="true" primaryKey="time_id">
   <Table name="time_by_day"/>
   <Level name="Year" column="the_year" type="Numeric"
     uniqueMembers="true" levelType="TimeYears"/>
   <Level name="Quarter" column="quarter" uniqueMembers="false" levelType="TimeQuarters"/>
   <Level name="Month" column="month_of_year" uniqueMembers="false" type="Numeric" levelType="T
   <Level name="Day" column="day_of_month" uniqueMembers="false" type="Numeric" levelType="Time
  </Hierarchy>
 </Dimension>
 <Dimension name="Product">

 </Dimension>
</Schema>
```

Note From now on we'll omit parts of our schema that we aren't focusing on now, to keep our code listing smaller.

Next, we'll define our default hierarchy. Because we need to join two tables together, we'll have to use an extra attribute. In XML

Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
... lines omitted ...
 <Dimension name="Product">
  <Hierarchy hasAll="true" primaryKey="product_id" primaryKeyTable="product">

  </Hierarchy>
 </Dimension>
</Schema>
```

Here we define our **hierarchy** just like we did before, but now we specify that the **primary key table** is the **product** table. Next, we'll add our join to `product_class`. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
... lines omitted ...
 <Dimension name="Product">
  <Hierarchy hasAll="true" primaryKey="product_id" primaryKeyTable="product">
   <Join leftKey="product_class_id" rightKey="product_class_id">
    <Table name="product"/>
    <Table name="product_class"/>
   </Join>
  </Hierarchy>
 </Dimension>
</Schema>
```

The **join** code is a long-winded way of specifying the SQL join between **product** and **product_class**, using the column `product_class_id` from both tables.

Next, we define the levels for our hierarchy. The product dimension has the following hierarchy and levels defined:

OBSERVE:

```
Product Family
 Product Department
  Product Category
   Product Subcategory
    Brand Name
     Product Name
```

We can define the levels for this dimension just like we did for the last dimension. The only difference is that we need to add an attribute for **table**. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
... lines omitted ...
 <Dimension name="Product">
  <Hierarchy hasAll="true" primaryKey="product_id" primaryKeyTable="product">
   <Join leftKey="product_class_id" rightKey="product_class_id">
    <Table name="product"/>
    <Table name="product_class"/>
   </Join>
   <Level name="Product Family" table="product_class" column="product_family" uniqueMembers="tr
   <Level name="Product Department" table="product_class" column="product_department" uniqueMem
   <Level name="Product Category" table="product_class" column="product_category" uniqueMembers
   <Level name="Product Subcategory" table="product_class" column="product_subcategory" uniqueM
   <Level name="Brand Name" table="product" column="brand_name" uniqueMembers="false"/>
   <Level name="Product Name" table="product" column="product_name" uniqueMembers="true">
   </Level>
  </Hierarchy>
 </Dimension>
</Schema>
```

Suppose we want to include the product's **SKU** as a property on the 'Product Name' level. We do this by adding a **Property** to the level. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
... lines omitted ...
 <Dimension name="Product">
  <Hierarchy hasAll="true" primaryKey="product_id" primaryKeyTable="product">
   <Join leftKey="product_class_id" rightKey="product_class_id">
    <Table name="product"/>
    <Table name="product_class"/>
   </Join>
   <Level name="Product Family" table="product_class" column="product_family" uniqueMembers="tr
   <Level name="Product Department" table="product_class" column="product_department" uniqueMem
   <Level name="Product Category" table="product_class" column="product_category" uniqueMembers
   <Level name="Product Subcategory" table="product_class" column="product_subcategory" uniqueM
   <Level name="Brand Name" table="product" column="brand_name" uniqueMembers="false"/>
   <Level name="Product Name" table="product" column="product_name" uniqueMembers="true">
   <Property name="SKU" column="SKU"/>
   </Level>
  </Hierarchy>
 </Dimension>
</Schema>
```

Now, as it often will be, is a good time to check your work. Click on 'Translate' - you'll see everything we have defined so far. Scroll down and you'll see the Product dimension:

Dimension: Product

Hierarchy: *default* Has All Member: true All Member Name: [All]

Level	Level In Hierarchy	Unique Members	Properties
Product Family	[Product].[Product Family]	true	
Product Department	[Product].[Product Family].[Product Department]	false	
Product Category	[Product].[Product Family].[Product Department].[Product Category]	false	
Product Subcategory	[Product].[Product Family].[Product Department].[Product Category].[Product Subcategory]	false	
Brand Name	[Product].[Product Family].[Product Department].[Product Category].[Product Subcategory].[Brand Name]	false	
Product Name	[Product].[Product Family].[Product Department].[Product Category].[Product Subcategory].[Brand Name].[Product Name]	true	SKU

This all looks great!

Cube

Now that we have our shared dimensions out of the way we can create our first cube. We'll base the cube on the sales facts, which are based in the table called `sales_fact`.

First, we should get familiar with `sales_fact`. Switch to Unix mode, and make sure you're still connected to the `foodmart` database. In MySQL Mode, type this code:

CODE TO TYPE:

```
mysql> describe sales_fact;
```

You will see the following results:

OBSERVE:

```
mysql> explain sales_fact;
+---------------+---------------+------+-----+---------+-------+
| Field         | Type          | Null | Key | Default | Extra |
+---------------+---------------+------+-----+---------+-------+
| product_id    | int(11)       | NO   | MUL | NULL    |       |
| time_id       | int(11)       | NO   | MUL | NULL    |       |
| customer_id   | int(11)       | NO   | MUL | NULL    |       |
| promotion_id  | int(11)       | NO   | MUL | NULL    |       |
| store_id      | int(11)       | NO   | MUL | NULL    |       |
| store_sales   | decimal(10,4) | NO   |     | NULL    |       |
| store_cost    | decimal(10,4) | NO   |     | NULL    |       |
| unit_sales    | decimal(10,4) | NO   |     | NULL    |       |
+---------------+---------------+------+-----+---------+-------+
8 rows in set (0.01 sec)

mysql>
```

This fact table has five keys to dimensions (as noted by the **MUL** under the 'Key' column), and three measures: `store_sales`, `store_cost` and `unit_sales`.

Recall from earlier in the lesson that cube definitions have this structure:

OBSERVE:

```
Cube
 Dimension Usage
 Dimension
 Measure
 Calculated Measures
```

Dimension Usage elements define how shared dimensions are used in the cube. **Dimension** elements allow you to define cube-specific dimensions, using exactly the same syntax as shared dimensions.

We'll use **Measure** elements to define the measures (facts) used in the cube. We'll go over **Calculated Measures** in the the next lesson; they're similar to calculated measures which we've used before in MDX queries.

Let's define our sales cube. Make sure you're using XML mode, then scroll to the very bottom. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
... lines omitted ...
 <Dimension name="Product">
  <Hierarchy hasAll="true" primaryKey="product_id" primaryKeyTable="product">
   <Join leftKey="product_class_id" rightKey="product_class_id">
    <Table name="product"/>
    <Table name="product_class"/>
   </Join>
   <Level name="Product Family" table="product_class" column="product_family" uniqueMembers="tr
   <Level name="Product Department" table="product_class" column="product_department" uniqueMem
   <Level name="Product Category" table="product_class" column="product_category" uniqueMembers
   <Level name="Product Subcategory" table="product_class" column="product_subcategory" uniqueM
   <Level name="Brand Name" table="product" column="brand_name" uniqueMembers="false"/>
   <Level name="Product Name" table="product" column="product_name" uniqueMembers="true">
   <Property name="SKU" column="SKU"/>
   </Level>
  </Hierarchy>
 </Dimension>
 <Cube name="Sales">

 </Cube>
</Schema>
```

Here we've added our **Cube** element for the **Sales** cube. The next step is to define the source table for fact data. Recall the source table is **sales_fact**. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
... lines omitted ...
 <Cube name="Sales">
  <Table name="sales_fact"/>
 </Cube>
</Schema>
```

Great! Now let's define our **shared dimensions**. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
... lines omitted ...
 <Cube name="Sales">
  <Table name="sales_fact"/>
  <DimensionUsage name="Time" source="Time" foreignKey="time_id"/>
  <DimensionUsage name="Product" source="Product" foreignKey="product_id"/>
 </Cube>
</Schema>
```

Here we can rename the shared dimensions if we feel like it, but the **source** attribute must point back to our shared dimension element: the **foreign_key** we found earlier when we ran `explain` on `sales_fact`.

The last step for us to take is to add a measure. We'll begin by adding **Unit Sales**, which we know is in the `unit_sales` column. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<Schema name="My First Schema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://courses.oreillyschool.com/dba4/schema/Mondrian.xsd">
... lines omitted ...
 <Cube name="Sales">
  <Table name="sales_fact"/>
  <DimensionUsage name="Time" source="Time" foreignKey="time_id"/>
  <DimensionUsage name="Product" source="Product" foreignKey="product_id"/>
  <Measure name="Unit Sales" column="unit_sales" aggregator="sum" formatString="Standard"/>
 </Cube>
</Schema>
```

The **Measure** element requires us to specify which **aggregator** we want to use for the measure. In this case we'll use *sum* - but we could have used any aggregate such as count, max, min, or avg. We'll also use the standard format string, but we could have used any other format string.

Our schema is complete. Check the syntax to make sure you haven't made any errors, then save your work. Switch back to MDX mode. It's time to test! In MDX Mode, type this code:

CODE TO TYPE:

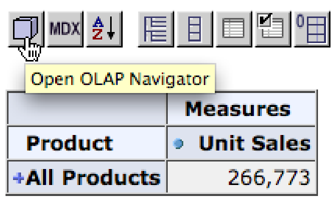
```
select
  { [Measures].[Unit Sales] } on columns,
  { [Product].[All Products] } on rows
from [Sales]
where [Time].[1997]
```

Run the query - you'll see these results:

	Measures
Product	Unit Sales
All Products	266,773

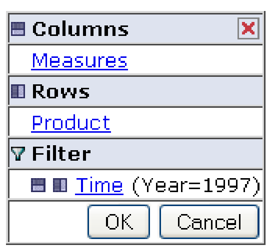
Slicer: [Year=1997]

How can we be sure *our* schema was used, instead of the default schema? For starters, you can open the OLAP Navigator to see what the cube looks like:



Slicer: [Year=1997]

You'll see a very small cube defined:



You can also click on the S icon to see the schema definition. In this case, you'll see that the query used the schema called **My First Schema**.

Your work is looking really good! You've created a basic schema to bridge the relational data warehouse and multi-dimensional data warehouse worlds. In the next lesson we'll expand this schema by adding calculated members and virtual cubes. See you there!

Advanced Schemas
DBA 4: Analyzing Data Lesson 11

Good to have you back! In the last lesson we learned how to create a simple schema for Mondrian. In this lesson we'll expand that schema, by adding calculated members and virtual cubes.

Calculated Members

Back in lesson 4, we went over calculated members. We declared a calculated member called `[Measures].[My Profit]` as `[Measures].[Store Sales] - [Measures].[Store Cost]`. Then we used that calculation in an MDX query.

Now let's use that calculated member to help us figure out profit for our business users. Determining how much profit is being earned is important to business users, of course. And once we've written the code that does this for us, it can be used in various MDX queries. Copying and pasting code is *always* a bad idea; MDX code is no exception.

Instead of giving in to the temptation to copy and paste our query, we can add our calculated member to the cube definition in the schema. This makes the member available to anyone who uses MDX. Nobody even has to know it is a calculated member!

Let's try this stuff out. To get started, open the schema that we worked on in the last lesson (including the projects).

Before we can calculate profit, we need to add two new measures to our Sales cube: **store sales** and **store cost**. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE Schema SYSTEM "http://courses.oreillyschool.com/dba4/mondrian.dtd">
<Schema name="My First Schema">
... lines omitted ...
<Cube name="Sales">
  <Table name="sales_fact"/>
  <DimensionUsage name="Time" source="Time" foreignKey="time_id"/>
  <DimensionUsage name="Product" source="Product" foreignKey="product_id"/>
  <Measure name="Unit Sales" column="unit_sales" aggregator="sum" formatString="Standard"/>
  <Measure name="Store Sales" column="store_sales" aggregator="sum" formatString="Standard"/>
  <Measure name="Store Cost" column="store_cost" aggregator="sum" formatString="Standard"/>
 </Cube>
... lines omitted ...
</Schema>
```

With those in place, let's work on the measure for profit. Switch to MDX mode, and type in this:

CODE TO TYPE:

```
WITH member [Measures].[Profit] as
[Measures].[Store Sales] - [Measures].[Store Cost]
select
  { [Measures].[Store Sales], [Measures].[Store Cost], [Measures].[Profit] } on columns,
  { [Product].[All Products] } on rows
from [Sales]
where [Time].[1997]
```

If you typed everything correctly, you'll see these results:

	Measures		
Product	Store Sales	Store Cost	Profit
+All Products	565,238	225,627	339,611

Slicer: [Year=1997]

Our calculation for profit looks correct:

OBSERVE:

```
[Measures].[Store Sales] - [Measures].[Store Cost]
```

Now we're ready to add it to the cube definition. Switch back to XML mode, and type in this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE Schema SYSTEM "http://courses.oreillyschool.com/dba4/mondrian.dtd">
<Schema name="My First Schema">
... lines omitted ...
<Cube name="Sales">
  <Table name="sales_fact"/>
  <DimensionUsage name="Time" source="Time" foreignKey="time_id"/>
  <DimensionUsage name="Product" source="Product" foreignKey="product_id"/>
  <Measure name="Unit Sales" column="unit_sales" aggregator="sum" formatString="Standard"/>
  <Measure name="Store Sales" column="store_sales" aggregator="sum" formatString="Standard"/>
  <Measure name="Store Cost" column="store_cost" aggregator="sum" formatString="Standard"/>
  <CalculatedMember name="Profit" dimension="Measures">
   <Formula>[Measures].[Store Sales] - [Measures].[Store Cost]</Formula>
  </CalculatedMember>
 </Cube>
... lines omitted ...
</Schema>
```

Now that we've created our calculated measure, let's try to use it in a query. Switch back to MDX mode and type this code:

CODE TO TYPE:

```
select
  { [Measures].[Store Sales], [Measures].[Store Cost], [Measures].[Profit] } on columns,
  { [Product].[All Products] } on rows
from [Sales]
where [Time].[1997]
```

As long as you typed everything correctly, you will see the same results as before:

	Measures		
Product	Store Sales	Store Cost	Profit
+All Products	565,238	225,627	339,611

Slicer: [Year=1997]

To double check our measure, click on the S button. Scroll down to the bottom, you'll see the new calculated member:

Cube: Sales

Shared Dimensions

[Time]
[Product]

Cube-Specific Dimensions

none

Measures

Name	Aggregator	Format String
[Measures].[Unit Sales]	sum	Standard
[Measures].[Store Sales]	sum	Standard
[Measures].[Store Cost]	sum	Standard

Calculated Members

Name	Formula	Properties
[Measures].[Profit]	[Measures].[Store Sales] - [Measures].[Store Cost]	

Since we can specify format strings for standard measures, you might expect Mondrian to accept format strings for calculated measures. It does, but in a slightly different way. Instead of using a `FormatString` attribute, we need to use an element called `CalculatedMemberProperty`. Switch back to XML mode and type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE Schema SYSTEM "http://courses.oreillyschool.com/dba4/mondrian.dtd">
<Schema name="My First Schema">
... lines omitted ...
<Cube name="Sales">
  <Table name="sales_fact"/>
  <DimensionUsage name="Time" source="Time" foreignKey="time_id"/>
  <DimensionUsage name="Product" source="Product" foreignKey="product_id"/>
  <Measure name="Unit Sales" column="unit_sales" aggregator="sum" formatString="Standard"/>
  <Measure name="Store Sales" column="store_sales" aggregator="sum" formatString="Standard"/>
  <Measure name="Store Cost" column="store_cost" aggregator="sum" formatString="Standard"/>
  <CalculatedMember name="Profit" dimension="Measures">
   <Formula>[Measures].[Store Sales] - [Measures].[Store Cost]</Formula>
   <CalculatedMemberProperty name="FORMAT_STRING" value="$#,##0.00"/>
  </CalculatedMember>
 </Cube>
... lines omitted ...
</Schema>
```

Switch back to MDX mode and rerun your query. You will see these results:

	Measures		
Product	Store Sales	Store Cost	Profit
+All Products	565,238	225,627	$339,610.90

Slicer: [Year=1997]

Great!

Before we move on, let's add one more cube-specific dimension to our Sales cube -- **Customers**. In XML Mode, type this code:

CODE TO TYPE:

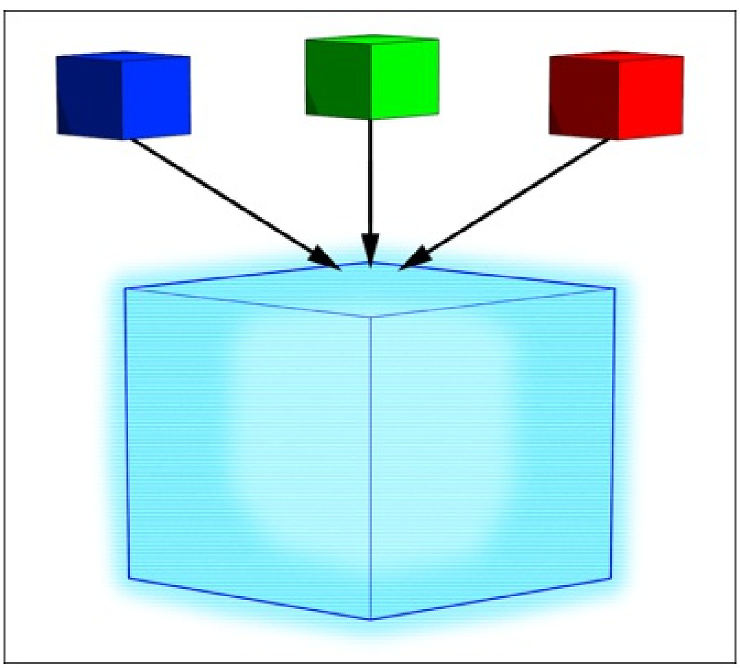
```
<?xml version="1.0"?>
<!DOCTYPE Schema SYSTEM "http://courses.oreillyschool.com/dba4/mondrian.dtd">
<Schema name="My First Schema">
... lines omitted ...
<Cube name="Sales">
  <Table name="sales_fact"/>
  <DimensionUsage name="Time" source="Time" foreignKey="time_id"/>
  <DimensionUsage name="Product" source="Product" foreignKey="product_id"/>
  <Dimension name="Customers" foreignKey="customer_id">
      <Hierarchy hasAll="true" allMemberName="All Customers" primaryKey="customer_id">
        <Table name="customer"/>
        <Level name="Country" column="country" uniqueMembers="true"/>
        <Level name="State Province" column="state_province" uniqueMembers="true"/>
        <Level name="City" column="city" uniqueMembers="false"/>
        <Level name="Name" column="customer_id" nameColumn="fullname" type="Numeric" uniqueMembers="tr
          <Property name="Gender" column="gender"/>
          <Property name="Marital Status" column="marital_status"/>
          <Property name="Education" column="education"/>
          <Property name="Yearly Income" column="yearly_income"/>
        </Level>
      </Hierarchy>
  </Dimension>
  <Measure name="Unit Sales" column="unit_sales" aggregator="sum" formatString="Standard"/>
... lines omitted ...
</Schema>
```

Virtual Cubes

Cubes in Mondrian typically have only one source table for fact data. In our Sales cube the source table was `sales_fact`. In the project for the last lesson you created a new cube called Warehouse, with a source table called `inventory_fact`.

Now let's suppose you wanted to compare sales and warehouse data, side-by-side. One option for doing that would be to combine the `sales_fact` and `inventory_fact` data into a single table, possibly using a view, then building a cube off of that structure. This may require a significant amount of work to implement - you might have to refactor many ETL processes to make it happen. Fortunately, you can avoid all of this trouble by using a *Virtual Cube*.

A *Virtual Cube* in Mondrian is made up of one or more "real" cubes, combined into a "virtual structure." Virtual cubes not only allow you to compare data from two "real" cubes, they also let you simplify existing cubes to provide a simpler interface to your users:



Note In order to work through the next part of this lesson, you need to have completed the projects from the last lesson.

Let's add a virtual cube to our schema. Switch to XML mode, and scroll to the very bottom.

We define a virtual cube using the **VirtualCube** element. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE Schema SYSTEM "http://courses.oreillyschool.com/dba4/mondrian.dtd">
<Schema name="My First Schema">
... lines omitted ...
 <VirtualCube name="Warehouse and Sales">
 </VirtualCube>
</Schema>
```

Now we'll add the *shared* dimensions to the cube, using the **VirtualCubeDimension** element. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE Schema SYSTEM "http://courses.oreillyschool.com/dba4/mondrian.dtd">
<Schema name="My First Schema">
... lines omitted ...
 <VirtualCube name="Warehouse and Sales">
  <VirtualCubeDimension name="Product"/>
  <VirtualCubeDimension name="Time"/>
 </VirtualCube>
</Schema>
```

We can also create cube-specific dimensions using the **VirtualCubeDimension** element, and an added **cubeName** attribute. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE Schema SYSTEM "http://courses.oreillyschool.com/dba4/mondrian.dtd">
<Schema name="My First Schema">
... lines omitted ...
 <VirtualCube name="Warehouse and Sales">
  <VirtualCubeDimension name="Product"/>
  <VirtualCubeDimension name="Time"/>
  <VirtualCubeDimension cubeName="Sales" name="Customers"/>
  <VirtualCubeDimension cubeName="Warehouse" name="Warehouse"/>
 </VirtualCube>
</Schema>
```

Finally, we define the measures we want within our virtual cube, using the **VirtualCubeMeasure** element. In XML Mode, type this code:

CODE TO TYPE:

```
<?xml version="1.0"?>
<!DOCTYPE Schema SYSTEM "http://courses.oreillyschool.com/dba4/mondrian.dtd">
<Schema name="My First Schema">
... lines omitted ...
 <VirtualCube name="Warehouse and Sales">
  <VirtualCubeDimension name="Product"/>
  <VirtualCubeDimension name="Time"/>
  <VirtualCubeDimension cubeName="Sales" name="Customers"/>
  <VirtualCubeDimension cubeName="Warehouse" name="Warehouse"/>
  <VirtualCubeMeasure cubeName="Sales" name="[Measures].[Store Sales]"/>
  <VirtualCubeMeasure cubeName="Warehouse" name="[Measures].[Warehouse Sales]"/>
 </VirtualCube>
</Schema>
```

Now that we have our virtual cube defined, let's try it out. This time we'll query our virtual cube, **Warehouse and Sales**. Switch back to MDX mode and type this code:

CODE TO TYPE:

```
select
  { [Measures].[Store Sales], [Measures].[Warehouse Sales] } on columns,
  { [Product].[All Products] } on rows
from [Warehouse and Sales]
where [Time].[1997]
```

If you typed everything correctly, you'll see this:

	Measures	
Product	Store Sales	Warehouse Sales
All Products	565,238	196,770.888

Slicer: [Year=1997]

To double-check our virtual cube, click on the S button. Scroll down to the bottom, you'll see the new calculated member:

Virtual Cube: Warehouse and Sales

Dimensions

Name	Cube
[Product]	
[Time]	
[Customers]	Sales
[Warehouse]	Warehouse

Measures

Name	Cube
[Measures].[Store Sales]	Sales
[Measures].[Warehouse Sales]	Warehouse

Calculated Members

Name	Formula	Properties

Everything looks good!

There isn't anything listed under the **Calculated Members** for our virtual cube, because we haven't defined any such members. Our members are defined exactly the same way as they are within a "real" cube - using a `<CalculatedMember>` element.

Wow! You covered a lot of material in these lessons! This is essentially the end of the line - the next lesson is a description of your final project.

We appreciate all of your hard work. Good luck!

Final Project
DBA 4: Analyzing Data Lesson 12

Your Final Project

You're almost finished!

Schema

The first step for your final project is to write a new XML schema. This schema will be specifically for the **HR** (human resources) cube in the FoodMart data warehouse.

The fact table for the HR cube is called `salary`.

The required dimensions are:

1. Time - from the `time_by_day` table
2. Employee - from the `employee` table
3. Store - from the `employee` and `store` tables
4. Pay Type - from the `employee` and `position` tables
5. Department - from the `department` table

The required measures are:

1. Salary
2. Employee Count - using the *count* aggregate
3. Average Salary - a *calculated* measure

Save your schema in an XML file.

Queries

Your next task is to write a few queries to demonstrate your ability to use these MDX tools:

1. Tuples and Sets
2. Named Sets
3. Crossjoin
4. Slicing *and* Filtering
5. A function for hierarchies - your choice
6. A date dimension function (like YTD) - your choice
7. Top/Bottom and Generate

Save each query in its own MDX file.

Take your time with this final project - it is intended to be a review of everything you've learned in this course. Remember, you can always consult prior lessons or ask your mentor if you have any questions. When you're finished, hand in all of your files.

It's been a pleasure working through the lessons with you. Good luck!