BusinessObjects and Oracle9i OLAP

Contents

Introduction

The release of Oracle9i AW (Analytical Workspace) makes it possible for the Business Objects semantic layer to integrate closely with Oracle OLAP technology. This is because AW allows Oracle OLAP data to be queried using standard SQL. Business Objects' patented semantic layer, which enables users to create complex SQL queries using visual objects, is already the industry-leading SQL generation technology. As a result, Business Objects users can benefit from the performance and calculation power of Oracle OLAP without leaving the familiar BusinessObjects query-building environment.

Oracle relational views also make it possible for Business Objects to exploit several trends in the OLAP market place. OLAP is moving away from pure multidimensional databases, and hybrid OLAP/relational solutions are becoming much more common. A Business Objects solution can hold both multidimensional and relational data in a manner that is completely hidden from the end user. Furthermore, the focus is moving away from proprietary OLAP APIs and query languages towards languages that are already industry standards. SQL is a prime example of this—with a Business Objects/Oracle solution you can query your multidimensional data using the most common database query language.

Advantages

Accessing Oracle cubes through the Business Object semantic layer has three big advantages:

- Performance. For common-size data sets, OLAP cubes provide better performance because they contain aggregated, pre-computed data that is instantly available to the query tool;
- Advanced OLAP calculations (see Using Oracle OLAP functions). The Oracle OLAP cube provides pre-calculated data and also allows users to compute advanced calculations such as growth ratios or trends on the fly;
- Transparent drill through from the cube to relational data (see Drilling outside the cube). With this solution, a single universe can encompass a cube and relational tables. This means that ausers can drill from aggregates to details within the same query-building environment.

Oracle9i and OLAP cubes

Oracle9i AW exposes Oracle OLAP cubes as relational views, which can be queried using standard SQL. Oracle exposes dimensions and rollups in a relational view. For BusinessObjects users to be able to profit from this capability, the BusinessObjects administrator must design a BusinessObjects universe (which transforms a visual query to SQL) around the cube view. The universe must handle rollups correctly, which imposes a non-standard approach to universe design. The next sections discuss the way in which the universe must be designed.

Universe design principles

In the example used throughout this paper, the view, `OLAPCUBE`, is derived from an Oracle cube using an AW query. The relational view contains Revenue (the measure) and two hierarchies—Time (Year, Quarter, Month) and Geography (Country, Region, City). The revenue in the Revenue column is aggregated according to the level, which means that any SQL statement returning data from the view must filter according to the values in the `time_level` and `geo_level` columns:

```
YEAR    QTR     TIME_LEVEL      COUNTRY         REGION  GEO_LEVEL               REVENUE
                ALL                                     ALL                     5000
2002            YEAR                                    ALL                     2000
2003            YEAR                                    ALL                     3000
2002    Q1      QTR                                     ALL                     500
2002    Q2      QTR                                     ALL                     500
```

The problem that needs to be overcome occurs because the cube contains rolled-up data, and BusinessObjects generates SQL per object, whereas to handle rollups it needs to generate per *object combination*. For example, if you create a Year object (based on the `year` column in the table) and place it in a query, BusinessObjects builds a `WHERE` clause that restricts time_level to '`YEAR`' (`WHERE time_level = 'YEAR'`). But if you include the Year and Quarter objects, you want BusinessObjects to restrict `time_level` to '`QUARTER`' (`WHERE time_level = 'QTR'`). BusinessObjects default behaviour is to create a `WHERE` clause that returns no rows (`WHERE time_level = 'YEAR' AND time_level = 'QTR'`).

To solve this you treat the cube view as the fact table in a snowflake schema, and surround it with dimension tables. You are not interested in the data in these dimension tables: they exist solely to force BusinessObjects to generate the correct SQL to filter the `OLAPCUBE` table.

The following sections:

- describe how to set up the universe;
- give examples of the SQL that the universe generates;
- explain how this SQL relates to the universe design.

You carry out the following tasks when designing the universe:

Task	Purpose
Include the relational view in the universe.	You access the OLAP cube data through the relational view.
Create tables hierarchies that correspond to the dimensional hierarchies in the cube. These hierarchies are build from Designer aliases on the SYS.DUAL table.	The dimension tables are used to generate the join conditions that restrict data in the relational view.
Join the dimension tables hierarchically using regular joins.	The joins between the dimension tables are used to generate the join conditions that restrict data in the relational view.
Join the dimension tables to the relational view using shortcut joins.	The shortcut joins are used to generate the join conditions that restrict data in the relational view.
Create dimension and measure objects	The user builds queries in the Query Panel using these objects

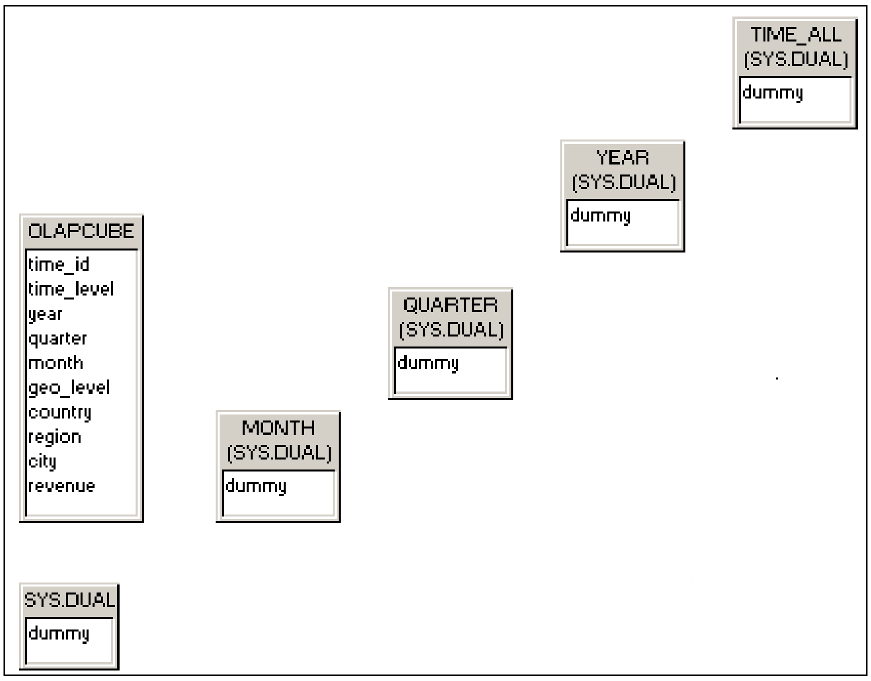
The following sections describe the universe design in more detail.

Setting up the universe

To include the relational view, simply insert it as a table into the universe.

Creating table hierarchies

In this example the relational view has two hierarchies (Time and Geography), so you need to create dimensional hierarchies to match them. You create these hierarchies using aliases on the Oracle system table `SYS.DUAL`, which always contains one row. You create a time hierarchy as follows:



Joining the dimension tables using regular joins

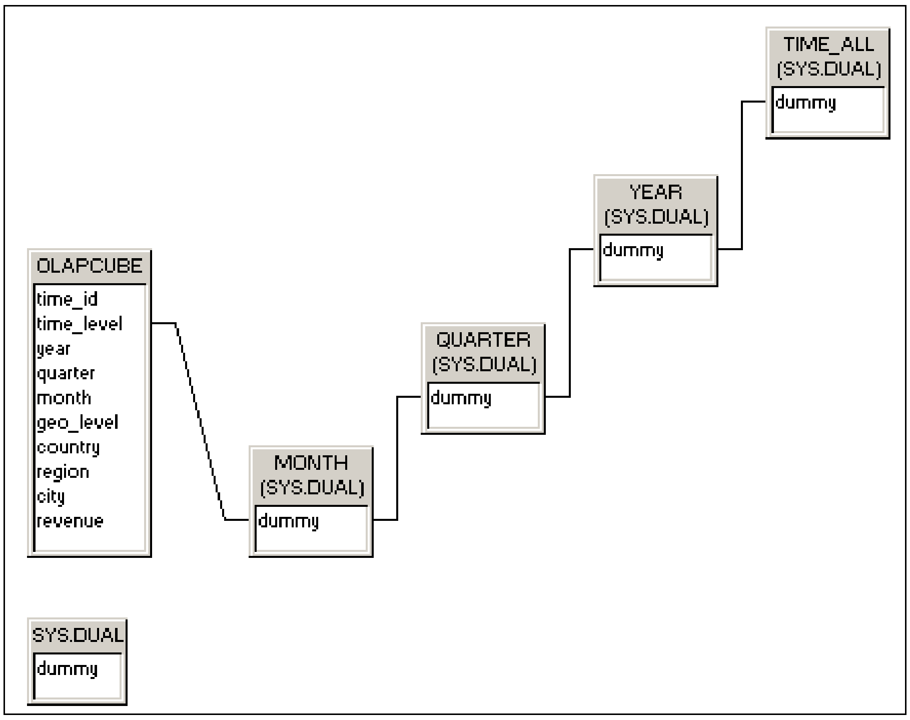
The hierarchy has four levels (All, Year, Quarter, Month). You need to join these levels using regular joins. The join expressions consist of a condition that will always be true, for example: `'1=1'`, but to force Designer to create a join (which it would not do because the expression does not reference any tables) you need to include the 'real' join expression in comments; for `example MONTH.DUMMY = QUARTER.DUMMY`. The full syntax of the join between `MONTH` and `QUARTER` is therefore:

```
/* MONTH.DUMMY = QUARTER.DUMMY */ 1=1
```

A similar pattern applies to all the other join expressions in the hierarchy, giving the following list:

Joined tables	**Expression**
Month, Quarter	`/* MONTH.DUMMY = QUARTER.DUMMY */ 1=1`

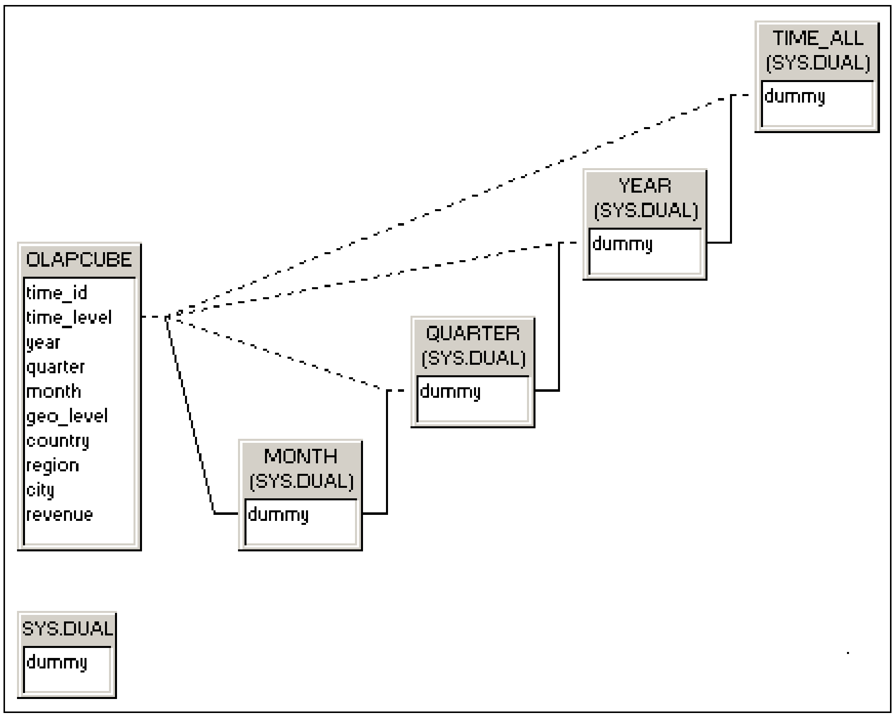
Quarter, Year	`/* QUARTER.DUMMY = YEAR.DUMMY */ 1=1`
Year, Time_All	`/* YEAR.DUMMY = TIME_ALL.DUMMY */ 1=1`



Joining the dimension tables using shortcut joins

Shortcut joins ensure that BusinessObjects generates SQL per object combination rather than per object. BusinessObjects uses shortcut joins when it can omit tables from a query and take a ‘shortcut’ between two tables that are not directly linked in a hierarchy. For example, if a shortcut join is defined between the `QUARTER` and `OLAPCUBE` tables, BusinessObjects does not need to join through the `MONTH` table to retrieve revenue by quarter.

Each table in the time hierarchy (except the lowest-level table) must be joined to `OLAPCUBE.time_level` by a shortcut join, as shown below:



The join expression must include the expression that will restrict the rows returned from `OLAPCUBE`; in the case of `QUARTER`, this is `OLAPCUBE.time_level = 'QTR'`. To ensure that Designer allows the join, the expression must also reference the `MONTH` table, which should appear inside comments (because it plays no part in the actual join expression that you are interested in generating). The full join expression is therefore:

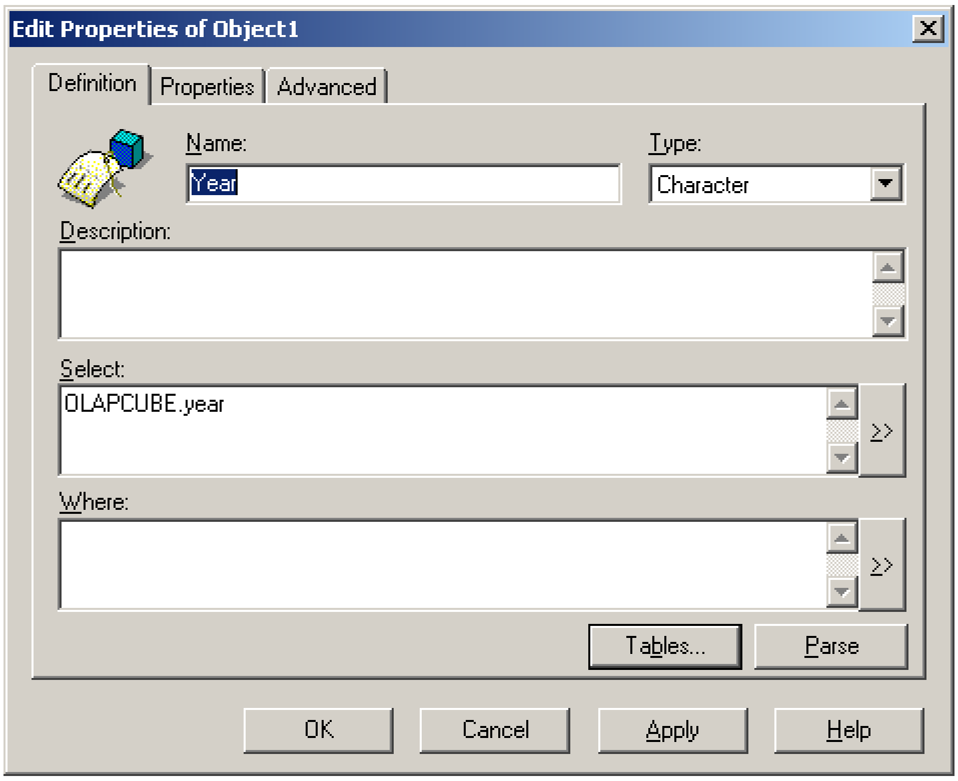
```
/* QUARTER.DUMMY */ OLAPCUBE.time_level = 'QTR'
```

The full list of shortcut join expressions for the time hierarchy is as follows:

Joined tables	Expression
MONTH, OLAPCUBE	/* MONTH.DUMMY */ OLAPCUBE.time_level = 'MONTH'
QUARTER, OLAPCUBE	/* QUARTER.DUMMY */ OLAPCUBE.time_level = 'QTR'
YEAR, OLAPCUBE	/* YEAR.DUMMY */ OLAPCUBE.time_level = 'YEAR'
TIME_ALL, OLAPCUBE	/* TIME_ALL.DUMMY */ OLAPCUBE.time_level = 'ALL'

Creating dimension objects

You create dimension objects based on fields in the `OLAPCUBE` table. For example, the Year object is based on `OLAPCUBE.year`. However, you need to make sure that each object is associated with the appropriate tables for join generation. You do this by clicking **Tables** in the *Edit Properties of [object]* dialog box and selecting the appropriate tables in the Tables dialog box:


Edit Properties of Object1

Definition | Properties | Advanced

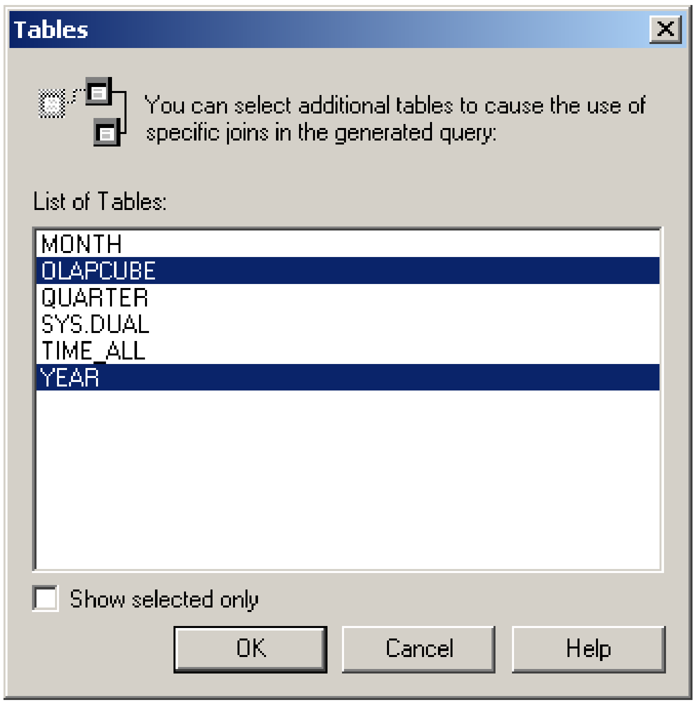
Name: Year

Type: Character

Description:

Select: OLAPCUBE.year

Where:


Tables

You can select additional tables to cause the use of specific joins in the generated query:

List of Tables:

- MONTH
- OLAPCUBE [✓]
- QUARTER
- SYS.DUAL
- TIME_ALL
- YEAR [✓]

☐ Show selected only

In the case of Year, you select the `YEAR` and `OLAPCUBE` tables. Remember that `YEAR` is an alias on the `SYS.DUAL` table, but you need its join definition to appear in the SQL generated by BusinessObjects.

Each object therefore needs to reference the `OLAPCUBE` table and the object's corresponding dimension table.

Creating measure objects

The Revenue measure object is simply based on the Revenue column in the `OLAPCUBE` table. In this case, the object must be associated with all highest level tables as well as the `OLAPCUBE` table. (In this example the highest level tables are `TIME_ALL` and `GEO_ALL`.) The complete list of associated tables is therefore `TIME_ALL`, `GEO_ALL` and `OLAPCUBE`. This ensures that, should the user include the Revenue object in a query without including any lower-level dimensions, the query will automatically apply `WHERE` clause logic associated with the highest-level aggregations.

Example universe queries

How does the 'virtual' snowflake schema that you have created generate SQL to filter the `OLAPCUBE` table? Look first at a simple query – Revenue only, with no dimensions selected.

Revenue only

The generated SQL for a query that returns Revenue only, with no selected dimensions, is:

```
SELECT OLAPCUBE.revenue
FROM   SYS.DUAL  Time_All,
       SYS_DUAL  Geo_All
WHERE  /* Time_All.dummy */ OLAPCUBE.time_level = 'All'
AND    /* Geo_All.dummy */  OLAPCUBE.geo_level = 'All'
```

You can see from this SQL that the sole purpose of the dimension tables you created is to cause BusinessObjects to generate the appropriate filter logic in the `WHERE` clause. In this case, `OLAPCUBE.time_level = 'ALL' AND OLAPCUBE.geo_level = 'ALL'`. BusinessObjects selected the shortcut joins at the highest level of the hierarchy (those that join `TIME_ALL` to `OLAPCUBE` and `GEO_ALL` to `OLAP_CUBE`) to generate the `WHERE` clause.

Revenue by year

For this query the user selects the Year and Revenue objects. The generated SQL is:

```
SELECT OLAPCUBE.year,
       OLAPCUBE.revenue
FROM   SYS.DUAL  Time_All,
       SYS_DUAL  Year,
       SYS_DUAL  Geo_All
WHERE  /* Year.dummy  = Time_All.dummy */ 1=1
AND    /* Year.dummy */  OLAPCUBE.time_level = 'Year'
AND    /* Geo_All.dummy */ OLAPCUBE.geo_level = 'All'
```

This query shows how the schema generates `WHERE` clause logic by object combination rather than object. The SQL references both the `TIME_ALL` and `YEAR` tables, but the join logic for both these tables (`OLAPCUBE.time_level = 'All' AND OLAPCUBE.time_level = 'Year')` returns no rows. (`OLAPCUBE.time_level` cannot be equal to “All” and “Year”.) What happens instead is that BusinessObjects includes the join logic between YEAR and TIME_ALL (which is simply the always-true expression 1=1), then uses the shortcut join between YEAR and OLAPCUBE, whose join expression is OLAPCUBE.time_level = 'Year'. Thus the WHERE clause contains the appropriate join logic to return Revenue at the Year level.

Revenue by month by region

For this query the user selects the Region, Month and Revenue objects. The generated SQL is:

```
SELECT OLAPCUBE.region_desc,
       OLAPCUBE.month_desc,
       OLAPCUBE.revenue
FROM   SYS.DUAL  Time_All,
       SYS_DUAL  Year,
       SYS_DUAL  Month,
       SYS_DUAL  Geo_All,
       SYS_DUAL  Region
WHERE  /* Year.dummy  = Time_All.dummy*/ 1=1
AND    /* Month.dummy = Year.dummy */ 1=1
AND    /* Month.dummy */ OLAPCUBE.time_level = 'Month'
AND    /* Region.dummy = Geo_All.dummy */ 1=1
AND    /* Region.dummy */ OLAPCUBE.geo_level = 'Region'
```

In this case BusinessObjects uses the shortcut joins between MONTH and OLAPCUBE and REGION and OLAPCUBE. The joins between the dimension tables appear as the always-true expression 1=1. The result is that the WHERE clause contains the correct filtering logic to restrict the query to Region and Month (OLAPCUBE.time_level = "Month" and OLAPCUBE.geo_level = "Region").

All queries generated by this schema return Cartesian products, because the SYS.DUAL table is referenced multiple times in the FROM clause without corresponding joins in the WHERE clause. This is unimportant because SYS.DUAL contains one row only. Furthermore, BusinessObjects does not warn the user that the query will generate a Cartesian product (which normally happens when you run such a query). This is because the commented portions of the join expressions lead BusinessObjects to believe that the schema contains all the appropriate joins for avoiding a Cartesian product.

Aggregate awareness

There is an alternative solution to the problem of handling rollups: you can use aggregate awareness. This solution is suitable only when there are few dimensions in a cube. In the example, aggregate awareness is feasible: you need a total of X aggregate tables to cover all possible combinations of dimensions. In general, when you have x dimensions, you need 2^x aggregate tables. But aggregate awareness quickly becomes unwieldy, and has major performance drawbacks, when there are a large number of dimensions. For example, to handle a cube with 10 dimensions, you need 1024 aggregate tables. As a general rule, it makes more sense to use a ‘virtual’ snowflake schema to handle rollups.

Using Oracle OLAP functions

Because Designer can generate SQL using any function supported by the DBMS, your schema can take advantage of Oracle AW OLAP functions to perform calculations. This is a very powerful capability, and allows you to create high-performance calculations at the cube level that would be much more difficult to implement at the BusinessObjects level.

A good example is the calculation of a measure for the previous period. You can do this using BusinessObjects, but it requires the data in the report to be sorted in a specific way. If you calculate at the cube level, the report can be sorted in any manner and the calculation will always return the correct value. Furthermore, you take advantage of Oracle's excellent calculation performance.

You can add a measure, Revenue Prior Period, to the example universe to calculate revenue for the previous period. The formula for this measure is:

```
olap_expression(olap_calc, 'lag(revenue, 1, time, LEVELREL
time_levelrel)')
```

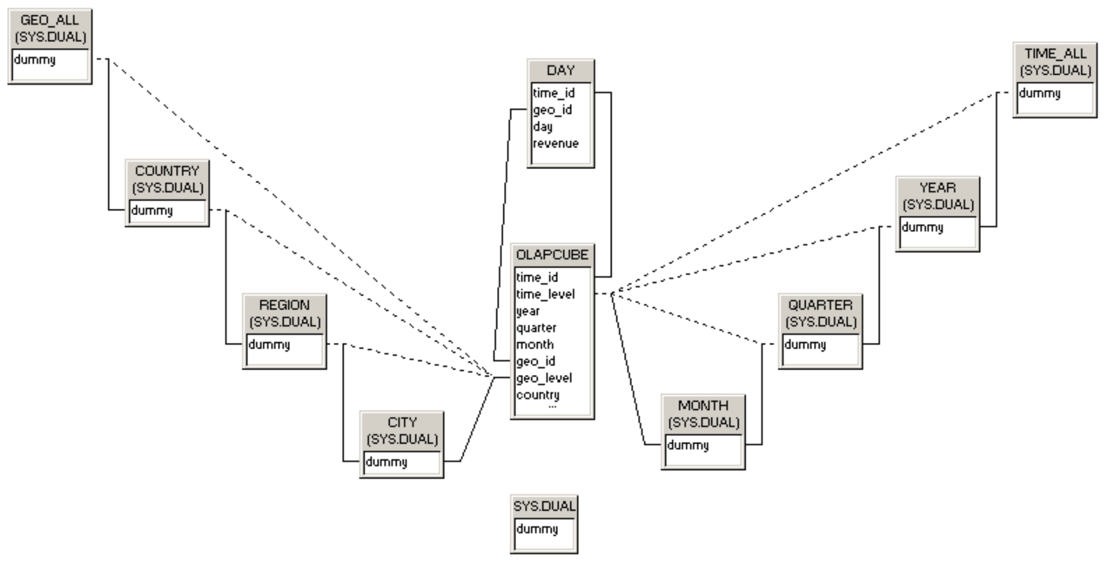
Drilling outside the cube

OLAP cubes contain aggregated data, so it is often necessary to drill outside the cube if you want to analyze data at a detailed, lower level. This data is often stored in a relational database, which means that you need to jump from an OLAP query to a relational query to continue your low-level data analysis.

Because the Oracle cube is exposed as a relational view within a BusinessObjects universe, drilling outside the cube is much more straighforward using a BusinessObjects/Oracle solution. The relational tables that store lower-level data can be referenced in the universe; this means that, from the user's point of view, the switch from cube data to relational data is seamless—both are possible within the same query-building environment.

Adding a DAY table

You can add a `DAY` table to the example schema to demonstrate this. This table holds revenue by day, as opposed to the OLAP cube, which goes down only as far as month. You join the table to the fact table in Designer:

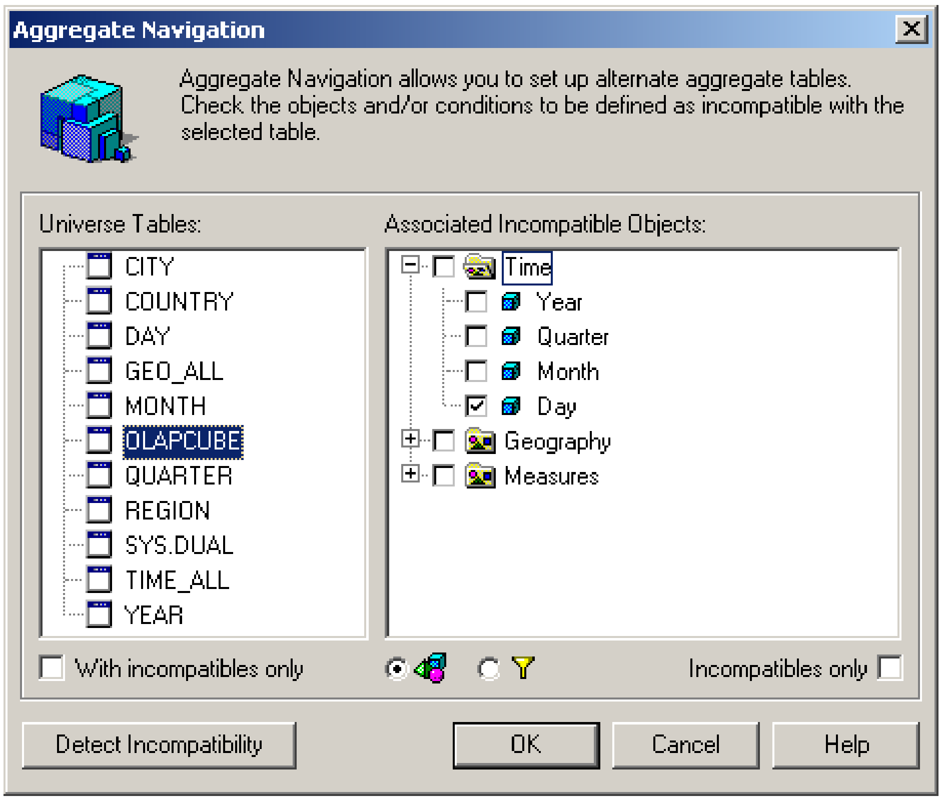


You need to modify the Revenue object to use the `DAY` table where appropriate. This involves:

- setting up aggregate navigation;
- forcing BusinessObjects to generate filters on the `OLAPCUBE` table.

Setting up aggregate navigation

Aggregate navigation needs to be set up so that BusinessObjects does not use the `OLAPCUBE` table to calculate revenue when the Day object appears in the query. To do this, set the Day object as incompatible with the `OLAPCUBE` table in the Aggregate Navigation dialog box:



Aggregate Navigation allows you to set up alternate aggregate tables. Check the objects and/or conditions to be defined as incompatible with the selected table.

Universe Tables: CITY, COUNTRY, DAY, GEO_ALL, MONTH, OLAPCUBE, QUARTER, REGION, SYS.DUAL, TIME_ALL, YEAR

Associated Incompatible Objects:

- ☐ Time
  - ☐ Year
  - ☐ Quarter
  - ☐ Month
  - ☑ Day
- ☐ Geography
- ☐ Measures

☐ With incompatibles only

Incompatibles only ☐

Forcing BusinessObjects to generate filters

In previous steps you used the Tables dialog box to tell BusinessObjects which tables to use to generate filter conditions in the `WHERE` clause. You cannot do this with the Revenue object because the set of tables is different depending on which table is used to calculate revenue. If the `OLAPCUBE` table is used, you want the hierarchies to be filtered at the ‘All’ level. If the `DAY` table is used, you want the hierarchies to be filtered at the lowest level for each (in this case `MONTH` and `CITY`). This is because the `DAY` table holds data at the level below the lowest levels in `OLAPCUBE`, and joins to `OLAPCUBE` using the keys that match the lowest-level rows in the fact table. The `WHERE` clause must filter to include these rows. If you did not ensure that the hierarchies were filtered at the lowest level, the `WHERE` clause would by default include the ‘next highest’filter in the hierarchy. For example, the query Year, Day, Revenue would include the condition `OLAPCUBE.time_level = ‘Year’`. The result would have no rows, because there are no rows in the `DAY` table to match the Year-level rows in `OLAPCUBE`.

You accomplish this by including commented references to the table sets before the appropriate object in the definition of Revenue. The formula for the example is as follows:

```
@Aggregate_Aware(
/* TIME_ALL.dummy GEO_ALL.dummy */
OLAPCUBE.revenue,
/* MONTH.dummy CITY.dummy */
DAY.revenue)
```

Now, whenever you use the Revenue object in a query, BusinessObjects determines whether to use the `OLAPCUBE` or `DAY` tables to calculate it, and generates appropriate `WHERE` clause filters on `OLAPCUBE`.