Chapter 1

Readme.doc – definitions you need to know

Sample data

We used a sample set of data and a sample cube in order to produce the screen shots that appear in this chapter. However, the cube was created just to provide the screen shots and has very little merit as a real cube so we haven't, therefore, included it on the CD-ROM.

Italics

One of the problems inherent in writing a book like this is the need to tread a thin line between defining terms in a readable way and ensuring that we are as precise as possible. Sometimes we've tried to do this by giving a general overview of a term and then giving a more formal definition. At other times, we have felt that even a general description needs to be qualified. In those cases we have often put the further qualifications in italics. Anything that you find in italics can be read as an aside to the main discussion and it should be possible to skip the italics on your first read through and still get the overall picture.

❛ *In fact, this applies as a general rule throughout the book. The comments in italics are asides to the general information.* ❜

Introduction

We are going to use this chapter to define many of the terms that are used in MDX. These will include:

- Dimensions
- Measures
- Members
- Cells
- Hierarchies
- Aggregations
- Levels
- Tuples
- Sets
- Member Properties

We briefly considered creating a glossary and defining each term individually, but it is difficult to maintain a sense of context in a glossary because all of the entries have to stand alone. So instead we will describe the factors that affect how a cube is constructed and use that to introduce the definitions as we go along.

If you have been building OLAP cubes for any length of time, the definitions of the first few terms are likely to be already firmly embedded into your brain. If so, simply fast forward until you hit one that you don't know. If that means you end up jumping to Chapter 2, that's fine.

What happens if you aren't familiar with the terms? Well, the obvious answer is to read through this entire chapter, making sure that you understand all the terms before moving on to the actual coding. That works fine for some people but it can work really badly for others. The 'others' are those who want to get started, now! They want to feel a keyboard working under their finger tips. They want to type code in, try it, see it fail, modify it, try it again, get it working. If they then hit a term that they don't understand, they are happy to divert for a little background reading. We've written this chapter, therefore, assuming that it will be read from beginning to end. But we've also tried to write it in a reasonably modular way so that you can jump straight to Chapter 2 and get started and flip back when you need a definition. The terms appear in the order listed above and there are headings to guide you, so you can scan through the chapter until you see the one you want.

Dimensions, measures, members and cells

OLAP cubes are stores of multi-dimensional data. MDX is all about manipulating OLAP cubes so in order to understand why MDX works in the way it does, it is an excellent idea to get a firm grip on the way in which multi-dimensional data is described, stored and defined.

An OLAP cube is made up of **Dimensions** and **Measures**. In the figure below, there are two dimensions, `Time` and `Product`. The `Product` dimension has four **Members** – `Sardines`, `Anchovies`, `Herrings` and `Pilchards`. The `Time` dimension also happens to have four members (`April` to `July`). There is one measure, `UnitsSold`, which is simply the number of cans of each product sold in each month.

For this two-dimensional 'cube' you can think of the members of the product dimension as being the labels for the columns of a worksheet. The members of the time dimension then form the labels of the rows and the values of the measure appear in the **Cells**.

UnitsSold

	Product			
Time	Sardines	Anchovies	Herrings	Pilchards
April	16	23	12	4
May	14	12	23	6
June	34	19	19	8
July	17	22	14	4

You can, of course, reverse the rows and columns without disrupting the meaning of the data.

UnitsSold

	Time			
Product	April	May	June	July
Sardines	16	14	34	17
Anchovies	23	12	19	22
Herrings	12	23	19	14
Pilchards	4	6	8	4

Clearly, each cell can be described in terms of one member from each dimension – thus we can see that the number of anchovies sold in June was 19. We could say more formally that each value for the measure `UnitsSold` occurs at a unique intersection between two members from the different dimensions.

❛ *Expressing it in this way imparts exactly the same information, you just get more street cred. for knowing the jargon.* ❜

It is easy for humans to visualize a cube like this that consists of two dimensions and a single measure. There is no reason, however, why a cube should be limited to one measure. As well as the unit sales figures shown

above, you might also want to store, say, profit. There are several ways in which we can represent (and you can visualize) this. You might be happy picturing it like this:

UnitsSold/Profit

	Product						
Time	Sardines		Anchovies		Herrings		Pilchards	
April	16	$40.00	23	$78.20	12	$23.88	4	$8.20
May	14	$35.00	12	$40.80	23	$45.77	6	$12.30
June	34	$85.00	19	$64.60	19	$37.81	8	$16.40
July	17	$42.50	22	$74.80	14	$27.86	4	$8.20

with the `UnitsSold` figure occupying the left of the cell and the profit occupying the right.

Alternatively, you could think of each measure represented by a single worksheet in a spreadsheet application, so you end up with a stack of sheets showing the same dimensions but different measures, like this:

UnitsSold

	Product			
Time	Sardines	Anchovies	Herrings	Pilchards
April	16	23	12	4
May	14	12	23	6
June	34	19	19	8
July	17	22	14	4

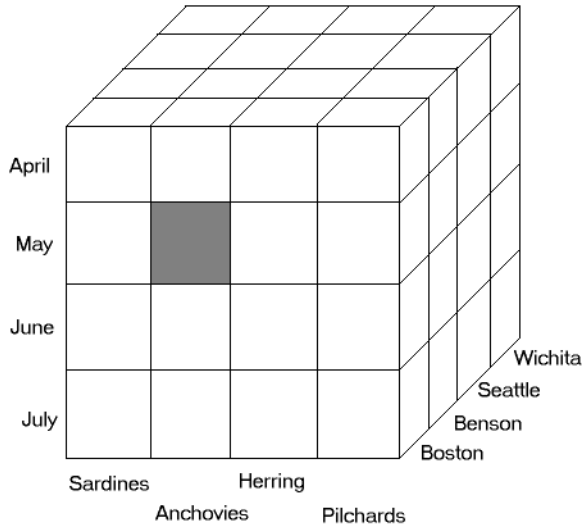
Profit

	Product			
Time	Sardines	Anchovies	Herrings	Pilchards
April	$40.00	$78.20	$23.88	$8.20
May	$35.00	$40.80	$45.77	$12.30
June	$85.00	$64.60	$37.81	$16.40
July	$42.50	$74.80	$27.86	$8.20

The way you choose doesn't matter too much; the important thing is to understand that a cube can have multiple measures – within reason as many as you want.

❝ *OK, so you want exact figures. A cube in Analysis Services can have up to 1,024 measures. For the record it can also have up to 128 dimensions, each with potentially thousands or millions of members.* ❞

As well as multiple measures, cubes can also have more than two dimensions. Suppose that our company has multiple stores and we want to see the `UnitsSold` figures broken down for individual stores. No problem, we just add another dimension, called `Store`, to contain the information about our outlets, looking like this:



(We've cut back to a single measure for this cube, just to keep the diagram relatively simple.)

The cell that's highlighted in the cube above sits at an intersection of the cube's three axes and each axis represents a dimension. The shaded cell lines up with `anchovies` from the `Product` dimension (the x axis), `May` from the `Time` dimension (the y axis) and `Boston` from the `Store` dimension (the z axis). The value that we would find in this cell tells us the number of anchovies sold in May in our Boston store.

Visualizing three dimensions is also relatively easy; we live in a three-dimensional world and so we're quite good at three-dimensional concepts. However, OLAP cubes can have many more dimensions.

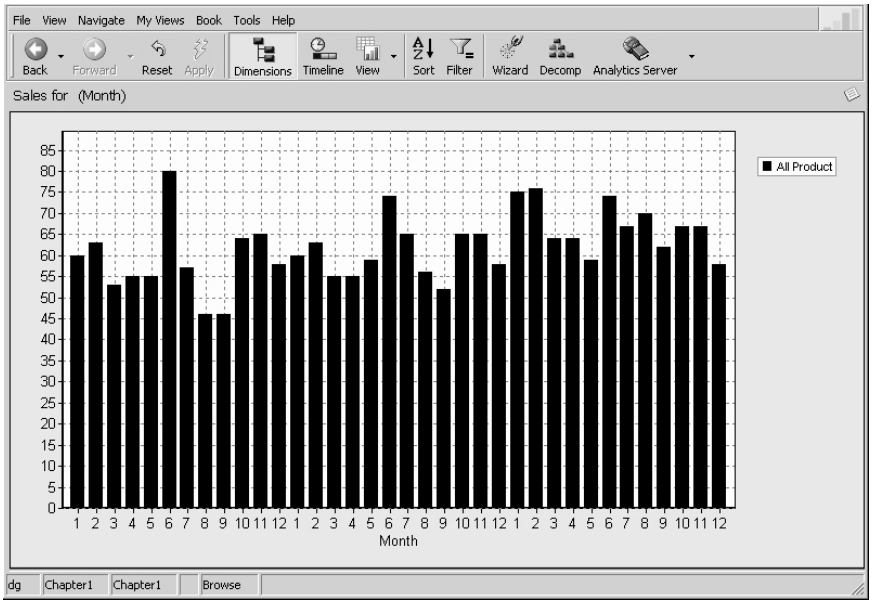
In our example, there could be an `Employee` dimension to tell us which member of staff made the sales, and a `Customer` dimension indicating to whom the sales were made. That's five dimensions already. This is a good time to stop trying the visualizations: mental pictures of two- and three-dimensional data are excellent for building a basic understanding of what OLAP cubes are all about but once we exceed three dimensions, it's much easier to rely on words rather than pictures.

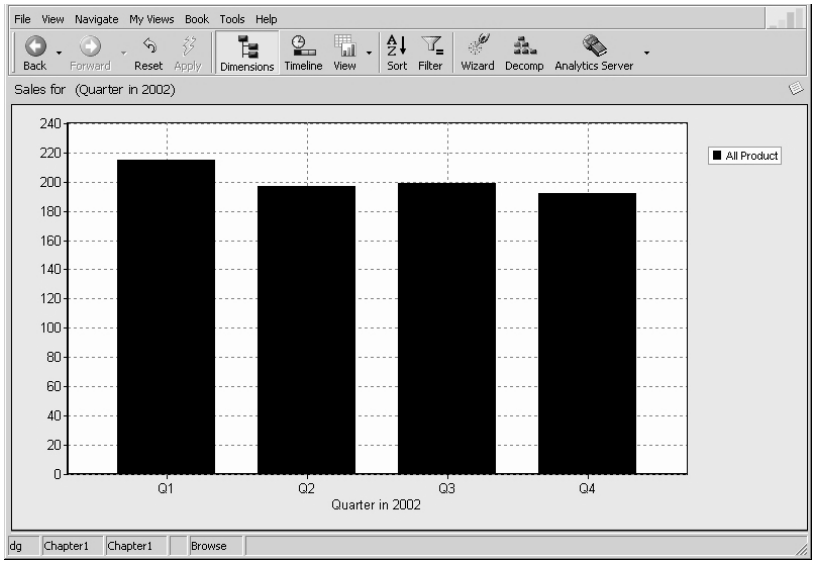
Talking about values in a five-dimensional cube turns out to be perfectly straightforward: in May, our man Steve in the Boston store sold five cans of anchovies to Katie for a profit of $12.50. In that simple sentence we used all five dimensions (`Time`, `Employee`, `Store`, `Product` and `Customer`). We also slipped in not one but two measures: `UnitsSold` (five cans of anchovies) and `Profit` ($12.50).

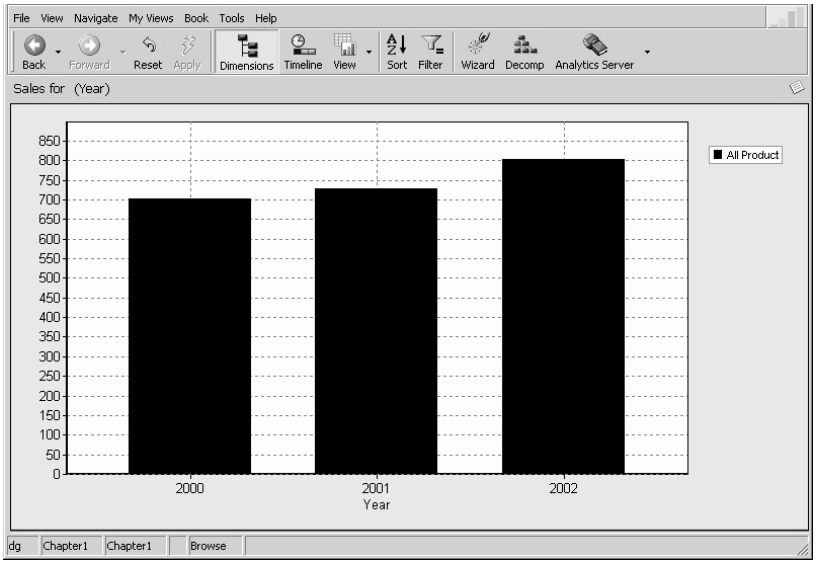
However, as long as we keep the number of dimensions and measures down to reasonable levels for the rest of this chapter, diagrams are still really useful to help explain the terms used in OLAP cubes and hence in MDX.

Cranking up the complexity

So far we have built up a set of words (`Dimensions`, `Measures`, `Members`, `Cells`) and definitions that allow us to describe simple OLAP cubes. Do we have to make it any more complex? Yes, because this is still too simple a model for the sort of analysis that business users actually want to achieve. Analysts and business people typically want to query their data in much more complex ways and so OLAP cubes have to be capable of handling that complexity. For instance, `UnitsSold` totals for each month of trading might be required, or totals for each quarter of the current year, or even totals for each year.







Our current level of complexity can't do this because it only caters for one unit type of member in each dimension – for example, we can only represent months in the time dimension. In order to give users rapid access to data totaled in this way, these quarterly and yearly totals need to be stored in the OLAP cube as well. This in turn means that we need to handle some of the dimensions as what are called **hierarchies**.

Hierarchies and aggregations

While there is no obligation for all dimensions in a cube to be hierarchical, experience suggests that many are in practice. Most cubes have a time dimension, for example, and time is almost always hierarchical.

We can imagine the different totals – those for each year, each quarter and each month – being held in worksheet-like grids. The total item `UnitsSold` for each year for, say, the Boston store might look like this:

UnitsSold

		Product			
Time		Sardines	Anchovies	Herrings	Pilchards
Year	2000	242	199	196	65
	2001	232	201	219	75
	2002	294	214	209	86

and for quarter, like this:

UnitsSold

		Product			
Time		Sardines	Anchovies	Herrings	Pilchards
Quarter	Q1	61	36	58	21
	Q2	64	54	54	18
	Q3	45	59	33	12
	Q4	72	50	51	14

and like this for each month:

UnitsSold

		Product			
Time		Sardines	Anchovies	Herrings	Pilchards
Month	April	16	23	12	4
	May	14	12	23	6
	June	34	19	19	8
	July	17	22	14	4

We could also represent these values in a rather more complex grid like this:

			Sardines	Anchovies
2000	Q3	July	17	22
		Aug	16	18
		Sept	12	19
	Q3 total		45	59
2000	Q4	Oct	27	19
		Nov	24	19
		Dec	21	12
	Q4 total		72	50
2000 total			242	199

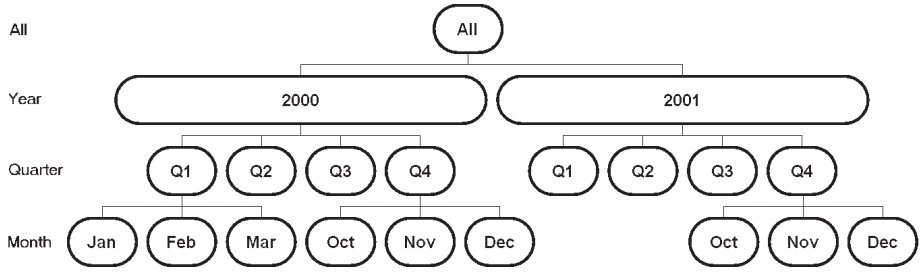
These representations of what is going on inside a cube are, as you can see, storing values derived by adding up, or aggregating, the original data in the cube. In practice, aggregations are not the only values that can be calculated from the original data – for example, a cube can also hold values expressed as percentages. However, no matter how they are calculated, such values are usually referred to as **aggregations**.

❻ *In fact, the term aggregation is very useful when discussing cubes, but it isn't one that is directly used in MDX. Aggregations are just an optimization that the underlying storage engine uses to speed up the response of the cube. The presence or absence of aggregations doesn't change any of the results of MDX expressions or queries.* ❟

In order to help us discuss hierarchies, it is useful to introduce the term **Level**.

Levels

Staying with a Time dimension, its hierarchy might look like this:



(In order to keep the diagram readable, only some of the members of the month level are shown.)

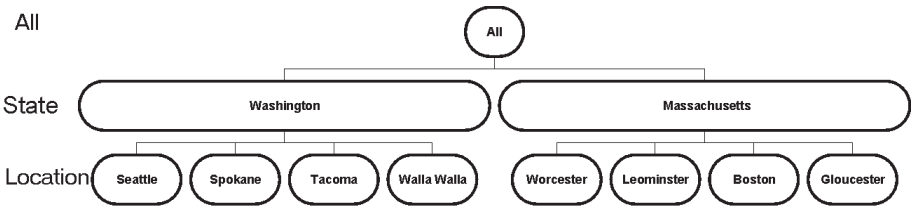
Just as ogres have layers (see Shrek), hierarchies have levels. In the example above, the Time dimension has four levels: All, Year, Quarter and Month. Year, Quarter and Month are just as you'd expect and All is simply a handy way of answering questions like "what is the total number of items sold for the period covered by data in the cube?" Most hierarchical dimensions have an All level at the top.

The top of a hierarchy is always the level that encompasses the greatest amount of information in the smallest number of members. Thus All is at the top of the hierarchy and you read 'down' that hierarchy to Month at the bottom. Or you can start with Month at the bottom (or 'leaf' level) and read 'up' to All at the top.

The leaf level is the level that's at the bottom of a branch of the hierarchy, and the term leaf node is used to mean a member of that leaf level. The leaf analogy comes from the branching, tree-like shape of a hierarchy, albeit a tree that's upside-down.

Levels have members, and a member is a single item in a dimension. It will sit at one of the levels in the dimension's hierarchy. To continue with the Time dimension example, at the Year level, you might have members called 2000, 2001 and 2002. At the Quarter level, there are likely to be members called something like Q1, Q2 and so on.

As we've said, most dimensions are hierarchical – take the `Store` dimension, for example. Stores could be grouped together into states so that analysis can be performed between individual stores and also between different states.



Naming conventions

Now we know that there are levels in a hierarchy and that each level has a descriptive name, like `All`, `State` and `Location` in the diagram above. We also know that each level contains members: `Seattle` and `Leominster` are members of the `Location` level. What good does this information do us? Well, once we start writing MDX code, we'll need a way of identifying precisely the specific members with which we want the code to work.

The most obvious way to identify a member is to start with the name of the dimension and work downwards, specifying the members at each level in the hierarchy until we reach required member. Working with the `Store` dimension shown in the diagram above, we'd indicate the `Leominster` member like this:

```
[Store].[All].[Massachusetts].[Leominster]
```

This is, in fact, the method we will be using almost everywhere in this book. It has the advantage of precision which outweighs its tendency towards the verbose.

There is a temptation, however, to take short cuts with these long path names. It's clear that in our simple example above, we could use just the dimension name and then go straight to the name of the member we want, like this:

```
[Store].[Leominster]
```

to point unequivocally to the `Leominster` member. This works here, but such short cuts will only work with certain data and with certain naming conventions.

If you look back at the diagram that shows the hierarchy of the Time dimension, you'll see that there will be two Octobers, one in 2000 and one in 2001. Here we couldn't take a short cut like:

```
[Time].[October]
```

and be sure we were pointing to exactly the right member.

With some dimensions it is relatively easy to impose a naming convention that uses unique member names and is therefore amenable to the use of short cuts. For example, the Time dimension could be re-structured so that the members at the Quarter level were called 2000-Q1, 2001-Q2 and so on, and the members at the Month level were called October-2000, May-2001 etc. The gain is that you can then reference them simply by dimension name and unique member name:

```
[Time].[June-2001]
```

rather than:

```
[Time].[All].[2001].[Q2].[June].
```

❝ *This works because we can control how Time is named. You can easily be caught out by other data because there are cases where even full path names don't help. For example, suppose that you discover that there are two places called Leominster in the state of Massachusetts. In that case even a full path name:*

```
[Store].[All].[Massachusetts].[Leominster]
```

wouldn't distinguish between them.

This type of duplication is all too frequent in real data. English counties (which are far smaller than US states) are littered with duplicates: there are two places called Ashton in Cornwall, for example.

In order to solve this problem Analysis Services provides a means of identifying members by means of their 'member keys' rather than by their member names – there's more about this in Chapter 14. ❞

It is worth noting that, despite appearances, client tools such as ProClarity, Excel etc. don't themselves ever create the names of the members that they subsequently use in the MDX expressions or queries that they generate. In fact, these tools are explicitly warned not to do so by the OLEDB for OLAP specification. Instead, the server generates the unique names for them, and it has all kinds of rules about how it can do this. Sometimes the server will generate a name of the type that we have already discussed here – dimension.name.name.name. For example:

```
[Store].[All].[Washington].[Tacoma]
```

However, it can also be in the form `dimension.level.name`, for example:

```
[Store].[Location].[Tacoma]
```

or even sometimes something completely different. As far as the tools are concerned, they never try to make sense out of the names. Instead they let the user click the objects they want in the interface and the tool uses the names it has been given for those objects to generate the MDX that is then sent back to the server as a query or an expression. Hand-written MDX, on the other hand, can use whatever the person writing it feels like at the time. As a general rule we recommend using fully qualified names, such as `dimension.name.name.name.name`.

Tuples and sets

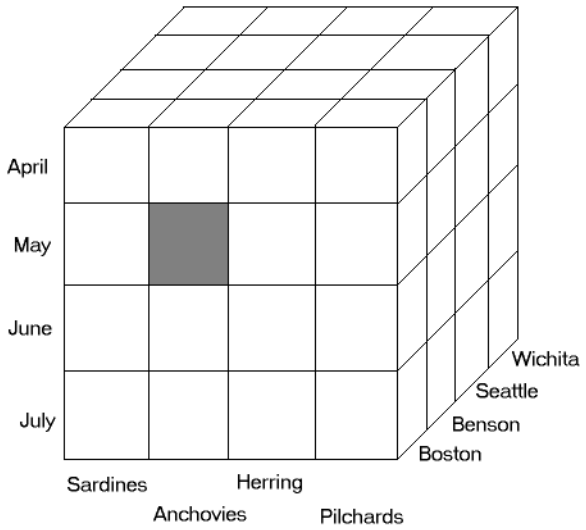
We need to define these terms accurately because they are of fundamental importance to an understanding of MDX. In later chapters you are going to meet expressions and functions that require you to give them, very specifically, a set or a tuple. However, we are quite well aware that defining the terms 'Tuple' and 'Set' has caused problems in the past, at least with reference to MDX.

Part of the problem is that, although these terms can be defined very accurately and succinctly in mathematical terms, defining them in more human terms tends to lead to very impenetrable definitions. This is because human language is so imprecise when compared to mathematics.

What we are going to do is to define them several times. The first definitions won't be totally accurate but will hopefully give a good feel for the important distinctions between tuples and sets. Then we'll add some more information and use examples to fill in some of the finer detail.

Tuples

❝ *We'll go back to the simple three-dimensional model with a single measure that we used earlier.* ❞



The highlighted cell sits at an intersection of the cube's three axes and each axis represents a dimension. The cell lines up with `Anchovies` from the `Product` dimension (the x axis), `May` from the `Time` dimension (the y axis) and `Boston` from the `Store` dimension (the z axis). The value that we would find in this cell tells us the number of anchovies sold in May in our `Boston` store.

We can express this description of the cell more neatly in pseudo-MDX as:

```
([Product].[Anchovies],[Time].[May],[Store].[Boston])
```

Here we are using the names of three members to point to the cell. In fact, the order in which we list the members is immaterial; we could equally well point to the cell like this:

```
([Product].[Anchovies],[Store].[Boston],[Time].[May])
```

Either way, we have a precise and unequivocal description of the location of the cell in the OLAP cube. In essence what we have done here is to identify a cell using its co-ordinates. The co-ordinates are members – one taken from each of the three dimensions. The name for this collection of co-ordinates is a **tuple**.

It is important to distinguish here between the tuple and the cell contents. One way to do so is to try to find an analogy from a more familiar system – a spreadsheet.

	A	B	C
1	12	32	45
2	37	23	12
3	65	45	32
4	78	56	44
5	98	23	34
6	290	179	167

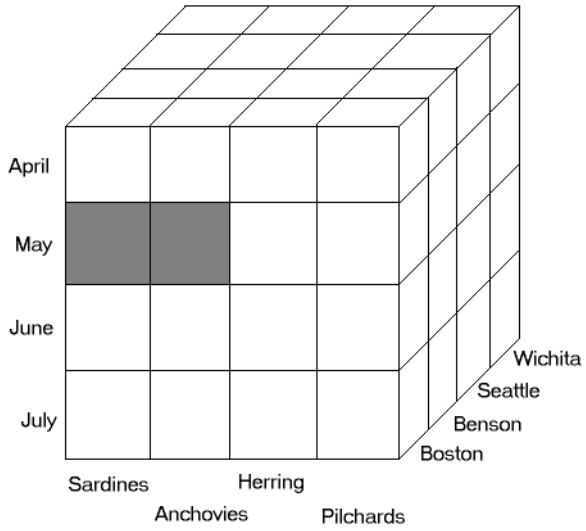
In this worksheet, cell C4 contains the value 44. The value is located at the intersection of C and 4. So 44 is the value that the cell contains and "C4" is the spreadsheet equivalent of a tuple in an OLAP cube.

So how are you supposed to pronounce 'tuple'? Answer – whatever. Arguments rumble on as to whether it rhymes with couple or pupil. It's possible that the former is more common in the US and the latter favored in the UK. The Brits would argue that, if it rhymes with couple, it should be spelt tupple. The Americans would counter with "OK, but if it rhymes with pupil, why isn't it spelt tupil?" In my opinion the only certainty is that anyone both loud and confident about the 'correct' pronunciation is wrong.

Since a tuple points to a single cell, it follows inexorably that each member in the tuple has to be from a different dimension. To put that another way, you can never have a tuple which has two or more members taken from a single dimension. Why not? Well, if you do, it is inevitable that the 'tuple' that you create will end up pointing to more than one cell. For example:

```
(([Product].[Anchovies],[Time].[May],[Store].[Boston]),
([Product].[Sardines],[Time].[May],[Store].[Boston]))
```

has two members from the `Product` dimension and therefore can't be a tuple because it is pointing to more than one cell, as the following diagram shows.



So a first definition of a tuple could be:

A tuple is the intersection of one (and only one) member taken from each of the dimensions in the cube. A tuple identifies a single cell in the multi-dimensional matrix.

Sets

Given the above definition of a tuple, a set becomes very easy to define because a set is simply a collection of tuples which have been defined using the same dimensions.

What do we mean by 'defined using the same dimensions'? Well, take these two tuples.

```
([Product].[Anchovies],[Time].[May],[Store].[Boston])
([Product].[Sardines],[Time].[May],[Store].[Boston])
```

Both have exactly one member from the `Time`, `Store` and `Product` dimensions, so they have been defined using the same dimensions.

❛ *In fact, we can say that they have the same 'dimensionality'.* ❜

So these two tuples, taken together, form a set. Although we go into the exact syntax in Chapter 4, it is worth knowing at this point that in practice, the set has to be wrapped up in curly braces like this:

```
{([Product].[Anchovies],[Time].[May],[Store].[Boston]),
([Product].[Sardines],[Time].[May],[Store].[Boston])}
```

So we can define a set like this:

A set is a collection of tuples with the same dimensionality.

In essence this definition is saying that a set is simply a collection of tuples; nothing too complicated there. However, in the interests of accuracy we need to extend the definition slightly because, as it stands, this definition **implies** that a set always has to contain two or more tuples. While that is often the case, it is also true that the collection of tuples in a set can also be one tuple or even zero tuples.

This may sound weird at first. You may want to ask "But if a set contains only a single tuple, doesn't that make the set a tuple?" You might even want to ask "How can a set possibly contain no tuples?" These are both fair questions.

The answer is that one of the reasons for defining sets in the first place is that some MDX expressions are built to expect multiple tuples. For example, there is a function called `AVG` (which appears in Chapter 7) which will work out averages for you. Clearly, you usually want to average more than one value so the `AVG` function expects to be passed a set rather than a tuple (in fact, it demands to be passed a set). However, we also want it to work under conditions when it is passed a tuple (which will point to a single cell) and even when it is passed an empty set. So the function is designed to expect a set, and a set is defined as being a collection of zero, one or more tuples. This means that our first definition can be extended to read as follows:

A set is a collection of tuples with the same dimensionality. It may have more than one tuple, but it can also have only one tuple, or even have zero tuples, in which case it is an empty set.

So, to summarize so far:

- A tuple points to a single cell, and cannot include more than one member from any particular dimension.
- A set is a collection of tuples.

Exploring the differences between tuples and sets

OK, given the definitions that we currently have, and bearing in mind that we are still using pseudo MDX, is the following a tuple or a set?

```
([Time].[May], [Store].[Boston], [Product].[Anchovies])
```

Answer – a tuple.

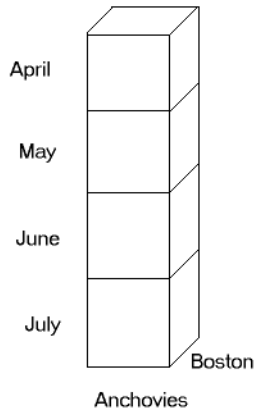
OK, now what about these two?

```
1 ([Store].[Boston], [Product].[Anchovies])

2 {([Time].[April], [Store].[Boston], [Product].[Anchovies]),
  ([Time].[May], [Store].[Boston], [Product].[Anchovies]),
  ([Time].[June], [Store].[Boston], [Product].[Anchovies]),
  ([Time].[July], [Store].[Boston], [Product].[Anchovies])}
```

Well, one big clue is that we have wrapped curly braces around the second one, but ignoring those briefly, it is worth trying to work out in your own mind the differences and similarities between these two.

We could argue that both are pointing to the same collection of cells in the cube:



The first statement:

```
([Store].[Boston], [Product].[Anchovies])
```

is made up of two members, one from the `Store` dimension and the other from the `Product` dimension. Since it doesn't give us any information about the third dimension, we will (for the present) assume no restriction for that dimension.

The second statement:

```
{([Time].[April], [Store].[Boston], [Product].[Anchovies]),
([Time].[May], [Store].[Boston], [Product].[Anchovies]),
([Time].[June], [Store].[Boston], [Product].[Anchovies]),
([Time].[July], [Store].[Boston], [Product].[Anchovies])}
```

actively points to the four cells shown above.

So, as we have said, the two appear to be pointing to the same set of cells. However, the first statement **is** a tuple, the second **is** a set. How can we be so sure?

Well, the second statement is clearly a set because even our simple definition of set tells us that "A set is a collection of tuples with the same dimensionality". This statement has four tuples. Each of these four tuples has exactly one member from the `Time`, `Store` and `Product` dimensions, so these tuples have the same dimensionality. Therefore it is clearly a set.

The first statement conforms to part of the definition of a tuple, the bit that reads "A tuple is the intersection of several members each taken from a different dimension in the cube."

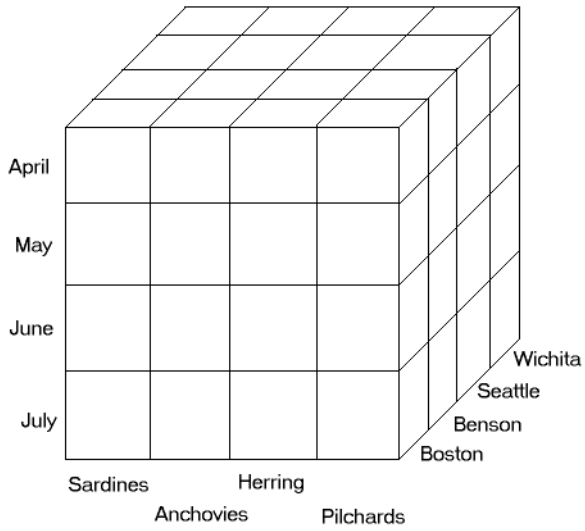
It describes the intersection of two members and each is taken from a different dimension. However, it appears to be failing the first part of our definition, the bit about "A tuple always identifies a single cell in the multi-dimensional matrix." But appearances can be deceptive!

And this is crunch time, this is where people have trouble with the definition of a tuple. So, let's be quite clear about what we are trying to say. This:

```
([Store].[Boston], [Product].[Anchovies])
```

is a tuple. The over-riding reason that we know it is a tuple is because it doesn't use more than one member from the same dimension. This may sound like a very fine distinction, but it isn't.

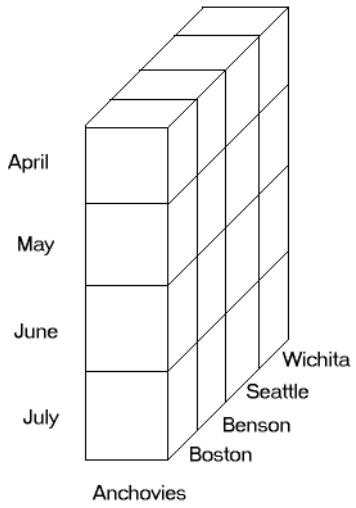
Think about it this way. Suppose that you have a three-dimensional cube like this:



You then define one member from a dimension, say:

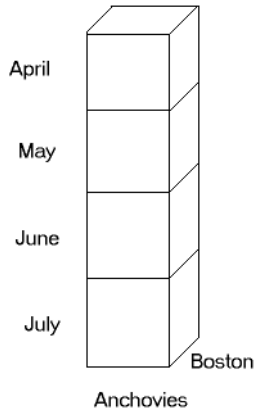
```
([Product].[Anchovies])
```

With that one statement you have trimmed the cube down to this:



Adding another member from another dimension further trims the cube:

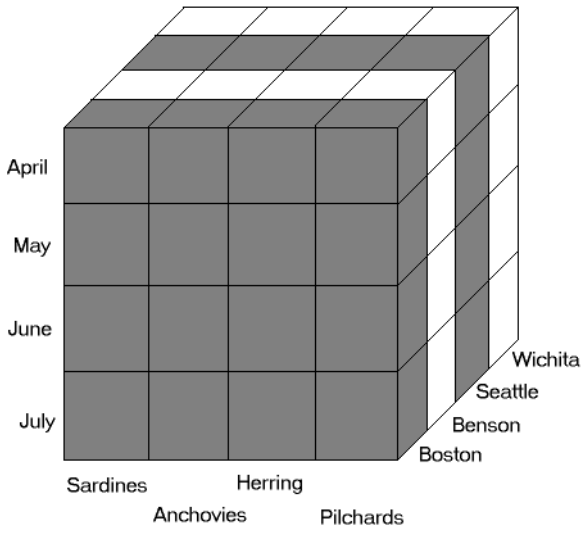
```
([Product].[Anchovies],[Store].[Boston])
```



Each time you add another member from another dimension you are further refining what you want from the cube and, if you choose enough members, each from a different dimension, you will inevitably end up with a single cell.

Now suppose that we start off with this:

```
([Store].[Boston], [Store].[Seattle])
```

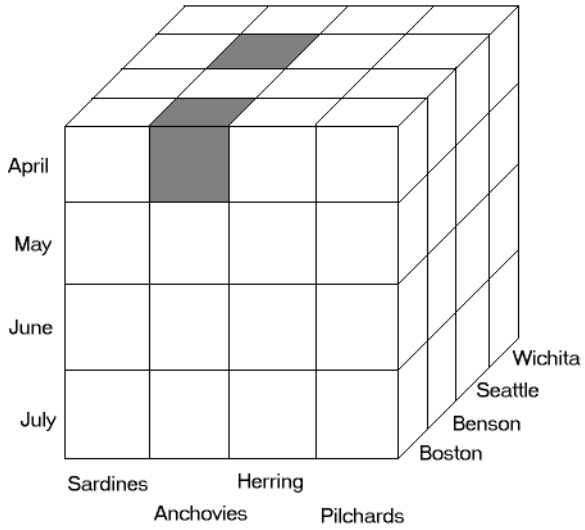


Once again we have only used a single dimension to restrict the cells in which we are interested, but already we are inevitably committed to ending up with a set rather than a tuple, and the reason why that is so is hopefully becoming clearer. No matter what members we use from the other dimensions, and even if we use members from every available dimension, we are going to end up with more than one cell because we started with more than one member from a given dimension.

So even if we restrict this with one member from each of the other two dimensions:

```
([Store].[Boston], [Store].[Seattle],([Time].[April],
[Product].[Anchovies])
```

we still end up with two cells:



So, as we said earlier, this:

```
([Store].[Boston], [Product].[Anchovies])
```

is a tuple. We know it is a tuple, not because it currently points to a single cell, but because it doesn't use more than one member from the same dimension and therefore has the potential to point to a single cell.

Tuples don't have to use a member from every dimension

Now, in case this is making it all sound too complicated, we haven't actually changed the original definition of tuple very much. Initially we said:

A tuple is the intersection of one (and only one) member taken from each of the dimensions in the cube. A tuple identifies a single cell in the multi-dimensional matrix.

This definition assumes that we are defining a tuple using one member from every dimension. If we do use one member from every dimension it is inevitable that our tuple will identify just one single cell. But we don't have to use a member from **every** dimension.

So now we are refining that definition to read as:

A tuple is the intersection of one (and only one) member taken from one or several of the dimensions in the cube. A tuple identifies (or has the potential to identify) a single cell in the multi-dimensional matrix.

We are sticking to the original point that a tuple is always defined by a single member from any given dimension; all we are dropping is the requirement that you have to use each and every dimension to define the tuple.

So the following are all tuples:

```
[Product].[Anchovies],[Time].[May],[Store].[Boston]
[Product].[Anchovies],[Time].[May]
[Product].[Anchovies]
[Time].[May]
[Product].[Anchovies],[Store].[Boston]
```

But do these tuples still point to a single cell?

Yes, because all dimensions have what can be considered to be a 'default member'. So if in an MDX query you don't specify a member for a particular dimension, then the default member for that dimension is implied.

So, if this tuple were used in a query:

```
([Store].[Boston], [Product].[Anchovies])
```

a 'default member' will be used from each of the missing dimensions, effectively turning the tuple into something like this:

```
([Store].[Boston], [Product].[Anchovies],[Time].[May])
```

to ensure that the tuple does point to a single cell.

So, where does the 'default member' come from? In practice, MDX will use the so-called current member (which may also be the default member if the user has not sliced the data). We only mention this here for completeness; when we introduce queries and expressions in later chapters, this will hopefully make more sense.

The take-home message from all of this is that you often don't have to use a member from every dimension when specifying a tuple.

Tuples and hierarchies

Next, it is worth discussing how tuples work with hierarchies.

Suppose that our cube has a hierarchical structure for `Time`. We have levels called `Month`, `Quarter` and `Year` and we have data for the years 1999, 2000 and 2001.

Is a pseudo-MDX expression like this:

```
([Product].[Anchovies],[Store].[Boston],[Time].[2000])
```

still a tuple? The acid test is "Does it still point to a single cell?" The answer is that it does because we have an aggregation member called `2000` and so somewhere in the cube there will be a single cell that holds the value for the total number of anchovies sold in Boston during the year 2000.

*You may well be aware that when you create a real cube not all of the aggregations are necessarily pre-calculated. So you might begin to think "If the aggregation has not been pre-calculated, does this affect whether this is a tuple?" Again it is a good question; the answer is that this is still a tuple. Think of it this way. In MDX a cell is considered to be an intersection of a set of co-ordinates, not a physical object. Therefore aggregated cells **always** exist, because the intersections of the co-ordinates always exist. Some cells will be materialized (that is they will already have been calculated and stored) and some cells will have to be computed on the fly, but they always exist as far as MDX is concerned.*

Sometimes measures behave like dimensions

What happens if we have essentially the same cube but with three measures, say `UnitsSold`, `Profit` and `Price`? Well, it doesn't make too much difference because the measures are going to act, in this case, pretty much like a dimension with three members.

So, we can specify the measure we want in just the same way as we specify a member from a dimension:

```
([Product].[Anchovies],[Store].[Boston],[Time].[May],
[Measures].[Profit])
```

Again, if you don't specify the measure, the expression will use the default measure (these are discussed at the end of Chapter 10).

So be aware that sometimes you'll hear people talking about measures as if they are dimensions. For example, "Try that query again, but this time use the `UnitsSold` member from the measures dimension." This is perfectly normal and, when you think about it, makes perfect sense. In fact, it explains why some of the GUI tools used to manipulate OLAP cubes show the measures as just another dimension.

Tuples revisited

So, hopefully, we've managed to convince you that a tuple is a relatively easy concept but just for completeness, here is a more formal definition.

A tuple is defined as an intersection of exactly a single member from each dimension (hierarchy) in the cube. For each dimension (hierarchy) that is not explicitly referenced, the current member is implicitly added to the tuple definition. A tuple always identifies (or has the potential to identify) a single cell in the multi-dimensional matrix. That could be an aggregate or a leaf level cell, but nevertheless one cell and only one cell is ever implied by a tuple.

Sets revisited

Our earlier definition of set:

A set is a collection of tuples with the same dimensionality. It may have more than one tuple, but it can also have only one tuple, or even have zero tuples, in which case it is an empty set.

still stands up to reasonable scrutiny.

Measures revisited

As we said when talking about tuples, there are times when we treat measures as if they were dimensions, and this is perfectly valid. However,

in case this leaves you with the impression that there is no difference, it seems worth stressing how measures and dimensions do differ.

For a start, measures are frequently numerical and, equally frequently, those numbers are continuously variable (they can contain any possible numerical value between two limits). Sales figures, prices, gross profit – all these values come from a continuously variable range of numbers.

Measures have special properties attached to them, for example Data Type, Format String etc.

Finally, measures are not hierarchical.

Dimensions, on the other hand, are typically character-based and the values they contain are often discontinuously variable (the level Year can contain 2001 and 1999 but not 1999.5).

Member properties

So, is all continuously variable data likely to end up as a measure? In the main, the answer is 'yes', but keep an eye out for exceptions. There are some pieces of data that look at first glance like just the sort of data you'd store as a measure. Take a value such as the floor area of each store: each value will be continuously variable and numeric, so it's a measure, right? Well, no.

Think about a measure – it is stored at the intersection of the members of the dimensions in the cube. Suppose that the Boston store has a floor area of 21,000 sq. ft. If we enter this as a measure, at the intersection of `Boston`, `Q1` and `Anchovies` we'll have a value of 21,000. At the intersection of `Boston`, `Q2` and `Anchovies` we'll find the value 21,000. And at `Boston`, `Q3`, `Herrings`... we'll find... err... 21,000 again. In other words, the value we have for floor area doesn't depend at all on the `Time` dimension or on the `Product` dimension. But measures are supposed to depend on all of the dimensions. So the bottom line is that a measure is not the appropriate place to store data like a store's floor area, nor for any data that depends upon the member in only one dimension for its value.

What do we do instead? Members have **Properties** and the role of a property is to hold information about a member. Our floor area data fits into this category beautifully: it is information about one particular member. In this case, each member is a store and each store's floor area is a piece of information that has relevance only to that particular store.

Summary

If you are new to this whole dimensional data business, there's a great deal of new information here so a quick summary of the main points that we've covered may help.

Data in an OLAP cube is organized into **dimensions** and **measures**:

UnitsSold

	Product			
Time	Sardines	Anchovies	Herrings	Pilchards
April	16	23	12	4
May	14	12	23	6
June	34	19	19	8
July	17	22	14	4

This simple cube has two dimensions – `Product` and `Time`. Both have four **members** and there is one **measure** – `UnitsSold`; so the cube has 16 cells.

The members of a dimension can be (and often are) organized into **hierarchies**; for example, time may be organized into several **levels** such as months, quarters and years. A cell which was the intersection of, say, `Sardines` and `Quarter2` would contain the **aggregated** values for sardines from April, May and June.

We need to extract subsets of data from OLAP cubes and for this we use either **tuples** or **sets**. A tuple is the intersection of one or more members, each of which is taken from a different dimension in a cube. A tuple always identifies a single cell in the multi-dimensional matrix.

A set is a collection of tuples. That collection is usually composed of multiple tuples but can be made up of one or even zero tuples. So, in practice, a set can identify zero, one or more cells in the multi-dimensional matrix.

We sometimes want to store additional data in an OLAP cube but we find that it can't be stored as a measure because it cannot logically be placed at the intersection of all the dimensions. Instead it logically relates to

members of a single dimension. For example, the floor area of a store depends simply upon which store we are considering, it doesn't depend upon the month, nor on the product. Such information isn't stored as a measure; it is stored as a **member property** of the appropriate member.