Creating Relational Universes: Best Practices

Applies to:

SAP BusinessObjects BI platform 4.0. For more information, visit the Business Objects homepage.

Summary

This document provides information and guidelines about designing relational universes in the information design tool, with a focus on performance-related features.

Author: Didier Mazoué

Company: SAP

Created on: 01 September 2011

Author Bio

Didier Mazoué is an expert product manager in the semantic layer team at SAP Business Objects Enterprise Information Management. Didier is the expert on most major OLAP databases, SAP NetWeaver BW, HANA, and relational databases. Before joining the product group in 2005, Didier was a strategic pre-sales field representative specializing in closing large database deals. He currently resides in France and works out of the SAP Lab in Levallois-Perret.

Table of Content

Audience

This document is intended for experienced universe designers who have a working knowledge of building universes using the information design tool.

Purpose

This document presents best practices for creating a relational universe, including recommendations for practices before you design the universe, and design considerations for the data foundation and business layer. A refresher on the steps to create a relational universe using the information design tool is also presented.

 This symbol indicates universe design recommendations dedicated to improving database and query performance.

Before designing the universe

Adopt standards

Standards for the components of a universe will help guarantee consistency and stability in the final product. If your enterprise has a data administrator, be sure to involve this person in the adoption of standards.

Standards can be specified for:

- Universe names
- Object definition guidelines
- Names for simple objects
- Names for complex objects
- Names for aggregate objects
- Folder names
- Alias names
- Filter names
- List of values names
- Parameter names
- Help text

The standards may be revised during the course of the universe development project as you become more familiar with the product.

Identify candidate objects for the universe

The best way to identify candidate objects is by interviewing end users, for example:

- *What type of information do you need to do your job*?
- *How do you know you are doing well?* or *How does your boss know you are performing well?*
- *What kind of information do others ask you for?*

Ask additional questions to capture unique *ad hoc* requirements. For example:

1. *When someone comes to you asking for specific information they can't get out of a report, what types of things do they ask for?*

As users answer these questions, record their answers in terms of class and object requirements.

Candidate classes and objects can also be identified by reviewing existing BI documents, for example special summary reports that are hand-produced for managers.

These questions help you determine which tables and joins to add to the data foundation and what objects to include in the structure of the business layer.

Identify objects for the business layer

Dimension objects describe actual things, and are used to qualify measures. Region and Product are examples of dimensions.

Dimensions can have attributes associated with them. For example, a Customer dimension may have Address and Phone Number details associated with it.

Tip: Don't assume that numeric data is always the source of a measure. It must make sense to aggregate the information for it to be suitable as a measure. For example, summing up sales revenue makes perfect sense; Sales Revenue is a measure. But it doesn't make sense to aggregate product list prices; List Price is a dimension, or perhaps a detail for the Product dimension. You will also find that you can create measures where there is no numeric data simply by counting things. This can result in measures like Number of Orders.

Identifying candidate objects as dimensions, attributes, or measures will facilitate slice and dice analysis. You can also plan for drill down and drill up analysis by identifying dimensional hierarchies (navigation paths).

Creating a relational universe with the information design tool

The basic steps to create a relational universe are summarized below. For a complete description of the procedures, see the Information Design Tool User Guide.

1. Create a local project: **File > New > Project**.

2. Create a relational connection in the local project: Right-click the project folder and select **New > Relational Connection**. See the recommendations for connection parameters in the next section.

3. Create a data foundation based on the new created connection: Right-click the project folder and select **New > Data Foundation**.
    a. Single Source: if you plan to use a single relational database
    b. Multisource-enabled: if you plan to use more than one relational data source

4. Edit the data foundation by double-clicking the data foundation name in the local project. See the design recommendations for the data foundation in the next section.

    - Select tables, SQL views and HANA Views you want to use in the universe
    - Create joins
    - Set join cardinalities
    - Create alias tables (optional)
    - Create derived tables (optional)
    - Run the check integrity tool to validate the data foundation
    - Edit the data foundation properties and SQL generation parameters
    - Save the data foundation

5. Create a business layer on the data foundation: Right-click the project folder and select **New > Business Layer**.

Note: The **Automatically create classes and objects** option is selected by default. If you leave the option selected, the business layer outline (folders and dimensions) are created automatically.

6. Edit the business layer by double-clicking the business layer name in the local project. See the design recommendations for the business layer in the next section.

    - If you automatically created the classes and objects, all objects are created as dimensions. You need to specify the measures explicitly using the **Turn into Measure** command. Otherwise, create the business layer outline:
        - Create folders and subfolders to organize the business layer
        - Drag and drop data foundation columns in the desired folders and rename the objects if needed
        - Turn the needed objects into measures and set the SQL aggregation function
    - Create attributes (optional)
    - Create predefined filters (optional)
    - Create list of values (optional)
    - Create parameters (optional)
    - Create navigation paths (optional)
    - Edit the business layer properties

- Run the check integrity tool to validate the business layer
- Save the business layer

7. You can create queries (optional) to validate and test the universe.

8. You can now publish the business layer to a local folder and test the published universe with Web Intelligence Rich Client: Right-click the business layer and select **Publish > To a Local Folder**.

9. To publish the universe to a repository, first publish the connection to the repository: right-click the connection in the local folder and select **Publish Connection to a Repository**. When asked, create a connection shortcut in the local folder.

10. Edit the data foundation and change the connection to use the connection shortcut. Save the data foundation.

11. Publish the universe to the repository: Right-click the business layer and select **Publish > To a Repository**.

Relational universe design recommendations

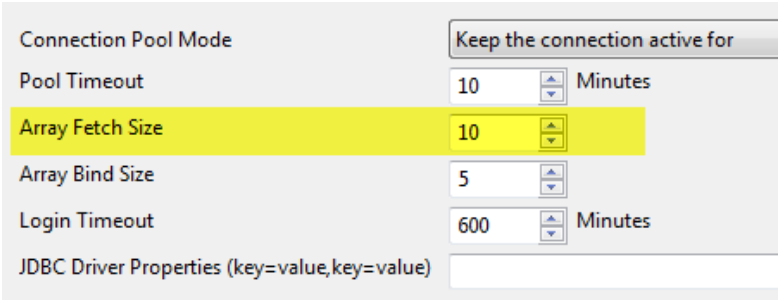
The following design best practices can be implemented in the connection, data foundation, and business layer editors. For a detailed description of the procedures, see the Information Design Tool User Guide.

Connection parameter recommendations

When you create or edit the connection, you can modify the following configuration parameters:

Array Fetch Size

The Array Fetch Size defines how many rows are returned from the database at each fetch. The default is 10, but it can be increased (to a maximum 1000) for better performance. Higher values of Array Fetch Size make queries run faster, but require more memory.



If the network can support large packets it is possible to increase the Array Fetch Size to increase the number of lines retrieved in each fetch and therefore decrease the number of fetch operations.

Note: If you need to retrieve 100 rows and have an Array Fetch Size of 20, then 5 fetch operations will be executed. If Array Fetch Size is 100 or higher, then only 1 fetch operation will be executed (but the network package will be bigger).

It is useful to test different values of Array Fetch Size to find the correct balance between fetch operations and the size of network transfers.

ConnectInit for optimization parameters

Using the ConnectInit parameter it is possible to send commands to the database when opening the session. These commands can be used to set database-specific parameters used for optimization.

- The syntax is defined in the connection parameters
- The string can contain variables (for example, user name: @Variable('BOUSER'))

For example:

With Sybase ASE it is possible to set the database query optimization algorithm to perform DSS queries instead of the default OLTP queries for the session.

With Teradata it is possible to set the row read lock level to "access" for the session. This generates an optimized tactical query:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL RU
```

Data foundation design recommendations

Limit the number of joins while keeping the record size low

If the user queries are known in advance and there is complete freedom in the configuration of the data model, the schema should be fine-tuned to decrease the number of joins and the quantity of data which is necessary for the generation of a correct answer, but which is not needed in the final answer.

Joins can be reduced by putting items which are often requested together in the same table. You can also create join indexes or tables to bypass the entire normal join path.

Joins: Chose the join type

Where table relationships are optional, the type of join to use must be chosen carefully. The use of standard versus outer joins will impact the results of user queries. Using the wrong type of join may produce results that are not what users expect.

In SQL, a standard join between two tables will return only rows where *both* tables meet the join criteria. If one of the tables has no corresponding row in the second table, its data will not be returned.

An outer join tells the database processing the SQL query to substitute a "null" row if one of the joined tables has no corresponding row in the other table. With an outer join, information in one table that does not have corresponding data in the second table is returned with "blanks" in columns from the second table.

Tip: Be careful! An outer join can alter the results of aggregate functions.

Define shortcut joins whenever possible

Shortcut joins are alternative join paths. The query generation engine, when defining the SQL query, looks for all possible paths and chooses the shortest one in terms of joins.

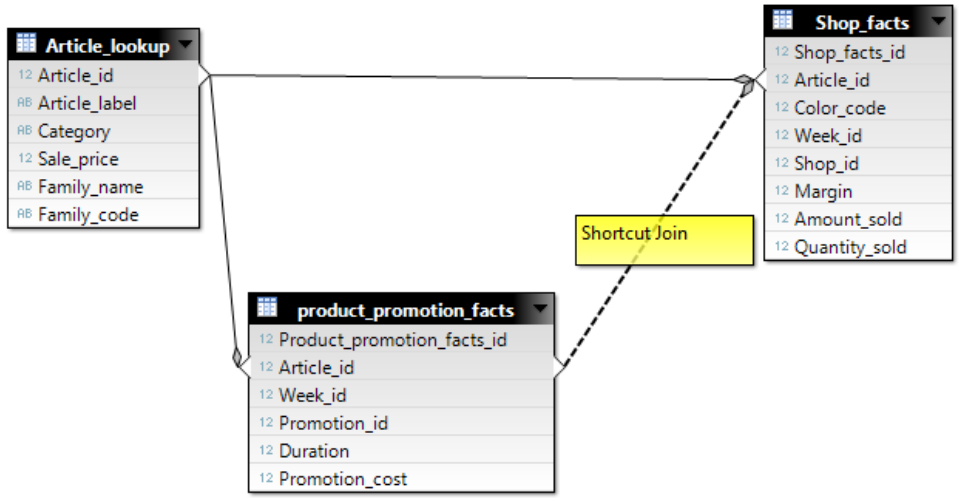
Note: Shortcut joins are not taken into account to define contexts, but only to decrease the number of joins whenever possible.

In the data foundation in

Figure 1, you can link to the `Amount_Sold` in `Shop_facts` directly without the need to join the `Article_lookup` table.

The SQL generator will decide the fastest path based on the objects requested in the query.

Figure 1



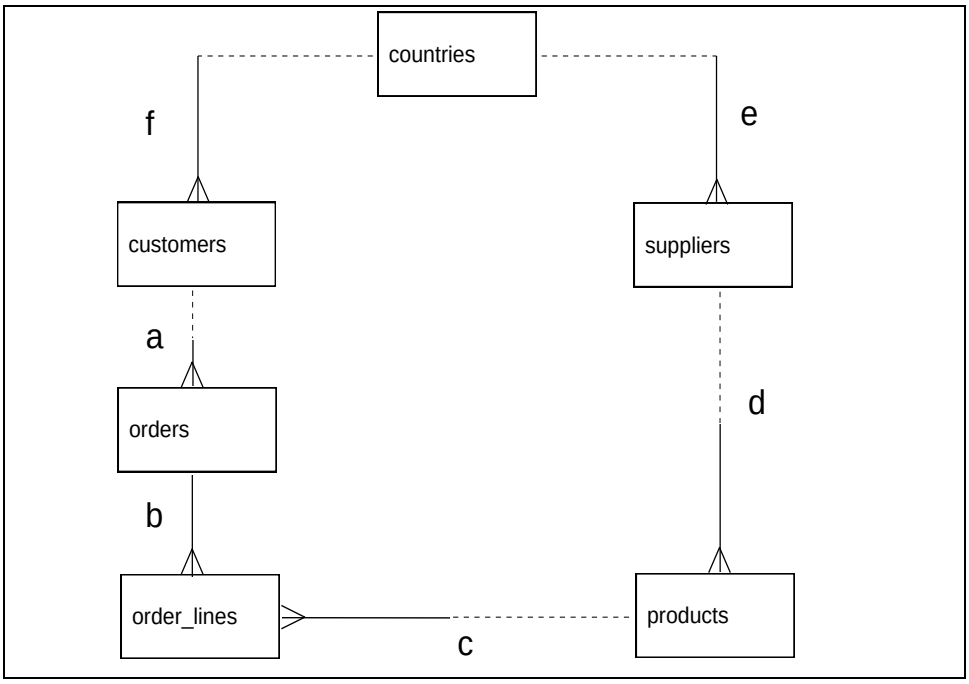
Resolve loops

Loops occur when there is more than one join path between two tables. The semantic layer query engine will not know which path to choose. Loops in the model must be resolved.

Tip: You can use the **Detect Loops** option in the data foundation to test for loops, but be aware that there are some situations where the problem may not be detected. If the loop involves only two tables, the information design tool will assume that the two joins are sub-components of a multi-column join rather than separate relationships. It will not detect the loop.

Consider the example shown in Figure 2, based on a simple order-entry system. The boxes represent tables, and the lines represent joins.

Figure 2

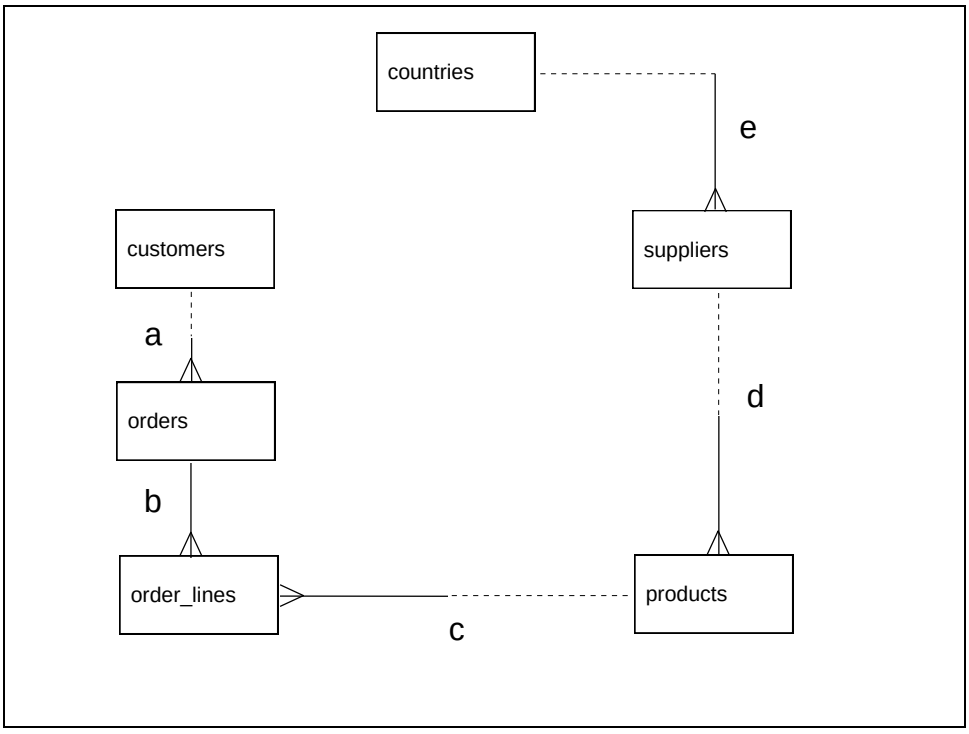


There are three possible solutions to resolving the loop:

1. Resolve the loop by eliminating a join, such that only one path exists between any remaining objects. Figure 3 shows a possible solution for the example loop. Join **f** between `countries` and `customers` has been deleted.

Tip: Be careful when removing a join. The joins that remain must produce query results that make sense to the users.

Figure 3



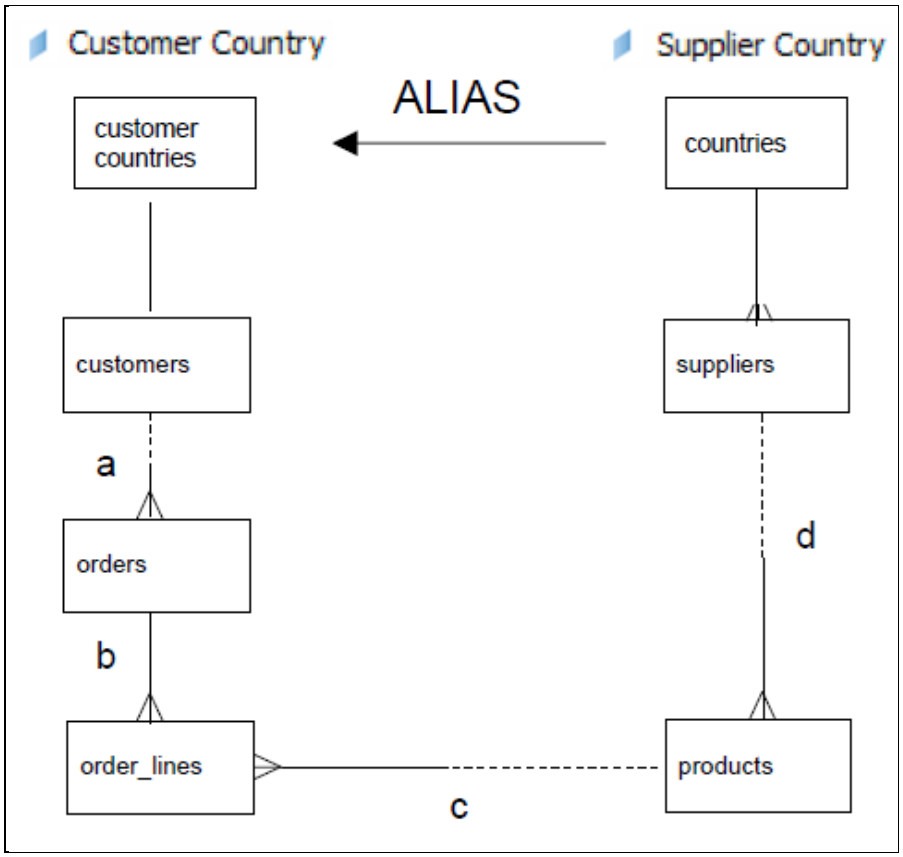
Tip: If the foreign key to countries from customers is made available as an object, and the primary key in countries is also made available as an object, users would have more than one way of retrieving the same information, each with a different meaning. The object names and the grouping of objects into classes must make it clear to the user what the result would be when using each object.

2. Resolve the loop with an alias. There may be times when you cannot resolve a loop by removing a join. In such cases, attempt to make use of an alias.

   In the solution shown in Figure 3, country information is only available in the context of a supplier. What if country information is also needed in the context of customers? If so, the solution does not support the business needs. An alternative solution involves using an alias for the `countries` table as shown in Figure 4.

   An alias is an SQL construct designed to allow multiple instances of a table in a data foundation.

Figure 4

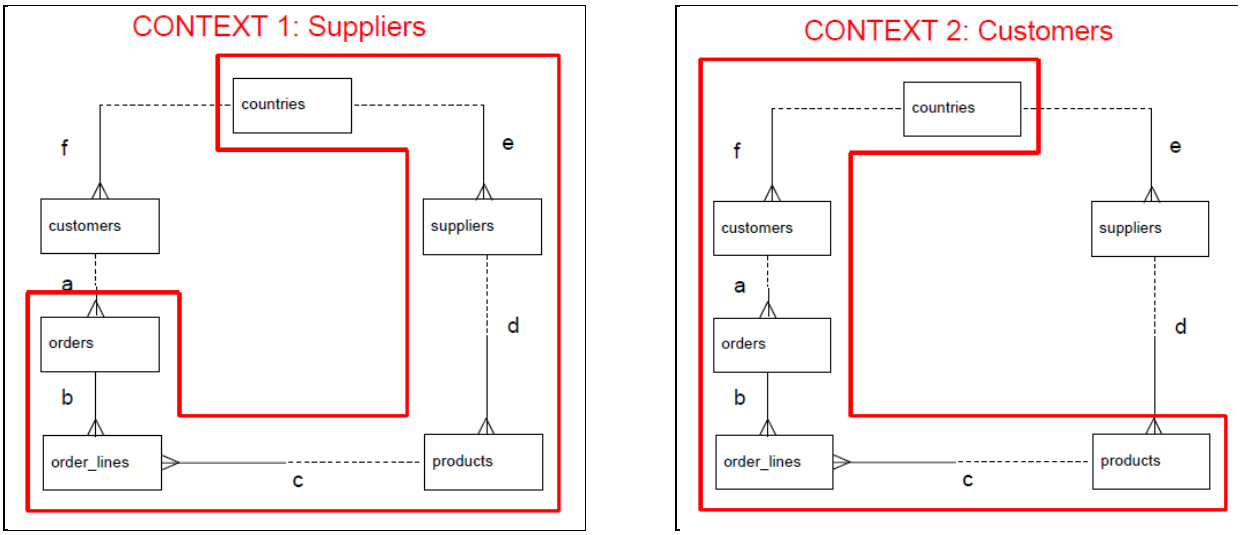


Tip: Where there are loops, look for a table that is on one side of multiple one-to-many relationships (for example, the countries table in Figure 2). This table may be a good candidate for creating an alias. In most cases it does not make sense to use the table in conjunction with both the related tables; such a query would produce a Cartesian product of the related rows.

3. Resolve the loops with contexts. A context is a collection of joins which provides a valid query path. Use contexts to resolve loops in the data foundation when the loop cannot be resolved by creating an alias table. In the information design tool, you define a context to resolve one join path in the loop by explicitly defining a join path. The user is prompted for the context to use at query time.

   Figure 5 shows the definition of two contexts. Each context determines a way to solve the loop ambiguity. At query time, the user selects a context, and the context defines the joins that are included to generate the appropriate SQL query.

Figure 5



Optimize queries on large tables using the BOUNDARY_WEIGHT_TABLE parameter

Large tables in the FROM clause have to be wholly parsed even if they are filtered by a condition afterward.

By setting the BOUNDARY_WEIGHT_TABLE SQL generation parameter, you can define a limit (in number of rows) to avoid this full parse.

If the table size in rows is greater than the value entered for BOUNDARY_WEIGHT_TABLE, the table is declared as a subquery in the SQL query:

```
FROM (SELECT col1, col2,...., coln FROM Table_Name WHERE simple
condition)
```

In the statement above, only the columns necessary for the query are retrieved and the condition in the subquery limits the number of records to parse.

Merge joins using the JOIN_BY_SQL parameter

When requesting measures in two different contexts (two different fact tables sharing the same dimensions), the semantic layer generates two SQL statements and joins the results after the data is returned from the database.

You can force the two result sets to be joined at the database level. Forcing the join in the database takes advantage of the calculation power of the database, which will likely improve performance.

To enable this feature, set the SQL generation parameter JOIN_BY_SQL to Yes.

BEGIN_SQL parameter

The BEGIN_SQL parameter lets you prefix SQL statements with the same parameters each time an SQL statement is generated. For example, this parameter can be used in some databases to do the following:

- Set the query priority
- Limit the number of rows fetched in a query

Universe variables can be used with this parameter:

```
/* ' User: '@Variable('BOUSER'), 'Dimension: @Variable('UNVNAME') */
```

END_SQL parameter

The END_SQL parameter lets you append a string to the SQL sent to the database each time a query is run. For example, this parameter can be used in some databases to do the following:

- Track database server activity: trace who is running queries in which universes
- Query banding and auditing
- Accelerating performance on SAP Hana (available with HANA 1.0 SP03):

```
WITH PARAMETERS ('request_flags'='USE_PARALLEL_AGGREGATION',
'request_flags'='ANALYZE_MODEL')
```

Universe variables can be used with this parameter:

```
/* ' User: '@Variable('BOUSER'), 'Dimension: @Variable('UNVNAME') */
```

Use derived tables to push calculations to the database

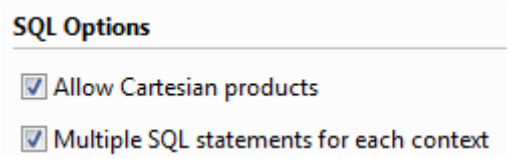
Some filtering and calculations that are done at the universe or report level can be pushed down to the database level without the need to modify the database.

This has two advantages: Less work for users because they do not need to create the calculation in the report or query; improved performance because the database calculation engine is more likely to retrieve data faster and less data is transferred.

Use derived tables to push down calculations to the database level. A derived table combines other tables using calculations and functions, without actually creating a new database table. You can create objects in the business layer on a derived table in the same way that you do for a standard table.

At query time, if such tables are needed, the SQL sentence defining them will be added to the query.

Check the SQL Options in the data foundation properties



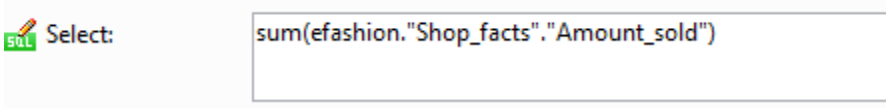
The **Allow Cartesian products** option is selected by default. This option allows end users to build a query containing objects belonging to tables that are not joined together either directly or indirectly. When retrieving data from the two tables, all rows in the first table are joined to all rows of second table. The result is called a Cartesian product. Cartesian products can have a negative impact on performance because of the potentially large number of rows retrieved.

The **Multiple SQL statements for each context** option is selected by default. This option allows end users to build a query containing objects belonging to different contexts. A frequent use case is to select measures coming from tables that belong to separate contexts, for example, `customers` and `suppliers` in the context definition in Figure 5, above. If this option is selected then one SQL query per context will be generated and the query engine will try to merge the two result sets into a single one.

Business layer design recommendations

Set aggregation for measures

Edit the SQL expression for each measure in the business layer, and define an aggregation function, if needed. In general, most measures require an SQL aggregation, for example:



Note: When you create a relational business layer with the Automatically create classes and objects option selected (default), all objects are created in the business layer as dimensions. You need to edit the business layer and specify the measures explicitly using the Turn into Measure command.

Set the projection function for measures

The projection function is used by Web Intelligence to do local aggregation when further aggregation is needed to project data onto the report. This avoids unnecessary calls to the database. The projection function is used when users create aggregations, breaks, or sections in a Web Intelligence report that require further aggregation of measures.

When you create a measure, specify the way the aggregate function will be projected onto a report. The available projection functions are summarized in the table:

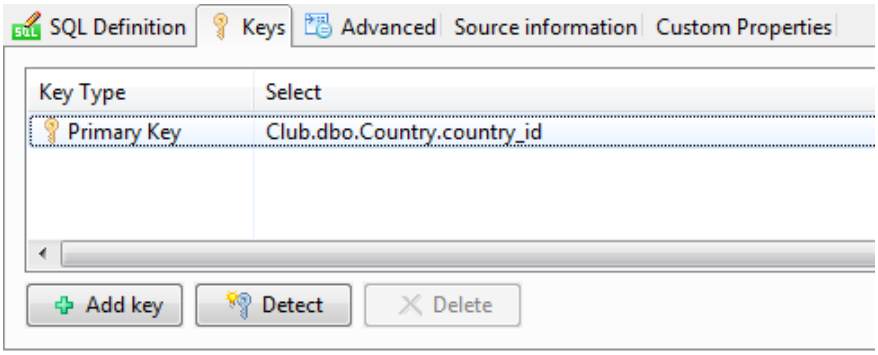
Projection function	Comments
NONE	No aggregation is done. Use this function if a measure has no SQL aggregation expression.
SUM	Use this function if a measure can be computed by a SUM in the report.
MIN	Same as SUM, but rarely used.
MAX	Same as SUM, but rarely used.
COUNT	Same as SUM, but rarely used.
AVERAGE	Same as SUM, but rarely used.
DELEGATED	Aggregations are done by the database. Use this function if the measure is non-additive (for example, a ratio, percentage, or average, weight). All user actions in a Web Intelligence document (drill operation, section, break, or report aggregation) query the database to retrieve aggregated results computed by the database engine.

Use index awareness

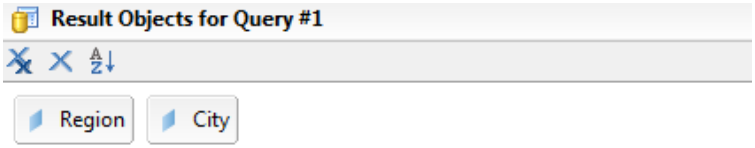
Index awareness takes advantage of the indexes on key columns to speed up data retrieval. Many business layer objects retrieve text-based data from database columns, such as names, that are meaningful to an end user. Most database tables have a primary key, usually a numeric value that is not meaningful to the end user, but is important for database performance. When you set up index awareness, you define which database columns are primary and foreign keys. The benefits of defining index awareness include the following:

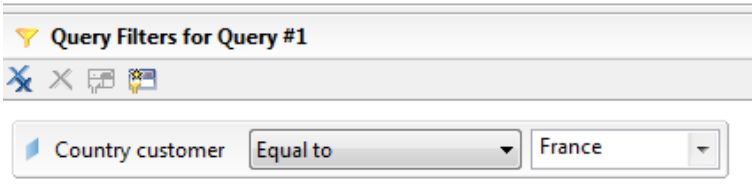
- Joins and filtering done on keys are faster than those done on non-key columns.
- Fewer joins are needed in a query, therefore fewer tables are requested.
- Guarantees uniqueness when used in filters or list of values (labels can be duplicated, keys can't)

Define index awareness by adding primary and foreign keys on the **Keys** tab in the business layer object properties:



The following example shows the difference in the SQL generated when using index awareness in a Country/Region/City when querying for Customer and Region, and filtering on Country=France.





The SQL generated without index awareness is as follows:

```
SELECT
Club.dbo.Region.region,
Club.dbo.City.city
FROM
Club.dbo.Country INNER JOIN Club.dbo.Region ON
Club.dbo.Region.country_id=Club.dbo.Country.country_id)
INNER JOIN Club.dbo.City ON (Club.dbo.City.region_id=Club.dbo.Region.region_id)
WHERE  Club.dbo.Country.country  = 'France'
```

The SQL generated with index awareness is as follows:

```
SELECT
Club.dbo.Region.region,
Club.dbo.City.city
FROM
 Club.dbo.Region INNER JOIN Club.dbo.City ON
(Club.dbo.City.region_id=Club.dbo.Region.region_id)
WHERE  Club.dbo.Country.country_id  = 2
```

Use aggregate awareness

Aggregate awareness is the ability of a universe to take advantage of database tables that contain pre-aggregated data (aggregate tables). When measures are pre-aggregated, query users can get more immediate response to high-level queries.

For an aggregate aware object, the query tries to get the data from the highest aggregation level and go to a more detailed level if the query is asking for detailed dimensions.

Aggregate aware techniques accelerate queries by processing fewer facts and aggregating fewer rows.

For example, in a data model there is a fact table for sales with detail on the transaction level, and an aggregate table with sales per day. If the final user chooses to see sales details, then the transaction table is used. If the user wants to see sales per day, then the aggregate table is used. Which table is used is fully transparent to the user.

Implementing aggregate awareness in the universe has several steps:

1. The first step is done at the database level. The database administrator must define and load the aggregate tables in the database.

2. Insert the aggregate tables into the data foundation.

3. Define aggregate aware objects. These are objects in the business layer that you want use the aggregate tables where possible instead of performing aggregation using non-aggregate tables.

   In the SQL expression for the object, define the SELECT statement to use the @Aggregate_Aware function:

   ```
   @Aggregate_Aware(sum(aggr_table_1), …, sum(aggr_table_n))
   ```

   In the @Aggregate_Aware function, aggr_table_1 is the aggregate table with the highest level of aggregation, and aggr_table_n the aggregate table with the lowest level or the detailed fact table.

4. Specify the incompatible objects for each aggregate table in the universe. In the business layer, use the **Set Aggregate Navigation** command. For each aggregate table in the data foundation, specify which objects in the business layer are incompatible:

   - If the object is at the same level of aggregation or higher, it is COMPATIBLE with the aggregate table.
   - If the object is at a lower level of aggregation, it is INCOMPATIBLE.
   - If the object has nothing to do with the aggregate table, it is INCOMPATIBLE**.**

Tip: You can use the Detect Incompatibilities button to help you with this process.

5. Resolve any loops in the data foundation using contexts if needed.

@Functions

For information, here is the list of functions that can be used in SQL or MDX expressions.

- **@Prompt**

  Asks the user for a value. This functions can be used in many places, most often in WHERE clauses.

- **@Variable**

  Returns the value of a prompt or of a variable (BOUser, Locale…).

- **@Select**

  Returns the SELECT clause of an object.

- **@Where**

  Returns the WHERE clause of an object.

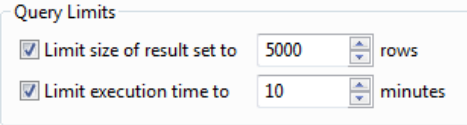
- **@DerivedTable**

  Returns the SQL expression of a derived table.

- **@Aggregate_Aware**

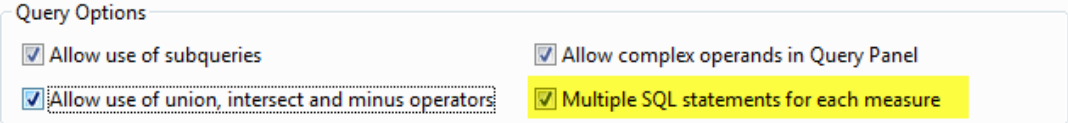
  Directs an object to query first of all the aggregate tables listed as its parameters. If the aggregate tables are not appropriate, then the query runs with the original aggregate based on the non-aggregated table

Check the query limits and options in the business layer properties



The default values for **Query Limits** are 5000 rows and 10 minutes. You can increase, decrease, or remove the query limits.

It is important to understand that incorrect results can be displayed in the documents if not all rows have been retrieved from the data source.



The **Multiple SQL Statements for each measure** option is selected by default. When selected, one SQL query is generated for each measure or group of measures belonging to a different fact table, or for measures having a WHERE clause filled (filtered measure). If the measure objects are based on columns in the same table, then the separate SQL queries are not generated, even if this option is selected.

For example, you can set a WHERE clause for different measures in order to return different aggregated data. These measures are filtered measures:

- Sales revenue for 2010: Time.Year = 2010
- Sales revenue for 2011: Time.Year = 2011

In this example, one query for each filtered measure is generated and the query engine tries to merge the two result sets into a single one.

Another example of when you want to generate a separate query for each group of measures in the same table is to avoid extra joins that can have a huge impact on database performance.

Related Content

Semantic Layer forum on SDN: http://forums.sdn.sap.com/forum.jspa?forumID=308

Information design tool tutorials: http://www.sdn.sap.com/irj/scn/info-design-tool-elearning?refer=main

Conversion of the relational universes in SAP BusinessObjects BI platform 4.0, from UNV to UNX: http://wiki.sdn.sap.com/wiki/display/BOBJ/Conversion+of+the+relational+universes+in+BI+4.x%2C+from+UNV+to+UNX

Idea place, SAP BusinessObjects Semantic Layer (including Universe Designer): https://cw.sdn.sap.com/cw/community/ideas/businessanalytics/sbosemanticlayer

For more information, visit the Business Objects homepage

Copyright