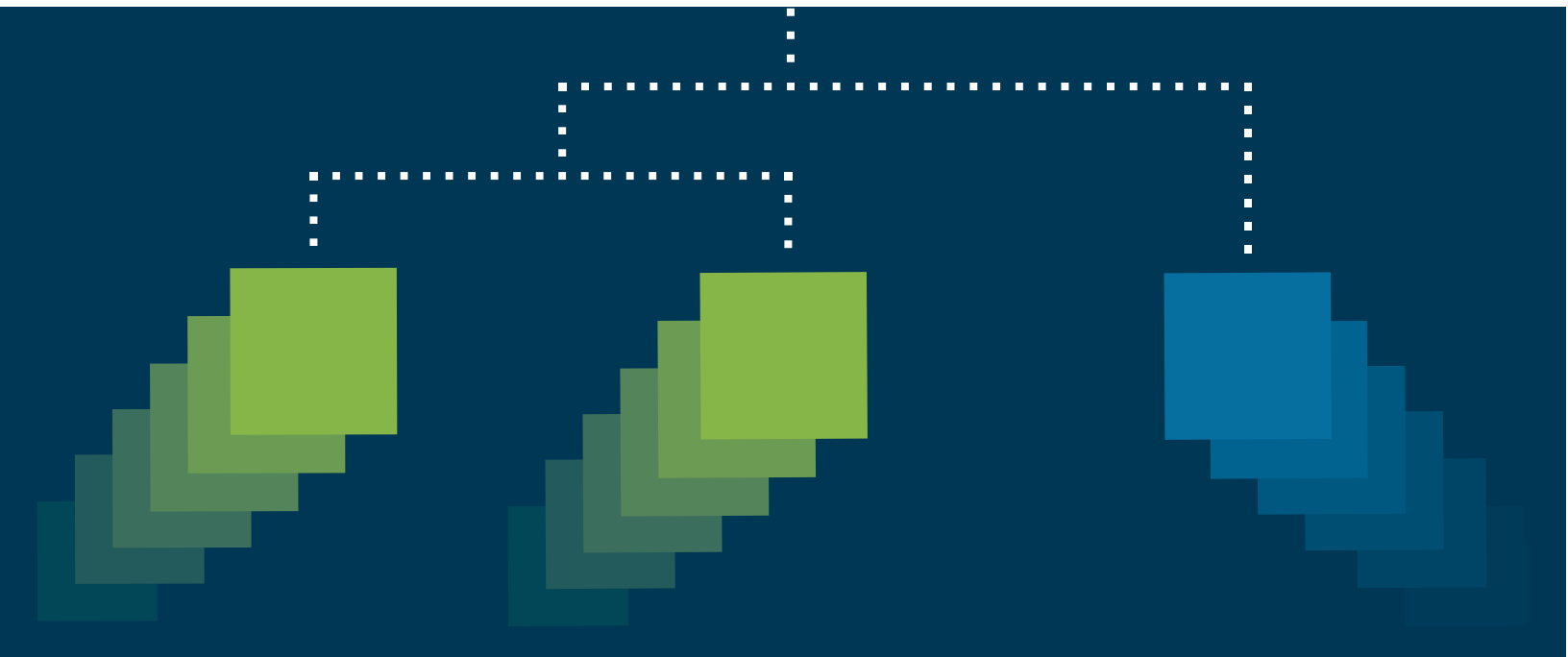


DENODO COOKBOOK

Query Optimization



04	INDEX OF FIGURES
06	INTRODUCTION
06	DV Cookbook Series
07	QUERY OPTIMIZATION
07	Before You Start
08	The Denodo Platform Compared to Relational Databases
09	THE LIFE OF A QUERY
11	ANALYZING QUERY PERFORMANCE
11	Execution Trace
13	Query Plan
13	Exploring an Execution Trace
16	Query Plans vs. View Trees
16	DELEGATION TO SOURCES (PUSH-DOWN)
17	The Benefits of Delegation
18	Showcase: The Performance Effects of Delegation
20	Optimizing for Delegation
20	<i>Delegation Through Caching</i>
23	JOINS
23	Join Methods
23	<i>Hash Joins</i>
25	<i>Merge Joins</i>
27	<i>Nested Joins</i>
29	<i>Nested Parallel Joins</i>
30	Join Methods and Data Source Query Capabilities
30	Inspecting the Join Method Chosen by the Optimizer
30	<i>Nested Joins in Query Traces</i>
32	Manually Specifying the Join Method
32	N-joins
34	Join Types
34	<i>Inner Joins</i>
35	<i>Outer Joins</i>
35	<i>Cross Joins</i>
35	<i>Join Type vs. Join Method</i>
36	MEMORY USAGE
36	Query Complexity vs. Dataset Size
36	<i>Row Shipping vs. Row Hoarding (Streaming vs. Blocking)</i>
39	<i>Caching and Memory Usage</i>
41	Java Heap Memory

41	Swapping
43	Analyzing Memory Usage
43	<i>Interesting Memory Patterns</i>
44	Server Memory Settings
45	STATIC OPTIMIZATION
46	Enabling Static Optimization
46	View Definition vs. Query Conditions
47	Sample Query Rewriting Scenarios
47	<i>Branch Pruning</i>
48	<i>Join Reordering</i>
48	<i>Join Type Change</i>
48	<i>Post-pruning Delegation</i>
50	<i>Aggregation Pushdown</i>
51	<i>Partial Aggregation Pushdown</i>
52	<i>Join Pushdown</i>
54	<i>Data Movement With Partitioning</i>
55	DYNAMIC OPTIMIZATION
55	View Statistics
55	<i>Statistics Used by the Optimizer</i>
56	<i>How to Gather Statistics</i>
57	<i>How Often Should You Gather Statistics?</i>
58	DATA MOVEMENT
58	Data Movement as an Optimization Technique
60	Rationale and Benefits
61	DATA SOURCE OPTIMIZATIONS
61	Connection Pools
64	<i>Pass-through Credentials</i>
65	Base Views From SQL Query
68	DENODO PLATFORM THREAD MODEL
70	VQL CONSIDERATIONS
70	Sorting
71	Aggregations
72	The DISTINCT Clause
72	Functions
72	<i>Delegable vs. Non-delegable Functions</i>
73	<i>Delegable Custom Functions</i>
74	The CONTEXT Clause
75	OTHER CONSIDERATIONS
77	SUMMARY

INDEX OF FIGURES

10	<i>Query execution pipeline.</i>
12	<i>The "Execution Trace" button in the query execution screen.</i>
12	<i>The "Execution Trace" button in the VQL shell.</i>
13	<i>Accessing the query plan.</i>
14	<i>An example of a query execution trace.</i>
19	<i>An unoptimized scenario without delegation.</i>
19	<i>An optimized scenario with full query delegation</i>
21	<i>Subqueries are delegable only if they use base views from a single data source.</i>
22	<i>A non-delegable query over three different data sources.</i>
22	<i>All three sources have been cached in full mode, and the query can be fully delegated to the cache.</i>
31	<i>An example of a nested join in a query trace (tree detail).</i>
31	<i>An example of a nested join in a query trace (advanced information).</i>
32	<i>Manually specifying the join method.</i>
33	<i>Creating an n-join.</i>
34	<i>View tree for an n-join.</i>
37	<i>Streaming operation: rows are shipped to the client as soon as they are received and processed.</i>
38	<i>Linear memory operation: Every source row is stored until the operation is applied to all rows, and then the results are sent to the client application.</i>
38	<i>Constant memory operation: Every source row is immediately aggregated into the partial results.</i>
40	<i>Rows are buffered due to the difference of speed between cache inserts and regular operations.</i>
40	<i>The throttling of operations to the optimal speed when using " query results = false".</i>
43	<i>A typical constant memory graph.</i>
43	<i>A typical linear memory graph.</i>
44	<i>A typical spiked memory graph.</i>
45	<i>Memory management options.</i>
47	<i>A typical partitioned data scenario.</i>
47	<i>Branch pruning when executing queries over a single partition.</i>
49	<i>Base case: a typical non-delegable query.</i>
49	<i>Intermediate step: pruning a branch of the original query.</i>
50	<i>Final query plan: delegation to source, possible after partition pruning.</i>
50	<i>Base case for aggregations over joins in data warehouse reporting scenarios.</i>
51	<i>Aggregation push-down vastly increases performance; no need to transfer the whole fact table.</i>

INDEX OF FIGURES

- 51** *Base case for aggregations using attributes of a dimension.*
- 52** *Splitting the aggregation in two steps dramatically increases performance.*
- 53** *Base scenario for join pushdown; the products base view references two identical copies of the data in datasources 1 and 2.*
- 54** *The join is pushed down below the union, and it can be delegated to both sources.*
- 56** *The View Statistics screen.*
- 58** *A data movement base case. The join+aggregation cannot be delegated.*
- 59** *Using data movement: The small table is temporarily copied next to the big table.*
- 59** *The whole query can be delegated after the temporary data movement.*
- 62** *A query execution trace showing a node waiting on a connection pool.*
- 63** *Each data source in the Denodo Platform maintains its own connection pool to the source system.*
- 64** *Multiple connection pools for a single data source when using pass-through credentials.*
- 66** *The "Create from query" button.*
- 66** *Creating a base view from a SQL query.*
- 69** *Server thread pool controls.*
- 70** *Server concurrency controls.*
- 73** *A non-delegable function displayed in a query execution trace.*

Introduction

This Data Virtualization Cookbook, dedicated to query optimization, provides a detailed explanation of the query performance optimization techniques in the Denodo Platform. This document explains the different decisions that an architect can make when designing new queries, and it showcases when, why, and how to use each of the available options at each step. In addition, we cover the different performance optimization methods that the Denodo Platform state of the art optimizer is applying to each query. With this detailed guidance, architects gain insight on how the Denodo Platform actually works and how to optimize their data virtualization solutions.

DV COOKBOOK SERIES

This Data Virtualization Cookbook is part of a series of in-depth manuals showcasing modern uses of data virtualization in the enterprise. In these books we explain common, useful data virtualization patterns, from the big-picture, business-decision-making level, to the architectural considerations, to the low-level, technical details and example data sets.

The Denodo Cookbooks will enable you to understand the architecture of current data virtualization best practices and get you ready to apply these patterns to your own business cases, ensuring you get the maximum value out of your data processing infrastructure.

Query Optimization

Query optimization is a critical aspect of any data virtualization project. The Denodo Platform provides state-of-the-art, automated optimization but every architect involved in a data virtualization solution should be familiar with the techniques that are automatically applied to every query received by the Denodo Platform. In many cases, out-of-the-box, the Denodo Platform will meet or even exceed the performance requirements of the project-at-hand, but in some complex scenarios, some degree of manual tuning can bring the platform's performance to the next level. This Cookbook will describe the areas of the Denodo Platform that include query-performance components and it will cover their internal behavior, what tuning opportunities they offer to the query designer, and in what situations each one should be used.

This is an introductory guide to help new users understand performance optimization in the Denodo Platform, but it is not exhaustive; users are encouraged to further explore these topics by reading other technical materials available from Denodo, such as the [Virtual DataPort Administration Guide](#), the [Virtual DataPort Advanced VQL Guide](#), and the [Denodo Community Knowledge Base](#).

Before You Start

The Denodo Platform optimizer is an extremely advanced piece of engineering that can apply optimizations whose functions might not be immediately clear to developers or architects. While these optimizations are applied in a very intelligent way, the optimizer does not operate in a vacuum: it needs view statistics, such as the number of rows or the number of distinct values per column, to make its optimization decisions.

The descriptions of the dynamic optimizer that are detailed in this book assume that the optimizer has access to the statistics of the views involved in the queries; it is the responsibility of the Denodo Platform users to actually gather these statistics in a timely manner. Without them, the dynamic optimizer cannot do its job. So remember: enable and gather the statistics for your views, if you want to have a fully functioning optimizer!

If you see queries that are not being optimized as they should be, check to see 1) if the dynamic optimizer is operating, and 2) if the view statistics for the views involved in the query are up-to-date

The Denodo Platform Compared to Relational Databases

Newcomers to data virtualization usually see this technology as equivalent to a relational database. While this analogy is appropriate for client applications querying the data virtualization layer through its JDBC or ODBC interfaces, the actual workings of the Denodo Platform are not comparable to those of an actual relational database.

For starters, the Denodo Platform holds no data itself: it retrieves the data in real time from the data sources (or the cache system, if the data has been previously cached) every time a query is received from a data consumer. This and other differences make the data virtualization layer different in its operation than other traditional information processing systems. Particularly in the area of query performance optimization, the concerns that developers and administrators should keep in mind are different than in typical database scenarios: optimizing queries is very different when you own the data versus when you have to retrieve the data from multiple, disparate remote systems.

The main motif throughout this book will be the optimization of all operations with two goals in mind:

- Reducing the amount of data processing performed at the data virtualization layer.
- Reducing the amount of data that is transferred through the network.

These two ideas are key in bringing the performance of a data virtualization solution in line with that of a traditional physical data scenario, and they are very different than the considerations that are taken into account by the architecture and design of relational databases and data warehouses. Many techniques are applied by the optimizer to this effect, and most of them try to follow the philosophy of making the data sources work as much as possible in preprocessing the data, and transfer the absolute minimum amount of data necessary from the sources into the Denodo Platform. This book will review all of these techniques, it will explain the different parts of the query optimization pipeline in the Denodo Platform, and it will showcase the areas that query designers and administrators should pay attention to when designing high performance data virtualization solutions.

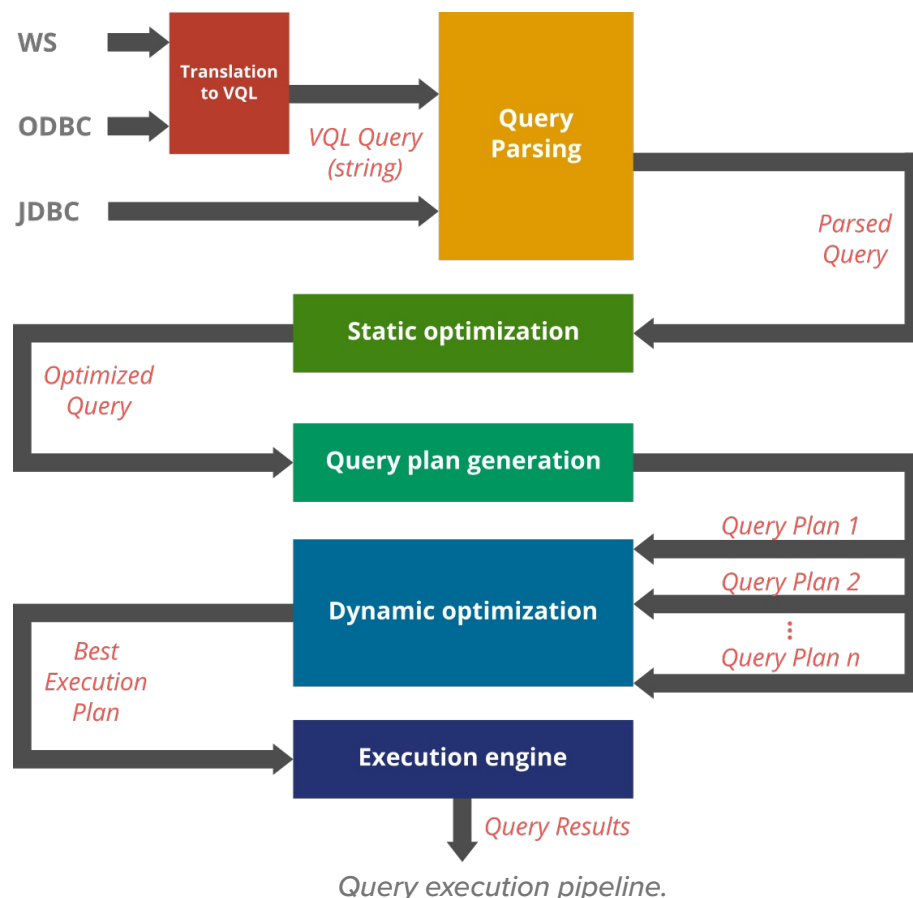
The Life of a Query

Understanding the behavior of a query is critical for tuning its performance. The first thing that a user should understand is the general process that the Denodo Platform follows, from the moment a query is received to the moment the query has been completed. The process looks as follows:

A note about this process: the description below is a simplified model that does not exactly match the way that the Denodo Platform executes queries, but it is an excellent, high-level mental model for understanding how the Denodo Platform works. The actual implementation may reorder operations, merge steps, and execute some in parallel, but the model explained below is very useful for understanding the actual behavior of the data virtualization software.

1. The query is received by the Denodo Platform through a connection opened by a client application. This query is written in VQL and received as a string through the JDBC connection.
 - a) If the query is received through an ODBC connection, the driver will translate the format used by the ODBC channel into VQL and send it to the Denodo server.
 - b) If the query is received through a web service, the published service will perform the translation of the query to VQL, and send it to the Denodo server.
2. Once the query has been accepted into the server, it gets parsed. This process involves transforming the string into a data structure that can be analyzed and modified by the query optimization and execution pipeline.
3. The parsed query is then fed to the static optimizer. In this step the general structure of the query is analyzed, and it is rewritten in a way that preserves the semantics of the query (the results of the rewritten query will be exactly the same as the results of the original query) yet improves the performance (for example, by removing unnecessary conditions or subqueries, reordering operations, etc.). We will see the details of this optimization later.
4. The statically optimized query is then passed on to the query plan generator. VQL is a declarative language, meaning that the user specifies what results are expected, but not how to compute them. The exact list of steps that need to be executed to get those results is called a query execution plan, and it is the responsibility of the query plan generator to analyze the query and come up with a plan. It is important to note here that multiple query execution plans can compute the same result set yet demonstrate different performance characteristics.

5. Once several query execution plans are available, the next step is choosing the best. This is where the dynamic optimization phase is applied. The optimizer calculates a cost associated with each execution plan, and the plan with the lowest cost is selected for execution.
6. The selected plan is then passed to the execution engine, which takes the plan and executes its steps to retrieve the data from the sources, combine and transform the results, and prepare the result sets for the data consumer.
7. After the execution of the query has finished, and all the associated resources have been freed, the query is considered complete.



Every query received by the Denodo server is processed in this way, although in some cases the process stops early, such as if the query is rejected for security reasons, if the query is not correctly formatted, if errors appear in accessing the sources, if calculations result in arithmetic errors (for example, a division by zero), etc. A query can fail in many ways, but the rest of this Cookbook focuses on how to optimize queries that finish their executions correctly, highlighting the static and dynamic optimization steps, and how they affect the execution step in combination with the decisions taken when designing the query and defining the virtual views used by it.

The process that happens before the execution of the query is performed very quickly, and in most cases it will not impact the execution time of the query. Typically, the biggest chunk of time is spent in the actual execution of the query; the overhead of the optimization steps is minimal, overshadowed by the performance improvement gained from those optimization steps. Short queries that manage small data volumes (executed in the order of milliseconds) may be impacted by the time spent in the optimization steps; those special cases may benefit from the manual disabling of the the automatic query optimization.

Analyzing Query Performance

The first step to optimizing the performance of a query is to analyze what is happening when we execute it. This is an often-overlooked step, as it is very common to take assumptions about the runtime behavior as fact, which then renders the corrective actions much less effective than they could be. When approaching optimization problems, try to gather actual data as much as possible, and draw conclusions from the gathered data.

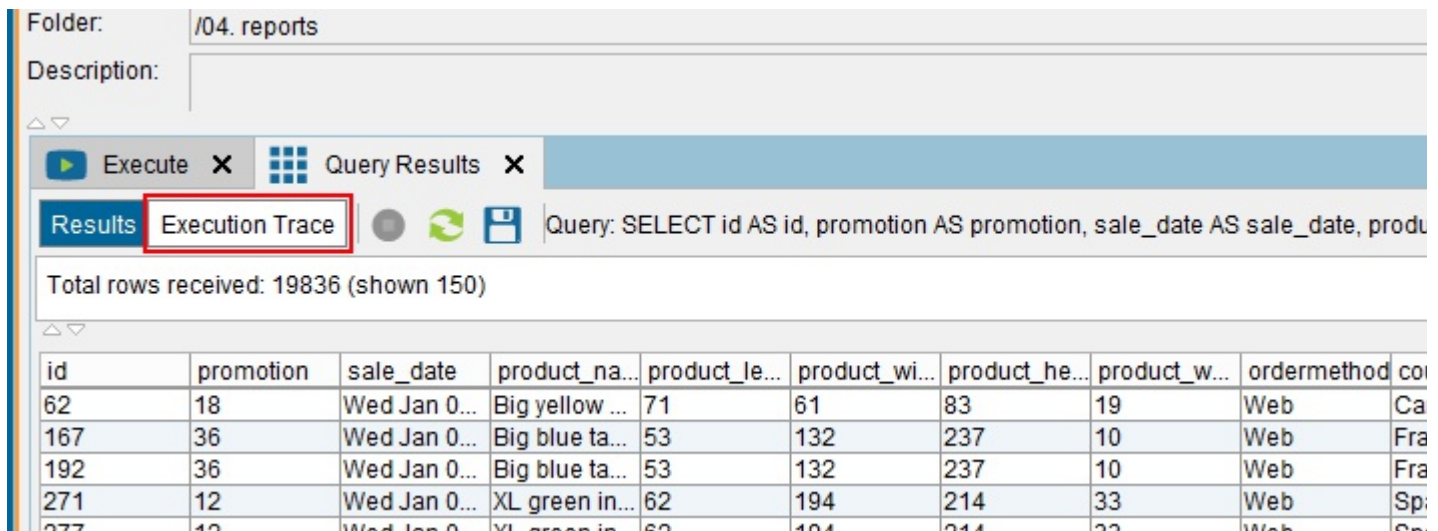
Execution Trace

Our main tool in this query analysis phase will be the execution trace. The execution trace of a query is a textual or graphical representation of the actual execution of its query plan. If we think of the query plan as a template for the execution of a query, the trace is akin to filling that template with actual values that describe a single execution of the query. As such, it gives us a detailed view of all the things that happened when running the query.

We can access two types of execution traces:

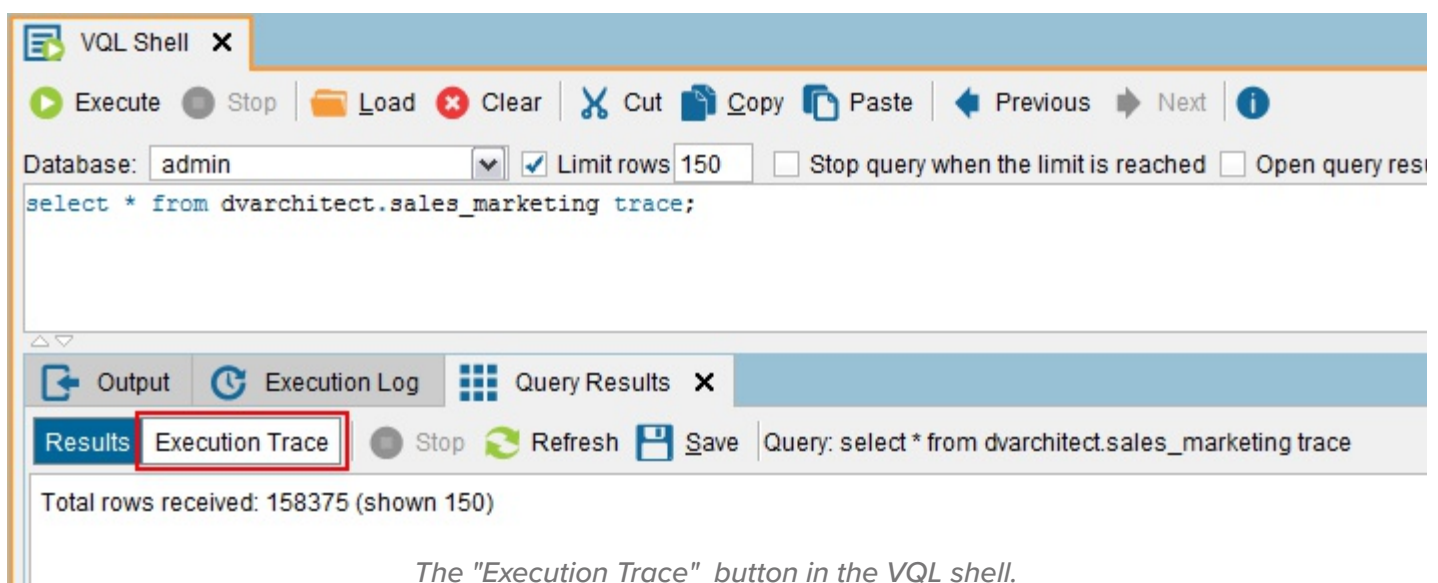
- After-the-fact: when the execution of a query finishes, we can access the trace of said execution; the way to access it depends on the way the query was executed
- If the query was executed graphically via the “Execute” button of a view, the trace will be available in the “Query results” tab, by clicking the “Execution Trace” button.

If the “Execution Trace” button is disabled, make sure that the option “Execute with TRACE” is selected in the previous screen.



The "Execution Trace" button in the query execution screen.

- If the query was executed through the VQL Shell, the trace can be examined by clicking on the "Execution Trace" button of the "Query Results" tab.



The "Execution Trace" button in the VQL shell.

If the "Execution Trace" button is disabled, you will need to add the keyword `TRACE` at the end of your query to make sure this information is recorded. For example, if you are running the query

```
SELECT * FROM myview;
```

modify it to

```
SELECT * FROM myview TRACE;
```

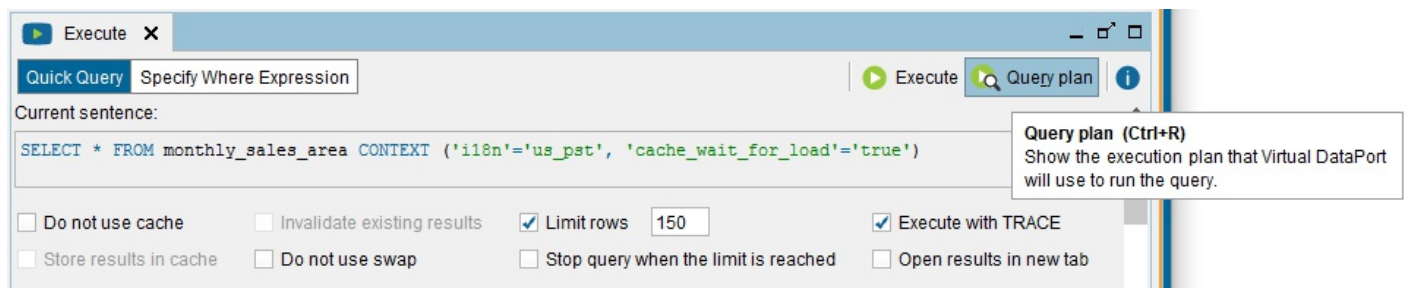
This will enable the "Execution Trace" button.

- Real time: we can also inspect the execution trace of a query in real time, while it is being executed. This is accomplished by opening the Query Monitor in the Virtual Data Port Administration Tool and clicking on the query we want to review. This will give us a snapshot of the current state of the query. We can tell the snapshot to be refreshed automatically by checking the option “Automatically refresh every” and specifying the refresh period in seconds.

As this method shows the execution trace of a query that is still being executed, there will be nodes that have not been executed yet.

Query Plan

These steps help us to get the actual execution trace of a query while or after it is run. The contents of this trace are a specific instance of the query plan with values related to that execution. In some cases, we might be interested in inspecting the query execution plan without actually executing the query (for example, because we might not want to wait for a long-running query, or we might not want to hit the source systems at that moment). We can do so by clicking on the “Execute” button of a view and then the “Query plan” button. Remember: this gives you a view of the execution plan of the query but not the details of a specific execution; we get a view of the template, but the details are not filled in. Regardless of that, it is always useful to take a look at the query plan if we cannot get the full execution trace.

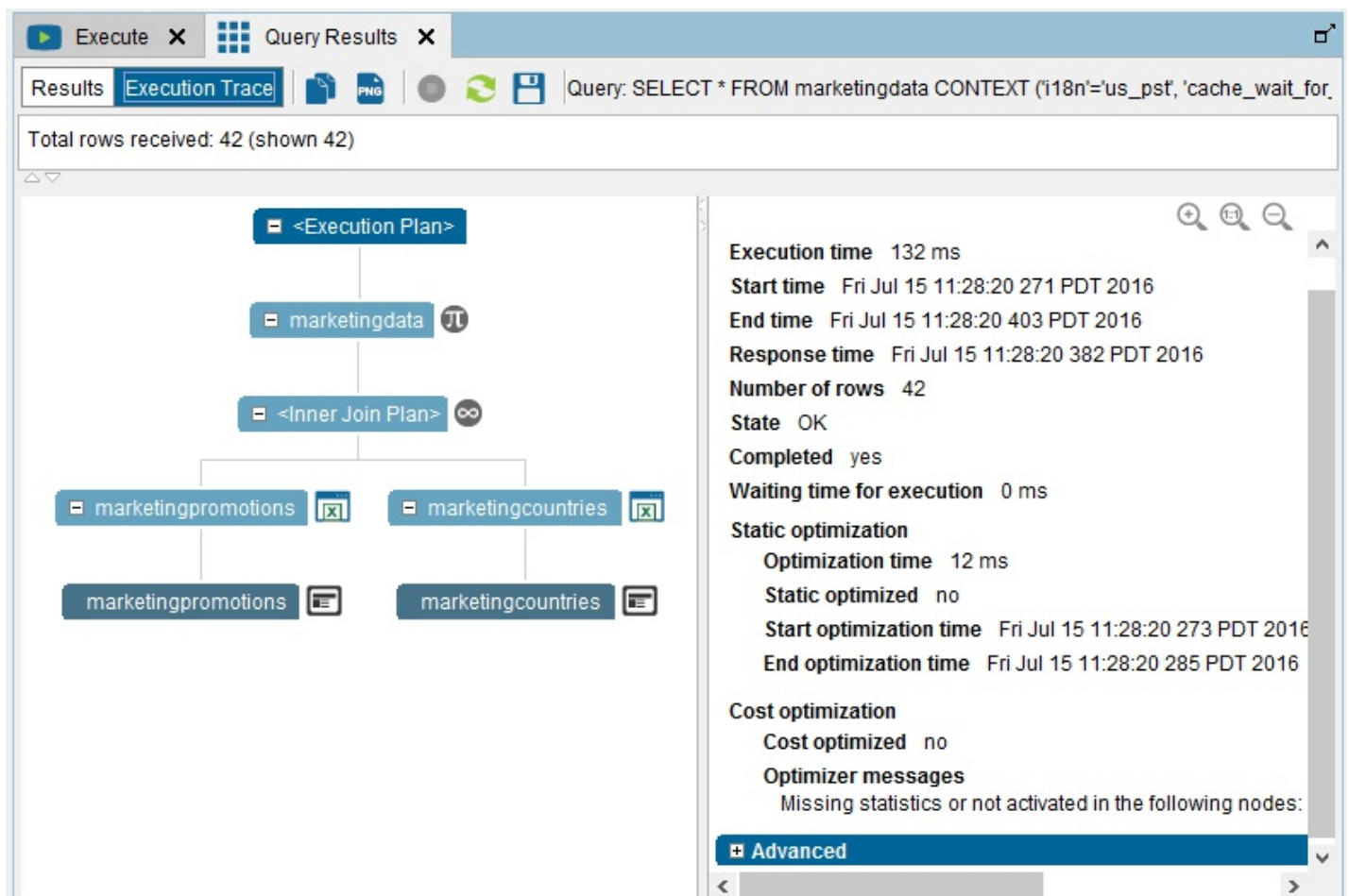


Accessing the query plan.

Exploring an Execution Trace

The graphical representation of the execution trace has two parts: the tree overview and the details panel. You should be very familiar with all the details in this screen, as they will be used across the whole book and will be of critical importance in your day-to-day interactions with the Denodo Platform.

The overview on the left side shows all the steps in the query plan and how they relate to each other. The top of the panel, at the root of the tree, shows a node that represents the whole query; when this node completes its execution, the query has finished. This node potentially contains several other nodes that it depends on, and those might contain other subnodes, etc. Each subnode represents a part of the original query (a subquery). If we continue our descent in the tree, we will eventually reach a leaf node (a node that does not have any children). Leaf nodes represent individual accesses to data sources (or the cache system, which is treated as yet another data source). If we step back to see the big picture, the Denodo Platform generates a query plan that consists of several accesses to data sources, combinations of those results, and then combinations of those combinations, etc., until obtaining the final result.



An example of a query execution trace.

Each node in the tree can represent a different type of operation, indicated by its icon and background color. In the [Virtual DataPort Administration Guide](#), you can find a comprehensive reference covering all the available types.

If you click on a node in the left side tree, the node will display its details on the right side panel. This panel shows all the information available about the execution of that specific node, which is critical for analyzing query behavior. The information is split in two sections, Basic and Advanced. Don't forget to expand the Advanced section for all the useful information for each node. The most frequently used values include:

- **Type:** The type of operation that this node executes (for example, a join, an aggregation, an interface view, etc.).
- **Execution time:** The node's total execution time. This enables you to search for bottlenecks in the execution of any query. Always check your assumptions about possible problems against the actual timing displayed in each node.
- **Execution timestamps:** Three timestamps are available: start, end, and response time, which mark the start of the execution of the node, the end of the execution of the node, and the time when the first result row was returned, respectively. The most common use for these is checking the difference between the start time and the response time.
- **Number of rows returned.**
- **State:** The status of the node. For example, the node may be running (PROCESSING), finished (OK), waiting for other nodes (WAITING), unable to finish (ERROR), etc.
- **Completed:** The panel will display Yes if the node has finished successfully. Otherwise, it will display No.
- **Search conditions:** The conditions used in this node. It's useful to review this information because the static and dynamic optimizations applied to the query plan could have modified the original conditions.
- **Filter conditions:** The same as search conditions.
- **Detailed row numbers:** A list of how many rows were discarded by the filter conditions, and how many were discarded because they were duplicates.
- **Memory details,** which are extremely useful for diagnosing slow queries:
 - **Memory limit reached:** This message will be displayed if the memory used by the node, for intermediate results, reached the limits configured for this view.
 - **Swapping:** If the panel displays Yes, intermediate results could not fit in the allocated memory space and were therefore stored in the physical hard drive, slowing the process down. If the panel displays No, then no disk swapping has occurred.

- **Number of swapped rows.**

Query Plans vs. View Trees

The query plan of a query is a tree-like structure, very similar to the view hierarchy of the view that is being queried. The two can match in their structure but they don't have to; the role of the optimization pipeline in the Denodo Platform is to identify opportunities for reducing the complexity and the number of executed steps, to improve query performance. The more the structure of a query plan matches the structure of the view being queried, the less optimization techniques have been applied to the query. An extreme optimization case occurs when a query is fully delegated to the data source, and the Denodo Platform does not perform any additional steps; we will see examples of this in the next section.

We have reviewed the big picture of what happens to queries that are received by the Denodo Platform and how to inspect the inner workings of the queries during and after their execution. Now we can start talking about the different factors that impact performance and how to use them to our advantage.

Delegation to Sources (Push-down)

The first important technique that the Denodo Platform uses for optimizing query performance is delegation to sources (or push-down). In this context, “delegation to sources” refers specifically to the delegation of processing (or operations) to the data sources; that is, making the data sources perform as much data processing as possible.

In a data virtualization scenario, several software elements can potentially have data-processing capabilities, at both the source level and the data virtualization level. The simplest case is a relational database that acts as a source for the virtualization layer—both systems have similar capabilities, so any operation on the data could be performed by either the source database or the data virtualization layer.

Suppose we have created a join between two tables in that database:

- One way to execute the join would be for the data virtualization layer to pull the data from both tables and then do the join in-memory.

- Another way to do it would be for the relational database to perform the join and have the data virtualization layer return the results.

In the second case, the join operation has been *pushed down* to the source. In general, any operation can be pushed down: relational operations (joins, unions, etc.), scalar functions (abs, upper, etc.), aggregation functions (avg, max, min, analytic functions, etc.), and more.

In order for an operation to be pushed down to the data source, it must meet two requirements:

1. The data source must have the capability to support the operation. This means that if the data virtualization layer is to instruct a source to execute an operation over the data, the data source must be able to carry it on. Data processing capabilities vary across different types of data sources, for example:
 - Traditional databases and data warehouses offer large sets of operations so in general they will be able to support a high degree of delegation.
 - Web services usually offer a much more limited set of operations. Most often, they just support selecting data from them, along with some limited filtering capabilities, but without the richness that relational databases allow through SQL.
 - Flat files, such as XML or CSV files, don't accept any delegation whatsoever.
2. The operation must be performed over a data set that is contained in a single source. This means that all operations must be performed on data that is already residing on that source, a source with visibility over the whole set of data being processed by the pushed down operations (this rule has an exception, which will be explained in the section that deals with data movement).

The Benefits of Delegation

Generally speaking, delegating operations to data sources will improve query performance. This is achieved through a variety of mechanisms:

- Less data being transferred over the network:
 - Fewer columns: Projection operations are usually included in the queries received by the data virtualization layer—often, client applications want only a subset of the columns present in the views/tables involved in the query. Pushing the projection to the source system cuts down the amount of data that is transferred and then discarded at the data virtualization level.

- Fewer rows: Queries usually include selection conditions that cut down the number of rows in the final result set. Aggregation operations, which are commonplace in analytical scenarios, vastly reduce the number of result rows compared with the source data sets. Moreover, on average, join operations produce fewer rows than the combined source tables, due to a low number of matching rows; although in some cases joins produce more rows—the worst example being a cross join—it’s not the most common situation. Delegating those operations to the source system will result in less data sent over the network.
- Source systems have more intimate knowledge of low-level details about the data, which helps in performing the operations with better performance. Source systems have a detailed understanding about the physical distribution of the data as well as details about the performance of physical storage regarding I/O, local indexes of data, status of memory buffers, etc. Each source system will also have its own query engine and optimizer, which will incorporate those additional insights into the system to further optimize each query sent by the data virtualization layer. So it makes sense, from the performance side, to send as much work as possible to the source systems.

Showcase: The Performance Effects of Delegation

Let’s see a concrete example of the performance benefits we get by using an optimizer with a “delegate first” orientation. Assuming the following data model:

- Table A with 100 million rows.
- Table B with 1 million rows.

We issue the following query:

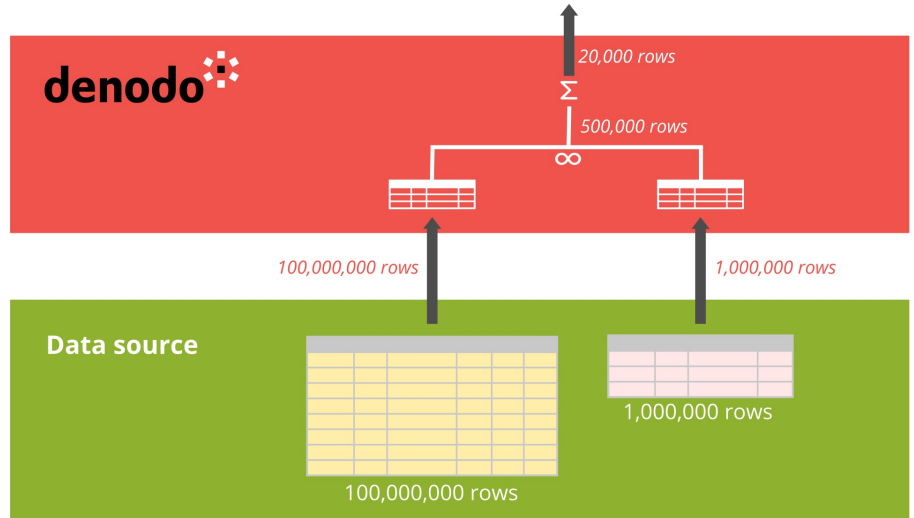
```
SELECT sum(value) FROM a JOIN b ON a.bid = b.id GROUP BY
a.somefield;
```

With the following assumptions:

- The number of matching rows in the join is in the order of 500,000.
- The results of the aggregation produce around 20,000 rows.

In a scenario with no delegation, for example, with both tables stored in different physical systems, the naive approach would be to:

1. Pull all the data from table A into the data virtualization layer (100 million rows).
2. Pull all the data from table B into the data virtualization layer (1 million rows).
3. Perform the join in memory in the data virtualization layer (500,000 result rows).
4. Perform the aggregation in memory in the data virtualization layer (20,000 result rows).

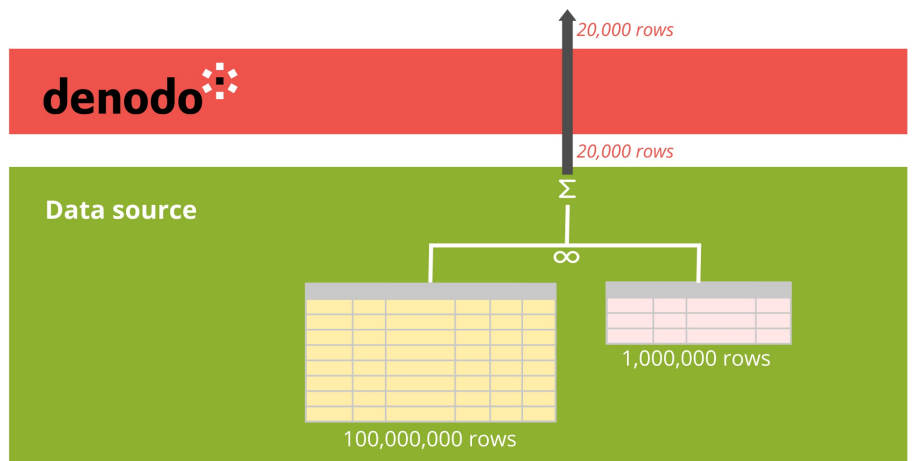


An unoptimized scenario without delegation.

With those steps, the data virtualization layer has transferred a total of 101,020,000 rows (including the 20,000 sent to the client application), and has performed a 100 million x 1 million join, in memory, and a further aggregation.

If those two tables were stored in the same relational database, the situation would be very different:

1. The full query can be delegated to the data source, including the join and the aggregation.
2. The results of the query would be transferred from the data source to the data virtualization layer (only 20,000 rows with a single column).
3. The results of the query are streamed by the data virtualization layer to the client application (20,000 rows with a single column).



An optimized scenario with full query delegation.

In total, we have transferred only 40,000 rows with a single column, so the benefits of this approach should be evident.

Note that in a real-world scenario, the optimizer would try to push the aggregation to the source, even if the join operation cannot be delegated, greatly improving performance. This will be explained in the “Static optimization” section of this book.

Optimizing for Delegation

The Denodo Platform is always looking for opportunities to delegate operations to source systems; it is one of the most important parameters guiding query optimization decisions. It is so important that it takes precedence over other considerations when choosing among the generated query plans. This means that sometimes the optimizer makes choices that may seem suboptimal at first but might make sense when taking into account the full picture. For example, the query engine might ignore a cache of a view in very specific scenarios if it results in a higher degree of delegation to the source.

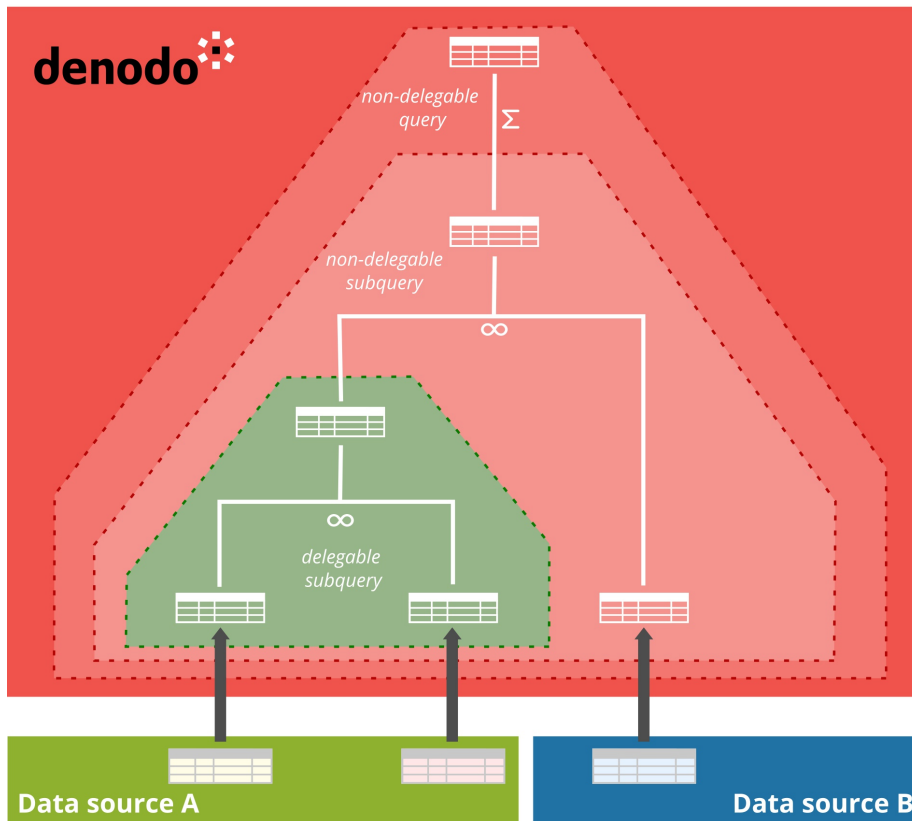
When designing queries and view hierarchies, the data architect or developer should always keep this in mind, and thus try to always group operations affecting each data source as closely as possible to maximize opportunities for delegation to those sources.

Always be on the lookout for spotting delegation opportunities. This should be at the top of your list of possible avenues for optimization.

DELEGATION THROUGH CACHING

There are multiple scenarios that prevent operations from being delegated. In a nutshell, any scenario that contains a violation of any of the two requirements for delegation that we defined earlier:

- If the data source does not include support for the operations being executed, then the query or subquery cannot be delegated, or if the data source is a passive system (such as a flat file) it cannot execute any operation on its own.
- If a query or subquery references tables that exist in different data sources it cannot be delegated, so it must be split into subqueries that are small enough so that they only reference data in a single source system; the exceptions to this are data movement and reordering of operations so that they can be delegated, and both of these topics will be discussed later in this book.



Subqueries are delegable only if they use base views from a single data source.

There is a way of circumventing these issues by using the cache (another possible alternative is to use data movement, which will be explained later). The Denodo Platform uses an external relational database as a cache system and this can solve both issues:

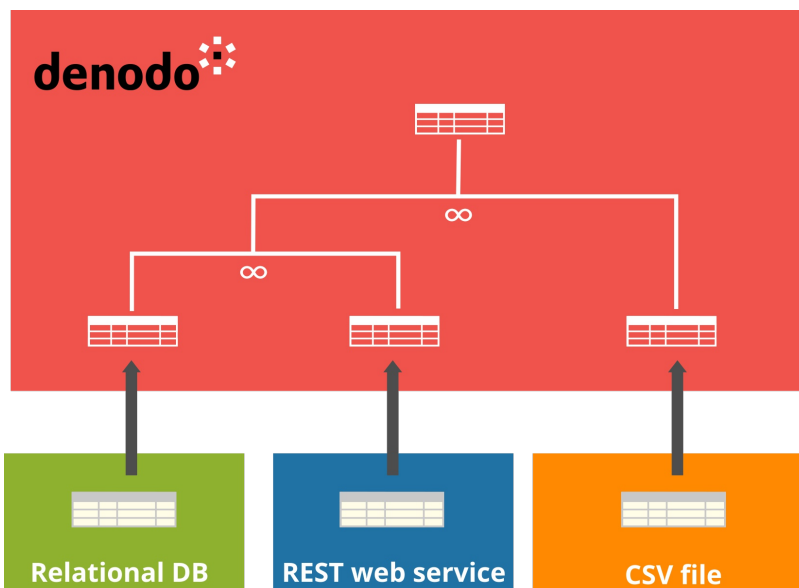
- The cache is a relational database. This means that the number of operations that are delegable to the cache is very high, as the native capabilities of relational databases are very rich. These systems are at the extreme end of the source capabilities spectrum, with passive systems being at the opposite end.

- The cache is always shared for all the views of a given virtual database. This means that the cache for a given virtual database is all stored in a single schema on the cache system, which in turn means that the data in the cache can be used in combined operations by the cache relational database.

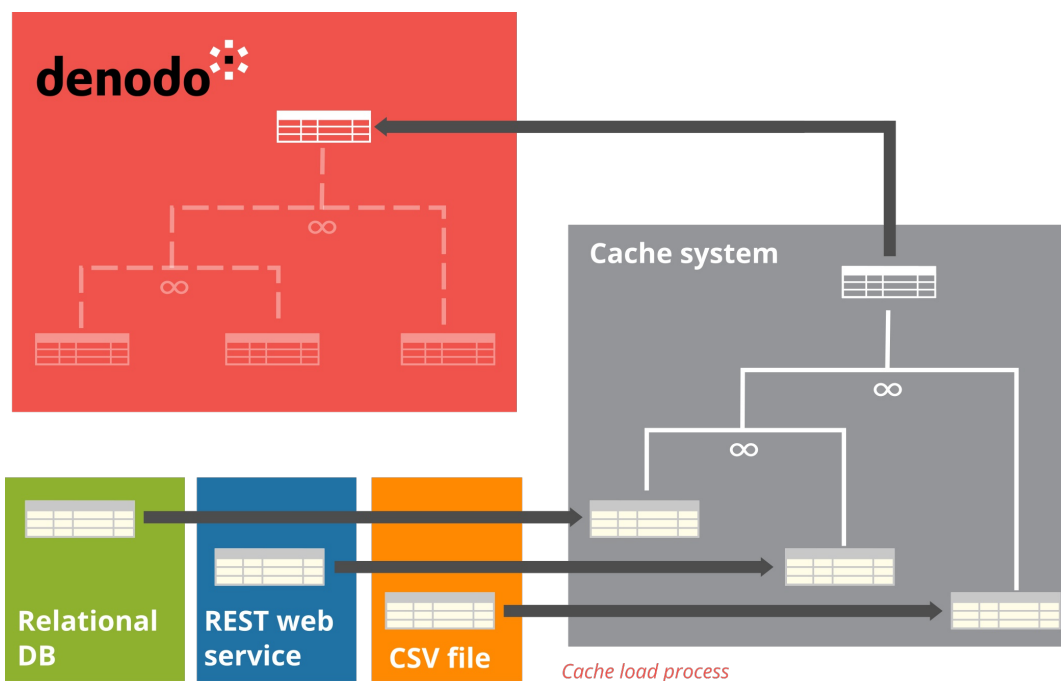
These two features combine in a way that enables the data architect to cache views from different data sources in a tactical way, so that when operations that would not be delegable using the naive approach (because they involve views over different data sources and/or they are data sources with non-delegable capabilities) become delegable because:

1. All the views involved in the query are stored in the cache so they can be combined together (circumventing the original non-locality of the data).
2. The cache relational database provides sufficient execution capabilities to run all the operations involved in the query (circumventing the reduced capabilities of the original data sources).

The only requirement for this to work is that the cache mode for the views involved in the queries must be set to Full. Partial cache mode does not allow general delegation of operations to the cache, only a small subset, such as projections and selections. Basically, it allows just the additional filtering of data sets that are already in the cache, but no combination of those data sets.



A non-delegable query over three different data sources.



All three sources have been cached in full mode, and the query can be fully delegated to the cache.

Joins

The join operation is arguably the most important operation defined in relational algebra, next to the selection operation. It is not surprising, then, that it potentially has the biggest impact in query performance in data virtualization scenarios, alongside query delegation. These two topics should be the starting point in any query performance diagnostics carried on by a data architect.

This section of the Cookbook will cover the most important aspects of join operations and how they relate to optimization scenarios in the Denodo Platform: the different join methods, how they relate to the data sources, how to inspect and modify them, and the different types of joins.

Join Methods

The most important aspect of join operations is the method used to execute the join. When a user issues a query that joins two views, the query only specifies that the two datasets have to be combined, without specifying exactly how. This is due to the declarative nature of SQL; the user specifies what is needed, and it is the responsibility of the query engine to generate a query plan with the specific steps for fulfilling the user's request. There are several ways to execute a join, so let's review all the options provided by the Denodo Platform, and the rationale behind each one.

Choosing the best join method automatically is one of the main tasks of the optimizer. You should always gather the statistics of the views in your project and enable the cost-based optimization in your server, to enable the optimizer to use the best join method and join order for each query.

The join methods described below are not specific to the Denodo Platform; they are the standard methods that traditional relational databases use to execute joins, so each is covered by extensive literature, describing their advantages and disadvantages and when they should be applied.

HASH JOINS

The hash join operates by building a hash table of the join attribute(s) of all the rows on the right side of the join and then checking for matches in the table for each hash value on the left side. The process works as follows:

1. The table on the right is queried and its results are obtained.
2. A hash table is built, using, as keys, the values of the join attributes for each of the rows obtained from the right side.
3. Once the table is built, the left side of the join is queried.
4. For each row received from the left side, its join attributes are used to build a key that is used to check for occurrences in the hash table built in step 2.
 - a) If the key does not exist in the hash table, the row is discarded.
 - b) If the key exists in the hash table, the join between the row on the left side and all the rows from the right side that share the same key is performed, and those new joined rows are added to the result set of the join.
5. The process continues until all the rows from the left side have been processed.

The most important aspects of this process, from the standpoint of performance, are:

- The whole data set from the right side is queried and processed before the query on the left side begins.
- Once the right side is ready, the rows from the left side are streamed from the source, compared to the hash table, and then returned. This happens row by row, so as soon as the right side has been finished, the join is potentially ready to start returning rows, and the number of rows per second returned will be only dependent on the speed the rows are retrieved from the left side. The join will never return any row before the right side has finished.
- The memory required for this type of join is directly proportional to the size of the result set from the right side, because before the join can return the first row, it needs to build a hash table of all the rows on the right side.

Those three factors guide when hash joins should be used and when they should be avoided:

When to use a hash join	When to avoid using a hash join
The right side of the join returns a small number of rows.	The data set on the right side of the join contains a large number of rows.

Remember that the optimizer can reorder the branches of a join when using statistics and cost-based optimization, so even if a small table is put on the left branch, the optimizer can swap it with the other branch so that the hash join can perform well.

One important aspect we mentioned in the previous description is the difference between operations (or parts of operations) that can operate on individual rows versus operations that need the full data set before continuing. In our example, the task of building the hash table is of the second variety: it blocks the whole join operation until it has been completed. On the other hand, the processing of the left branch belongs in the first category: once the hash table is built, each row from the left side can be processed individually and returned as part of the join results (if it finds a match in the hash table). We will review this concept in more depth later, as it is very important when analyzing performance; keep your eyes open so you can spot these two kinds of operations throughout our discussion.

A note about join ordering: The description of the hash join, as explained here, applies to joins that use the natural order. That is, they execute the hash join by building the hash table on the right branch of the join. If you run into a situation where a hash join seems like the right choice but your small dataset is on the left side instead, then you can still use a hash join effectively by configuring the join to use reverse order. In that case, the join will build the hash table on the left branch and then stream the rows from the right side.

MERGE JOINS

Merge joins work by doing a simultaneous linear scan of both branches of the join. The process works as follows:

1. The merge join operation requests the data from both branches of the join to be sorted by the join attributes.
2. The sources start streaming rows to the merge join.
3. The merge join takes all the rows from the left side that contain the first value for the join attribute. This could be a single row or a set of rows that have that value. We will consider the case of just a single row for this (imagine a join using a primary key attribute), but the concept is easily extensible to multiple rows with the same join attribute value.
4. The merge join takes all the rows from the right side that contain the first value for the join attribute. Let's again assume this is a single row.

5. The rows from the left and right side are compared
 - a) If the values of the join attribute match, then the joined row is returned as a result of the join. The merge join then draws new rows from both branches of the join.
 - b) If the values of the join attribute don't match, then the row that has the smallest value for the join attribute is discarded and a new row is taken from that branch of the join.
6. Step 5 is repeated until we run out of rows in one of the branches of the join.

To understand why this operation works, remember that both branches are sorted by the join attribute, so any time we draw a new row, we are increasing the value of that attribute. When the values of the join attribute from both sides of the join don't match, then we draw rows from the side that has the smallest value, thus eventually increasing that value, until they both match (or until the side that was smaller becomes bigger, in which case the merge join starts drawing rows from the other side to try to balance the values on both branches and find a match).

This explanation only considers situations with non-repeating join attributes: the process described above draws rows one at a time. In the real world, we will often find a join between two tables where there are many rows that share the value of the join attribute. The extension of the algorithm for calculating the merge join in this case is simple: Instead of drawing a single row, the merge join draws all the rows with the same value for the join attribute, and when a match is found instead of joining two rows, the merge join performs a cartesian product between the partial sets pulled from both sides of the join (matching each row from the left side with all the rows from the right side) and returns all those rows as partial results of the join.

From a performance point of view, the behavior of this join has the following implications:

- The merge join requires both sides of the join to get their data sorted. This implies that the source systems or subqueries on both sides of the join need to 1) be able to sort the data and 2) will potentially have a slight performance penalty due to this sorting operation. The exact magnitude of this impact will depend on the source system, but in general, relational systems that have indexes defined over the appropriate columns will demonstrate high performance during this operation.
- Once the results start coming in from both sides, the merge join does a simple linear scan of both tables, with very simple comparison operations at each step. This is a very lightweight process that offers excellent performance.

- The merge join operates on the streaming operation model: As soon as some rows that match the join condition are found, they are returned as results of the join. So if there is a downstream operation over these results, it can start its execution even before the merge join has finished.
- The merge join offers the best performance versus other types of joins, when there is a need to join two extremely big data sets.

The guidelines for using this type of join are:

When to use a merge join	When to avoid using a merge join
Sorting both branches of the join is an acceptable performance tradeoff.	Sorting both branches of the join would have too big an impact to performance.
Both branches of the join have big data sets.	At least one of the data sets is small.

One last consideration: Merge joins are symmetrical so they are not affected by the order that the join specifies (as it was the case with the hash join).

NESTED JOINS

Nested joins work by querying one branch of the join first and then querying the other branch based on the results of the first branch. In detail:

1. The left branch of the join is queried and the results are obtained.
2. For each row received from the left branch a new query is issued to the right branch, adding a `WHERE` clause to select exactly the rows that match the join condition derived from the value coming from the left side. For example, if we receive the `id_right` 7 from the left side (assuming the join condition is over the `id_right` field), then the right branch will be queried with a condition `where id_right = 7`.
3. The results of the query on step 2 are matched to the results of the left side.
4. The results of step 3 are returned as results of the nested join.
5. The process continues until there are no more results from the left branch.

This means that nested joins make sure that 100% of the rows that come from the right branch are returned as results of the join, given that they are specifically selected to fulfill the join condition. This is in contrast with other types of joins, which select extra rows from both branches and then discard them as needed.

Nested joins are very common in scenarios where the right side imposes a restriction on the types of queries that can be accepted; the most typical example is a web service that has an input parameter. In that case, you can think of the nested join as first getting rows from the left branch and then feeding those one at a time to the right branch to meet the query capabilities of the right branch.

In the description of how the nested join works, the right side was queried once per row on the left side. In reality the Denodo Platform optimizer tries to batch the calls to the right side when possible to reduce the number of calls as much as is feasible. The exact details on how this batching is done depends on the specific data source or view tree that is on the right side—this happens transparently when querying relational databases and LDAP data sources. Common techniques that are used by the optimizer to specify several values on a single query are:

- OR conditions
- IN clauses
- Common table expressions
- Using inner joins with a subquery that selects the specific values

The size of the blocks on the right side when batching is controlled by the property

```
com.denodo.vdb.engine.wrapper.vdb.BindingJoinAccess.JDBCdefaultBlock  
Size
```

Nested joins are a streaming operation: As soon as the left branch of the join returns a row, the join can query the right branch, and as soon as the right side rows for that first query are returned, the join will begin to return results.

When to use a nested join	When to avoid using a nested join
The right branch has source constraints and cannot answer free-form queries, instead forcing the use of input parameters.	Any time a merge or a hash join can be used.
The left branch of the join has relatively few rows.	
Getting individual rows or groups of rows from the right branch is preferable to doing a full or almost full table scan.	

Nested joins are asymmetrical: The left and right branches of the join have different roles, so make sure that your data sets and data sources match the patterns described in the nested join requirements; if the roles in your data are switched, remember to specify `REVERSE` order in your join definition.

Nested joins offer an additional layer of performance optimization: When the join returns only rows from the right branch, the optimizer identifies the nested join as a semi-join and further optimizes the execution by not trying to match the results from the right side with the results from the left side, instead just shipping the results from the right branch as results of the join.

NESTED PARALLEL JOINS

A nested parallel join works in the same way as a nested join, but instead of executing every query to the right branch sequentially, it executes them in parallel, with a maximum of concurrent requests that is configurable by the user.

When to use a nested parallel join	When to avoid using a nested parallel join
Use it over a regular nested join any time it is possible.	The source on the right side cannot cope with the load imposed by a large number of parallel iterations.

Join Methods and Data Source Query Capabilities

As you have seen in this description of the join methods, the optimizer has three main options when deciding what type of join to use. These options are not equivalent, and as explained in the previous section, each one makes different assumptions about the data on each branch of the join.

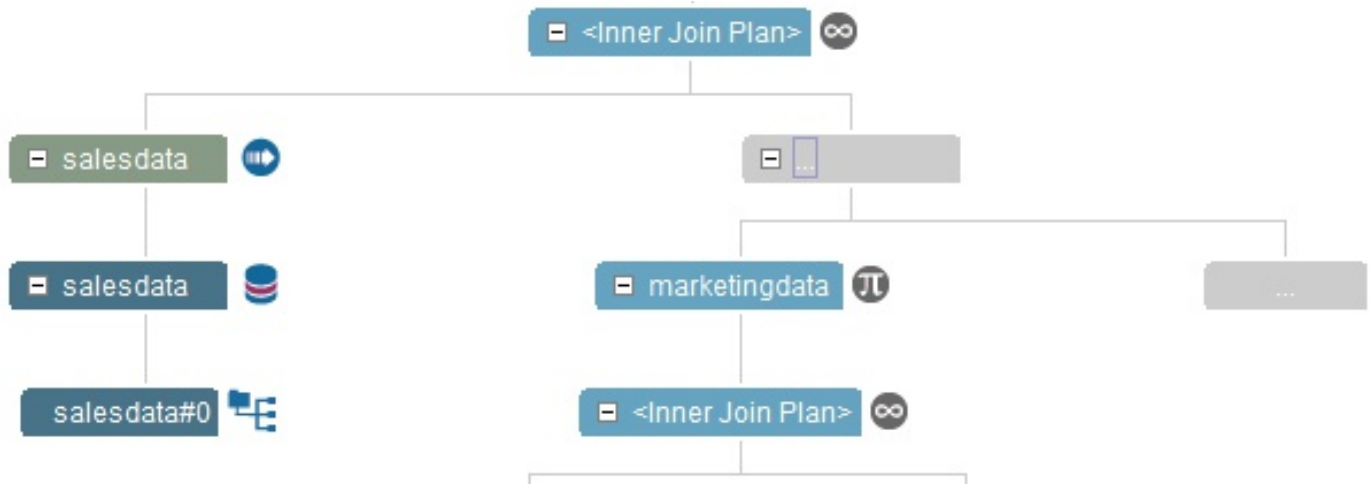
A very important consideration is the querying capabilities of the underlying views on each branch of the join. When the optimization process decides what method to use, the available options are restricted by the capabilities of those views; the most typical case is a view with mandatory fields (such as a view representing a SOAP web service operation with a mandatory input field), which does not accept a `SELECT *` query (it requires a `WHERE` clause with a condition setting a value for the mandatory fields). In this situation the only available join method will be a nested join, because that type of source cannot retrieve the full underlying data set in a single query, and so it cannot be used in a merge or hash join. Moreover, because the view specifies input parameters, it will always appear on the right side of the nested join (the side that receives multiple queries), never on the left side (which requires a naked `SELECT *` without a `WHERE` clause). By having a single restriction on the query capabilities of the underlying views, the optimizer's usual room for maneuvering gets reduced, and the parameters for the join get automatically fixed. Other similar situations may include capabilities related to sorting data. These might come into play, for example, when the optimizer is presented with the option of executing merge joins.

Inspecting the Join Method Chosen by the Optimizer

Checking the join method chosen by the optimizer is easy; inspect either the query plan (before the query execution) or the execution trace (during or after the query execution), and click on the node of the execution plan that represents the join that you want to check. The right panel will display the method used for that particular join execution, in addition to other detailed information. Remember, this automatic selection is highly dependent on the optimizer having access to the statistics of the views being queried!

NESTED JOINS IN QUERY TRACES

Nested joins will appear slightly differently than other join methods in the execution trace of a query, as they potentially execute many queries on the right branch of the join. This is what you will see on the query trace when you execute a nested join:



An example of a nested join in a query trace (tree detail).

Type PROJECTION

Execution time 57 ms

Start time Fri Jul 15 12:00:09 052 PDT 2016

End time Fri Jul 15 12:00:09 109 PDT 2016

Response time Fri Jul 15 12:00:09 109 PDT 2016

Number of rows 1

State OK

Completed yes

+ Advanced

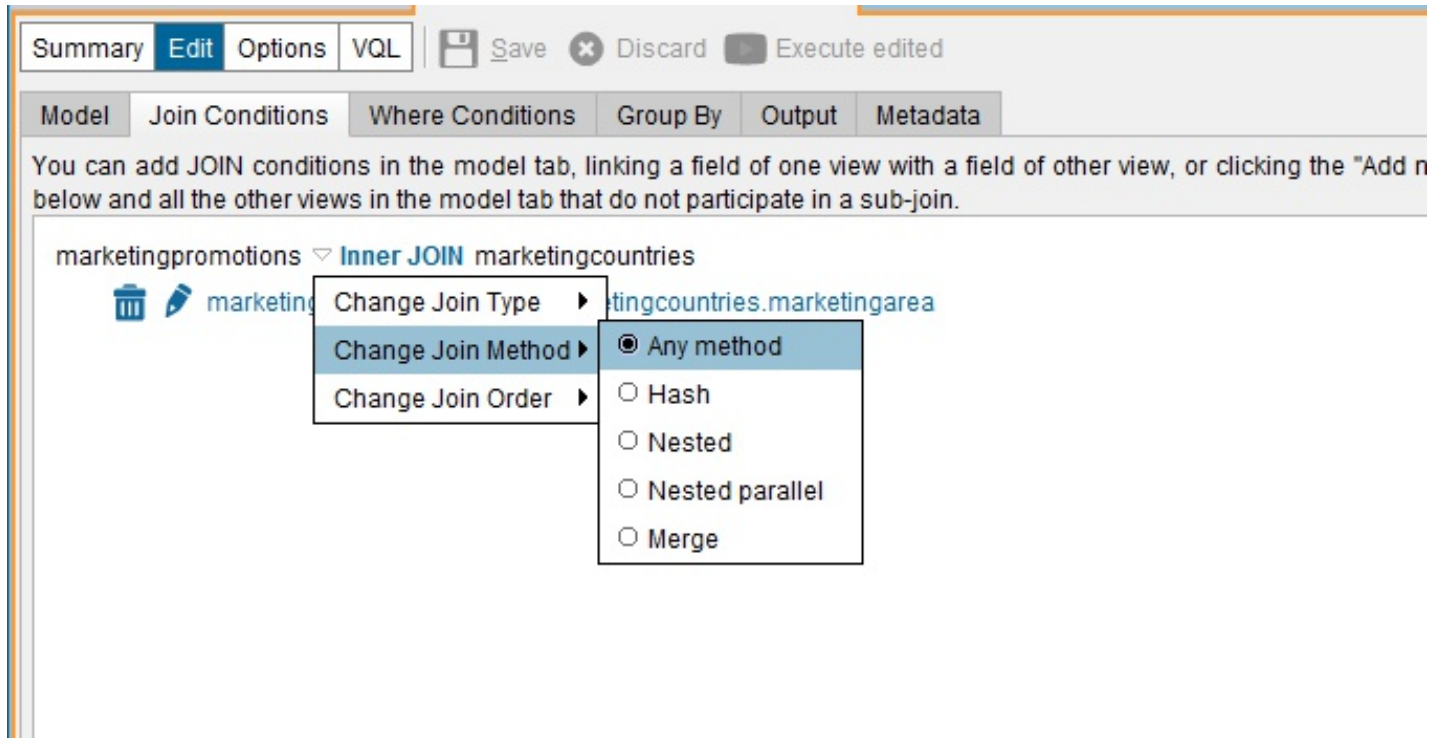
The remaining 383 subplans are not shown

An example of a nested join in a query trace (advanced information).

Notice that this will not happen when inspecting the query execution plan (pre-execution), only in the query execution trace (post- and during execution). The query plan does not know the results that will be obtained from the left branch of the join before execution time, thus it cannot predict the exact number of queries that will be issued to the right side (although it can estimate the number based on view statistics as will be shown later). Remember, the query execution plan and the query trace are related, but they do not convey the same information.

Manually Specifying the Join Method

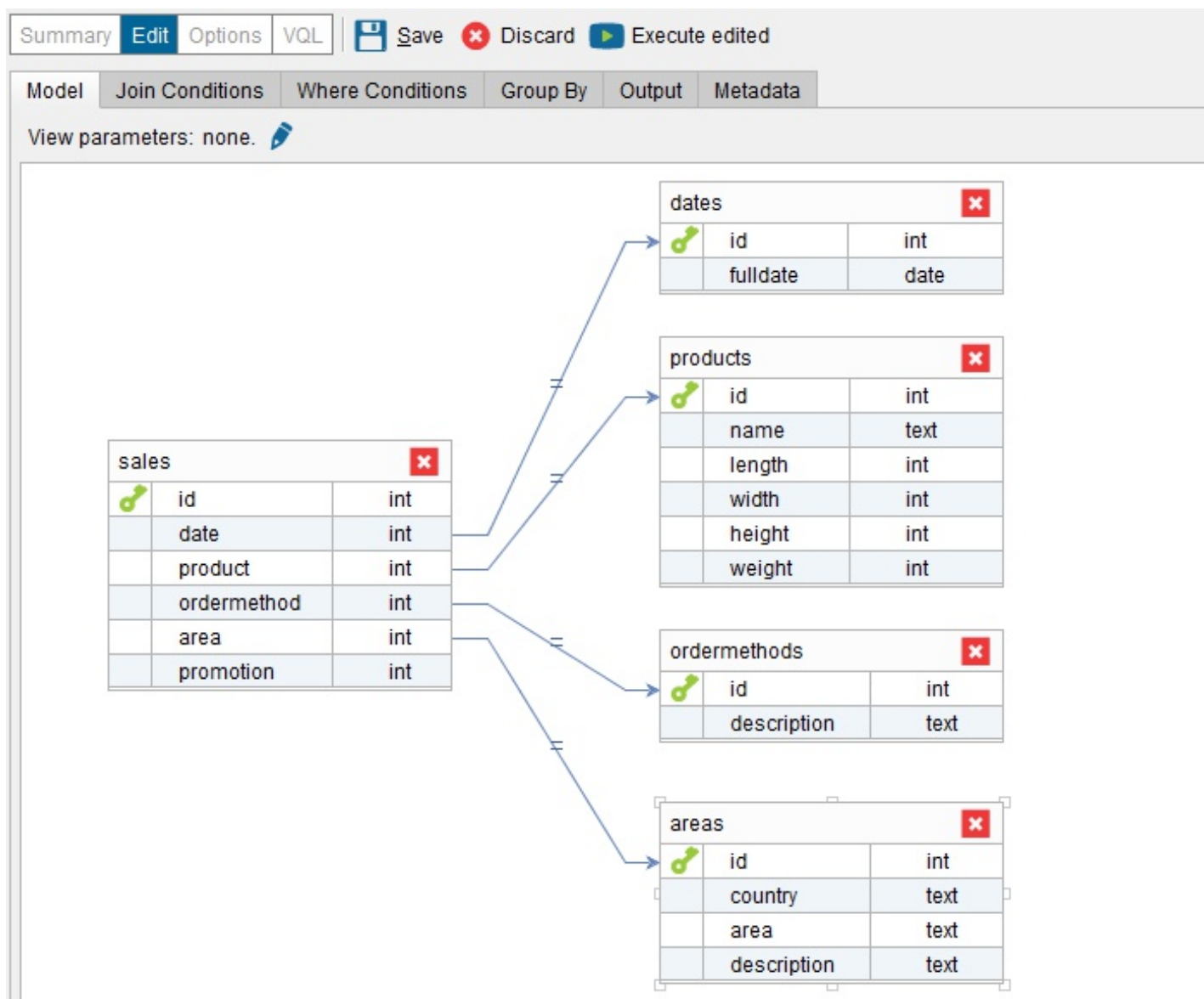
Denodo Platform users can choose to let the optimizer do its job and select the best join method for each query (or what the optimizer believes to be the best method). However, the optimizer may not always choose the most appropriate method for every situation. If this happens, the user can override the default option and provide an alternative to be used at runtime. In that case, the join would not be further optimized, and it would be left as the user specified. This setting is found in the “Join Conditions” tab of the join operation.



Manually specifying the join method.

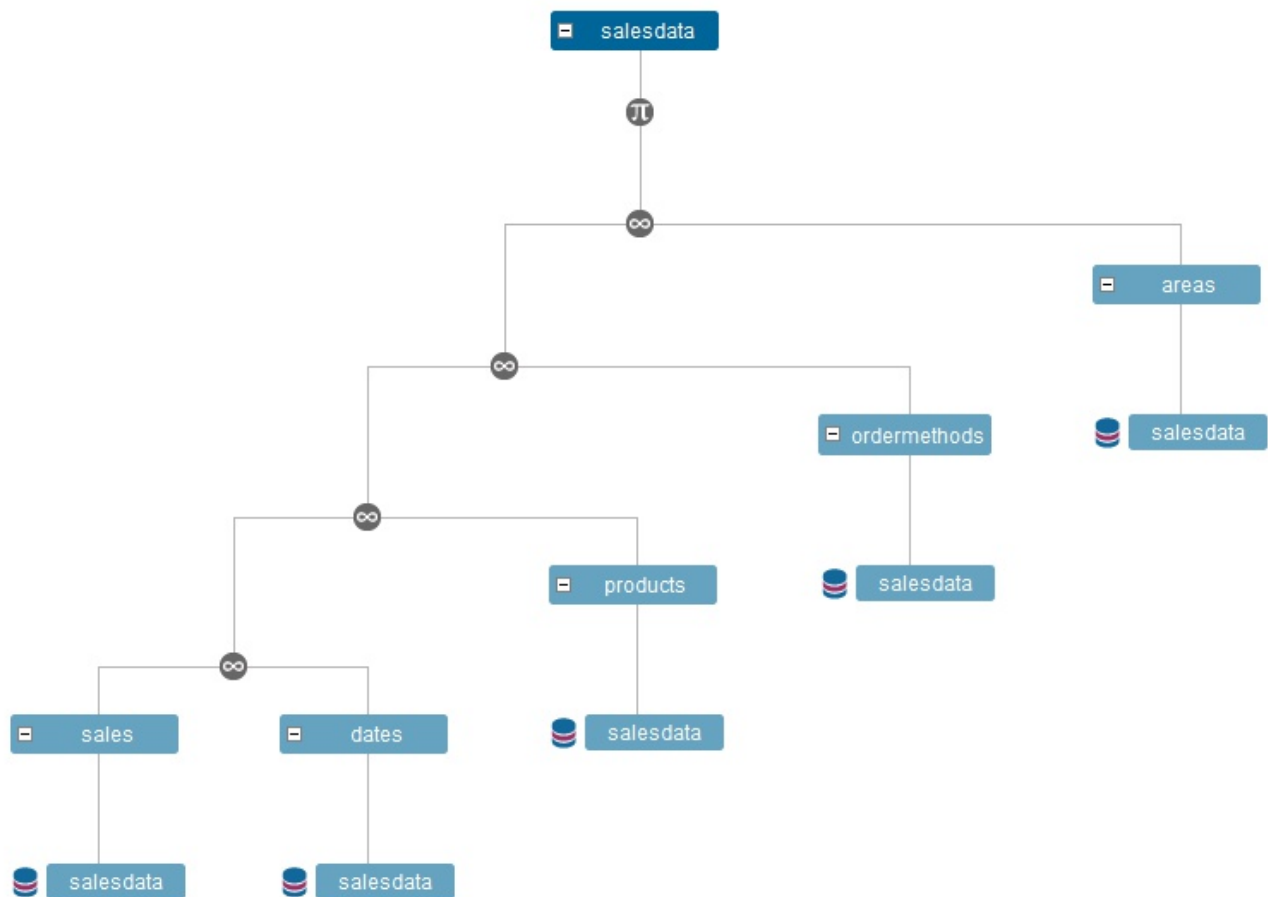
N-joins

N-joins are joins that are performed over more than two views. When we create an n-join, we drag all the views that take part in the join and then connect them by dragging fields from one view to another to specify a join condition between those two views. As the conditions are defined exclusively between two views, we need several join conditions to connect all the views involved in an n-join (at least N-1 conditions if we have N views). This results in a join between several views but with somewhat of an internal structure. The join will be always interpreted by the Denodo Platform as a sequence of two-way joins.



Creating an n-join.

Having multiple consecutive 2-way joins offers the optimizer an opportunity for reordering them to increase performance. For example, if we have a 5-way join, in which four of the views come from the same source database, the optimizer will probably reorder the join so the first three pairs of views that are joined are the ones that come from the database, and the last view joined will be the other view that comes from a different source; this would enable the optimizer to delegate the join between the first four views to the source system, increasing performance as we have seen before.



View tree for an n-join.

Join Types

INNER JOINS

Inner joins represent the most basic type of join, in which only the rows that match the join condition are returned from both sides of the join. This is the type of join that is selected by the optimizer when possible, as it is the most performant, and it returns the least rows. The optimal join type to use is of course determined by the specific use case more than the performance considerations, as the semantics of using an inner join and an outer join are different (the result sets of both operations are different) but under some circumstances they can become equivalent; in these cases the static optimizer will try to convert outer joins into inner joins.

OUTER JOINS

Outer joins produce a superset of the results of an inner join. Depending of the specific type of outer join, it will include all the results of the inner join, plus:

- In the case of a left outer join, all the non-matching rows from the left side of the join.
- In the case of a right outer join, all the non-matching rows from the right side of the join.
- In the case of a full outer join, all the non-matching rows from both the left and right sides of the join.

Outer joins are worse than inner joins from the performance standpoint, both because they return more rows than inner joins and because of the mechanics of calculating the result sets. The optimizer of the Denodo Platform server will always try to transform outer joins into inner joins using static optimization techniques that will be discussed later.

CROSS JOINS

Cross joins, or “cartesian products of two views,” generate all the possible combinations of rows from the rows coming from each branch of the join. If we have N rows on the left side and M rows on the right side, the join will return NxM rows—it’s very important to always use cross joins judiciously because that NxM resulting data set can grow very quickly even when the source views have small data sets. For example, if we do a cross join between two 10,000-row views the result set will have 100,000,000 rows. We have moved from a situation with two small views into a situation with a respectable data set; depending on the intended usage of this hundred million rows, the performance of the query could be extremely poor. Always use cross joins as the last resort!

The Denodo Platform optimizer will also try to avoid cross joins in queries. For example, reordering join operations for maximizing delegation to the data sources may produce cross joins between views; in these cases the optimizer will avoid reordering as this would probably perform worse than the original query.

JOIN TYPE VS. JOIN METHOD

The type of join defines the data set that will be returned as the result of the join operation: which rows are included and which rows are discarded. The join type is orthogonal to the method used to execute said join; for example, an inner join could be executed using a hash join and a left outer join could use a merge join, or vice versa. Any combination of join type (inner, outer, cross join) and method (merge, hash, nested, nested parallel) is possible.

Memory Usage

Memory is a critical topic when discussing the performance of a Denodo Platform solution. Each Denodo server will have a memory pool, assigned by the operating system, which represents the absolute maximum memory that the process can consume. It is therefore very important to recognize patterns in memory usage that can be optimized to reduce the footprint of each query so the server can execute more concurrent queries.

Query Complexity vs. Dataset Size

When faced with the question of “how much memory will this query need?” most novice users try to answer by looking at the size of the source data sets. Intuitively a query that processes 1 billion rows should need more memory than a query that processes 1 thousand rows. Unfortunately this simple approach is wrong; the correct way to estimate how much memory a query will need is to look at the query’s complexity.

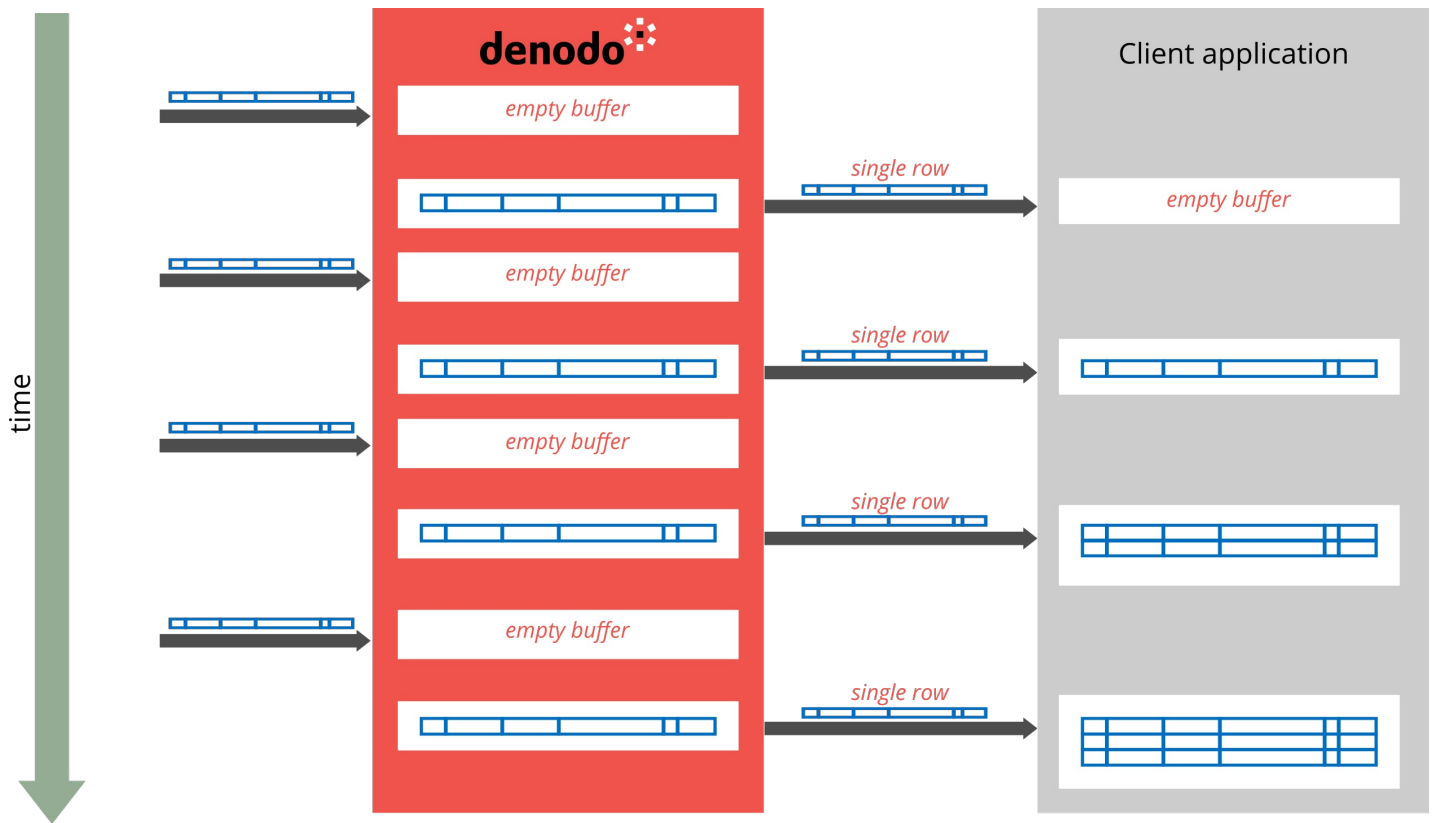
From the standpoint of CPU load, the most important consideration is the computational cost of the functions that are applied to each individual cell.

From the standpoint of memory consumption, the most important parameter is the type of operations that we execute over the data, splitting them into operations that stream tuples vs operations that hoard tuples in memory. Let’s take a look at each of those.

ROW SHIPPING VS. ROW HOARDING (STREAMING VS BLOCKING)

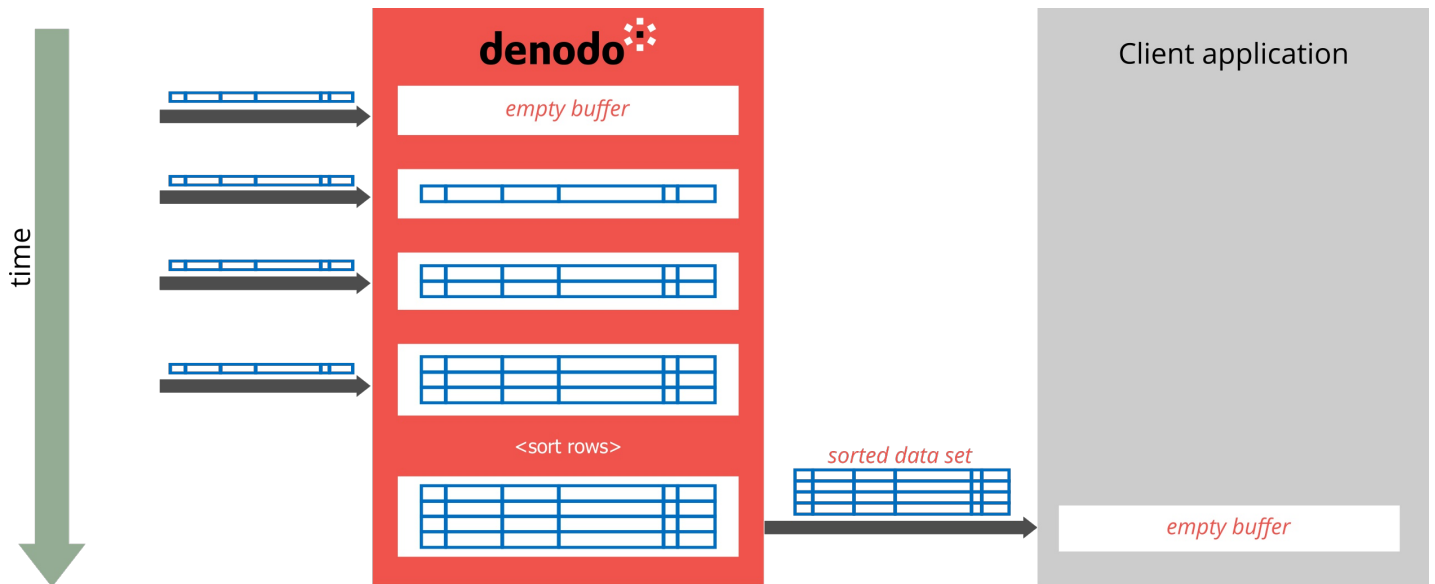
When we need to process a data set, operations can behave in two ways:

- First, operations can be applied over each row of the source data set individually, so the result of the operation over a single row only needs that row to be calculated. For example, if we are projecting a subset of the columns of a dataset, we can grab the desired columns from each row individually. Another example is a selection based on a condition over the columns of the dataset—when we grab a single row, we can check whether that row verifies the condition or not; we don’t need other rows to know that. This type of operation is called a **streaming operation**. These operations ship rows individually to the next stage of execution.



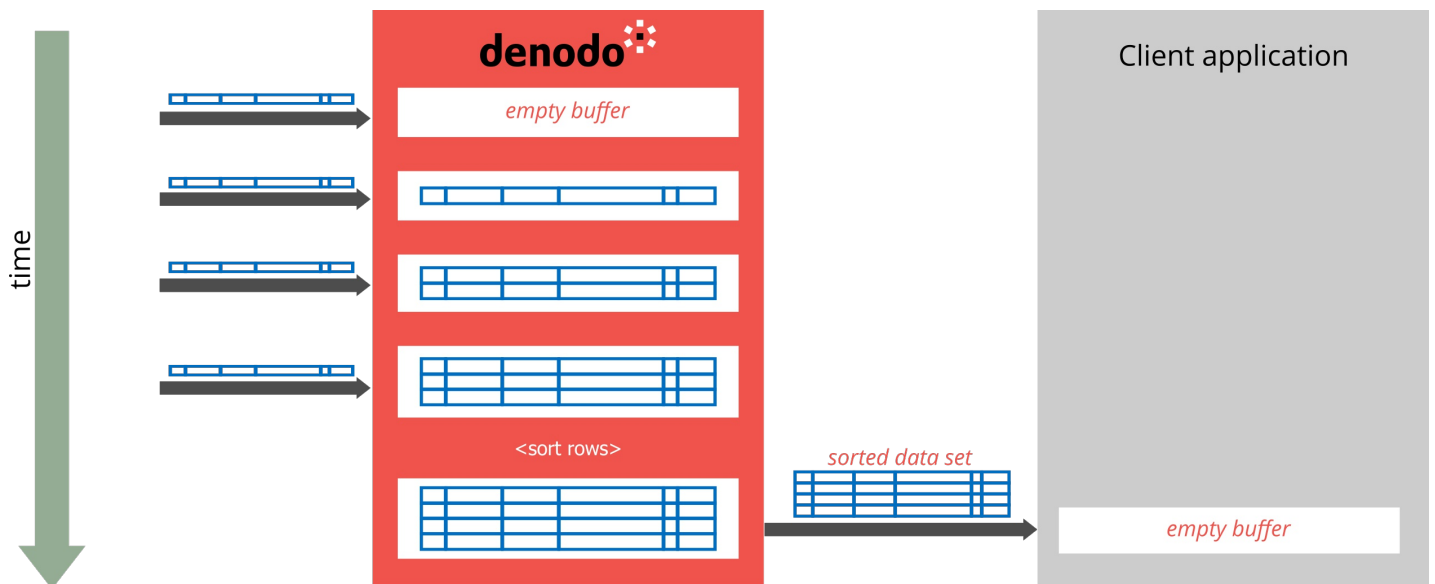
Streaming operation: rows are shipped to the client as soon as they are received and processed.

- On the other hand, some operations cannot calculate their results based on individual rows, so instead they need a collection of rows before they can return the results. One example of this type of operations is sorting a data set, because if we grab a single row we cannot determine where that row should be returned; the process needs to grab all the rows in the source data set and then sort them in memory. Another example is an aggregation of rows, such as when summing a column of the source data set. We cannot return the result of the operation until we have added all the values of all the rows in the source. These operations are called **blocking operations**, and they may hoard rows in memory until they have enough to produce one or more result rows. These operations can be further split in two subtypes:
 - The first subtype is exemplified by the sort: It is an operation that needs to actually hold the source data in memory before it can return the results of the query, since the results of the query are actually all or part of the rows of the original data set. These operations are called **linear memory operations**.



Linear memory operation: Every source row is stored until the operation is applied to all rows, and then the results are sent to the client application.

- The second subtype is illustrated by the aggregation function: While it needs to process all the source rows before returning a value, it does not need to actually hold the source rows in memory. Instead, it can process each row individually and accumulate the results of that single row into the temporary result. In the addition example, there will be a temporary value holding the sum of the values of all the rows processed so far, but we don't need the actual values that were used to calculate it. These operations are called **constant memory operations**.



Constant memory operation: Every source row is immediately aggregated into the partial results.

From a performance standpoint, the ideal situation is to use a combination of streaming operations. This will result in the smallest row latency for the operation (as the first result rows can be calculated early, as soon as the first rows of data are received from the sources), and very importantly, the memory usage will be low.

From a memory perspective, the obvious best choices are to always use streaming or constant memory operations. If your query only uses those types of operations, then the memory usage will be more or less constant **independent of the size of the data set** being processed. This notion can be counterintuitive, but it should be one of the first criteria in query design.

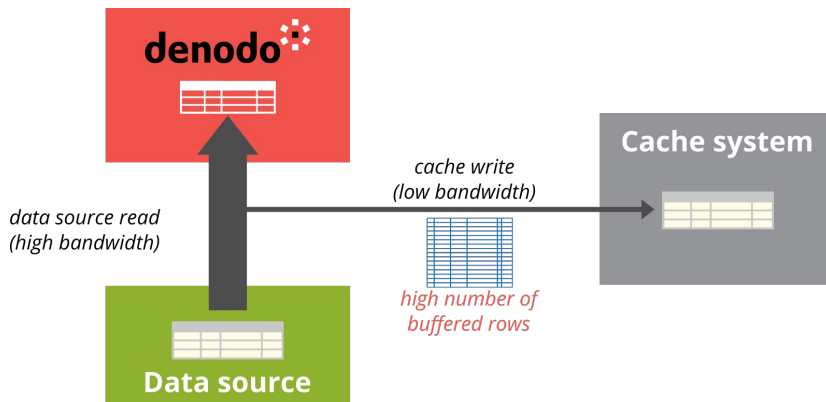
Obviously, sometimes we cannot avoid adding blocking operations, but always try to reduce these operations as much as possible. A single blocking operation will have a huge impact on row latency, as no result row can be returned before the whole data set is in memory, although the total execution time may not be affected (vs. a streaming operation). If you are chaining several operations, always evaluate whether it is to your advantage to execute the blocking operations earlier or later in the chain. As a rule of thumb, earlier is better most of the time, but every situation should be assessed individually. Of course, all of this discussion assumes that it is possible to reorder operations without changing query semantics. If that is not possible, there are no avenues for query optimization, and you should choose the operation order that gives the required results.

A typical mistake when designing queries is to add unnecessary sorting clauses either in the definition of some views in the hierarchy or in the query that is sent to the Denodo Platform. Sorting clauses in intermediate views can easily become irrelevant, as any further operation on sorted data can modify the order of the rows (due to the nature of relational operations). Always double check if an issued sort operation is actually needed or if it can be avoided, as such operations have a very big impact on performance, both in latency and in memory usage.

Now that we've covered streaming vs blocking operations, we recommend that you review the discussion above about the different join methods that the Denodo Platform offers, paying attention to the behavior of each branch of each join, and identifying which areas are streaming, constant memory, and linear memory, so you are familiar with the runtime behavior of your queries.

CACHING AND MEMORY USAGE

Caching can be another source of undesirable linear memory consumption: When we enable cache load for an operation, the Denodo Platform performs the operation as usual, and as soon as rows are returned as results of that operation, they are sent to both the data consumer and the cache system.



Rows are buffered due to the difference of speed between cache inserts and regular operations.

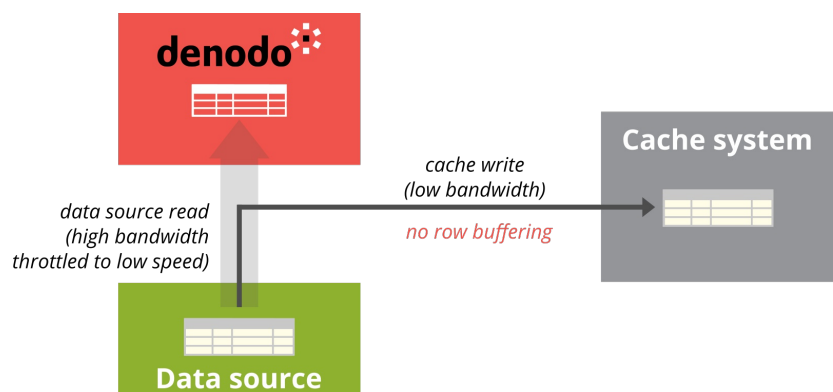
Usually, inserting data in the cache is slower than simply returning it to the calling application. So suddenly, we find that the results are fed to two different pipelines with very different processing speeds. The typical behavior in this scenario is that result rows are calculated more quickly than can be inserted in cache, so the stream of rows that are shipped to the client application determines the speed of

calculations. (If the client application cannot keep up with the speed of the Denodo server, the calculations will be throttled to accommodate for the slower data consumption rate.)

However, the stream of rows that are inserted into cache will store more and more rows in memory, roughly at a linear pace, so by using the operation to populate a cache table, the operation will be turned into a linear memory operation (independent of the operation's original behavior).

There is a way of avoiding this behavior: Tell the Denodo Platform to cache just the results of the query, without returning any row as a result of the operation, when the query is issued. This is done through the context clause `'cache_return_query_results'='false'`. When using this setting the result rows will not be returned as results of the query; instead the Denodo Platform will only instantiate the pipeline that feeds the cache system, which in turn will be used as the moderator of the speed at which the operation will be performed.

If the cache inserts cannot keep up with the rate at which the results are calculated (which is a common situation), then the operation will be throttled to keep the memory consumption as constant as possible. By using this setting, constant memory operations or streaming operations will remain performant from a memory usage standpoint.



The throttling of operations to the optimal speed when using "query results=false".

This is the recommended setting for cache load queries; issue specific queries for full cache loading that don't return data to any client, but only load data into the cache system.

Java Heap Memory

The Denodo Platform runs as a Java process regardless of the operating system where it is executed. This means that the memory that the server will have available at runtime will be the amount that the operating system assigns to the Java process at startup. Inspecting the memory consumed by the Java process that executes the Denodo Platform may provide an incorrect view of the actual memory being consumed by the data virtualization operations. The operating system assigns the Java Virtual Machine (JVM) an initial heap size, and the JVM can choose to grow or shrink the amount of memory at any point. At any given point the heap will be partially used by the Denodo Platform running on that JVM—the virtual machine will always have some free memory on reserve so if the Denodo Platform needs more memory, it can provide it right away without having to request more from the operating system.

The bottom line is that the memory consumption numbers obtained by examining the Denodo Platform process at the operating system level will not provide accurate information about the memory being consumed by the server at a given point in time, as those numbers refer to both the memory consumed by the server and the free memory that the JVM is holding.

Always use the Diagnostics and Monitoring tool or the logging facilities included in the Denodo Platform to check the memory consumption of the server.

Swapping

Swapping refers to the practice of using hard drive space to temporarily store data that does not fit in the main memory. Hard drive space is then used as an extension of the computer's RAM, albeit a much slower one. This technique is a tradeoff that allows a calculation to complete successfully even when there is not enough physical memory present to complete the calculation, although at a reduced performance level, as access to the hard drive can be several orders of magnitude slower than access to main memory. This loss of performance is an unfortunate side effect, but in most cases it is better than the alternative: the operation failing due to lack of memory.

The Denodo Platform offers a flexible swapping system that can be enabled both system-wide and on a per-view basis. This system accomplishes two goals:

- It allows the system to execute queries that require more memory than what is assigned to the Denodo Platform server process.
- It protects the Denodo Platform server from running out of memory when heavy queries are received, keeping the service available for other queries.

The way to tell if a query is using swapping is to look at the query's execution trace. Using the query monitor, look at nodes in the execution plan that seem to be taking a long time, and click on them to show the details; if the node is using the swapping subsystem, it will show both the "Swapping YES" status and "Number of swapped tuples X", where X stands for the number of rows that overflowed the allotted memory size and were therefore sent to the hard drive.

When you are designing queries, one of your goals should be to minimize the amount of swapping. Of course sometimes swapping is unavoidable, given the size of the data sets being processed and the types of operations that are performed. Regardless, you should be aware of the type of operations that your queries are executing, memory-wise, and if they have a constant memory profile, or if they are blocking (either constant memory type or linear memory type). If you find some of your queries are being swapped, that could mean they are being executed as linear memory type, and the most common situation is that the wrong join method is being applied.

For example, remember how hash joins work. They retrieve the full data set from the right branch of the join, then they build a hash table with the values of that branch, and finally they stream the rows from the left side, comparing each one with the contents of the hash table. The optimal scenario is having a very small table on the right side and an arbitrarily large table on the left side. If we reverse the situation, and place a huge table on the right branch of the hash join, the hash table that is to be built will be very big, which means it will consume a lot of memory, and this will probably force the query to use swapping. This situation would be easily solved by changing the order of the join so the big table is on the left branch and the small table is on the right branch of the join.

Always check the join methods that are executed in swapped queries and verify that they are suitable for the data sets being processed.

Analyzing Memory Usage

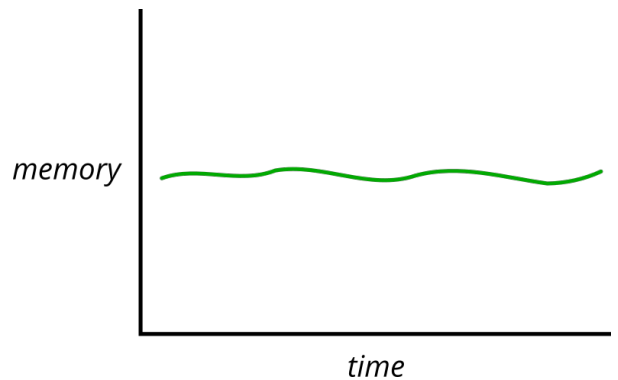
The main tool for analyzing the memory usage of the Denodo Platform server is the Diagnostic and Monitoring tool, included in the core distribution of the Denodo Platform. This tool is a web application that needs to be started in the same way as the data virtualization server. To do so, open your Denodo Platform Control Center and in the Virtual Data Port tab, click the start button for the diagnostics tool; afterwards, you can click on the Launch button to open a browser pointing to the running tool.

Check the official documentation of the Diagnostic and Monitoring tool for complete setup and initial configuration instructions.

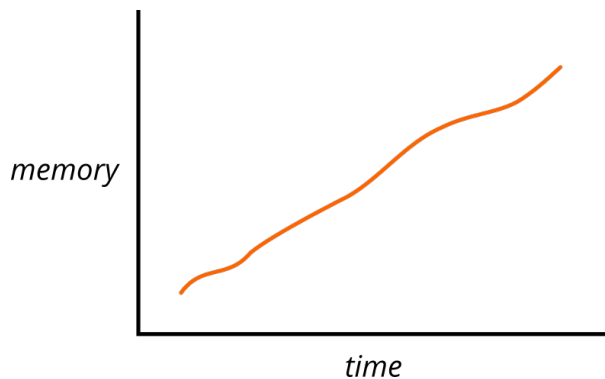
INTERESTING MEMORY PATTERNS

One of the most important diagnostics that we can perform is to analyze the behavior of memory consumption by the server. Typical patterns that are possible in the memory consumption graphs are:

- **Constant memory queries:** These appear as flat lines in the memory graph. This is the ideal scenario from a performance standpoint: These types of graphs will appear when your queries contain streaming operations (row shipping).
- **Monotonically increasing memory:** These will appear in row hoarding scenarios, such as queries that include sorting operations. In these cases the memory grows linearly and the graphics in the diagnostics tool will display this upward tendency. As discussed earlier, these kinds of situations should be avoided as much as possible.
- **Memory spikes:** A graph with spikes can appear when queries perform operations that hoard rows in memory but also periodically clear

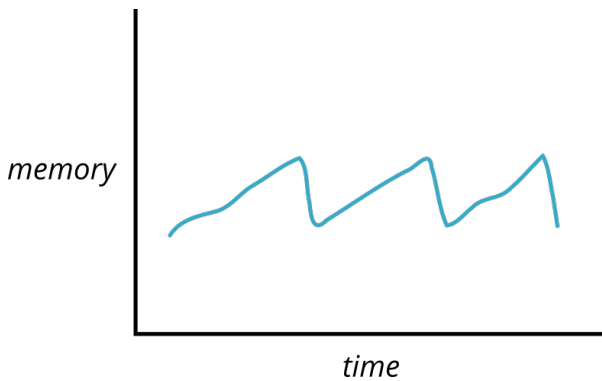


A typical constant memory graph.



A typical linear memory graph.

those hoarded rows. A typical example would be a group by operation that is performed on a data set that is sorted over the aggregation field. In that case, rows are hoarded until a full group is in memory (this can be detected because the data is sorted). At that point, the whole group can be replaced with the aggregated calculation, causing the memory consumption to go down sharply.



A typical spiky memory graph.

Always take into account your server settings, regarding query memory limits and swapping, when you try to visualize the memory consumption of your server (these are in the Administration > Server configuration > Memory menu of the administration tool). It may be more difficult to discern patterns in the graphs if the server is set to limit the memory consumption of queries, and enable swapping.

Memory consumption trends will only be recognizable in the diagnostics tool when processing medium to high volumes of data. The graphs will not register enough change for queries that are executed very quickly and which do not process enough rows to make a difference in the overall memory state of the server.

Server Memory Settings

The Denodo Platform optimizer offers some controls over how the memory is managed when executing queries:

- Setting the maximum amount of memory that any query will consume.
- Setting the maximum amount of memory that unprocessed, intermediate results can use.
- Swapping settings: enabling/disabling swapping and setting swapping size controls.

These settings can be found under the “Administration > Server configuration > Memory usage” menu in the VDP Administration Tool.

vdp Server Configuration

Cache

Concurrent requests

Default I18n

HTTP proxy

Identifiers charset

Kerberos configuration

Memory usage

Queries optimization

Server connectivity

Stored procedures

Threads pool

Limit query memory usage: ☐ Off ☒ On

Maximum query size (MB):

Maximum size in memory of intermediate results (MB):

Swapping status: ☒ Off ☐ On

Maximum size in memory of each node (MB):

Maximum size of the blocks stored in swap (KB):

Ok Cancel

Memory management options.

Static Optimization

Static optimization is the first step that the optimizer takes when a new query is received. In this step, the optimizer rewrites the original query into a new form that will be executed with higher performance. The transformations that are applied are at the SQL level; they do not define how to execute each operation (that will be the role of the dynamic optimizer), but what operations to execute and in which order.

The static optimization process takes into account the original query (after being parsed into a format that is amenable to analysis) and the metadata available about both the views that are affected and the data sources that will be used in the query. The details about the capabilities of each data source are of special importance during this process, as they will constrain the number and type of operations that in the end will be delegated to the data sources.

The static optimization process modifies the SQL of the query that was received, but of course the resulting query will maintain the semantics of the original query: The results of the optimized query will always be correct!

Enabling Static Optimization

The Denodo Platform server should have the static optimization enabled by default, but should you want to check if your server has it enabled, or if you would like to disable it for testing, open the Administration > Server configuration > Queries optimization menu in the Virtual DataPort Administration tool.

View Definition vs. Query Conditions

A virtual data model is composed of a collection of views. These are split into base views, which represent physical tables or entities that exist in remote systems, and derived views, which are virtual views created as a combination of other views in the data virtualization layer. Every derived view that is created in the data virtualization layer is created from a list of relational operations and clauses (joins, aggregations, projections, `WHERE` clauses, ordering clauses, etc.). These operations are codified in the definition in the view and are immutable.

The Denodo Platform receives queries from client applications. These queries are dynamic, in the sense that they don't need to be defined within the data virtualization layer: The client application can assemble any SQL on the fly and send it to the data virtualization server to be executed. These queries can also define operations and values in the same way as the view definitions.

When a query is received and executed, the final query plan that is calculated for the query execution is generated by combining the parts defined in the dynamic query sent by the client application with the parts defined in the views that the query uses. Two different queries over the same virtual view can lead to very different query execution plans due to the conditions and operations specified in the runtime part of the query.

Some of the opportunities for optimization stem from collisions between the conditions defined in the views and the conditions that appear in the query sent by the client application; any conflict between those two (for example, the conditions $a > 0$ and $a < 0$ are in conflict) will result in branches of the query plan that are guaranteed to return no rows. the optimizer can safely remove those branches from the query plan.

Always take into consideration the runtime behavior of the views under querying conditions, as opposed to the design-time behavior.

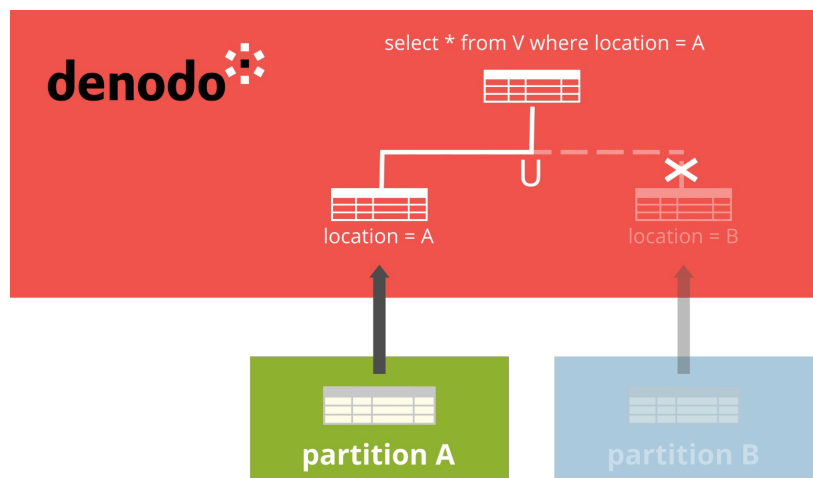
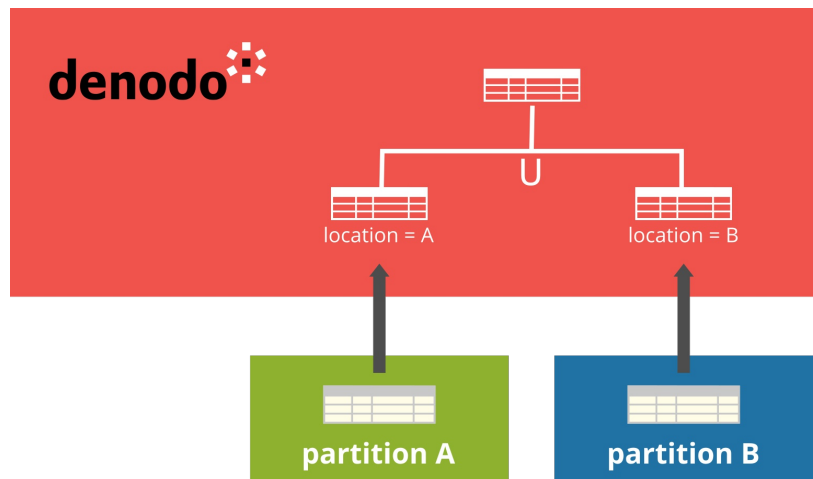
Sample Query Rewriting Scenarios

The static optimizer will apply a variety of optimizations depending on the specific situation that it finds; it applies an extensive catalog of optimizations that enables the optimizer to work under many different scenarios. Some examples of these optimizations are showcased below; this sampling of techniques should provide a general sense of how the static optimizer works and convince the reader of the strong performance optimization features present in the Denodo Platform, but it is by no means exhaustive. The Denodo Platform has many more tricks up its sleeve.

BRANCH PRUNING

Query plans are a tree of nodes, each executing an operation in the overall plan that answers the query sent by the client application. These nodes sometimes return zero rows, as mentioned earlier, and in these cases the optimizer can remove these branches (or subtrees) of the execution plan; this is known as branch pruning, and it can massively simplify the query execution plan.

Branch pruning can be useful in several scenarios, such as when queries that use partitioned data. When that is the case, a number of base views in different data sources contain data with the same schema but with values that refer to different partitions. (It is common for data that refers to geographical information, with each partition containing data for a separate geographical location). Partitioned unions are defined in



the Denodo Platform as a union operation over a set of selection views. Each of the selection views defines a `WHERE` clause with conditions that match the data stored in the underlying partition (so these conditions do not actually filter any of the rows that are in the data source when issuing a full scan over the selection view).

Queries received over the union view will by default hit all the partitions in the union, and subqueries will be delegated to each underlying data source accordingly. But in certain scenarios, some queries will only need data from specific partitions. These queries specify conditions that match one or more partitions but conflict with the definition of the other partitions. For such queries, the optimizer would simplify the query execution plan and remove the partitions that are known to be useless for that specific query so that only the needed data sources would be queried.

Another common scenario for branch pruning is join operations over several tables. Consider the example of a star schema, in which a `sales` fact table is joined with dimensions such as `product`, `customer`, etc., that are then queried, retrieving fields only from part of the base tables (such as `sales` and `product`, but not `customer`). In many of these cases, the unused branches of the join are pruned and not executed.

JOIN REORDERING

The optimizer can also apply optimizations to n-joins. As explained above, the Denodo Platform executes n-joins in pairs, by splitting them into a series of 2-way joins. This means that there is an order to the sequence of joins when executing the n-join, so the optimizer has some degree of freedom to seek performance improvements. The optimizer's main goal when reordering joins is to delegate part of the n-join (at least one of the 2-way joins) to the underlying data sources.

JOIN TYPE CHANGE

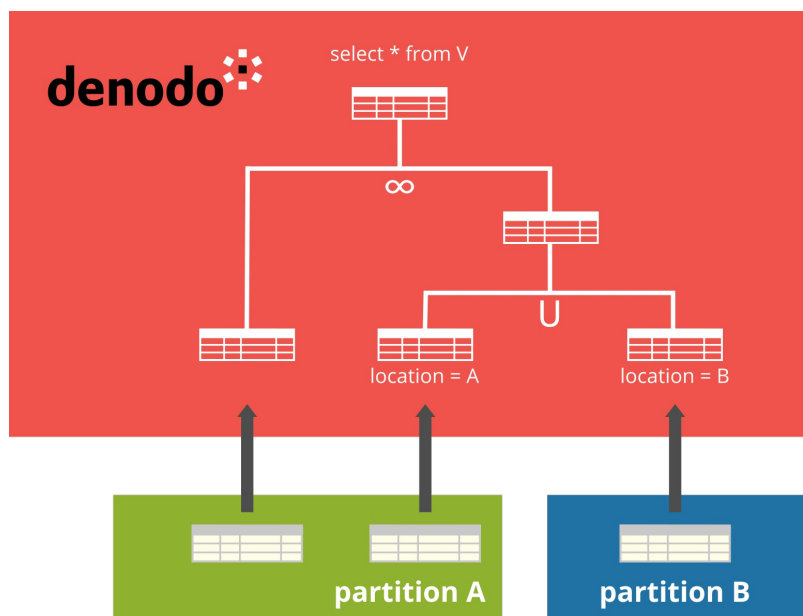
In addition to changing the method used to execute joins, the optimizer can also change the join type. When possible, the optimizer will execute outer joins as if they were inner joins. But this is only possible when the outer join is either used as input of some other view or when it is used within a larger dynamic query. Regardless, in some cases the larger query in which the join is embedded defines conditions that are more restrictive than the join condition, so the outer join is actually equivalent to an inner join, but only in the context of the larger query.

POST-PRUNING DELEGATION

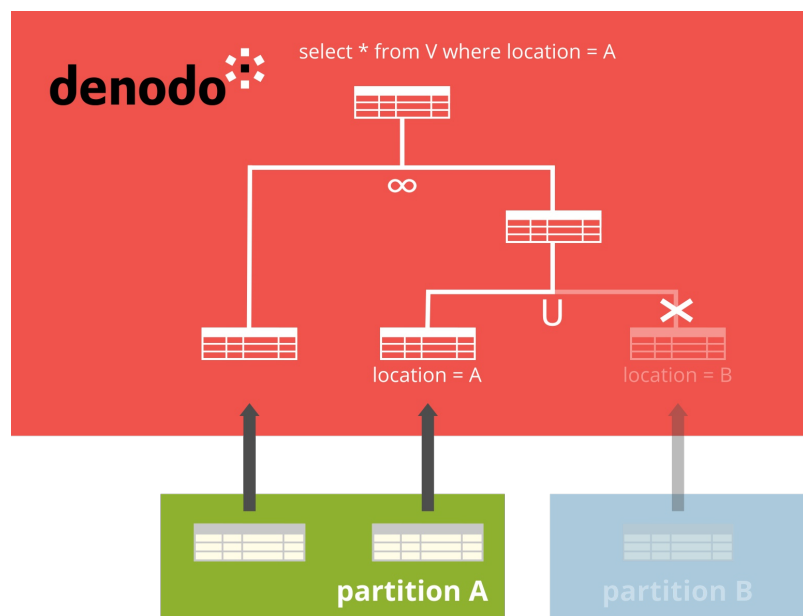
Most data virtualization scenarios include data combinations across different physical systems, which prevents the delegation of operations if naive query plans are used. Some of these scenarios include pruning the branches of the query plan, which allows for further query optimization. A typical scenario results in the pruning of branches due to unused

partitions or incompatible conditions, which makes all the resulting nodes of the execution plan use data that resides in the same data source. The immediate result is for the operations in those nodes to be delegated to the underlying data source.

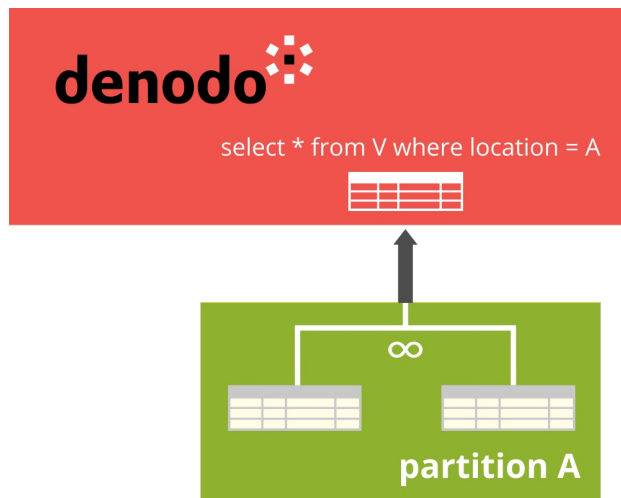
This example shows that many times during the optimization process the application of one technique opens the door for further optimization tactics, chaining them in a process that enhances a query from displaying unacceptable performance to demonstrating performance levels that are equivalent to those of physical systems.



Base case: a typical non-delegable query.



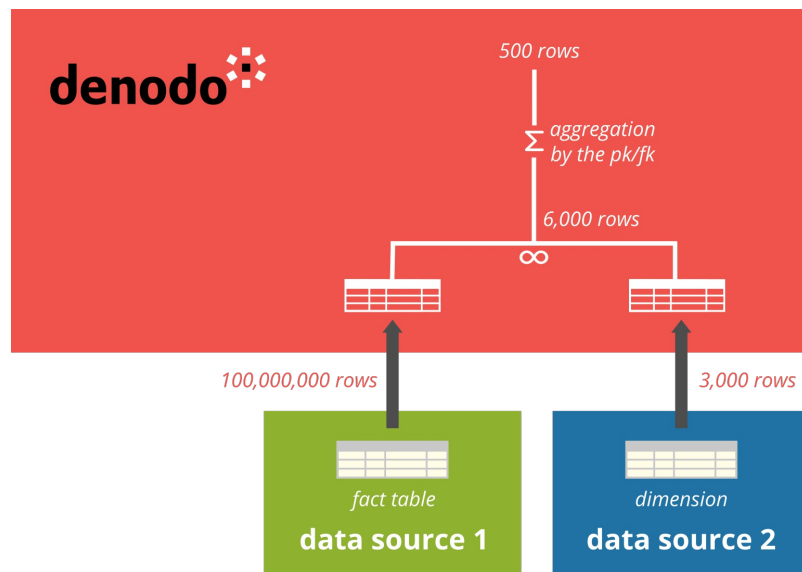
Intermediate step: pruning a branch of the original query.



Final query plan: delegation to source, possible after partition pruning.

AGGREGATION PUSHDOWN

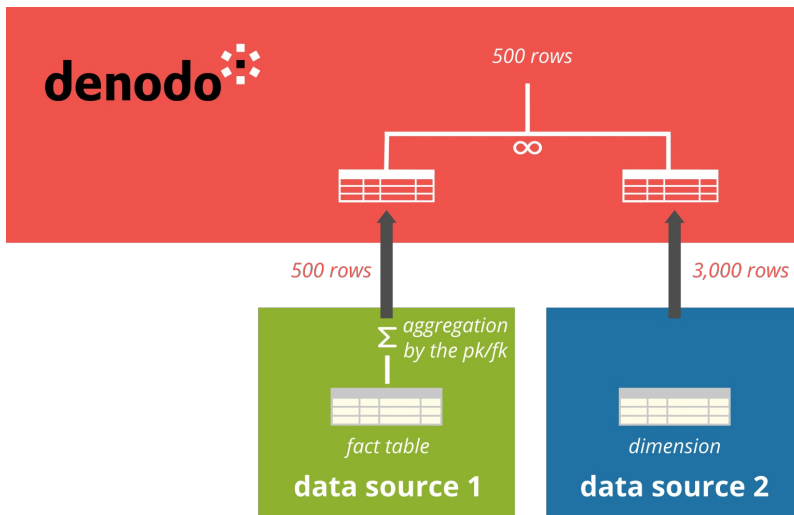
Aggregation pushdown can be applied in many scenarios but it is very typical of logical data warehouse situations. A large fact table in a physical system is joined with one or more dimensions stored in separate physical systems, and aggregated over the primary key of one or more dimensions. This represents a classical query in reporting scenarios and it is probably the most common type of query that data warehouses will receive.



Base case for aggregations over joins in data warehouse reporting scenarios.

In this scenario, the join operation cannot be delegated to any of the data sources, as none of them contains the whole data set; a naive execution plan would pull the whole fact table into the data virtualization layer, do the join in memory, and then do the aggregation, also in memory.

The optimized plan, on the other hand, recognizes that the order of the join and aggregation operations is not important: The aggregation is done over the primary keys of the dimension(s), so it can be pushed down to the fact table first, and then the results of the aggregation can be joined with the attributes coming from the dimension.



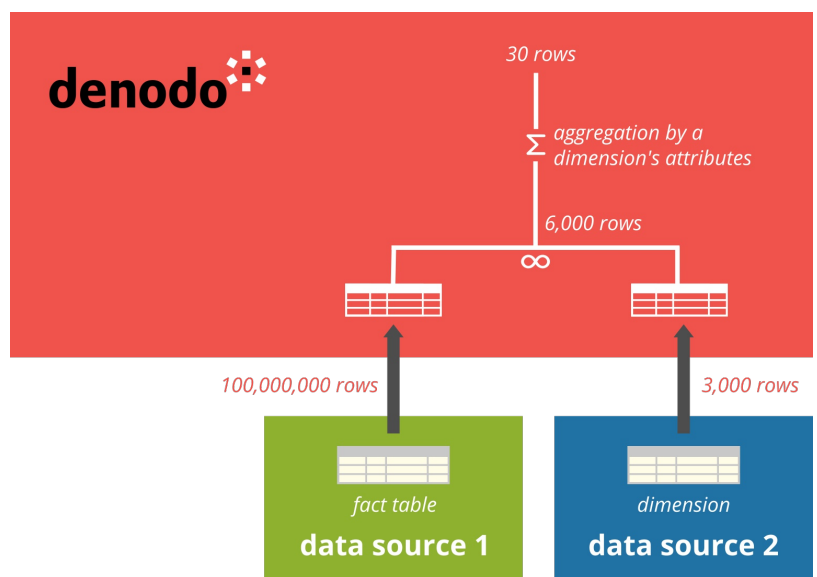
Aggregation push-down vastly increases performance; no need to transfer the whole fact table.

This optimization results in huge performance gains due to avoiding the massive transfer of rows that is performed in the naive scenario. The fact table before the aggregation will be orders of magnitude bigger than after the aggregation, so pre-aggregating it enables both the transfer and the join of a (comparatively) small number of rows.

For a more detailed logical data warehouse scenarios explanation, visit [this entry in the Denodo Community Knowledge Base](#).

PARTIAL AGGREGATION PUSHDOWN

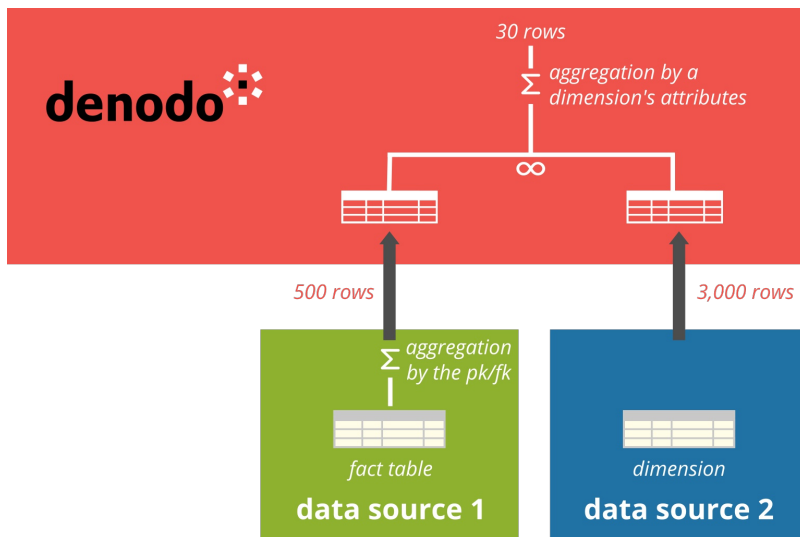
Partial aggregation pushdown is applied when some restrictions prevent full pushdown. The most typical example is a logical data warehouse scenario in which a query joins a fact table with one or more dimensions (that live in a separate physical system from the fact table), and in which **the query aggregates the results over one or more attributes of the dimensions** (or more precisely, when there are expressions—GROUP BY, SELECT, or WHERE—that use fields from both sides of the join). This is the main difference when compared with the full aggregation pushdown scenario; instead of aggregating by the primary/foreign keys, the query



Base case for aggregations using attributes of a dimension.

aggregates the results by an attribute of the dimension. This means that the aggregation cannot be executed by the system containing the fact table, as it is performed on attributes found on a separate server. The answer of the optimizer is to split the aggregation in two steps:

1. Pre-aggregate the results using the primary/foreign keys in the fact table. This operation is fully delegated to the data source.
2. Join the results of the first step with the dimensions and then perform the aggregation over the attributes of the dimension.



Splitting the aggregation in two steps dramatically increases performance.

This process works because any two rows from the fact table that have the same value for the dimension foreign key will have the same value for the dimension's attribute that the query is aggregating over; it does not matter that different rows in the dimension may have the same value for the group attribute. Those rows will be aggregated together in the second aggregation step.

The benefits of this process stem from the first aggregation performed over the fact table. This step takes a

first stab at reducing the number of rows that have to be 1) transferred into the data virtualization layer and 2) joined with the dimensions. That area is where the bulk of the processing resides., and using a partial aggregation pushdown technique makes the data virtualization optimizer approximate the performance of a native solution.

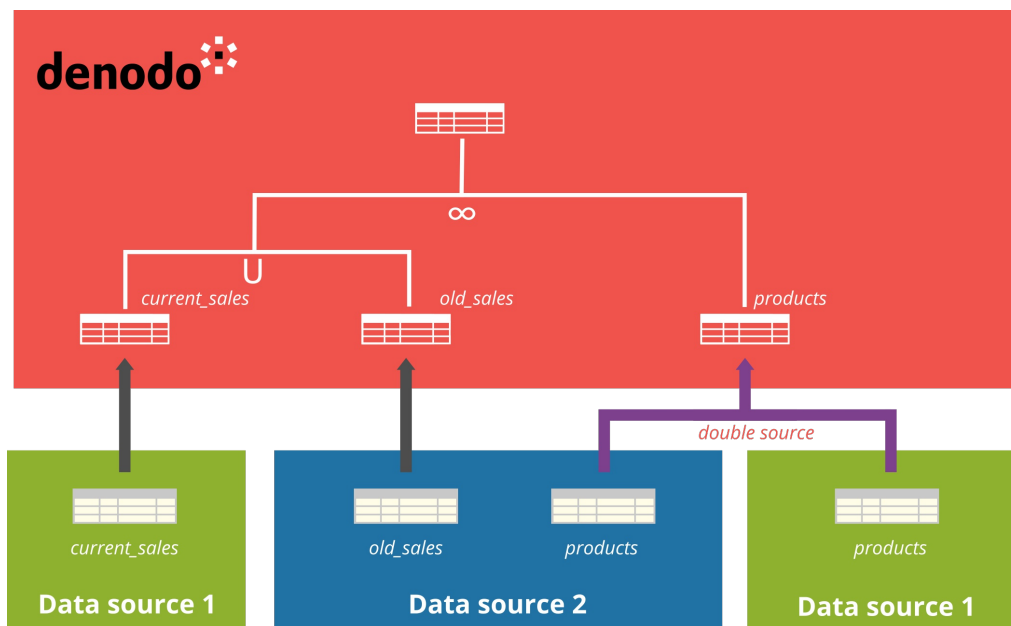
JOIN PUSHDOWN

Another useful optimization technique is pushing down joins under partitioned unions so they can be delegated to the data sources (in line with one of our main optimization themes). This technique is applied in situations like this: A union of two views in two different data sources (for example, a partitioned fact table) is joined with another view (for example, a dimension) and the dimension is physically stored twice, once in each data source.

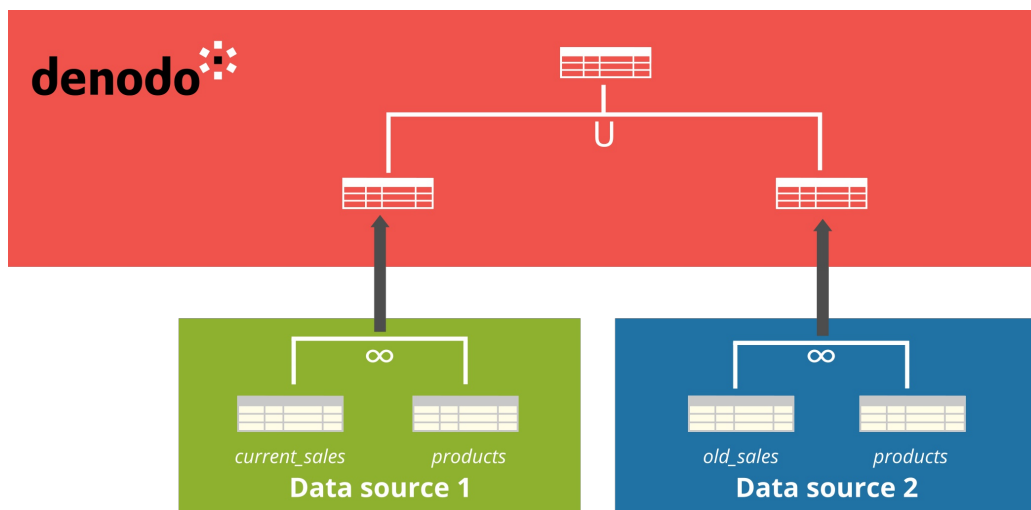
The Denodo Platform optimizer understands that a table can be replicated in two or more sources and it will use the copy that performs best for each specific query that is executed. Setting a secondary source for a base view requires adding an alternative wrapper in the “Options > Search methods” tab of the base view in question. By clicking “+” you will be able to specify an alternative source for the data of that base view.

The optimizer can change the order of the join and the union and choose to execute the join first (as the dimension table is present in both data sources). After this reordering, the joins can be delegated to the data source.

This is a simplified example. In a real-world logical data warehouse scenario it is very common to add aggregations on top of the join; in most of those situations, the optimizer would apply the join pushdown technique in combination with the full/partial aggregation pushdown, resulting in the aggregation operation(s) being executed at the data source level. This scenario would present the maximum gain in performance as the aggregation pushdown will result in almost all processing done by the sources and smaller data sets transferred through the network.



Base scenario for join pushdown; the products base view references two identical copies of the data in datasources 1 and 2.



The join is pushed down below the union, and it can be delegated to both sources.

Of course, the optimizer can apply many performance-tuning techniques to the same query; it is not limited to a single optimization per query. All the methods shown in this Cookbook are routinely combined together to further improve performance.

DATA MOVEMENT WITH PARTITIONING

The previous technique, join pushdown, requires the second branch of the join to be duplicated across both data sources (and referenced in the data virtualization layer as a single view with several alternative sources). There are many cases in which that requirement cannot be satisfied, such as when the right branch of the join (the product dimension in our previous example) is only present in one data source.

In such cases, the optimizer can apply automatic data movement to satisfy the requirements of the join pushdown technique:

1. First, the dimension data is copied into all data sources involved in the union, before the query execution.
2. Once the dimension is copied, the requirements for the join pushdown are satisfied, and the optimizer applies that technique (pushing the join and possible aggregations under the union, then delegating those operations to all the data sources).

The specifics of data movement will be explained later in this book, in the “Data movement” section.

Dynamic Optimization

View Statistics

The optimizer is an extremely advanced system that can take a wide range of actions to optimize the performance of queries received by the Denodo Platform. As mentioned earlier, the optimization pipeline has two broad optimization phases: static and dynamic. Static optimization works on the SQL of the received query on its own, while dynamic optimization takes into consideration the context in which the query is executed.

The goal of the dynamic optimizer is to generate multiple execution plans for the given query and then select the optimal one. In this process, the optimizer decides on the best way to execute each operation in the plan. For example, for each join operation, it assesses data about each one of the branches of the join and selects the best join method, type, and ordering for that specific case.

This set of decisions is based on the knowledge the optimizer possesses about the data to be queried. This knowledge is not available initially; the operators of the Denodo Platform must feed it to the optimizer by gathering statistics on the views involved in the virtual data model.

Gathering statistics is a crucial process that is absolutely required for the correct functioning of the optimizer, and it should be one of the first steps taken by the administrators of the data virtualization platform.

STATISTICS USED BY THE OPTIMIZER

The optimizer needs knowledge about the data itself if it is going to optimize queries correctly; this data is not available at the Denodo Platform level, as the data virtualization layer does not hold data. Rather, it queries the data sources (and cache system) in real time when new queries are received.

Statistics about the data gathered by the Denodo Platform are stored for each view, and include values such as:

- The average size of each column.
- The minimum and maximum values for each column.

- The number of distinct values and number of nulls for each column.
- The existence of indexes.

The last item is specially important. Data sources often allow the definition of indexes over the data. These indexes usually cover a subset of the columns of the whole table and queries that use them are much faster than queries that do not. Therefore, defining indexes on the data is of extreme importance during the traditional process of designing the database layer of any application. As the Denodo Platform is querying these sources, it becomes extremely important to know which columns have the performance advantage of being included in an index, and the optimized queries at the Denodo Platform level will take advantage of that fact and always favor delegating queries that hit the available index over queries that use other columns.

Check the [Virtual Data Port Administration Guide](#) for details regarding indexes in the sources and in the Denodo Platform. Some data sources (such as some parallel databases) require extra configuration steps for declaring their indexes correctly in the data virtualization layer.

Selector	Column Name	Average Size*	Min Value	Max Value	Distinct Values	Null Values
<input checked="" type="checkbox"/>	country	4.90677966101...	N/A	N/A	7	0
<input checked="" type="checkbox"/>	total_sales	8.0	25	6315	102	0
<input checked="" type="checkbox"/>	year	8.0	2014	2014	1	0
<input checked="" type="checkbox"/>	month	8.0	1	12	12	0
<input checked="" type="checkbox"/>	area	7.89830508474...	N/A	N/A	2	0
<input checked="" type="checkbox"/>	in_promotion	1.0	N/A	N/A	2	<null>

The View Statistics screen.

HOW TO GATHER STATISTICS

Statistics are gathered and stored on a view-by-view basis. This means that administrators are free to gather statistics only for the views whose performance is critical for the use case at hand, and each view can have its own period of statistics gathering.

The process of gathering statistics is very simple, and it can be done either manually or automatically:

- For the manual gathering of statistics, use the “View Statistics Manager” in the Virtual DataPort Administration Tool (in the Tools > Manage statistics menu). This tool will show a selector of the database you want to gather the statistics in, and it will display all the views available in the selected database. You can select multiple virtual views and then click on the “Gather” action. After confirmation, the Denodo Platform will start gathering the data from the sources, by querying each one in turn.
- Programmatic gathering of statistics is done through a VQL query that uses one of the available statistics-related stored procedures in the Denodo Platform. These are `GENERATE_SMART_STATS_FOR_FIELDS` and `GENERATE_STATS`. The difference between them is that the former will use the source’s system metadata when available, and the latter will always issue a `SELECT` statement to the source systems to gather the data. These stored procedures can be called as part of a VQL query that can be scheduled to be triggered periodically, either through Denodo Scheduler or any other scheduling software that you may have available.

Gathering statistics is done by querying the data sources from the data virtualization layer and asking them about the summarized knowledge about the data; the Denodo Platform will always try to use the most performance-aware method for issuing these queries such as querying the metadata tables directly when the source makes them available, as is the case with relational databases. The Denodo Platform uses metadata tables when the source supports this method. In those cases, some of the statistics (such as the minimum and maximum values) may have to be calculated using queries over the data itself. This extra step is enabled by selecting the checkbox “Complete missing statistics executing `SELECT` queries”; note that the minimum and maximum values are only useful for range queries (for example, conditions like `myField < 10`), so if your queries do not use that type of `WHERE` clause, you may not need to retrieve the additional statistics and the metadata-based statistics will suffice.

Keep in mind that statistics gathering may take a long time (especially when non-relational, slow sources that do not have accessible statistics-related metadata are involved in the data combinations).

HOW OFTEN SHOULD YOU GATHER STATISTICS?

This question often comes up, and the answer depends on the specifics of your data and the specifics of your use cases. In the end, statistics are only used to compare execution plans.

The actual values that are reached when calculating the cost of each plan do not matter; what matters is only the comparison between those relative values. This means that the process does not need to arrive at a perfectly accurate value as a result of the calculation. Ballpark estimates will be enough to choose the right combination of operations.

Statistics only need to be updated when there are changes in the contents of the data sources that may be significant enough to sway the decision from one execution plan to another. If some table contains one million rows, there is no need to gather statistics every time a hundred rows are added, as that small number of new rows will have no impact on the selection of the final query execution plan. This is compounded by the fact that updating the statistics of a view, if the underlying data has not changed, will have no effect whatsoever, so the frequency of gathering the statistics should be dependant on the frequency of changes in the data sources.

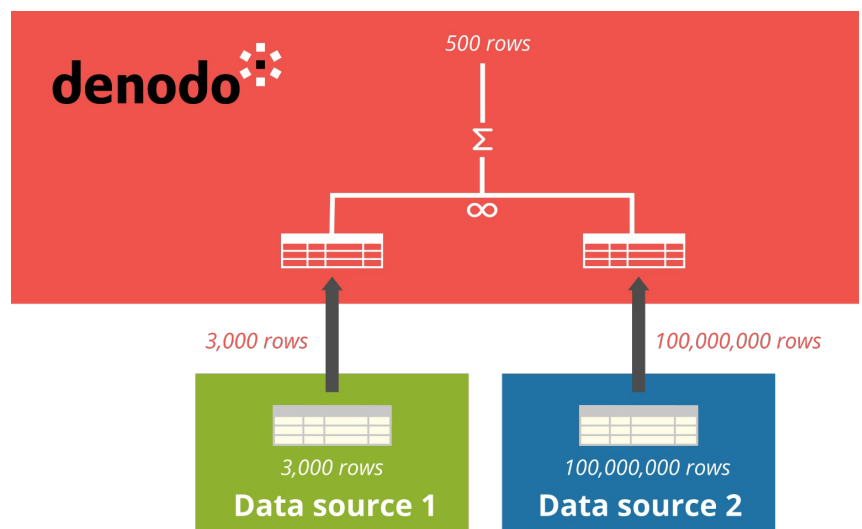
In addition, the statistics of each view can be gathered independently of each other. Thus, gathering statistics should not be seen as a monolithic process, but instead as a set of small processes that must be triggered at the right time. The frequency of changes in the data for each of the underlying tables in the data sources will drive the need for setting the periodicity of gathering statistics for each independent view.

Data Movement

Data movement (sometimes also referred to as “data shipping” in the literature) is one of the advanced techniques that the Denodo Platform implements for dramatically increasing performance in some specialized scenarios.

Data Movement as an Optimization Technique

Data movement is a performance optimization technique that is relevant when executing queries

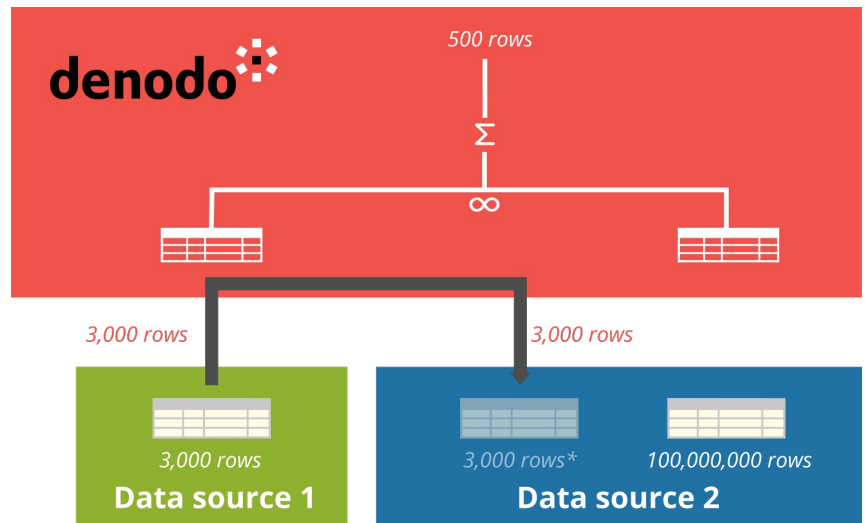


A data movement base case. The join+aggregation cannot be delegated.

across distributed data sources. Imagine a join between two tables—one in data source A and another in data source B. If the data virtualization layer executes this query in the standard way, as we have seen before there is no possibility of delegating any operation to the sources; data source A cannot complete the join because it does not have access to the data in data source B, and conversely data source B cannot execute the join because it does not have access to the data in A.

Data movement optimizes this scenario by:

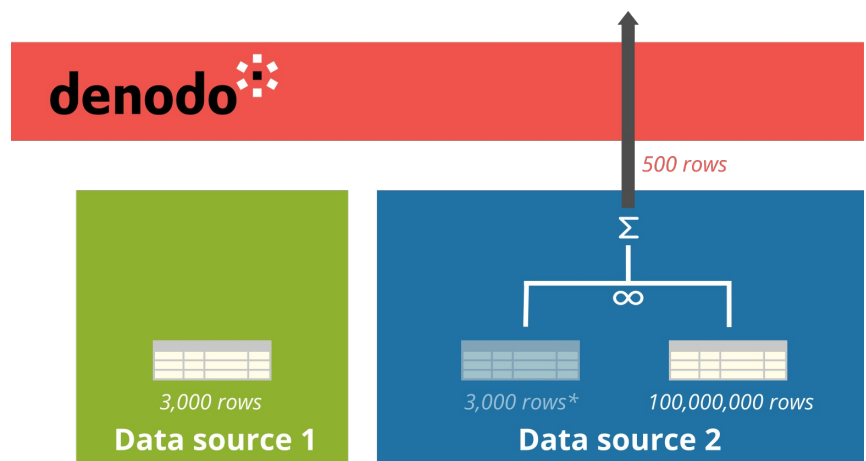
1. First, creating a temporary table in data source B.
2. Second, inserting in that temporary table all the rows from the table in data source A.
3. Third, delegating the join of both the temporary table and the original table in B to data source B, and retrieving the results of the query.
4. Fourth, deleting the temporary table in data source B.



Using data movement: The small table is temporarily copied next to the big table.

You can see that the point of this technique is to align the data in a way that the join operation can be delegated to one of the data sources.

Data movement is a technique that piggybacks on the benefits of delegating queries to sources (as seen at the beginning of this book).



The whole query can be delegated after the temporary data movement.

Rationale and Benefits

Why is data movement used as an optimization technique? A priori, it seems counterintuitive as we are performing additional data transfers (including row insertion operations, which are usually expensive) and we do not avoid the execution of the join operation. But if we look at it more carefully we can imagine a scenario in which it makes sense to operate this way.

Imagine that the table in data source A contains 1,000 rows. It's a small table with a small number of columns. In data source B we have a table that contains 1,000,000,000 rows. We then join these two tables together using the Denodo Platform, and let's assume there is a relatively low number of results of the join. Let's say only 2% of the billion rows match a row on the thousand rows.

This kind of situation is not uncommon for scenarios in which a database or a data warehouse contains a set of dimension tables, and one or more of those are joined with a very big fact table that is living on a big data store (for example, Hadoop).

Let's do some quick calculations for the naive scenario. The Denodo Platform cannot optimize the join so it will pull one thousand rows from the left branch, plus one billion rows from the right side. The join is then done in memory (although if done properly with a hash join it should not require a big amount of memory), and the resulting twenty million rows are transferred to the client application. The total number of rows transferred in this scenario is 1,020,001,000.

If we use data movement, the numbers will be quite different: First we transfer the 1,000 rows from data source A to the Denodo Platform and then to data source B, making it a transfer of 2,000 rows. (In some cases, it may be possible to transfer the rows from data source A to data source B without doing a round trip through the data virtualization layer, but this is not true for the general case). Then we delegate the join operation to data source B and we get the resulting 20,000,000 rows and ship them to the client application. This means a total of 40,002,000 rows transferred. This means that the data-shipped join transfers fewer than 4% of the rows the naive join transfers, which represents a huge improvement in performance.

The gains are even more evident in a typical reporting scenario, where on top of the join between the dimension and fact tables, we aggregate the join to calculate some metrics over the whole history in our data systems. In this case, if the distributed join was preventing any delegation, by copying the dimension table to the second data source we can probably delegate both the join and the aggregation, so the resulting data set that we pull as a result

of the query in data source B is much smaller than the combined size of both tables involved in the combination. Thus, the performance gains obtained by data movement are even greater.

Assuming that aggregation reduces the number of rows in the result data set by 99%, then the numbers are 1,000,201,000 rows for the naive join and 402,000 rows for the data shipped join. The gains are staggering (the number of rows transferred by the data movement join is around 0.04% of the rows transferred by the standard join).

Data Source Optimizations

Connection Pools

The Denodo Platform receives queries from data consumers; these queries are processed by the query engine in the Denodo Platform server, and during the query plan generation they are iteratively decomposed into a series of subqueries that are sent to each data source involved in the original query. This means that a query received from a data consumer will ultimately result in one or more queries sent to data sources; the Denodo Platform will then open a connection to each of the data sources, send each subquery, retrieve the data and close the connection. This process is usually optimized by using a connection pool.

A connection pool works by generating a set of connections to a given data source when the data virtualization server is initialized. Those connections are then kept alive and reused for all the queries that need to use the data source, so when a new query needs to access it there is no need to create a new connection from scratch. If the pool has a connection available (one that is unused by any other query) then it is assigned to the next query. Once the query has finished, the connection is returned to the pool and made available for other queries. If no connection is available, the pool will either create a new one for the query (so the pool size will grow) or pause the query until a connection is available.

This situation has two sides, when viewed from a performance- optimization perspective:

- On one hand, having a pool of connections to a source improves performance because connections don't need to be created and destroyed for every single query received by the server. This performance improvement can have a significant impact in operational use cases for data virtualization (use cases with many simultaneous transactions of small size), and it's usually irrelevant for analytical use cases (use cases that typically have a low transaction volume, but each transaction is long-lived and requires a high resource commitment).

- On the other hand, having a connection pool means that there is a maximum number of connections that will be created against a data source. This in turn means that while you are guaranteeing that the data source in question will never receive more simultaneous queries than what's specified in the pool (thus protecting the performance of the data source in the cases when it is needed), you are also restricting the amount of queries involved in that specific data source that the Denodo Platform server will execute concurrently.

The second point is very important and is often overlooked by newcomers to data virtualization. It is critical to review all the connection pools that are enabled in the Denodo Platform server and make sure they are appropriately sized; otherwise, connection pools can easily become performance bottlenecks in scenarios with a high volume of transactions (even when the Denodo server is under light load, it will not process more queries if the affected connection pools do not have available connections). The main tool for diagnosing this is the set of server logs; use them to characterize your server load and predict the average and maximum loads that each data source is likely to go through, and then perform any corrective actions on the pool size configuration.

Remember that the connection pools in the Denodo Platform come in a default size of 4 initial connections and 20 maximum connections. There are no “one-size-fits-all” use cases so make sure you size your connection pools according to your needs.

In summary, remember to check your connection pool sizes if your queries seem to be stuck at the base view level in your performance testing. In these cases, the execution trace will look like this:

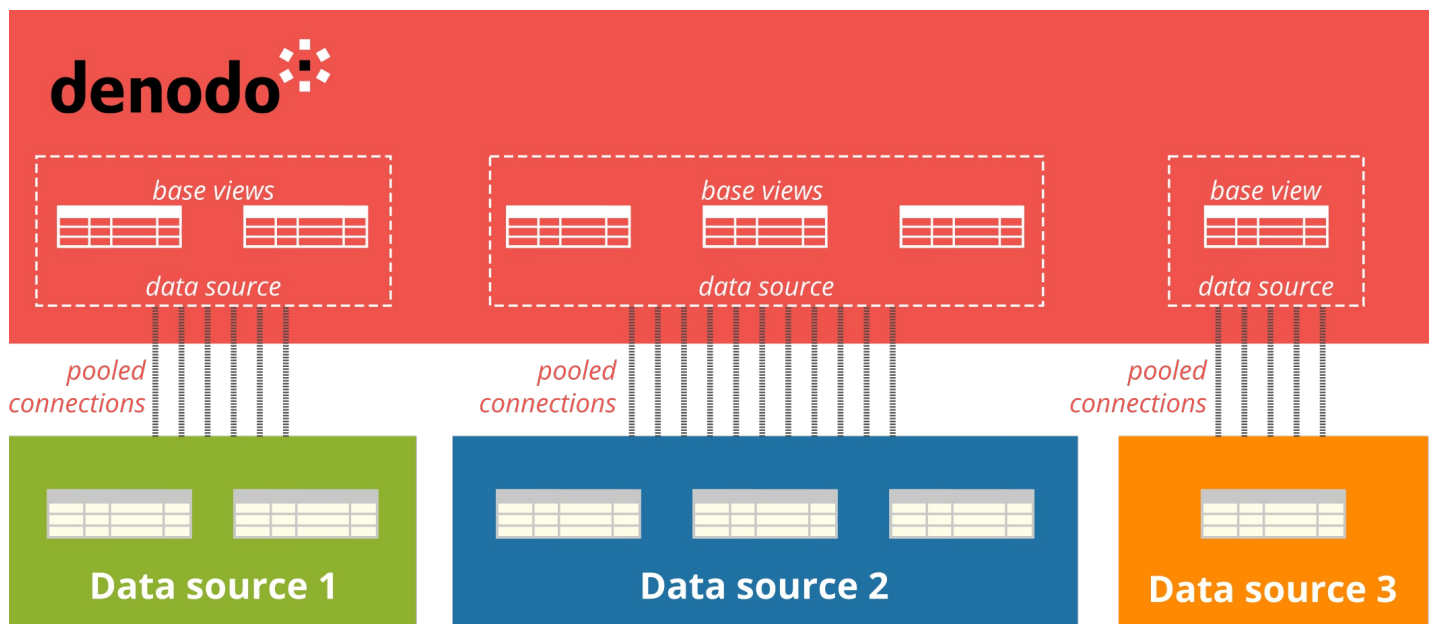
The screenshot displays a Denodo execution trace. On the left, the execution plan shows a sequence of steps: a root node labeled '<Execution Plan>', followed by a 'sales' table node, another 'sales' table node, and finally a 'sales#0' node. On the right, the details for the selected 'sales#0' node are shown. The details include: Type: JDBC Route; Execution time: unknown; Start time: Fri Jul 15 11:21:42 895 PDT 2016; End time: unknown; Response time: unknown; Number of rows: 0; State: PROCESSING; Completed: no; DB URI: jdbc:mysql://localhost:3306/sales; User name: denodo; Connection time: 0; and Cached connection: no.

A query execution trace showing a node waiting on a connection pool.

The data source node shows as processing but the node does not specify a SQL query. When the node is not waiting for the connection pool, it will show a specific SQL query that was sent to the data source.

If you find that situation in your logs, then increasing the size of the affected pool(s) should solve the issue and make the queries run normally.

Finally, a word about multiple connection pools. The Denodo Platform will create an independent connection pool for each data source that actively maintains connections to other systems (for example all the data sources of type JDBC, ODBC, multidimensional DB, etc.). This means that each pool is configurable separately, so the performance can be tuned to exactly meet your needs, and queries waiting on the connection pool of a specific data source will have no impact in the connection pools of other data sources.



Each data source in the Denodo Platform maintains its own connection pool to the source system.

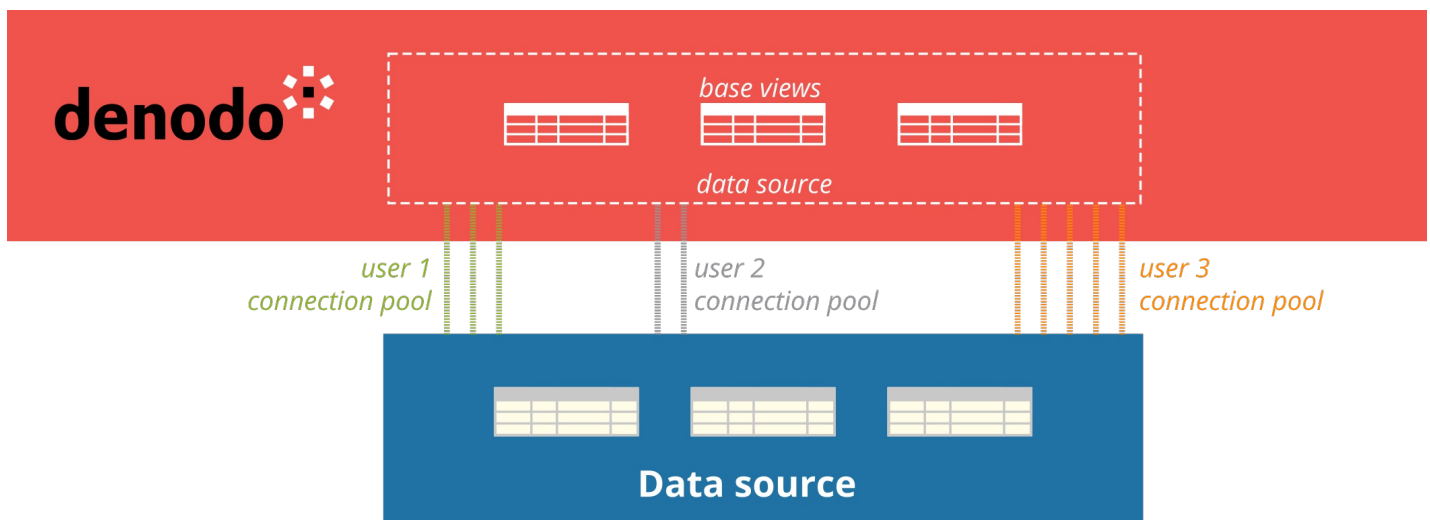
There is one connection pool that usually has a critical status, with a higher importance over the rest of connection pools: the connection pool configured to the cache system used by the Denodo Platform server. This pool is more critical because it will potentially be involved in answering many queries, as all queries in a database (and sometimes a whole server) use the same cache data source. So, independently of what other data sources they need, if they hit the cache they will all go through the same pipe. This makes it extremely important to size the pool to the cache system appropriately, otherwise the cache may become a performance bottleneck, which is the exact opposite of what the cache is for.

PASS-THROUGH CREDENTIALS

When using connection pools, it is important to keep in mind how they interact with pass-through authentication. At first, security concerns may seem unrelated to performance, but there is a connection. Pass-through authentication is a security scheme wherein the credentials that the data consumer used to connect to the Denodo Platform (the northbound connection) are reused in the connection from the Denodo Platform to a specific data source (the southbound connection). This is usually done with the idea that the data source itself has authorization rules that are dependent on the user that is requesting data, so by passing the northbound credentials to the data source, it can apply the additional security checks and rules that are already configured.

What is the problem, then, with connection pools? Remember how a connection pool works: When the server starts, it creates a set of connections to the data source. This is done before any query is received so these connections must be created with the same user credentials, and these credentials will potentially be different than the credentials that the Denodo Platform server wants to pass-through to the data source when a query is received. So pre-creating the connections does not work.

Another option would be to populate the server with connections as the queries are coming. Under this schema, the pool would be populated with connections using different credentials that depend on which users sent the queries to the server, but a future query issued by user A may receive a connection that was created with the credentials of user B. This would obviously not be acceptable so this solution does not work either.



Multiple connection pools for a single data source when using pass-through credentials.

What the Denodo Platform does when both the connection pool and pass-through authentication are enabled at the same time is to create a new separate connection pool for each user. This means that user A will have a connection pool for his queries that only that user will have access to, user B will have another connection pool, etc. Each connection pool will be created the first time that a user sends a query that accesses the specific data source, so the connection pools will be created on demand.

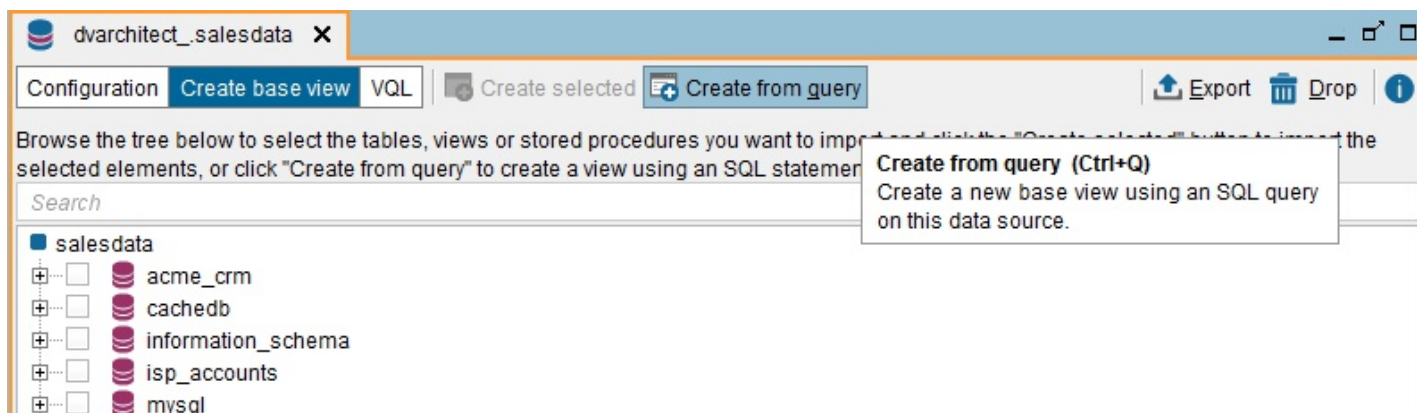
The implication of this schema from the performance angle is that the data source in question will create an unbound number of connection pools. Each new user that issues a query related to that data source will create a new connection pool to the source, and each pool may have multiple connections to the data source. This could result in a data source receiving more connection requests than it can process, so the performance could suffer. It is therefore very important that you use a combination of pass-through credentials and a connection pool only in the cases where you know that the number of Denodo Platform users accessing the data source will be small, or when the data source has enough capacity (in both maximum number of open connections received and processing power) to satisfy all the requests from the data virtualization server.

Base Views From SQL Query

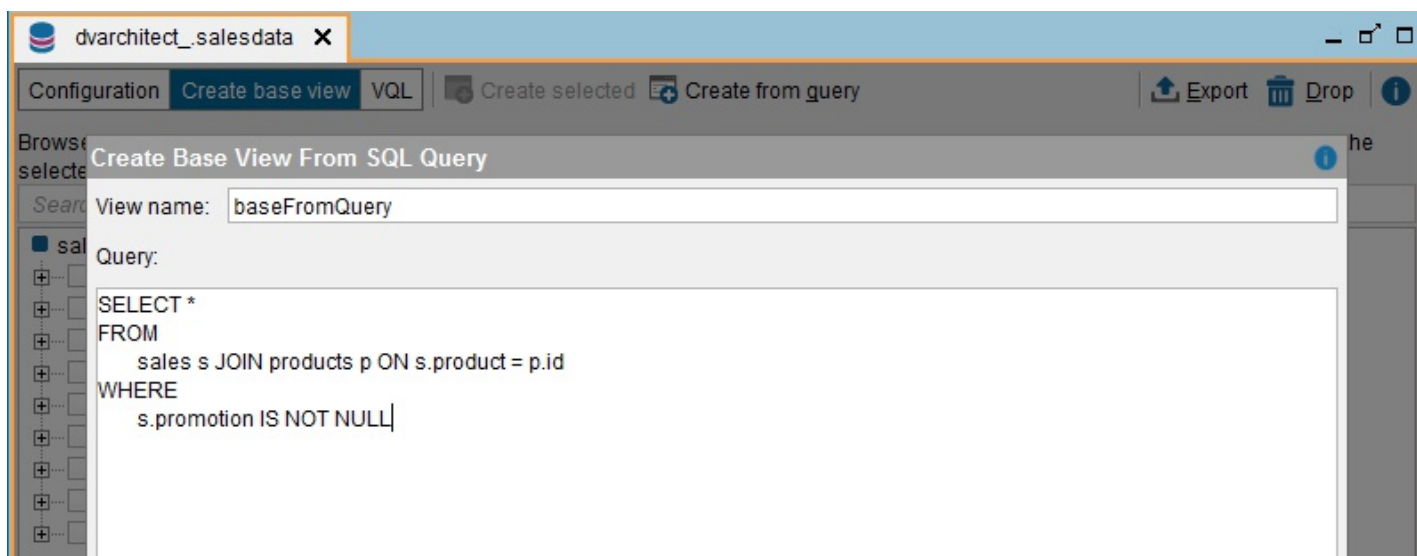
Another optimization option that the Denodo Platform provides is the ability to create base views from specific SQL queries. The most common way to create a new base view is to graphically introspect a data source, select one of the tables or views that the data source contains, and then click on “Create selected base views.” This, as you know, creates a base view in the Denodo Platform which refers to a physical table (or a view) in the remote data source.

Read [this entry in the Denodo Community Knowledge Base](#) for a more detailed explanation of base views created from a SQL query.

There is another way to create a base view in data sources that accept SQL: We can create a base view from a SQL query that we as users specify. This is done using the “Create from query” button in the data source introspection screen:



The "Create from query" button.



Creating a base view from a SQL query.

With this approach, we can specify arbitrary SQL queries in the definition of the base view, which offers two avenues for performance optimization:

- We can use vendor-specific hints and dialects, if a database vendor provides a specific hint or optimization that we want to use for our use case, and the default behavior of the Denodo Platform does not suit our needs. As this mechanism allows arbitrary SQL to be sent to the source, we have full flexibility in using vendor-specific syntax, optimization hints, or any other specialized feature that we may need.

Remember that the Denodo Platform uses vendor-specific SQL dialects by default, so we always recommend checking what is the standard behavior in your use case before going the base-view-from-query route.

- Using queries that are complex to express in the virtualization layer: In some cases, we already have a query defined elsewhere (for example, in a reporting tool) that gets us the data we need from the data source. A common approach is to reverse-engineer the query and create it in the data virtualization layer through regular base and derived view combinations, but sometimes this process may be too time-consuming or prone to introducing bugs in well-established queries. In those situations, we can just take the query that we already have and directly turn it into a base view in the data virtualization layer.

Once a base view is created in this manner it becomes a SQL black box. If we execute a simple `SELECT *` query over that base view, the Denodo Platform will just issue the underlying SQL query to the data source. But we are not just limited to these simple queries. The base view is a first class citizen of our data virtualization layer, so we can use it in combination with other views. If we end up combining this base view with other views that live in the same data source, the Denodo Platform will still be able to delegate the data combination to the source (as it would do with regular base views). In this situation, the base view created from a SQL query is treated as a text blob, so in the resulting query that blob is added as a named subquery in the `FROM` clause and then used in the general query sent to the data source, referencing it by name.

Note: delegating the definition of the base view as a subquery is the default behavior starting in version 6.0 of the Denodo Platform. In previous versions, you need to change the value of the field “Delegate SQL Sentence as Subquery” to “Yes” by opening the base view in question, clicking “Advanced,” selecting the “Search Methods” tab and clicking the “Wrapper Source Configuration” link.

For example, let's suppose a base view `myBaseView` was created from the following query:

```
SELECT * FROM sales s JOIN products p ON s.product = p.id WHERE  
s.promotion IS NOT NULL
```

If we issue the following query to the Denodo Platform:

```
SELECT count(*) FROM myBaseView GROUP BY product
```

The naive way to execute the query would first execute the query defined by `myBaseView`, retrieve all its rows into the data virtualization layer, and then execute the group by query. Instead, the Denodo Platform optimizes the query by sending the following SQL to the data source:


```
SELECT count(*) FROM
    (SELECT *
      FROM sales s JOIN products p
        ON s.product = p.id
     WHERE s.promotion IS NOT NULL)
GROUP BY product
```

The SQL defined in the base view is embedded as a subquery in the outer query, and the whole SQL query is delegated down to the source, thus benefiting from all the performance gains of source push down.

The takeaway is that this method of creating base views does not preclude delegation, so in the Denodo Platform it is possible to leverage performance optimizations that use native hints specific to individual data sources.

Note that creating base views from SQL queries should only be done when necessary; if the use case can be solved by defining the views graphically using the wizards in the VDP Administration Tool, that method should be preferred as it will allow for greater maintainability of the views during the regular operation of the system.

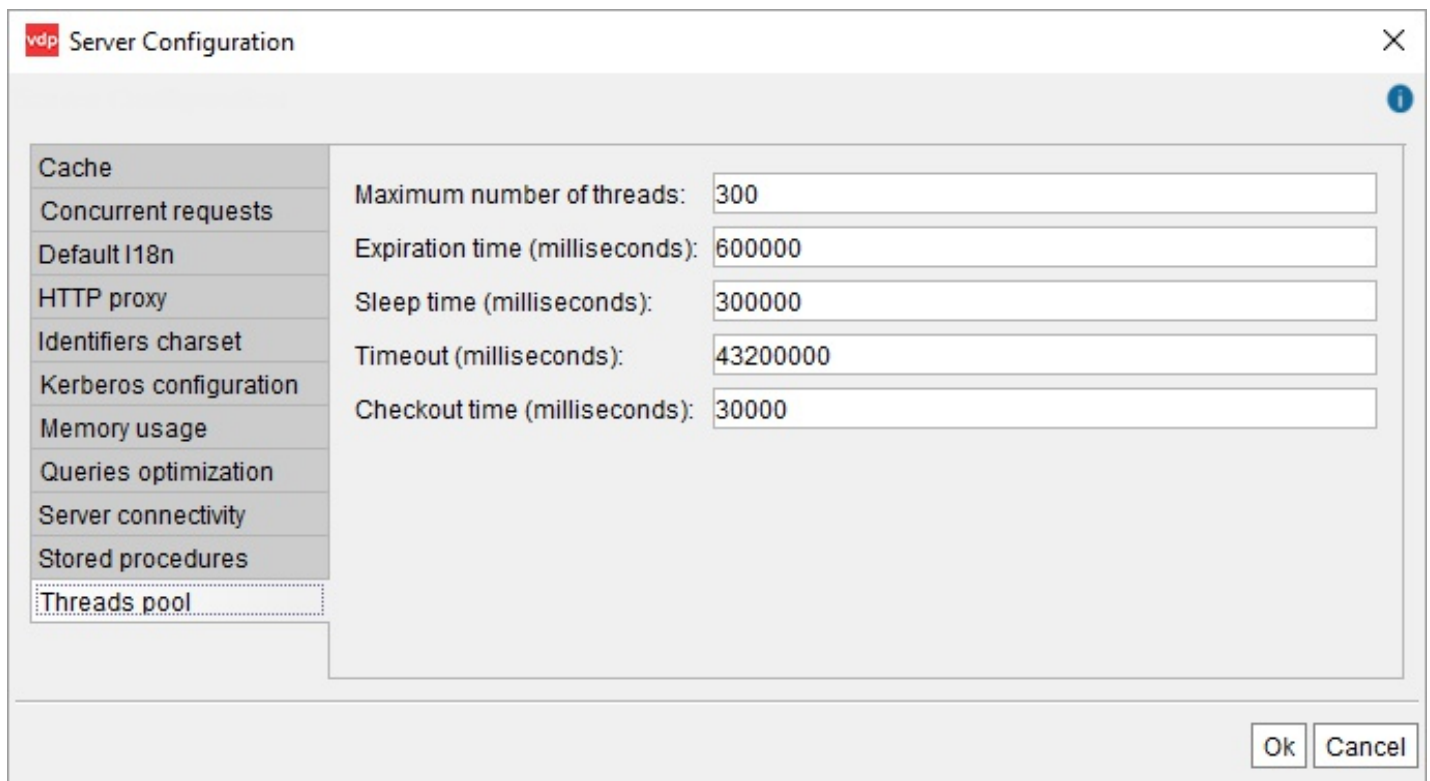
Denodo Platform Thread Model

The Denodo Platform can execute several operations concurrently, because the server is implemented in a multithreaded environment. In this context, a thread represents a lightweight version of an operating system process. It can execute operations on data independently and being lightweight, each thread requires fewer resources to be created and destroyed than does a traditional process.

Threads require fewer resources than OS-level processes, but the Denodo Platform has to maintain a balance between the number of operations that are executed concurrently and all the resources that are available to be shared across all threads (such as CPU cores, RAM, and network bandwidth). Finding this balance is key to achieving the right performance in each possible scenario. This is the policy that the Denodo Platform implements for thread creation:

- When a query is executed, the Denodo Platform creates a single thread to answer it. This means that the execution of each query is independent of any other query that is running on the server, unless there are shared resource bottlenecks (like CPU cores, available memory, or available connections in shared connection pools).
- When a query plan needs to query a data source, the Denodo Platform creates a thread for each specific access to that data source in the query.
- When data needs to be loaded in the cache, the Denodo Platform creates a new thread for each independent load process.
- The Denodo Platform creates separate threads for the embedded Apache Tomcat web server.

These threads are all assigned from a common thread pool that is created by the Denodo Platform server at startup time. The size of the thread pool is configurable by the user by tuning the values found in the Administration > Server configuration > Threads pool menu.



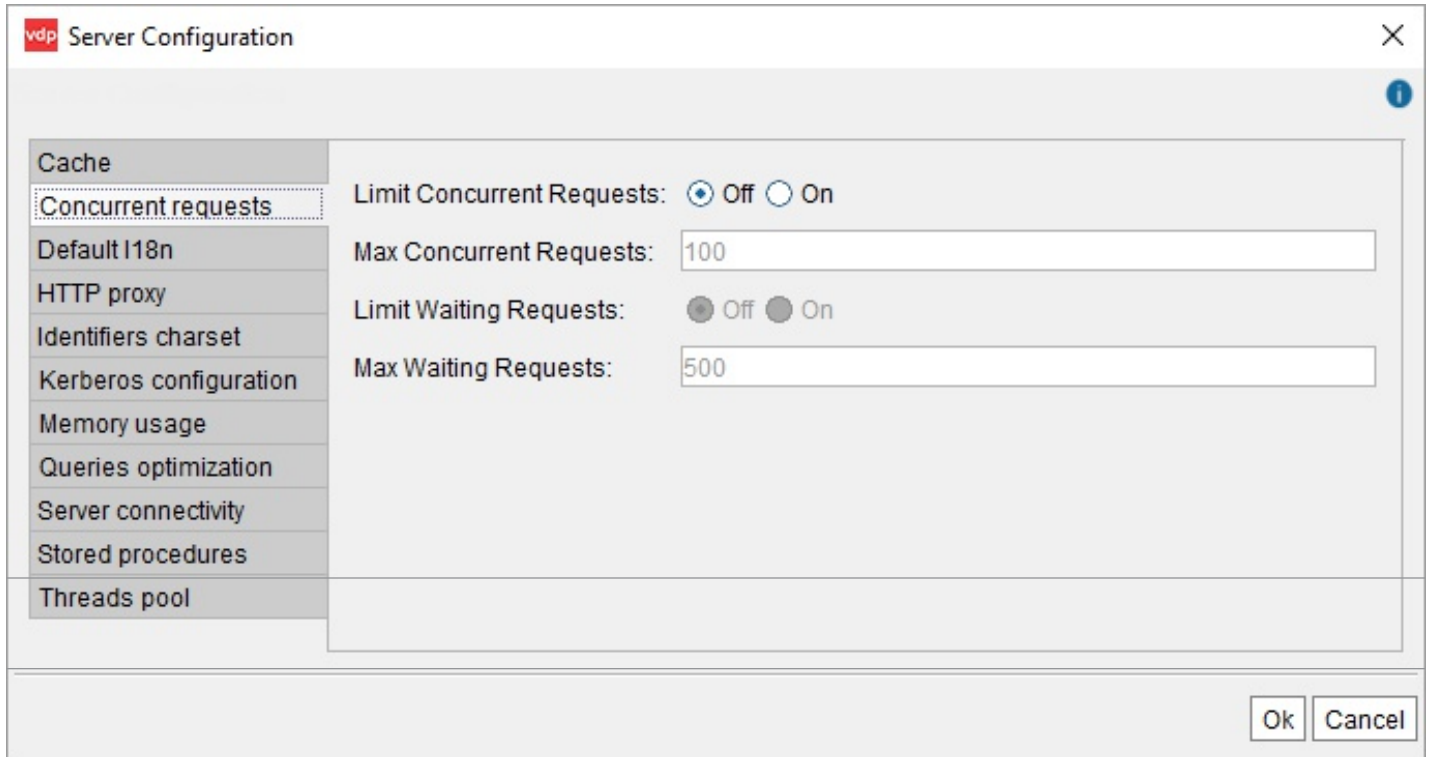
The screenshot shows the 'Server Configuration' dialog box with the 'Threads pool' section selected in the left-hand menu. The right-hand pane displays five configuration parameters for the thread pool, each with a corresponding text input field:

Configuration Parameter	Value
Maximum number of threads:	300
Expiration time (milliseconds):	600000
Sleep time (milliseconds):	300000
Timeout (milliseconds):	43200000
Checkout time (milliseconds):	30000

At the bottom right of the dialog box are 'Ok' and 'Cancel' buttons.

Server thread pool controls.

The biggest factor in the number of active threads in the Denodo Platform server is the number of concurrent queries received by the server. The server does not limit this number by default, but the administrators can throttle the number of concurrent queries that the server will process, to ensure the proper distribution of available hardware and network resources. This is done by modifying the values found in the Administration > Server configuration > Concurrent requests menu.



The screenshot shows the 'Server Configuration' window with the 'Concurrent requests' tab selected. The settings are as follows:

Setting	Value
Limit Concurrent Requests:	<input checked="" type="radio"/> Off <input type="radio"/> On
Max Concurrent Requests:	100
Limit Waiting Requests:	<input type="radio"/> Off <input checked="" type="radio"/> On
Max Waiting Requests:	500

Server concurrency controls.

Administrators can set both the maximum number of concurrent requests in the server and the maximum number of queries waiting to be executed.

VQL Considerations

Sorting

Sorting data is achieved in VQL/SQL using the `ORDER BY` clause. The section “Memory usage” described different types of operations based on their memory footprint; in that regard, sorting is always going to be a linear memory operation. It will always need to process the full data set before the results are returned. As such, you should avoid sorting

data as much as possible, as it will turn any operation you include it in into a blocking operation that consumes memory linearly. Always analyze the need for sorting and only use it when absolutely needed.

From a pure performance standpoint, sorting is a relatively fast operation, with typical average complexity of $O(n \log n)$, which is usually in the realm of acceptable performance even for big data sets. This operation is usually pushed down to the sources, which may include indexes that will assist in the sorting, making it even faster.

Finally, although sorting data incurs a penalty hit, the Denodo Platform optimizer sometimes uses it for performance optimization. The classic example is to request the two branches of a join to be sorted so the optimizer can use a merge join, which in most cases has the best performance profile of all join methods. This again follows the theme of the optimizer choosing suboptimal execution plans for the subnodes because that results in a better execution plan for the combined query; the combination of sorting both branches and using a merge join may be faster than retrieving both branches as they are and then using another join strategy like a hash join.

Aggregations

Aggregations are a basic operation in VQL (expressed using the `GROUP BY` clause) and one of the most important operations in reporting scenarios, so it is always important to understand the performance implications of executing aggregations.

In general, aggregations will have two kinds of behaviors:

- If the source data that is being aggregated is not sorted by the aggregation field(s), then the `GROUP BY` will behave as a blocking operation. The specific subtype will depend on the specific aggregation operation that is performed: Counting rows (for example, with a `count (*)` query) will be a constant memory operation, whereas nesting rows into an array will be a linear memory operation.
- If the source data is sorted by the aggregation field(s), then the operation will be semi-blocking. This means that the query will start reading rows until the value of the aggregation field changes; at this moment, there are more results pending, but those pertain to different result rows (because the aggregation value is different, and the data is sorted), so the results of the current group of rows can be calculated at that point. In effect, the query will block for each one of the groups, but once all the rows of a group are available the result row for that group can be calculated. This has a much lower memory footprint than aggregating unsorted data; sometimes the

optimizer will decide to sort the data before aggregating it if it estimates that the performance will be better that way. (This is especially true if the sorting can be delegated to the operations executed before the aggregation, including pushing it down to the data source level).

When we add an aggregation to a view tree, we are adding an extra operation that will add extra CPU and memory requirements to our query. Counterintuitively, however, joining views and aggregating the results may result in better performance than just joining the views:

- As we saw in the query rewriting section, the Denodo Platform will try to reorder the operations to get maximum performance. This means that sometimes the aggregations will be pushed under the joins (the data will be first aggregated, then joined); this is very common when querying star schemas, where a fact table is joined with one or more dimensions and then aggregated. This reordering in turn makes for the better performance of joins in the general case, as once the source data is aggregated the cardinality of the join will potentially be much lower (as after the aggregation there are fewer rows to join with the other side).
- If the aggregation is delegated to the data source, the number of rows that will be transferred through the network will be lower, achieving better performance.

The DISTINCT Clause

The `DISTINCT` clause in a selection operation removes duplicate rows. This introduces a slight change in behavior from the memory perspective, as once a row is returned as part of the result, the Denodo Platform will have to keep some data in memory as a reminder to not return further rows that have the same column values; in this sense the `DISTINCT` operation will be mostly equivalent to a `GROUP BY`. This means that operations that include the `DISTINCT` clause have slightly larger memory footprints, in proportion to the number of distinct rows in the source data set. This is usually not important, except for very big data sets with a lot of distinct rows.

Functions

DELEGABLE VS. NON-DELEGABLE FUNCTIONS

When using a function in a query you must be aware of the implications of using it from the performance standpoint. The biggest negative impact that a function can have is making a query (or subquery) non-delegable when it was originally delegable. The most common situation for this is using a function in the data virtualization layer that cannot be executed by the source because it does not provide such function.

The Denodo Platform is always going to recognize when functions are not delegable and will always try to move the application of those functions up in the view execution tree so as to delegate as much logic as possible to the data sources, of course while respecting query semantics. Unfortunately the semantics of complex query trees are often restrictive in the amount of reordering of function application that the Denodo Platform can apply.

The Denodo Platform will also help you understand when a function is not delegable through the query trace; when it finds this kind of situation the function will be displayed in this way:

The screenshot shows the Denodo Query Results interface. At the top, there's a toolbar with buttons for 'Execute', 'Query Results', 'Copy trace to clipboard', 'Save PNG', 'Stop', 'Refresh', and 'Save'. Below this, the query text is 'SELECT * FROM p_products'. The main area shows the execution trace with a tree of nodes: '<Execution Plan>', 'p_products', 'products', 'products', and 'products#0'. A tooltip is visible over the 'products' node, stating 'The function 'acos' cannot be delegated to this database'. On the right, a sidebar displays metadata for the selected node 'p_products': Name p_products, Database dvarchitected, Type PROJECTION, Execution time 136 ms, End time Fri Jul 15 11:02, Response time Fri Jul 15, Number of rows 100, State OK, Completed yes. There is also an 'Advanced' button at the bottom of the sidebar.

A non-delegable function displayed in a query execution trace.

In summary, always be mindful of the capabilities of the underlying data sources that you are querying and plan your views to use functions accordingly.

DELEGABLE CUSTOM FUNCTIONS

The Denodo Platform offers an API for extending the core engine by adding custom, user-defined functions. These functions are written in Java and allow the user to add the functionality of any Java library to their data virtualization solutions.

By default, such functions are not delegable, as they are arbitrary code that may or may not have any mapping to the source capabilities; but sometimes a custom function may have an equivalent in some data sources. The Denodo Platform enables users to specify a delegation equivalent for custom functions against specific data sources (this is only available for JDBC data sources). For example, we can notify the Denodo Platform that the new custom function

X can be delegated to Oracle using a specific syntax, to SQL Server using a different syntax, and nothing else (if executed against any other source, the query sent to the data source will not include the function, and it will instead be executed at the Denodo Platform level).

The specific mechanism for creating these delegable functions is described in the [Virtual DataPort Developer Guide](#), in the section “Developing Custom Functions that Can Be Delegated to a Database.” In a nutshell, for each data source that accepts this function, we define a record with three fields:

- Database name.
- Database version.
- Pattern, which defines the syntax that will be used when delegating the function to the source. This pattern is a string with placeholders for parameters specified with the syntax `$0`, `$1`, `$2`, etc. For example, a pattern could be `“FUNCTION_IN_DB($0,$1,$2)”`.

With these three items the Denodo Platform will be able to delegate the function successfully to that specific source.

This mechanism can also be used to effectively “import” functions that are supported in a specific source but not by the Denodo Platform. The procedure would be to create a new custom function, marking it as a function with no implementation (using the annotation `implementation=false`) and configured to be delegated to the source through the delegable custom function syntax seen above. Normally the Denodo Platform does not accept a custom function that is not implemented, but this is overlooked when using the delegation syntax. This enables the Denodo Platform to use the function in any query, but with a caveat: If the query is such that the function is applied at a point that is not being delegated to the data source, the query will fail because the custom function does not have an implementation and thus it needs to be delegated to the data source.

The CONTEXT Clause

The `CONTEXT` clause of VQL is a very important tool when designing queries to be used in a production environment where they are issued automatically from either a BI/reporting tool or a custom-made application (web, mobile or desktop). This VQL clause is specified at the end of the query string and enables the user to configure many of the options that have been discussed throughout this book (and many others), in a programmatic way as opposed to graphically. Some of the options that are available through the `CONTEXT` clause are:

- Cache control (loading, invalidating, disabling for individual queries, etc.).
- Overwriting the join method for a specific query.
- Specification of data movement parameters.

The `CONTEXT` clause has a very rich set of options, so it is strongly recommended to check the VQL reference found in the “Denodo Virtual DataPort Advanced VQL Guide” that lists all the possible hints that can be specified in the `CONTEXT` clause and details the syntax and semantics for each.

Other Considerations

This book has covered performance topics from different perspectives, with the intention of pushing the readers to do their own testing in their Denodo Platform environments and find optimal solutions for their data virtualization scenarios. Here are a few important considerations when dealing with real world performance measurements.

The first refers to the methods used to reach your conclusions. The most common mistake when evaluating query performance is to run them a single time and draw conclusions from that execution. This is very problematic because the execution time of a query is not a set quantity that will be perfectly repeatable every single time. Execution time varies every time the query is run. If these variations are not taken into account when drawing conclusions, the insights gained by the analysis could be very flawed.

In general, you should take the same steps you would in any other type of statistical analysis. At a minimum:

- Run each query a multitude of times and write down all the resulting execution times. For extra points, write down not only the total execution time but also the execution times of intermediate nodes that may be important (such as base views and key intermediate calculations). An easy way to do this is actually saving the full execution traces of each query as a text file and then gathering the data from the saved traces.
 - An often-overlooked data point is the execution time and the latency of each of the data sources involved in a query. Always pay attention to those values; users are often surprised at the actual behavior of their data sources and how it relates

to their expectations. Pairing that data with the actual queries sent to the data sources (also present in the execution traces) is excellent support material when talking to the DBAs in your organization. Execution traces prove to be invaluable, once again.

- Once the raw execution times are gathered, calculate average times as needed. But don't stop there. For each time, calculate at least the mean value and the standard deviation. These are very important data points in understanding the behavior of the query. For example:
 - A big difference between the average and the mean could signal the presence of outliers and/or a skewed distribution.
 - The standard deviation of each value will indicate how much “spread” there is. A low standard deviation means the values are tightly packed around the mean value, whereas a large standard deviation shows the values are spread across a bigger range.
- When you draw and report on your conclusions, always state the sample size that was used (the number of runs for the query), in addition to the conditions under which the test was executed.

Those are the most basic steps that should be done, at a minimum. As with any other statistical analyses, it can be useful to analyze further to try to characterize your queries. Plot the histogram of values, try to fit them to a distribution, and check for the presence of outliers, as they can have a big impact in a real world scenario, even if they are not common events.

Second, realize that data virtualization scenarios, by their nature, have multiple moving parts that will affect the performance of the running queries. Many factors may influence performance, including:

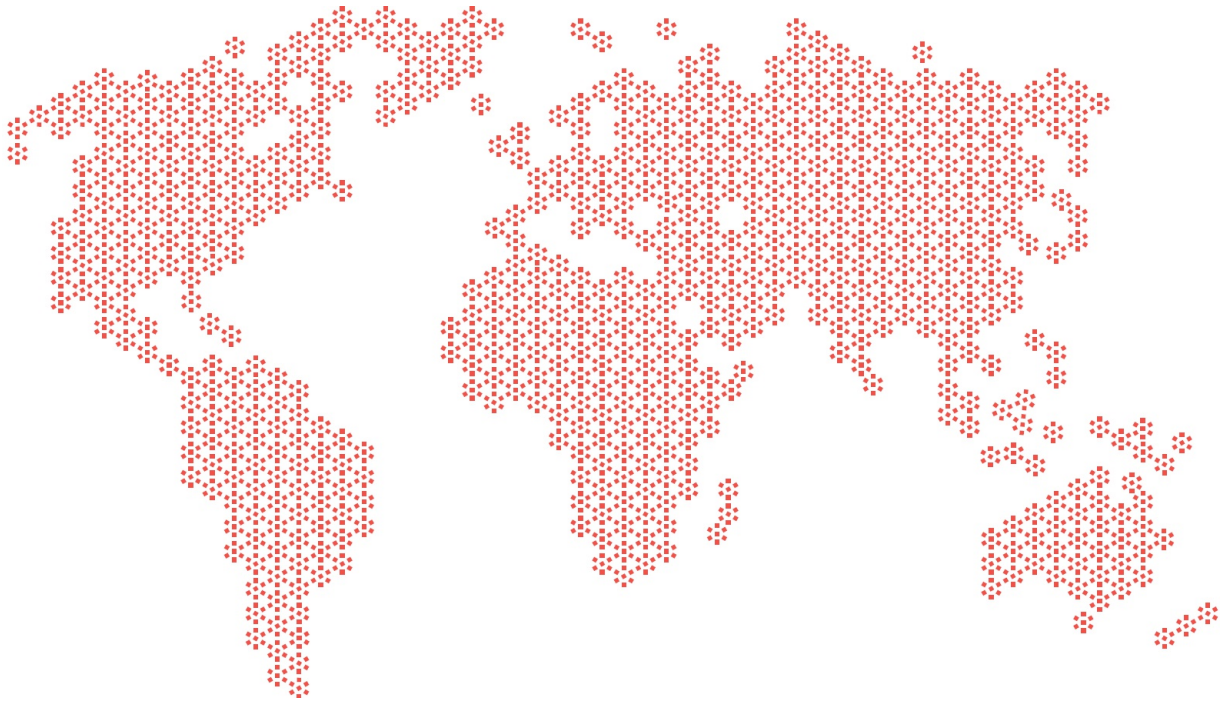
- Data source behavior: the latency, average execution time, bandwidth, etc.
- Network performance, including bandwidth, latency, packet drop rates, etc.
- Hard drive behavior, including maximum and average read and write speeds (both sequential and random), seek time, traditional spinning media vs SSD performance issues, etc.
- CPU and memory load in the server, including the other processes that are running concurrently with the Denodo Platform. Are there any processes scheduled to run at specific times in your production servers? Could they affect the performance of your data virtualization solution?

All these factors, and many others not listed here, mean that you must test each query several times in different scenarios so you actually assess the performance of each query under different circumstances; the same query that performs bad in scenario A (for example, under heavy transactional load) may perform perfectly well in another scenario (for example, under heavy analytical load). Do not assume that a query tested in a specific configuration will behave similarly in a different real world scenario.

The takeaway should not be surprising: Taking the proper time to fully understand the behavior of your queries is critical in your path to success with data virtualization.

Summary

In this Cookbook, we have reviewed the basic components and techniques involved in query optimization in data virtualization scenarios with the Denodo Platform. This basic introduction to the topic covered both the features that the Denodo Platform offers for manual intervention and the automatic capabilities that assist the user in obtaining the best performance at all times. We recommend using this manual as the starting point for anybody interested in designing and optimizing queries using the Denodo Platform, and we encourage everybody to expand their knowledge of these important topics, starting with the resources at the [Denodo Community](#), including the [Questions & Answers](#), the [Knowledge Base](#), and Virtual DataPort documentation such as the [Virtual DataPort Administration Guide](#), the [Virtual DataPort Advanced VQL Guide](#) and the [Virtual DataPort Developer Guide](#).



Denodo

Denodo is the leader in data virtualization – the only platform that delivers Information-as-a-Service across disparate structured, web and unstructured sources. Innovative leaders in every industry use Denodo to dramatically increase flexibility and lower costs of data integration for agile BI and reporting, call center unified desktops, customer portals, web and SaaS data integration, and enterprise data services. Founded in 1999, Denodo is privately held.

Denodo North America & APAC

530 Lytton Avenue, Suite 301
Palo Alto, CA 94301, USA
Phone (+1) 650 566 8833
Email info.us@denodo.com & info.apac@denodo.com

Denodo EMEA

Portland House, 17th floor, Bressenden Place
London SW1E 5RS, United Kingdom
Phone (+44) (0) 20 7869 8053
Email: info.emea@denodo.com

Denodo DACH

3rd Floor, Maximilianstraße 13
Munich, 80539, Germany
Phone (+49) (0) 89 203 006 441
Email info.emea@denodo.com

Denodo Iberia & Latin America

C/ Montalbán, 5
28014 Madrid, Spain
Tel. (+34) 912 77 58 55
Email: info.iberia@denodo.com & info.la@denodo.com