STATUS	TOTAL_2003	TOTAL_2004	TOTAL_2005
Shipped	3.573.701,25	4.750.205,89	1.513.074,46
Resolved	28.550,59	24.078,61	98.089,08
Cancelled	75.132,16	187.195,13	0
On Hold	0	26.260,21	152.718,98
Disputed	0	0	72.212,86
In Process	0	0	144.729,96

How the SQL CASE WHEN construct can help you in pivoting data?

kettle.bleuel.com |

Within Kettle you have the Normalizer and Denormalizer steps to help with pivoting (a simplified use case for pivoting is changing the row/column axis, see also transpose). Imagine you would like to have a cross table, you may use a Kettle transformation to accomplish this. (In these days Pentaho Reporting's cross tab functionality is on the road map but not implemented, yet. One solution would be to use a Kettle Transformation as a data source.).

Another solution for a cross tab is to use the SQL CASE WHEN construct. Here is an example:

The result of the following sample query *select status, year_id, sum(totalprice) as totalprice from orderfact group by status, year_id* looks like this:

STATUS	YEAR_ID	TOTALPRICE
Shipped	2003	3.573.701,25
Resolved	2003	28.550,59
Cancelled	2003	75.132,16
Shipped	2004	4.750.205,89
Cancelled	2004	187.195,13
Resolved	2004	24.078,61
On Hold	2004	26.260,21
Shipped	2005	1.513.074,46
Resolved	2005	98.089,08
On Hold	2005	152.718,98
Disputed	2005	72.212,86
In Process	2005	144.729,96

When we want to put the years (2003, 2004, 2005) into separate columns, we can use the SQL CASE WHEN construct:

CASE WHEN Boolean_expression THEN result_expression [...n] [ELSE else_result_expression]

to accomplish this, e.g.:

select status,
sum(case when year_id=2003 then *totalprice* else 0 end) as TOTAL_2003,
sum(case when year_id =2004 then *totalprice* else 0 end) as TOTAL_2004,
sum(case when year_id =2005 then *totalprice* else 0 end) as TOTAL_2005
from orderfact group by status

Et voilà, you have data that can be used as a cross tab:

Another use case for dealing with planning data

In another use case, we store actual and planning data (e.g. for turnover, quantity, weights etc.). To make it a bit more challenging, planning data is often stored as a budget and multiple forecasts or different plan versions (scenarios).

One solution for this problem is to store these data in separate columns, e.g. have columns for actual and other columns for planning data. Having multiple measures (see above) needs to have multiple columns for each type of the measures, e.g. turnover_actuals, quantity_actuals etc. and have this also for all types of planning scenarios. Thus you would get a big amount of columns and the biggest issue is: What happens when you need a new planning scenario? You would need to add more columns and your data model is not flexible enough and needs to be changed.

Another solution is to store the definition of an actual or planning type as an extra TYPE_ID in our table. This means each row is marked whether it is an actual or any kind of planning type.

But, when you sum up the whole table, the result would be wrong since it contains a mix of actual and planning data.

This can be solved also with the CASE WHEN construct even within a Mondrian schema definition:

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

<Measure name="Sales Budget" aggregator="sum" formatString="#,###.00">
<MeasureExpression>  <SQL dialect="generic">
    (case when type_id=1 then TOTALPRICE else 0 end)
</SQL>  </MeasureExpression>  </Measure>
```

[for more details see the Mondrian documentation about Measures]

With this solution you are flexible with your data model and the result will be correct.

Another more complex use case is to combine this with a reporting solution for project costing or contribution accounting. When you have a large number of different costs and would store these costs in separate columns you will reach a non manageable amount of columns and your data model is not flexible enough. It's better to store these different types of costs in rows and you are much more flexible.

By all the flexibility with the data model you need to consider performance aspects of your database, e.g. if column based databases behave different with the CASE WHEN (I have not tested this) and different approaches should be tested. Any comments and findings are much appreciated.