Denodo Query Optimizations for the Logical Data Warehouse

Revision 20180524

NOTE
This document is confidential and proprietary of Denodo Technologies. No part of this document may be reproduced in any form by any means without prior written authorization of Denodo Technologies.
1 OVERVIEW

This document is aimed at Data Architects and Developers interested in optimizing the use of Denodo for scenarios which require integrating data sources containing large volumes of data. This is just one case of Logical Data Warehouse and Logical Data Lake scenarios.

The document includes:
- A general explanation of the context and the optimization techniques applied by Denodo.
- A step-by-step tutorial on how to configure Denodo to automatically apply those optimizations.

This is the first part of a series of articles about Denodo Optimizations for Logical Data Warehouses. The second part deals with optimizations for scenarios where fact tables are partitioned across several systems, and can be found here.

2 INTRODUCTION

When used in Logical Data Warehouse (LDWH) and/or Logical Data Lake architectures, Denodo provides unified query access and data governance on top of several data sources (see Figure 1). These data sources typically include one or several physical data warehouses such as a Hadoop Cluster, SaaS applications and additional databases.
The appeal of the Logical Data Lake and Logical Data Warehouse comes from the huge time and cost savings obtained in comparison to the “physical” approach, where all data needs to be previously copied to a single system. Replicating data means more hardware costs, more software licenses, more ETL flows to build and maintain, more data inconsistencies and more data governance costs. It also means a much higher time to market. In turn, the logical approaches leverage the existing systems while minimizing or removing the need for further replication.

Nevertheless, in these logical approaches, the data virtualization server needs to execute distributed queries across several systems. An initial thought may be that, in such a data intensive situation, Data Virtualization requires moving big amounts of data through the network to integrate them and that it will be slow. However, the reality is that these scenarios require processing large amounts of data, but not necessarily transferring them. Actually, the execution of the queries can usually be organized in such a way that each data source performs most of the work locally with its own data, and Denodo only needs to combine the partial results from each data source to generate the final results. Therefore, even complex distributed queries that need to process billions of rows can usually be resolved by moving very little data through the network, provided the right optimization techniques are used.

The Denodo optimizer uses a variety of techniques for this purpose including query delegation, query rewriting, advanced join optimization, ‘on-the-fly’ data movement and parallel query processing. In the following subsection an example of a Logical Data Warehouse scenario is presented and some of the main optimization techniques are explained using this example. In the part 2 of this series, a step-by-step tutorial describes how to configure Denodo to ensure these optimizations are applied.
2.1 LOGICAL DATA WAREHOUSE SCENARIO

Let’s imagine a big retailer sells around 10 thousand products per day. For each product, a conventional relational database stores its ‘identifier’, ‘product name’, ‘product category’ and ‘list price’. Let’s say the retailer also operates a Data Warehouse with sales information. For each sale, the Warehouse stores the ‘identifier’ of the sold product, the ‘sale date’ and the ‘sale price’. The Data Warehouse stores only sales from the last year, and contains approximately 200 million rows. Sales data from previous years is offloaded to a Hadoop-based repository running Hive which currently stores around 1 Billion historical sales. The data fields stored in Hive for each sale are the same as in the Data Warehouse.

With that setup, let’s suppose the business users want to run a report to obtain the total amount of sales of each product in the last two years. This requires integrating product data with sales information from last year from the Data Warehouse and the year before that from the Hadoop-based system. Please note that the size of the final report is only ten thousand rows; one row for each product. In this case, a naive execution plan (see Figure 2) would involve pushing the filtering conditions, sales in the last two years, down to the data sources, retrieving all the rows verifying those conditions, and then computing the report. Although pushing-down the filter conditions retrieves “only” 150 million rows instead of the total of one billion rows from Hadoop, the overall plan still requires transferring ~350 million rows and will be painfully slow in this hypothetical situation.

![Figure 2 Naive execution plan.](image-url)
Instead of this naive execution plan, Denodo can automatically apply a more sophisticated execution plan to answer the same query (see Figure 3):

1. It pushes down a query to the Data Warehouse to compute the total sales by product in the last year. This query returns one row for each product (10 thousand rows).
2. In parallel, it also pushes down a query to the Hadoop cluster to compute the total sales by product in the year previous to the last one. This will also returns one row for each product (10 thousand rows).
3. For each product, the Data Virtualization server only has to add the partial results from steps 1 and 2 together in order to obtain the total sales by product.
4. In parallel with those calculations, the Data Virtualization server also queries the ‘Products’ table in the conventional relational database to obtain the additional data needed for each product (e.g. product name, product category...) and merges them with the results from step 3.

![Query plan with optimization.](image)

In sum, each data source computes the total amount of sales by product on its own data, and transfers the partial results to Denodo. Then, Denodo computes the final result for each product by adding the partial sales amount computed by each data source. This reduces data transfer from ~350 million rows to ~20 thousand. This is 17,500x less or about 4 orders of magnitude less data transferred. This volume of records can be sent through modern Gigabit local networks in sub-second times, making the network overhead of the query insignificant.

Intelligent query rewritings like the one shown in this example are only one among many optimization techniques available in Denodo that can be used for virtualizing data-intensive scenarios. This particular example uses what is called **full aggregation pushdown**.
In this post from the Data Virtualization blog, an overview of other optimization techniques that Denodo applies can be found. The overall goal of these optimizations is to push down processing to the sources, in order to reduce the amount of data that must be moved across the network to satisfy the request, and to leverage the power of the underlying data sources (for instance, parallel databases and Hadoop-based databases are specialized in processing large aggregation queries).

3 TUTORIAL

In this part of the article, a step-by-step tutorial is provided on how to set up a sample Logical Data Warehouse Scenario and configure Denodo appropriately for it. The tutorial is divided into the following sections:

1. Preparing the datasets: the first step is getting the datasets and loading them into the different data sources used for the example.
2. Preparing Denodo: in this step the Denodo views are configured to ensure they can use the relevant optimizations.
3. Executing the queries: some sample queries will be executed to illustrate several scenarios and the optimizations applied in each one.
4. Single-View approach: an alternative way for modelling the same scenario will be introduced, where all dimensions and facts are joined in the same virtual view (single join view). This approach is often useful because it makes report creation easier in some self-service BI tools.

3.1 PREPARING THE DATA SETS

For this example, the datasets from TPC-DS will be used, TPC-DS is the most important standard industry benchmark for decision support systems. The TPC-DS schema models the sales and sales returns processes of an organization that employs three primary sales channels: store, catalogs and the Internet. The schema includes 7 facts tables and 17 dimension tables, but in this tutorial, only the following ones will be used (see Figure 4):

- Dimension tables: customer (2,000,000 rows), item (204,000 rows).

**NOTE:** TPC-DS can be scaled to different data sizes. The number of rows shown assume TPC-DS has been scaled to a total size of 1 Gbyte.

After registering in order to download the TPC-DS data generation tool, called DSGen, the datasets can be generated by following the instructions from the How_To_Guide.doc (under the /tools folder). The data files need to be loaded in the databases to use:

- Facts table: for this example, the store_sales table will be stored into Netezza\(^1\), as it is a parallel database widely used to implement data warehouses. Let’s name it SALES_DB. Following Netezza’s best practices, in addition to loading the data:
  - Statistics for the table will be generated.
  - The table will be created as a Clustered Base Table (CBT), specifying two organizing keys: the product identifier and the customer identifier.
- Dimension tables: each dimension is loaded into a different database. Let’s call them CUSTOMERS_DB and PRODUCTS_DB for the customer and item dimensions, respectively\(^2\).

---

\(^1\)Generating TPC-DS database and data in Netezza
Finally, a data source for each database is created in Denodo Virtual DataPort, as shown in Figure 5.

3.2 PREPARING DENODO
Denodo uses a variety of information for query optimization, including statistics of the views’ data, information about the available indexes in the data sources, primary keys, and the referential constraints defined in Denodo. It also takes into account some configuration parameters describing certain features of the data sources.

The following subsections describe in detail how to configure Denodo to collect and use each type of information mentioned above. Before that, automatic optimizations must be enabled from the Administration > Server Configuration > Queries optimization menu of the VDP Administration Tool as shown in Figure 6:

CUSTOMERS_DB and PRODUCTS_DB can be loaded into any RDBMS database: Oracle, SQL Server, etc. In the case of Oracle, this document explains how to load the data using the SQL*Loader utility.
3.2.1 Declaring Primary Keys and Associations

One of the most important optimizations that Denodo applies is **aggregation pushdown** (a simple example was shown in Figure 3). It can avoid the movement of large amounts of data through the network, therefore greatly reducing execution times. To use this technique, Denodo needs information about the Primary Keys (PKs) of the views involved in the queries.

Denodo is able to automatically retrieve PKs information for base views coming from databases accessed through JDBC or ODBC. For instance, for the customer view in CUSTOMERS_DB it automatically detects that c_customer_sk is the PK when the base view is imported, as shown in Figure 7. It is also possible to manually modify the PKs for every view in the Virtual DataPort server, if needed.

![Figure 6 Queries optimization screen.](image)

### Table: Primary Keys for customer view.

<table>
<thead>
<tr>
<th>PK</th>
<th>Field Name</th>
<th>Field Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_customer_sk</td>
<td>long</td>
<td></td>
</tr>
<tr>
<td>c_customer_id</td>
<td>text</td>
<td></td>
</tr>
<tr>
<td>c_current_cdemo_sk</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>c_current_hdemo_sk</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>c_current_addr_sk</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

In Denodo 5.5 and lower associations between views needed to be defined to let Denodo apply aggregation pushdown. This is no longer needed from Denodo 6.0 onwards. Associations are still needed for other optimizations discussed in the last section of this document.

**3.2.2 Collecting Statistics**

The cost-based optimization process needs the statistics of at least all the base views participating in the query (the average size of a field, the maximum and minimum values, the number of distinct values, etc.) to estimate the cost of prospective execution plans. The statistics allow the Denodo query engine to decide whether to apply such optimizations as join reordering, choosing a join execution method (merge, hash, nested,...), applying ‘aggregation pushdown’, or carrying out data movement automatically.

To gather the statistics of a view, open the Options > Statistics dialog for that view, enable them, and click the ‘Gather statistics for selected fields’ button, as shown in Figure 8. When that button is pressed, Denodo performs the following process:
1. If the source maintains its own statistics in metadata tables (in its catalog), Denodo obtains the statistics from those. This is the best scenario, as querying the catalog tables is usually very fast.

2. Some sources do not allow accessing all the required statistics information from their catalog tables (for example, Netezza does not allow obtaining the maximum and minimum values for a field). In this case, selecting ‘Complete missing statistics executing SELECT queries’ tries to get the information that could not be retrieved from the catalog by sending queries to the sources (for instance, select max(field), min(field), ... from <view>).

3. Some sources, such as Hive and Spark SQL, do not expose information about statistics in their catalog. In this case, Denodo has to use queries to get the statistical information about each selected field (by default, every field is selected).

If the sources do not have their statistics in the catalog (for instance, because they have not been generated yet) and an additional workload executing queries to

3Some sources need their statistics to be generated (they are not generated automatically). For instance, in Oracle the GATHER_TABLE_STATS procedure has to be executed. In other cases, the DBA could have disabled them, so they are not stored.
generate the statistics cannot be introduced into the data sources, the option ‘Do not delegate the execution of SELECT queries’ can be enabled. By doing this, Denodo retrieves all the data from the table (executes a SELECT * FROM view) and executes the aggregation functions locally. This option makes sense when dealing with heavily-loaded operational data sources. In this case, as those calculations are time consuming, this process can be moved from the sources to Denodo. The process can also be scheduled with Denodo Scheduler during periods where the system is not expected to be under heavy load. (see “How to update the statistics of a view automatically” Denodo KB article for more information).

It is important to note that Denodo does not need precise values in the statistics to do a good job with cost optimization. Providing approximate values is usually good enough. Therefore, if it is not possible to obtain statistics for some data sources (e.g. web services with mandatory input parameters) but approximate values for the most commonly used fields are known, it is possible to manually enter those estimates and receive the same benefits.

To achieve good estimations of both the required number of I/O operations and the network transfer cost, it is crucial to have available at least the following statistics for the view used in a query:
- Number of total rows of the view.
- Field statistics of all the fields of the view that are used in the query (more specifically, in the select and where clauses). Note that a field might not be referenced in the actual query performed, but may be required in a lower view.
  - For these fields, at least their average size must be provided.

To see more information about cost-based optimizations and how to enable and use it in Denodo, see section ‘Gathering the Statistics of Views’ of the Denodo Virtual DataPort Administration Guide.

3.2.3 Declaring Data Movement Targets
The Data Movement optimization can execute some federated queries more efficiently. When a query involves two views and one of them is much larger than the other, Denodo can transfer the data of the smaller view into the data source of the larger view and execute the entire operation in the second data source. This may offer great performance improvements for some queries versus pulling the data from both sources and having the Denodo server process the results. The performance of the following operations, in particular, can be improved with Data Movement: Join, Union, Minus and Intersect.

In order to allow Denodo to apply this optimization automatically, the data sources involved in the query must be enabled data movement targets, as shown in Figure 9. Being a ‘data movement target’ means that Denodo can create a temporary table in that source with the data coming from other branch of the join, if that is the best option for a certain query. We are going to set each database made in this tutorial as a data movement target and the optimization engine will decide automatically whether to apply this technique or not for each query.

Notice that the user specified for the connection needs privileges to create tables in that schema (it is possible to specify a different schema for the data movement if the user does not have write permissions in the schema configured when creating the data source).
3.2.4 Declaring Indexes in Denodo

When calculating the cost-estimations of the execution plans, Denodo examines the indexes defined in the views to determine if they apply to the query. Then, it uses different formulas to estimate the number of I/O operations based on the index type. The number of I/O operations can change dramatically depending on if an index is applicable or not, and depending on index type. Therefore, to achieve good estimations it is crucial to ensure the accuracy of the indexes’ information. 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 (see section ‘Specifying Indexes’ of the Denodo Virtual DataPort Administration Guide to see how to tune the imported indexes information for several databases).

In this example, Netezza is the database for the store_sales table. In the section ‘Tuning the Imported Indexes Information for Netezza’ of the Denodo Virtual DataPort Administration Guide, even though Netezza does not have the concept of index, every field that is part of a Clustered Base Table (CBT) can be considered a cluster index. These indexes have to be manually created in Denodo, and they have a big impact in cost estimations. An index of type “Cluster” for the distribution columns of the table must be declared (that is, the columns used to divide the data into partitions - in this example, its primary key -). In Figure 10 the physical properties of the STORE_SALES table in Netezza are shown (mainly, the distribution and organizing keys) and Figure 11 shows the indexes created in Denodo for that view.

Figure 9 Enabling data movement.

4The number of I/O operations can be heavily influenced by some I/O parameters of the data source. Denodo provides adequate defaults for the most typical configurations in the most common types of data sources, but these parameters can also be modified if necessary.
EXECUTING QUERIES

Let’s see how to execute some queries in the sample scenario and how the optimizations work in practice.

3.3.1 Full aggregation pushdown: total sales by customer

Let’s start with a query which applies the same optimization technique (‘full aggregation pushdown’) illustrated at the beginning of this article. The query computes the total sales by customer (see Figure 12). As in that example, Denodo can delegate the full group-by operation to SALES_DB. Then, Denodo joins the results with the tuples from the customer view to add the rest of projected fields (see Figure 13).
SELECT
c_customer_sk,
c_first_name,
c_last_name,
c_preferred_cust_flag,
c_birth_country,
c_login,
c_email_address,
SUM(ss_ext_sales_price) year_total,
's' sale_type
FROM customer
INNER JOIN store_sales
ON (c_customer_sk = ss_customer_sk)
GROUP BY
c_customer_sk,
c_first_name,
c_last_name,
c_preferred_cust_flag,
c_birth_country,
c_login,
c_email_address

Figure 12 Raw SQL query for total_sales_by_customer view.

Figure 13 Tree view for total_sales_by_customer view.
Figure 14 Query Plan for total_sales_by_customer view: no group-by after join.

Figure 15 Query Plan for total_sales_by_customer view: customers branch.
The store_sales view contains 287,997,024 rows that would be transferred over the network and processed in Denodo if no optimization was applied. Also from the statistics there are only 1,999,984 different customers in that view, so, thanks to this optimization, only 0.69% of the total is transferred, which is a big improvement in terms of performance.

In the images above: the left branch of the join only transfers 1,999,984 tuples from the store_sales view, and the full group-by is delegated (as shown in Figure 16). The right branch of the join retrieves the total number of customers, as shown in Figure 15. And the final result (Figure 14) contains the total_sales_by_customer with 1,999,984 tuples and shows that the “Aggregation Push-down” optimization has been applied. It is a full push down, as there is no group-by after the join.
3.3.2 Partial aggregation pushdown: total sales by item category

This is a very similar scenario, except the results are grouped by item category (i_category field from the item table). The query sent to the server and its tree view can be seen in Figure 17 and Figure 18, respectively.

In this case, the group by operation cannot be fully computed in the facts table, so the ‘full aggregation pushdown’ technique cannot be applied. What is the difference with the previous case? In the previous example the results had to be aggregated by customer, and the facts table included the customer id as a foreign key. Therefore, it was possible to compute the groups directly in the facts table, adding the remaining fields of the dimension table in the join operation. Nevertheless, in this case, the results have to be grouped by a dimension field (i_category) which has no correspondence in the facts table.

In cases like this, Denodo can apply an alternative technique called ‘partial aggregation pushdown’. The difference with ‘full aggregation pushdown’ is that the aggregation operation is performed in two steps:

1. One of them is pushed down to the data source (in this example, computing the total sales by item, grouped by the item foreign key, as shown in the bottom screenshot in Figure 19).
2. The second one is performed by the Data Virtualization server (in this case, obtaining the totals for each category, after joining the results with the item view), as shown in the upper screenshot in Figure 19.

The rules that Denodo follows to determine the exact operations that need to be performed at each of the two steps are somewhat complicated, but it is easy to see that the new query plan will get the desired results..

```
SELECT
    item.i_category_id,
    item.i_category,
    SUM(ss_sales_price) sum_agg
FROM store_sales
JOIN item
    ON (store_sales.ss_item_sk = item.i_item_sk)
GROUP BY
    item.i_category,
    item.i_category_id
```

**Figure 17** Raw SQL query for total_sales_by_item_category view.
Figure 18 Tree view for `total_sales_by_item_category` view.
3.3.3 Data Movement: total of sales by item filtered by category and price

In this scenario (see the SQL query in Figure 20), the query engine detects that only a few tuples are needed from the item view (because of the filtering in the where clause), so the best execution plan is to move the data from the item view to SALES_DB (the data source of the store_sales view) and push down the whole query execution to this data source, as shown in Figure 21.

```
SELECT i_item_sk,
       SUM(ss_sales_price) sum_agg
FROM store_sales
JOIN item
```
ON (store_sales.ss_item_sk = item.i_item_sk)
WHERE i_current_price > ss_list_price
AND i_category = 'Women'
GROUP BY item.i_item_sk

Figure 20 Raw SQL query for total_sales_by_item_price_women view.

Figure 21 Tree view for total_sales_by_item_price_women view.
Figure 22 shows that the optimization applied was ‘Data Movement’ and that the complete query was delegated to the data source. The second screenshot in this figure shows that the needed data from `item` was copied to a new temporary table (the name is shown in green rectangles) in the data source of the `store_sales` view. This way, Denodo can push down the query to this data source, as shown in the third screenshot.

### 3.4 SINGLE-VIEW APPROACH

In this section, an alternative approach for modelling the same scenario is presented. A single view joining the facts table with all the dimension tables will be created as shown in Figure 23. With this approach a single join view will contain all
the data. This makes it easier to create reports for external reporting tools, as they only have to query that unified view, avoiding join operations among several tables.

![Diagram](image)

Figure 23 Single join view for reporting.

What happens if the first query (total of sales by customer) is executed against this single join view? Note that this query is not projecting any of the fields from the item view, but is included in the definition of the new view. Denodo’s query engine is smart enough to notice that and it removes the branch of the item view from the join. This is an optimization known as “Branch Pruning” and is shown in Figure 24. In this figure, the query sent to the `store_sales` view is displayed in blue (with the ‘full aggregation pushdown’) and in orange the query sent to the `customer` view. Note that no query has been sent to the `item` view.
The information that Denodo needs to apply this technique depends on the cardinality of the associations between the joined views and on the join method, as explained in the following two subsections.

3.4.1 Case A: every sales has a known customer and item

To use this technique, Denodo needs information about the Foreign Keys (FKs) of the views involved in the queries. To that purpose, it is necessary to define associations between the fact table and each dimension: in the example, between store_sales and item and between store_sales and customer. Let’s see an example on how to create an association between store_sales and customer. Figure 25a and Figure 25b show an association named store_sales_customer, where store_sales contains the foreign key (note that “Referential constraint” has been selected) to customer.

![Figure 24 Query plan for total sales by customer with single join view.](image-url)
Another association between `store_sales` and `item` has to be created in the same way.

It is **very important** to notice that this optimization can only be applied when the cardinality on the side of the unused table (`item`, in this case) is exactly 1. This is because, even if the `item` view does not contribute fields to the output of the query, its join condition could filter rows from the output. In our example, this could happen if the `sales` table had rows without any associated item (this means `item_sk` would be ‘null’). Declaring the 1..N cardinality allows Denodo to know that this cannot happen and, therefore, that the optimized execution tree is equivalent to the original one.
3.4.2 Case B: sales can have unknown customers and items

If it is possible to have sales made to an unknown customer or of an unknown item (this is, there is a 0..1 cardinality on the side of the dimension table), it is likely all this sales information has to be included in the final report. If this is the case, the join method can be changed from inner to left outer (being store_sales the left side), so no rows from store_sales are filtered out from the output. With this approach, it is not necessary to have the associations defined in order to let Denodo apply the ‘Branch Pruning’ optimization.
4 REFERENCES

Denodo Query Optimizations for the Logical Data Warehouse (Part 2): Working With Partitioned Fact Tables