Best Practices to Maximize Performance I: Modeling Big Data and Analytic Use Cases

Revision 20201019

Denodo North America & APAC: 525 University Avenue, Suite 31, Palo Alto, CA 94301. USA
Denodo Iberia & Latino América: Montalbán 5, 28014 Madrid, Spain
Denodo EMEA: 21st Floor, Portland House, Bressenden Place, London SW1E 5RS. UK
Denodo DACH: Karlstraße 10, 80333 München. Germany
www.denodo.com

CONTENTS

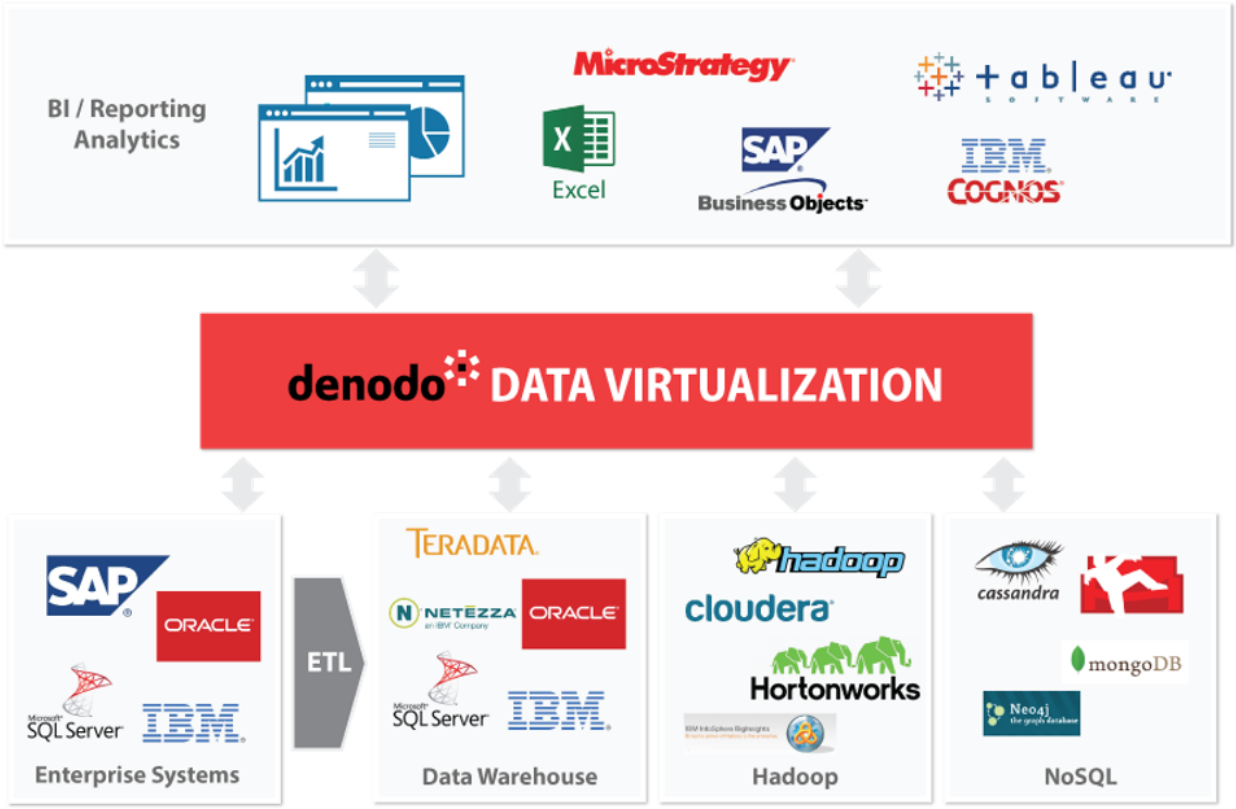
1 OVERVIEW

This document is the first 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.

This first document is aimed at Data Architects and Developers and it contains the recommendations for designing a model that can make the most of the query optimizer capabilities.

2 LOGICAL DATA WAREHOUSE

The term Logical Data Warehouse(LDW) was introduced by Gartner to define a system architecture that offers data consumers the appearance of a traditional data warehouse, but where the data can be distributed among different systems. The LDW provides a unified infrastructure for querying, metadata management, security and governance across all these systems.
You can easily build a Logical Data Warehouse or Logical Data Lake architecture using Denodo: Denodo efficiently combines data from multiple data sources, allows enforcement of global security and governance policies across such data sources, and provides a layer of abstraction which allows data consumption without having to deal with the underlying data management infrastructure. The data sources in a LDW typically include physical data warehouses, Hadoop clusters, SaaS applications, NoSQL systems and additional databases.



In a scenario like this where you need to combine large amounts of data from different data sources, Denodo's query optimizer module plays a crucial role in achieving high performance.
The query optimizer is able to apply different techniques in order to resolve each query in the most efficient way, minimizing data transfer. These techniques include query delegation, query rewriting, advanced join optimization, 'on-the-fly' data movement and parallel query processing.

In order to obtain the best performance and make the most of the query optimizer there are some modeling considerations that are important to take into account. This document will focus on the design of a logical data warehouse or data lake from a performance point of view.

2.1 LOGICAL DATA WAREHOUSE: SAMPLE SCENARIO

To illustrate the different models and approaches we will use a sample scenario. Let's imagine a big retailer company that stores:

- Information about items and customers in a conventional relational database.
- Information about sales in two different systems:
  - An enterprise data warehouse (EDW) containing the sales from the current year; and
  - A Hadoop-based repository containing the information for the sales from previous years.
- A date table stored in both the EDW and the Hadoop cluster to filter by different date criteria.

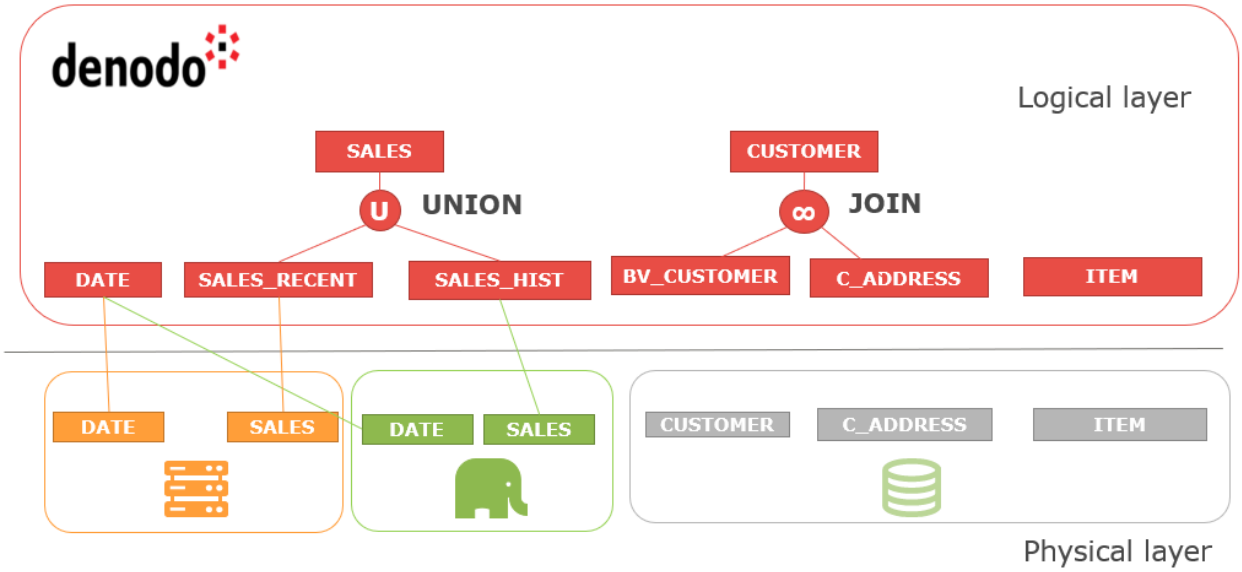
2.2 LOGICAL DATA WAREHOUSE: GENERAL RECOMMENDATION

The general recommendation for modelling this type of scenarios is:

- Create a single view for each fact entity and dimension entity. This maximizes the chances that Denodo will use certain optimization techniques (group by push-down and branch pruning), and makes the model clearer from a semantic point of view and easier to use for client applications:
  - If the data for this fact table or dimension is **partitioned horizontally** among different systems you should build a view as a partitioned union. This way Denodo will only access the partitions that are necessary to resolve each query. In our example, create a unified view 'sales' as a partitioned union from both the data warehouse and the Hadoop repositories (See section **Partitioned Unions**).
  - If the data is **partitioned vertically** and you therefore need to join several views in order to obtain all the necessary columns, follow the conventions for building "prunable" views (See section **Vertically Partitioned Views**). This way, if the only purpose of a join is enriching

the fact table with more columns (and not filtering rows) Denodo will only perform that join if those columns are required in the query. In our example, create a unified customer view joining the data between customer and c_address.

- If there are tables in different systems containing the same data, provide alternative sources (see section **Alternative Sources**). This way Denodo will decide which is the best one to access depending on the query. In our example, create a base view over the date table in the data warehouse and specify the Hadoop cluster as an alternative source or vice versa.



The following sections provide details about each of these steps, as well as other modeling considerations such as when to use view parameters or the 'Create Base View from Query' option. Finally, the last subsection describes an alternative modelling approach called 'single view' that can be useful in certain cases.

3 UNION VS UNION ALL

Like in Standard SQL, using Denodo it is possible to create a union view that selects only distinct values, or a union view that allows duplicates (UNION ALL).
Our recommendation is **to use UNION ALL whenever is possible**, as the default UNION, or UNION DISTINCT, prevents two of the most important query rewritings: Aggregation Push-down and Join-Union push-down. This happens because the UNION is performing an implicit DISTINCT operation, and therefore calculating a join or group by operation before or after the distinct is not equivalent in general.

4 PARTITIONED UNIONS

It is common that the data for the fact table (and sometimes the dimensions) is located in several different systems. In the retailer company from our example, the information about sales is stored in two systems: an enterprise data warehouse and a data lake based on Hadoop.
In this case, in order to create the canonical view containing all the sales information, you can create a view in Denodo called 'sales' defined as a UNION of both tables.

However, we know that each system has a specific part of the data, defined by a certain criterion:

- The enterprise data warehouse contains the sales data from the current year
- The data lake based on Hadoop contains the sales data from previous years

Therefore, if a query asks for the sales from 2016, for instance, it is not necessary (and it wouldn't be efficient) to access the data warehouse system as we know that the information from past years is in the data lake only.
In cases like this you need an extra step to provide Denodo the information about the partition criterion. To build this partitioned union you would:

- Create an intermediate selection view over each partition and specify there the condition that is true only for that partition (year = getyear(now()) for instance) as a regular WHERE condition, and
- Create the union as the union of these intermediate selections.

The conditions in our example would be:

- year = getyear(now()) for the data warehouse, and
- year < getyear(now()) for the data lake

This way, if a query contains a filter condition that is not compatible with some of the partitions, the query optimizer will detect that situation and it will remove all the union branches that are not necessary for that particular query.
Using the previous example, if the query contains the condition year = 2016 it will detect that the condition year = getyear(now()) is not compatible with the current selection and the only partition that is compatible is the one accessing the Hadoop system.

You can find a full example about how to create a partitioned fact table by date in the document titled Denodo Query Optimizations for the Logical Data Warehouse (Part 2): Working With Partitioned Fact Tables in the Knowledge Base of the Denodo community.

Finally, there are cases where the partitioning criterion is not defined by a pre-existing field.
For example, let's imagine our company sells products worldwide and the sales information is partitioned in three systems:

- One containing the information for EMEA (Europe, the Middle East and Africa);
- One containing the information for America; and
- One containing the information for APAC (Asia-Pacific).

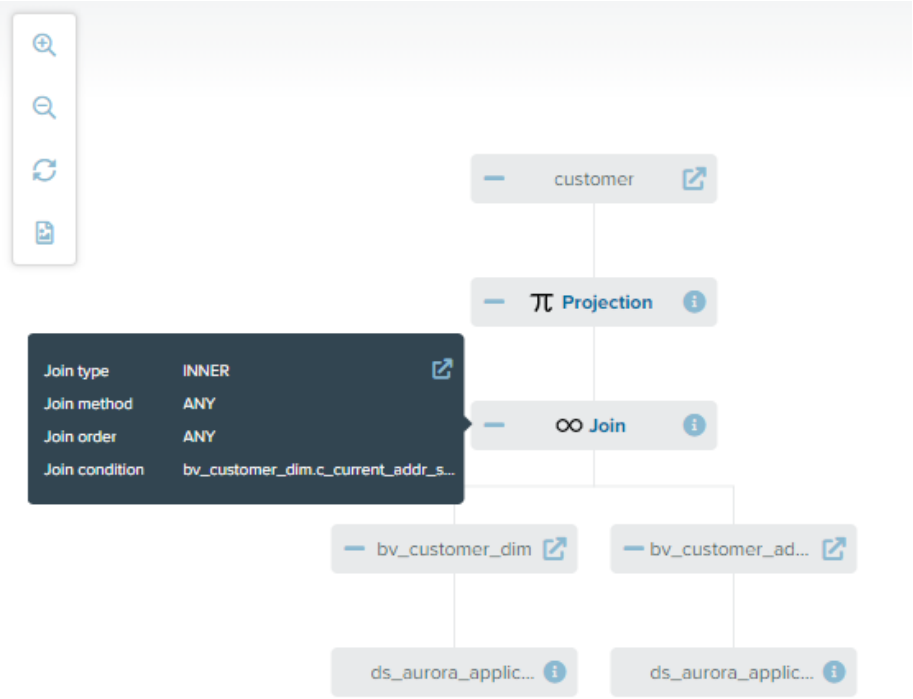
The partition in this case is made by region, but we don't have a column in sales specifying a region. In order to build the partitioned UNION, you can create the

intermediate selections using view parameters. You can find an example of this scenario in the Virtual DataPort Administration Guide.

5 VERTICALLY PARTITIONED VIEWS

If the necessary columns to build a unified view are distributed among different tables you may need to join several views in order to obtain the complete information. For example, in the retailer company scenario, we have a customer table with the basic information and another table containing the address information.

In this case we recommend building a unified customer view joining both tables following the considerations in the **Building Joins** section. This means (i) making sure you have set the primary key on customer_address, (ii) defining a referential constraint between both views, and (iii) specifying a simple join condition using the FK-PK fields. This will ensure that Denodo will access just the necessary tables in order to get the data for each specific query.



5.1 SLOWLY CHANGING DIMENSIONS

Slowly changing dimensions are data warehouse dimensions that store and manage both current and historical data over time. In order to do so, two common approaches are used:

- Adding a column containing a flag ‘current_flag’ that is ‘Y’ for the current value or ‘N’ for the old ones.

- Adding two columns start_date and end_date that store the date range while that data was effective.

For example, imagine that customer_address contains both the current address and all the previous addresses for each customer and contains a column current_flag that is 'Y' for the current address.

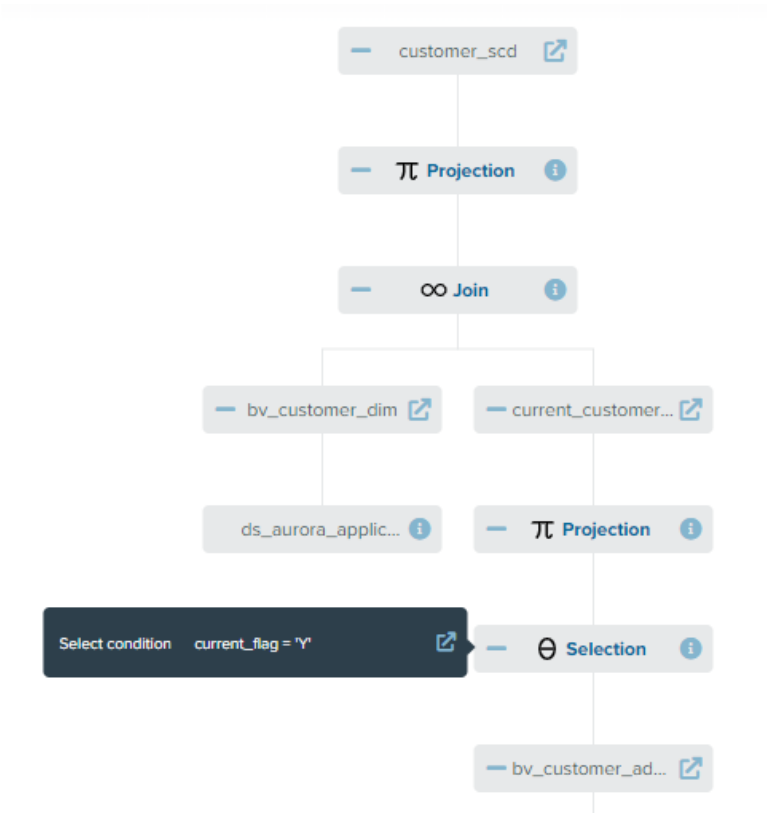
customer_id	street_number	street_name	city	state	country	current_flag
1	18	Jackson	Fairfield	AZ	US	N
1	362	Washington 6th	Fairview	NM	US	N
1	585	Dogwood Washington	Pleasant Valley	PA	US	Y
2	111	Smith	Oak Ridge	CO	US	Y

Take into account that customer_id is not unique on address (as the same customer can have several rows for the different addresses that person had). This means that if we want to combine some data with the current address for a certain customer, we need to consider only the one for which current_flag = 'Y'. One way of doing this would be adding the condition to the join, but as described in the **Building Joins** section, in order to build 'prunable' views, it is important to build joins between FK-PK relations with simple conditions.

Therefore, the general **recommendation for modeling tables storing complete history of data** is:

1. Create a derived view containing a WHERE condition to obtain only the current values:
    a. current_flag = 'Y' or
    b. end_date is null (depending on whether it uses a flag or start_date/end_date columns to trace the changes).

    In our example, create a derived view current_customer_address with condition Current_flag = 'Y'.
2. Set the primary key on this new derived view. In our example, we can set customer_key as the primary key on this new view because customer_key is unique after applying the filter condition.
3. Every time you want to obtain the current address in a join, use this new derived view instead of filtering the current value on the join condition. In our example, create the unified customer view using this new current_customer_address. This way Denodo will be able to prune the join when the query does not require information about the address.



For more details about modeling slowly changing dimensions in Denodo see the article titled [How to model slowly changing dimensions in Denodo](#).

6 ALTERNATIVE SOURCES

If the same table is replicated in several systems, you can create a base view over one of them, and then add the information about the alternative tables that Denodo can use to obtain the same information. This way Denodo can decide which one is better to access depending on the query.

This is especially useful when the fact table is partitioned among different systems. When this happens, it is also common that some of the dimensions, such as the date dimension in our example, is replicated in all of them. In that case, Denodo can rewrite the query so each system performs the join between its partial fact table and its replica of the dimension, and the virtual layer just needs to combine these partial results with a union.

This way the join in each partition will be executed in parallel, the data transfer will be minimized (if there was a filter by date, for example) and the query execution will take advantage of structures like indexes that could exist in the source.

You can see a complete example of this feature in the article titled Denodo Query Optimizations for the Logical Data Warehouse (Part 2): Working With Partitioned Fact Tables (Section “Total Sales by Customer: Alternative Sources“)

7 VIEW PARAMETERS

A view in Denodo can include not only regular columns but also parameters.
Parameters are especially useful when you want to build a view containing a WHERE condition but you don't want to specify a fixed filter value on the view definition, as you want it to be dependent on the query instead. For example, a view obtaining the total sales between a start date and an end date where the specific date interval will be provided in the query (notice that this view only contains a column with the total, so you can't just simply query the view adding a WHERE condition).
You can provide a default value for the parameter to use in case no other value is provided.
If the parameter doesn't have a default value it will be mandatory to specify a value in a WHERE or JOIN condition.

There are two situations where view parameters can be useful to improve the performance:

- Enforce a filter parameter. You can add a parameter to force the user to always query a view specifying some filter. For instance, in cases where a fact table contains a high volume of data, you could use view parameters to force users to always query within a range of dates (start_date - end_date).
- Partitioned Unions. When you are building a partitioned UNION, each union branch needs to have a condition specifying the partitioning criterion (See section **Partitioned UNIONs** in this document). However, sometimes the tables do not contain a column that can be used for defining this criterion and you can use a view parameter instead.

You can find detailed information about view parameters and these scenarios in the Virtual DataPort Administration Guide.

8 OTHER MODELING CONSIDERATIONS

8.1 BUILDING JOINS

Building joins in Denodo is as easy as dragging and dropping the views you want to join and specifying the desired conditions, but in order to obtain the best performance you should take into account the following recommendations when building a new join:

- Check if the join condition references a field or group of fields that are unique in one of the views and make sure there is a **primary key or a unique index** defined over those fields. This is necessary for branch pruning and group by push-down optimizations.
  - For example, if you join sales and item using item_id, make sure item_id is defined as the primary key on item.
  - **Special case**: The field is not unique in the table but is unique for that particular join condition.
    - Example: customer JOIN address ON(customer_id = customer_id AND current_flag = 'Y')
    - In this case, create a derived view over address with the condition current_flag = 'Y' and primary key 'customer_id'; and modify the join to use this new view (See section **Slowly Changing Dimensions**).
- Check if the joined views fulfill an FK-PK relationship and, in that case, make sure that there exists a **referential constraint** defined between them. This is necessary for branch pruning and to push down window functions to one side of a join (See section Analytic Functions (Window Functions) in the Denodo Virtual DataPort VQL Guide).
- If you are using Outer joins, **link the views from the same data source first**:
  - When joins are inner, the query optimizer can reorder them automatically to maximize the number of joins that are pushed to the same source.
  - Example: (sales JOIN item) JOIN date WHERE year = 2017 will be reordered to (sales JOIN date) JOIN item, so the database can perform the JOIN between sales and item and transfer to Denodo only the sales during 2017.
  - However, this join reordering only applies to Inner joins. Therefore, if you are using outer joins, start joining the ones that belong to the same data source first. In our scenario, join sales with date first, and then sales with item.
- **Avoid using view parameters on the join condition**. It forces Denodo to take into account extra considerations for the possible join strategies and makes the internal view management less efficient.
- Take into account that **if you set a specific join strategy, the Cost Based Optimizer will not consider other alternatives**. Therefore, specify a particular join strategy only if you are not using the Cost Based Optimizer or you really want to fix that strategy for all cases.

- Use **simple join conditions** if possible. Conditions like fieldA = fieldB allow the optimizer to estimate join cardinalities and decide if a branch pruning is possible or a group by can be totally pushed down to a data source.

> NOTE: Take into account that a join with a condition like UPPER (idA) = idB can always be transformed into a join with a simple condition creating an intermediate projection with a derived field. In this case, you can create a derived view over A with a new field upperIdA defined as UPPER (idA) and join that new view with the condition upperIdA = idB.
>
> **Special case:**
> The merge join strategy cannot currently be applied if some of the fields in the join condition are derived fields like in the previous example upperIdA = idB.

8.2 WHEN TO USE BASE VIEW FROM QUERY

When building the physical layer in Denodo, the user can create the base views over the physical JDBC/ODBC data sources in two ways:

- Selecting the individual tables from the ones available in the data source
- Creating a base view from a query.

The general recommendation is to create the views by selecting the individual tables from the database instead of using the “Create from Query” option.
Using the “Create from Query” option has several disadvantages, such as missing the data lineage and internal information about the columns (like the type, the column size, or the scale and precision) on the original source. This information can be important if you plan to cache that view or the optimizer decides to move that view to a temporary table (using the data movement or MPP acceleration optimizations) as the new table should use the same column properties as the original one.

However, there are situations where this option can be the best or the only one. For example:

- If the query was already built, and it is a complex query, difficult to re-build that represents a canonical entity.
- If the query uses database-proprietary functions or constructions (e.g. Teradata recursive queries)
- To invoke stored procedures in those databases where graphical introspection is not supported.

For more information about the use of the “Base View from Query” option and its pros and cons please see the Knowledge Base article Using the Create Base View From Query Option.

8.3 AVOID SUBQUERIES INSIDE THE WHERE CLAUSE

A query containing a subquery usually has an equivalent alternative using a JOIN instead of a subquery.

If the subquery cannot be pushed to a data source (and therefore Denodo will be the one performing the subquery), our recommendation is to avoid the use of subqueries and use the equivalent JOIN instead. This is because the use of subqueries in the WHERE clause in Denodo has some limitations:

- The Cost-based optimizer cannot make cost estimations over queries containing subqueries in the WHERE clause
- The rule-based optimizer cannot apply all the query rewritings
- Depending on the subquery, performance can be worse than the equivalent query using a JOIN instead

Let's take a look at some examples:

***Query 1**: Obtain the employees whose salary is higher than the average salary*

```
SELECT *
FROM EMPLOYEE
WHERE salary > (SELECT AVG(salary) FROM EMPLOYEE)
```

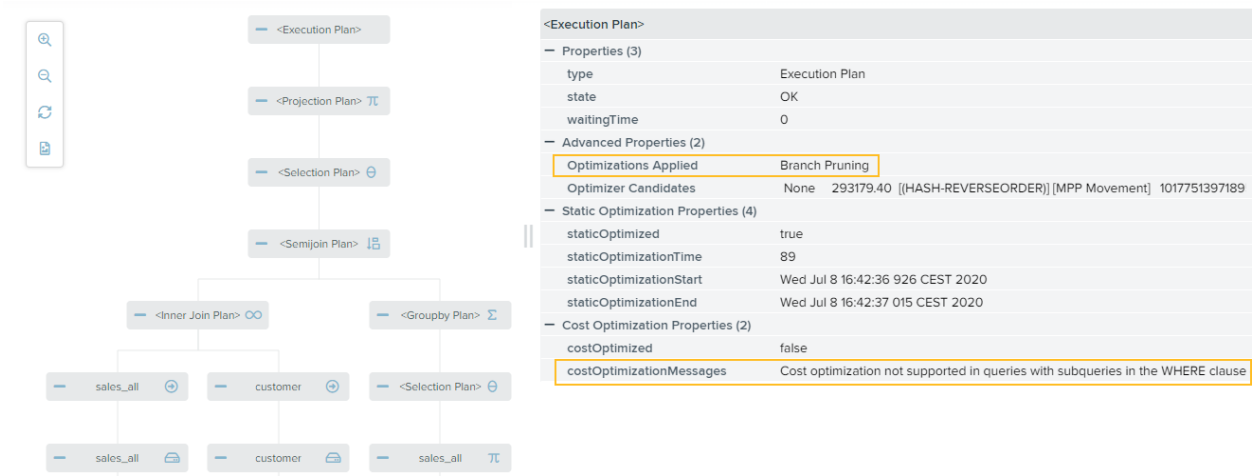
In this case, because the subquery operation is pushed completely to the data source it is not necessary to rewrite the query.

***Query 2**: Obtain the details of the products that were sold by the maximum price through online channels*

- Using subquery:

```
SELECT c_first_name,
C_birth_day,    c_birth_month,    c_birth_year,    c_birth_country,
```

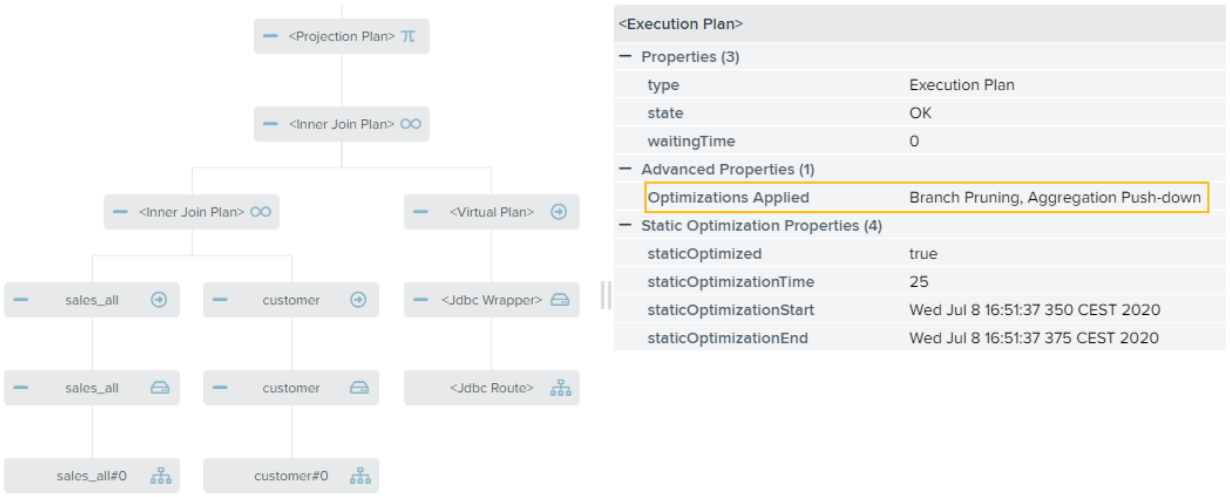
```
c_email_address
FROM   sales_all   s   JOIN   customer   c   ON(s.ss_customer_sk   =
c.c_customer_sk)
WHERE sale_channel = 'online'
and ss_sales_price = (SELECT MAX(ss_sales_price)
                                                FROM sales_all s2
                                                where sale_channel =
'online'
                                                                GROUP BY
ss_customer_sk)
```



You can see the cost optimizer could not apply any optimizations due to the subquery, and although it applied branch pruning for the main query to access only the partition with online data (bottom-left), it accessed the two partitions in the subquery and it does not push a partial aggregation to each one (right-side of the semi-join plan).

- Not using subqueries

```
SELECT c_first_name,
C_birth_day,    c_birth_month,    c_birth_year,    c_birth_country,
c_email_address
FROM   sales_all   s   JOIN   customer   c   ON(s.ss_customer_sk   =
c.c_customer_sk)
JOIN (SELECT MAX(ss_sales_price) max_price FROM sales_all s2
      where sale_channel = 'online'
         GROUP BY ss_customer_sk) max_customer ON(ss_sales_price =
max_price)
```



In this case you can see the cost optimizer could operate and the rule-based optimizer could apply more query rewritings: It only accesses the online partition for both the main query and the “subquery“ (Branch pruning) and it pushes the aggregation to the data source (Aggregation Push-down).

8.4 SINGLE-VIEW APPROACH

As we have described at the beginning of this document the general recommendation is creating one view for each fact table and one view for each dimension table. However, in some scenarios it can be useful to offer a single view containing all the information available in the data warehouse including the columns from all the dimension tables. For example, for some reporting tools, using the single-view approach can make things easier for the user. Nevertheless, this option has some disadvantages:

- Depending on the join conditions, primary keys, etc., used in that single-view, some of the optimizations may not occur.
- It can be hard to manage views with a high number of columns.

Therefore, our general recommendation is:

- Provide the different entities in the star schema individually and let the application combine the entities they need for each case, and,
- Use the single-view approach only if there is a clear advantage to doing so.
- Even if you use the single-view approach, we still recommend building one view for each different entity as a first step, and then building the single-view using these predefined canonical views. This will make the model clearer and easier to manage and maintain.

You can find more information about the single view approach in the Knowledge Base article Denodo Query Optimizations for the Logical Data Warehouse, section **Single-View Approach**.