1 OVERVIEW

This document is the second part of a series of articles about how to achieve the best performance while working with the Denodo Platform. These articles will provide best practices on a variety of different topics such as the way you should build your model, the different configuration settings you should use and the caching method that best suits your scenario. This guidance is especially important in analytic use cases which require integrating large volumes of data from different sources.

Like the previous article, this document is aimed at Data Architects and Developers interested in optimizing the use of Denodo for analytic scenarios. In the first one, Modeling Big Data and Analytic Use Cases, you can find general guidelines on how to design views that are optimal from a performance point of view. In this one we describe all the additional meta-information and configuration settings that are relevant for the Query Optimizer module.
2 THE QUERY OPTIMIZER MODULE

VQL, like SQL, is a non-procedural language. This means that it does not enforce the database server what is the list of steps to follow to obtain the desired result. Instead, it is the database the one in charge of deciding what is the best plan to execute a particular query.

For this purpose, the query optimizer is a key component of Denodo. This module explores the different but equivalent execution plans that can obtain the result for a certain query, and decides which one would achieve the best performance.

The query optimization process consists of two different phases:

- The **static optimization phase**: In this phase the optimizer analyzes the query and performs those query rewritings that improve the performance regardless of the specific data we need to manage. These transformations include removing unnecessary JOINs or UNION partitions (branch pruning) or pushing conditions and group bys under Joins and Unions.

- The **cost-based or dynamic optimization phase**: In this phase the query optimizer analyzes the execution plan returned by the static optimization phase and estimates the costs of different possible plans. The alternatives considered include, among other variables: the different join strategies to use for each join (MERGE, HASH, NESTED) or moving partial results from some of the operations to another data source or to a Massive Parallel Processing system.

In analytic scenarios, the query optimizer module is critical as the execution times of two equivalent plans can differ in orders of magnitude. Therefore, it's essential to make sure the optimizer is enabled and all the information that can be relevant for this analysis is available and it is accurate. This information includes:

- Primary keys
- Indexes
- Referential constraints
- View statistics
- Data movement targets
- MPP acceleration data source

The following sections will describe the key concepts you need to consider for each one and in the article Denodo Query Optimizations for the Logical Data Warehouse you can find a step-by-step tutorial that will guide you in this configuration process for a sample scenario.
3 PRIMARY KEYS AND INDEXES

3.1 THEIR ROLE IN THE OPTIMIZATION PROCESS

During the static optimization phase, primary keys, as well as unique indexes, let Denodo know what fields are unique in a view, and this information is extremely powerful specially when those fields are used in a JOIN condition, since in that case the optimizer can be sure that the join operation will not duplicate rows from the other table.

This is a necessary condition in order to apply two important rewritings, Join pruning and Aggregation push-down:

- **Join pruning** is especially important in single-view scenarios and in general when the fact or the dimension tables are partitioned vertically (see section 'Vertically Partitioned Views' in the previous article of this series).

- **Aggregation push-down** is essential in queries asking for aggregated data over star schemas, as it avoids the movement of large amounts of data through the network, and therefore it drastically reduces the execution times.

Example:
Let’s say we have information about 200M sales in a data warehouse and information about 2M customers in a CRM and we want to obtain the total sales by each preferred customer with some additional information about the customer:

```
SELECT c_customer_id, c_first_name, c_last_name, c_email_address, ca_county,
       SUM(s.ss_sales_price)
FROM store_sales s INNER JOIN customer c ON (s.ss_customer_id = c.c_customer_id)
WHERE c_preferred_cust_flag = 1
GROUP BY c_customer_id, c_first_name, c_last_name, c_email_address, ca_county
```
Since the join is using a primary key, the query optimizer can push the whole aggregation to SALES, and therefore Denodo will just retrieve one row for each customer instead of all sales information (2M rows vs 200M).

During the **cost-based optimization phase**:
- primary keys are taken into account to estimate the number of rows returned by joins, group bys, selections and distinct operations.
- Indexes are taken into account to estimate if an execution plan will use that index or not and therefore this can affect dramatically the cost estimation.

### 3.2 CONFIGURING PRIMARY KEYS

In Denodo you can specify the primary key for base views and derived views and this information is not propagated between views.

#### 3.2.1 In which views should I set a primary key?

**3.2.1.1 Base views**

You should define the primary key for all **base views**:
- For those base views coming from databases accessed through JDBC or ODBC, Denodo will automatically retrieve their PKs information.
- For those base views that belong to other kind of data sources, you should set the primary key manually.

**3.2.1.2 Derived views**

You should define the primary keys for all derived views that participate in joins. If a derived view is built using joins or unions it may not be possible for the optimizer to know if the fields used in a join condition are unique just looking at the information on
the base views. For this reason we recommend to set a primary key in the derived view over those fields that you will potentially use in joins if you know they are unique.

In a star schema scenario, this means that you should define a primary key on those views modeling a dimension or a fact table. For example, let’s say that following the recommendations for vertical partitions, we have built a customer dimension view as the join between by_customer_dim and by_customer_address.

This way we offer one view with the complete information from customers including the details of their current address, and we will use this derived view in all queries requiring data about customers.

Now, if we want to get the total sales by zip_code we could run a query like:

```
SELECT  c.ca_county, c.ca_state, c.ca_zip, SUM(s.ss_sales_price)
FROM store_sales s JOIN customer c ON(s.ss_customer_id = c.c_customer_id)
GROUP BY c.ca_county, c.ca_state, c.ca_zip
```

As we know that c_customer_id will still be unique in this new view (as each customer only has one current address), we should edit this new view customer and set c_customer_id as primary key.

Visit the section Primary keys of views in the Denodo Virtual DataPort Administration Guide for more details.

### 3.2.2 Natural key vs surrogate key

In cases where a table has a natural key but the primary key is a surrogate key, we recommend leaving the original PK from the source on the base views (the surrogate key) to simplify the governance of the model, as base views should mirror what you have in the source.

However, in multi-source data models it can be common to use the natural key for joins between views from different data sources, and use the surrogate key only for the joins between views from the same data source.
For example, let’s say that our customer table has a surrogate key c_customer_sk and a natural key c_customer_id, which represents a customer code recognized company wide.

If we want to join customer and customer_address, both in the same ERP system, we need to use the surrogate key, but if we want to join customer with sales, that is in another system, we may need to use the natural key.

To ensure that the optimizations are applied in either case, we need to model this in a way the optimizer can know that both fields are unique. Currently, Denodo does not have support for unique constraints, so in these scenarios we recommend the following:

● Leave the surrogate key as the primary key on the base views
● Create a new view over the base view assigning the natural key as the primary key. This will also make the model clearer from a semantic point of view:
  ○ It will be easier for developers to know what fields should they use in multi-source joins.
  ○ It will be easier for end users to discover related views using the data catalog.

In a scenario where our semantic view is built from several tables, like the JOIN between customer and address from our previous example, set the primary key on the final view as the natural key, as in this case this will be the semantic view that we will use in multi-source operations.

● Define a UNIQUE index on the surrogate key if it doesn’t already exist:
  Take into account that if this surrogate key will be used in join operations, it’s highly recommended to have an index in the physical database. Therefore, in most cases this index will already exist (depending on the data source it may not be an index but a similar structure, check the Configuring indexes section for more details). This way, the optimizer will know the natural key is unique due to the primary key and it will know the surrogate key is unique thanks to the unique index.

NOTE: Keep in mind that when you specify an index in a base view Denodo does not actually create an index in the data source (except for the cache indexes). You are just adding metadata so that Denodo "knows" about that index.
3.3 CONFIGURING INDEXES

In Denodo, base views include the information about what indexes are available in the data source. The VDP server automatically introspects the available indexes and their types for most databases, but the administrator may need to modify the information in specific cases:

- The information provided by the driver is not complete or accurate (Oracle, MySQL, Informix, Teradata, Greenplum, Hive, SAP HANA),
- The concept of index does not exist in the database but it has similar structures that can be considered as indexes for cost estimation purposes (Netezza, Redshift)
- The data source does not have indexes but it is acting as a front-end for another database that actually has indexes (REST/SOAP Web Services or SAP BAPIs)

Visit the section **Specifying Indexes** in the Denodo Virtual DataPort Administration Guide for more details.
4 REFERENTIAL CONSTRAINTS

In Denodo you can model relationships between views creating associations. Associations are useful for different reasons (for example they allow users to navigate between views using a RESTful Web service and discover related views using the Data Catalog), but in this document we will focus on their role during the optimization process. To know more about associations check the article ‘Associations in Denodo’ in our Knowledge Base.

An association is also a referential constraint if each row from the dependent side always reference one (and only one) row from the principal table.

For example, an association from SALES to ITEM using the item_id field is a referential constraint as every row in sales reference a specific product in ITEM.

On the other hand, let’s say we have a dimension BOOK that only contains information about book items. An association from SALES to BOOK is not a referential constraint if SALES contains information about purchases of any kind of products, not only books. In this case some of the rows from SALES do not reference any row in BOOKS.

NOTE: It is important to make sure that only the foreign key - primary key relationships are marked as referential constraints. Otherwise the optimizer may do query rewritings based on this information that could not be equivalent to the original one and therefore it could lead to incorrect results.

4.1 THEIR ROLE IN THE OPTIMIZATION PROCESS

If there is a referential constraint between two tables, like SALES and ITEM, and the cardinality for the principal side is 1, the optimizer knows that a Join operation between those two tables will not filter any of the rows from SALES, as the referential constraint ensures every sale will match with one item.

This allows the optimizer to remove unnecessary joins from queries.

For example, let’s say we are using the extended customer view from previous examples that is built as a JOIN between customer and address, and let’s say that we want to obtain the total sales by customer age:

```
SELECT d.d_year - c.c_birth_year as age, SUM(s.ss_sales_price) 
FROM store_sales s INNER JOIN bv_date_dim d ON(s.ss_sold_date_sk = d.d_date_sk) 
INNER JOIN customer c ON(s.ss_customer_id = c.c_customer_id) 
GROUP BY 1
```

This query will perform the JOIN with customer_address although it’s easy to see that is not necessary for this particular query.

If we define a referential constraint between bv_customer_dim and bv_customer_address, the query optimizer can remove that JOIN because:
The only columns we need from customer is the id and the birth year, both from bv_customer_dim.
The referential constraint ensures that the number of rows with and without the JOIN is the same, as it does not filter or duplicate any rows.

4.2 BETWEEN WHICH VIEWS SHOULD I CREATE ASSOCIATIONS?

Our recommendation is the following:

- Create referential constraints between those **base views with relationships from the same data source**. If those views had a foreign key in the source, you can import that information:
  - select all the JDBC base views for which you want to create its associations
  - right-click one of the views and click Discover associations.
- Create referential constraints between the **semantic views with FK-PK relationships from different data sources**. In a star schema scenario this means:
  - Create a referential constraint between the related views that make up a semantic dimension or facts table that is partitioned vertically (like in the customer example).
  - Create a referential constraint between each facts table and the dimensions it is related to (for example between SALES and customer).

Visit the section **Creating an association** of the Denodo Virtual DataPort Administration Guide for more details.
5 VIEW STATISTICS

5.1 THEIR ROLE IN THE OPTIMIZATION PROCESS

The cost-based optimizer estimates the cost of different possible execution plans and choose the one for which the estimated cost is minimum. Base on this, it can make decisions like: which is the best join strategy to use, if it’s better to push a partial aggregation under a join or transfer part of the data to another data source and delegate more operations to that data source.

In order to make these estimations it needs some statistics from the views like the number of rows and the number of distinct values for each column.

5.2 WHAT VIEWS REQUIRE STATISTICS?

You should gather statistics for:

- All the base views
- Derived views that have cache enabled
- Flatten views

In most cases Denodo can gather the statistics automatically. In the cases where it is not possible (e.g. web services with mandatory input parameters or views with complex types like arrays) you can manually enter the estimated values.

To know if the query optimizer will apply cost-based optimizations for a certain query you can use the command DESC QUERYPLAN <query> and check the execution plan. Example:

```
DESC QUERYPLAN SELECT c_customer_id, c_first_name, c_last_name, c_email_address, ca_county, SUM(s.ss_sales_price) FROM store_sales s INNER JOIN customer c ON(s.ss_customer_id = c.c_customer_id) WHERE c_preferred_cust_flag = 1 GROUP BY c_customer_id, c_first_name, c_last_name, c_email_address, ca_county
```
In this case you can see that it could not apply cost-based optimizations because one of the views does not have statistics.

Denodo does not need exact values for the statistics to do a good job making cost-based decisions. What it really matters is that the statistics are complete and they don’t have missing values for:

- **Number of total rows of the view.**
- **average size and number of distinct values** for all the fields that are used in a query (more specifically, in the select and where clauses as well as in join conditions and group by fields). Note that a field might not be referenced in the actual query performed, but may be required in a lower view.

These values are crucial to achieve good estimations of the number of rows on each operation, the required number of I/O operations and the network transfer cost. Therefore, make sure there is always an approximate value for these fields.

NOTE: The cost-based optimizer is very powerful when the statistics are complete. However, estimations based on wrong or incomplete statistics can lead to very inefficient execution plans. Therefore, in cases where it is not possible to provide reliable statistics it is better to disable the cost-based optimizer. Keep in mind that many optimizations like branch pruning or aggregation push-down will still apply in the absence of statistics and the query optimizer will make decisions based on heuristics that obtain good results in most cases.

### 5.3 CACHED VIEWS

When you gather statistics from a view with FULL cache enabled it will access the cache database if possible, and therefore the statistics corresponds to the data in cache.

In these cases, make sure you gather the statistics once the cache has been loaded and you clean your cache frequently enough to avoid keeping a lot of invalid tuples in the cache (See section **Cache Maintenance Task** of the Denodo Virtual DataPort...
Having a lot of invalid rows would slow down the operations in cache and the statistics would not mirror the data considering the invalid rows.

NOTE: Denodo can obtain the statistics from the cache database only if it has been configured specifying a custom CATALOG/SCHEMA (See section Configuring the Cache of the Denodo Virtual DataPort Administration Guide)

For more information about view statistics check the section ‘Collecting Statistics’ in the Knowledge base article Denodo Query Optimizations for the Logical Data Warehouse and the section Gathering the Statistics of Views of the Denodo Virtual DataPort Administration Guide.
6 DATA MOVEMENT CONFIGURATION

The Data Movement optimization can execute some federated queries more efficiently. When there is a Join/Minus/Intersect/Union operation where one of the views is much larger than the others, Denodo can transfer the data of the smaller views into the data source of the larger view and this way execute the entire operation in this data source. This may offer great performance improvements for some queries compared with pulling the data from both sources and having the Denodo server process the results.

For example, let’s say we want the total sales from 18-year-old customers and we use the following query:

```
SELECT SUM(s.ss_sales_price)
FROM store_sales s JOIN customer c ON(s.ss_customer_id = c.c_customer_id)
JOIN bv_date_dim d ON(s.ss_sold_date_sk = d.d_date_sk)
WHERE d.d_year - c.c_birth_year = 18
```

As `bv_date_dim` and `customer` belong to different data sources, it’s not possible to push the condition `WHERE d.d_year - c.c_birth_year = 18` to any of the two systems. Without data movement enabled, the query optimizer would choose the following plan:

1. Push a partial aggregation to the data warehouse to obtain the total sales by customer and year.
2. Transfer to Denodo:
   - the result of this partial aggregation and,
   - the id and birth year from customer.
3. Join those tables on the virtualization layer
4. Filter all the sales except the ones from 18-year-old customers.
5. Aggregate the remaining data to obtain the total sales

This means: (1) reading and transferring 2M rows from customer and (2) let's say 5M (year, customer_id) groups from the data warehouse and (3) perform 2 aggregations, one in the data warehouse and one in Denodo.

If the data source corresponding to the data warehouse is configured as a possible target for data movement, the cost-based optimizer would choose a better execution plan instead:

1. Create a temporary table in the data warehouse and insert there the data from customer.
2. Process the full query in the data warehouse.

In this case we just need to (1) read and transfer 2M rows from customer, (2) move this 2M rows to the temporary table and (3) perform one aggregation in the data warehouse.

Keep in mind that by default, Denodo will create the temporary table on the schema specified in the connection details of the data source, but we recommend specifying a different schema for these temporary tables, as shown in the figure below. Take into account that the user specified in the connection will need enough privileges to create tables in the selected schema.
Visit the section ‘Declaring Data Movement Targets’ in the Knowledge base article [Denodo Query Optimizations for the Logical Data Warehouse](#) and the section [Data movement](#) of the Denodo Virtual DataPort Administration Guide for configuration details.

### 6.1 DATA MOVEMENT TO PARTITIONED UNIONS

When the facts data is distributed in more than one system, the data movement technique can also create temporary tables on each system and then insert the data in parallel to maximize delegation.

For example, let’s say we have the sales information in two different systems:

- One data warehouse system containing ~200M sales from physical stores.
A Hadoop-based MPP database like Impala with information about ~8M online sales.

And we have built our sales view as the UNION of both sales tables:

The date dimension is replicated on each system, so we have defined an alternative source (See the Alternative Sources section):

Now, let's say we want to obtain the total sales by year and month for the last 5 years and we use the following query:

```
SELECT d_year, d_moy, SUM(s.ss_sales_price)
FROM sales s INNER JOIN bv_date_dim d ON(s.ss_sold_date_sk = d.d_date_sk)
INNER JOIN item c ON(s.ss_item_sk = c.i_item_id)
WHERE d_year > getyear(addyear(now(), -5))
GROUP BY d_year, d_moy
```
The option the query optimizer choose in this case is:

- Moving the data from item to a temporary table on each data source (Data movement),
- Pushing the JOIN with item to each database (Union-Join Push-down), and,
- Pushing the partial aggregation to each database (Aggregation Push-down).

With this strategy, (1) each data warehouse calculates the total by year and month and (2) those partial results are transfer to Denodo that (3) will do the final aggregation to obtain the result.

This way the data transfer is just 2M rows (transferred in parallel) during the data movement operation plus 60 (5 years x 12 months each) coming from each source.
7 MPP ACCELERATION

Since version 7.0, the cost-based optimizer has a new possible alternative to consider: If you have a SQL-on Hadoop system like Impala, Presto or Spark, you can import it in Denodo as any other data source and allow the query optimizer to create temporary tables to accelerate some queries.

The data movement from the previous section only applies to Join/Minus/Intersect/Union operations and the temporary data is stored in one of the data sources participating in those operations. On the other hand, the MPP acceleration applies wherever Denodo detects that it would have to perform costly operations on large amounts of data. Therefore, it can apply to aggregations, some Joins, DISTINCT and ORDER BY operations.

For example, let’s say we want to obtain the customer information for those customers that spent more than 1000 dollars in purchases during a certain time period and order the result by certain columns:

```sql
SELECT c_first_name, c_last_name, c_email_address, ca_street_name, ca_street_number, ca_city, ca_zip, ca_state, ca_county
FROM store_sales s JOIN customer c ON(s.ss_customer_id = c.c_customer_sk)
JOIN bv_date_dim d ON(s.ss_sold_date_sk = d.d_date_sk)
WHERE d.d_year between 2000 and 2003
GROUP BY c_first_name, c_last_name, c_email_address, ca_street_name, ca_street_number, ca_city, ca_zip, ca_state, ca_county
HAVING SUM(s.ss_sales_price) > 1000
ORDER BY ca_county, ca_state, ca_zip, ca_city
```

The option the query optimizer choose in this case is:
- Push a partial aggregation to get the total sales by customer
- Perform the join and the final aggregation in the virtualization layer
- Order the result
Even if the data source performs a partial aggregation, the final aggregation in Denodo will be costly. This is because in order to calculate the total operation, it's necessary to hold the partial sum for each group in memory until all the rows have been processed. If the number of groups is high like in this case (2 million customers), it will reach the memory limit set on the server and therefore it will change the aggregation strategy: it will first save the rows on disk ordered by the group by fields and then perform a sorted aggregation holding just one group at a time. This will avoid monopolizing the system resources at the expense of slowing down the aggregation because of the IO operations to disk.

Finally, the order by operation will force Denodo to swap data to disk again for the same reason.

If we have a Hadoop-based MPP database, we can configure it for query acceleration. This requires two steps:

- Edit the data source to allow the optimizer to create temporary tables for query acceleration
● Enable the MPP acceleration and select this data source as the one to use (Administration -> Server Configuration -> Queries optimization):

If we run the previous query with this configuration, the query optimizer will choose the following plan:

- Push a partial aggregation to the data warehouse containing the sales
- Obtain the information about customers from the CRM
● Combine the data using a MERGE JOIN
● Transfer the join result (2M) to the MPP data source
● Let the MPP perform the final aggregation and order by

Compared to the previous plan, it would have the extra cost of transfer and insert 2M million rows but then the MPP will perform the costly aggregation and order by in parallel so the overall performance will be better.

For more details about the MPP query acceleration check the document ‘How to configure MPP Query Acceleration in Denodo’