denodo

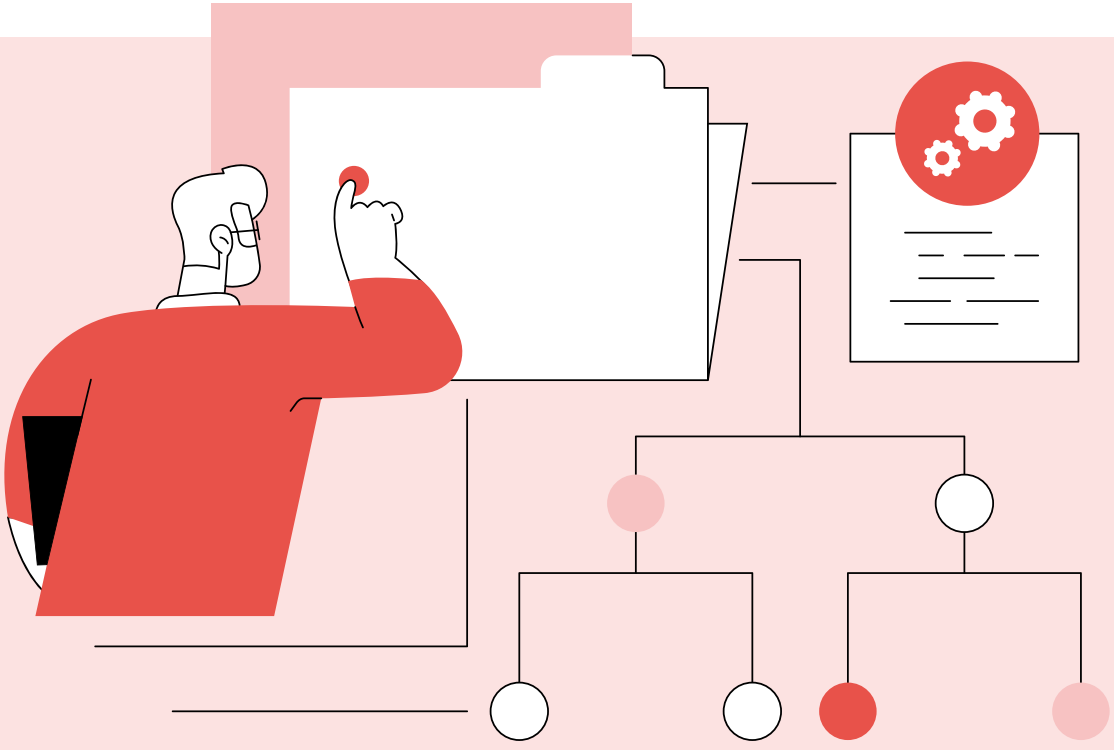
Whitepaper

Performance in a Data Fabric

Comparing Data Virtualization Architectures

By **Pablo Álvarez Yáñez**

Table of Contents



Overview

One of the critical ideas behind data fabric is the capability to access any data asset in the organization through a central, easy-to-use access point. End users should not have to deal with the complexities of the data ecosystem behind the scenes, nor should they have to understand the nitty-gritty details of each and every database and application in the organization.

This can be achieved using a data virtualization layer that abstracts the complexity and offers a central access point. In addition to centralized access, this layer often provides other capabilities, like caching, security, modeling, and cross-source federation, which can be enforced with consistency across the organization. All these features enable end users to feel that all company data is consolidated and stored in a single system, even when the data is spread across dozens of heterogeneous systems.

The main goal of a data fabric is, from a business perspective, to create an agile platform that improves time-to-data by means of a self-service data layer that exposes data in a language that the business can understand and use.

Data fabric vendors implement two main architectures to provide this capability:

- Specialized data virtualization layers
- Data engines with data virtualization extensions

In this paper, we will explore both architectures in detail, and we will focus on the implications that these implementation decisions have in terms of the performance of query execution.

To further illustrate the differences between these two architectures, we have performed extensive benchmarks using TPC-H, which shows how both architectures perform under different scenarios. You can find the detailed explanation of the test methodology and environment specs in the Benchmarks section below, but here is a quick summary of the results:

Scenario	Specialized data virtualization layer: The Denodo Platform	Data engines with data virtualization extensions: Leading data lake vendor
Access to an external source	26.52 seconds	5 Hours 8 Minutes 28 Seconds
Federation of 2 external sources	3 Minutes 19 Seconds	2 Hours 27 Minutes 15 Seconds
Federation of data lake with small external source	2 Minutes 29 Seconds	2 Minutes 13 Seconds
Federation of data lake with large external source	6 Minutes 16 Seconds	4 Hours 10 Minutes 32 Seconds

These results showcase the power of specialized data virtualization layers in distributed landscapes, in which the complexity and specialization of their engines surpass the potential advantage of data lake parallel engines for access to external datasets and multi-source federation.

Introduction: Data Virtualization Architectures

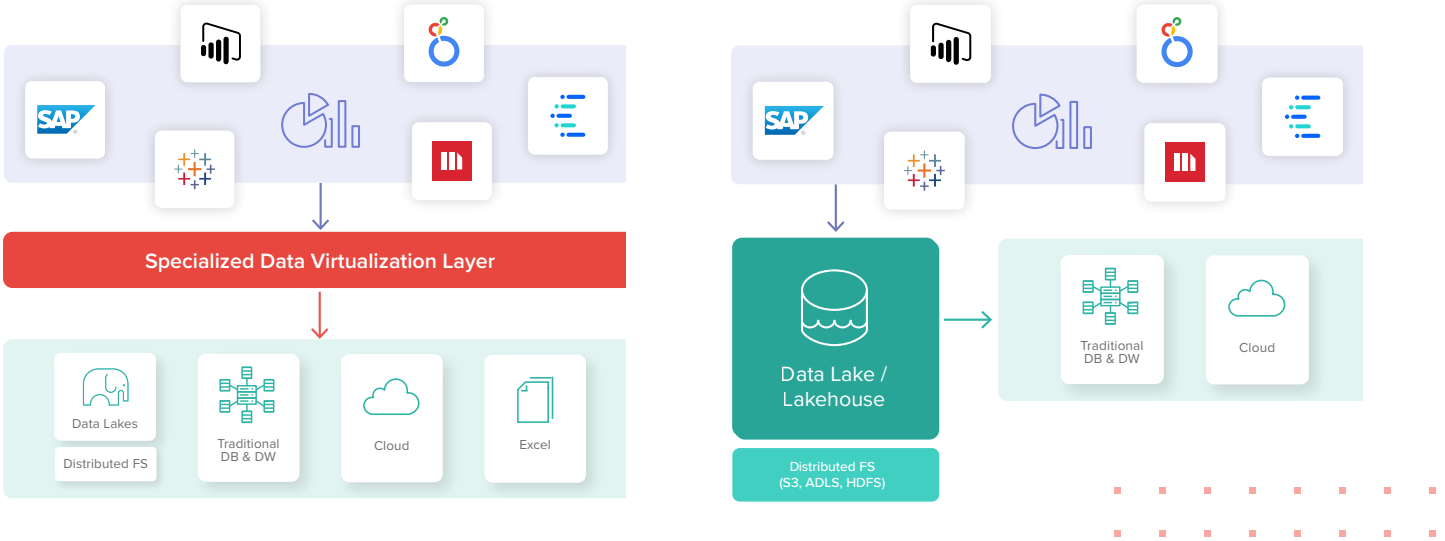
There are two major types of data virtualization technologies used by data management vendors to provide a common access layer across multiple data sources. In this section, we compare their commonalities and differences.

SPECIALIZED DATA VIRTUALIZATION LAYERS

In this type of architecture, a layer sits above all data sources to provide a centralized access point. It analyzes incoming queries and routes each request to the data source that contains the corresponding data. This process is called “query push-down” or “query delegation.” As a query may involve tables from multiple sources, this type of software needs to include an engine with cross-data source federation capabilities and a purpose-based optimizer for those scenarios. Other techniques such as caching, and aggregate-aware acceleration, are also frequently used. Denodo is a vendor in this category.

DATA ENGINES WITH DATA VIRTUALIZATION EXTENSIONS

In this architecture, a data system includes an extension so that, in addition to its own data, it can also link to external data sources. Early examples of this architecture are tools like Oracle DB links or Microsoft SQL Server Linked Servers. Currently, many data lake engines with MPP capabilities implement this type of architecture, such as Spark, Dremio, or Starburst (Trino). Therefore, the focus of this paper for this category will be on data lake engines. In these systems, when external data is requested, the external tables are queried by the worker nodes and fed into the parallel engine processing pipeline. Techniques like caching are also available for vendors in this category.



Specialized vs Data lake extension

Both architectures enable end users to run queries in a distributed data landscape, but the processing styles are significantly different. In the next section, we will dive into the repercussions that these design differences have in query execution performance.

But first, it is worth noting that specialized data virtualization solutions typically include additional capabilities for creating and governing a semantic layer across multiple data sources, such as advanced modeling, data lineage, and governance. Data lake vendors tend to be more focused on the execution of queries with data in object stores. Although an analysis of those functionalities is out of the scope for this paper, you can find a more in depth discussion in the whitepaper “Unleashing the Full Potential of Your Data Ecosystem.”

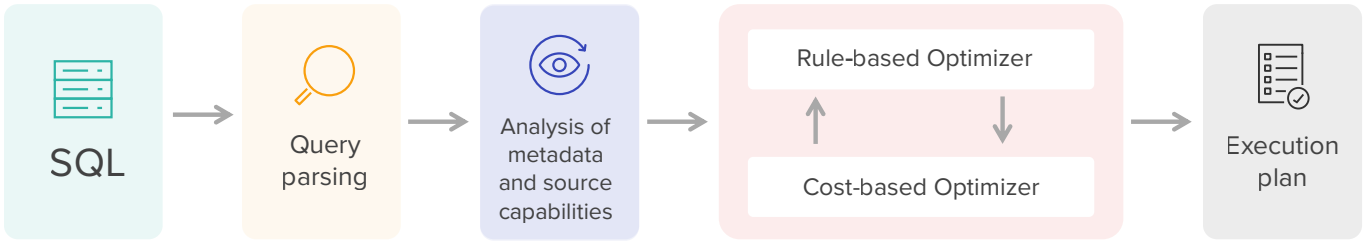
Comparing Query Execution in Data Virtualization Architectures

Let's now focus on how these architectures operate and the implications these architectural differences have on query execution and performance.

SPECIALIZED DATA VIRTUALIZATION LAYERS

Specialized data virtualization layers act as relational database systems, but with one big difference: they only host metadata. They represent the metadata of objects like tables, views, and stored procedures, just like any database, but they do not actually host the data. Data results are always fulfilled from the original data sources, or from the cache, and are queried on demand. This metadata contains not just table names, columns, and data types, but also all of the information necessary to perform the queries to the underlying sources. Some of these details are the data source type, version, and vendor; data type and structure mappings; data statistics for cost estimation; and specific configuration, like bulk load setting, pagination in APIs, etc. This architecture usually supports a broader range of data sources than data engines with data virtualization extensions, which are often restricted to RDBMS and noSQL.

When an end user issues a query, the optimizer builds an execution plan. This involves several steps, starting with an analysis of the metadata of the tables/views involved in the query.



A specialized data virtualization engine's query optimization pipeline.

During the initial metadata analysis, the engine can detect two possible scenarios and will use this information to compare different possible query execution techniques:

1. **All data required to answer the query is in the same system.** Based on the characteristics of the data sources, the engine can decide to:
   a. Leverage the processing of the underlying system. This technique is called "query push-down," and it is the preferred approach if the underlying data source has processing capabilities, like a relational database, MongoDB, or even systems like Salesforce.
      i. This step is critical, as it enables the engine to take advantage of all of the processing-power-related features of the sources, such as parallel processing, indexes, caches, partitions, and other artifacts that the source engine has available to accelerate execution.
      ii. To do this efficiently, the engine needs to have a full understanding of data type mappings, SQL dialects, and the syntax of the transformation functions for each vendor/version. The better the quality of the SQL translation, the better the performance will be.
   b. Use its own engine. This technique is normally called "post-processing," and it is the preferred strategy if the data source does not have processing capabilities (e.g. an Excel spreadsheet). In this case, the execution plan would bring the entire dataset to the virtualization engine and perform any additional operations (e.g. column transformations, aggregations, etc.).

2. **Data is spread across multiple systems.** In this case, the data virtualization optimizer needs to choose among a variety of techniques for operations like joins or aggregations (in-memory merges, hash joins, nested loops, on-the-fly data movement to temp tables, etc.) and query rewriting rules (branch pruning, partial aggregation splits, etc.). The cost- based optimizer plays a significant role, as it estimates partial volumes using comprehensive data source statistics and other data source details (e.g. size of the cluster for an MPP system) and uses them to weigh the cost of different execution options before choosing the optimal one. This is the most complex part of the engine in this type of architecture, and the performance of the query will depend enormously on the decision made by the optimizer. We will see an example of this below.

3. **A combination of both techniques.** In most queries, even if the data is spread across multiple sources, execution uses both of the techniques described above, as it may be possible to push down some operations and post-process others. For example, you may be joining three tables, but if two of the tables are in the same source, that one JOIN can still be pushed down in a single execution branch.

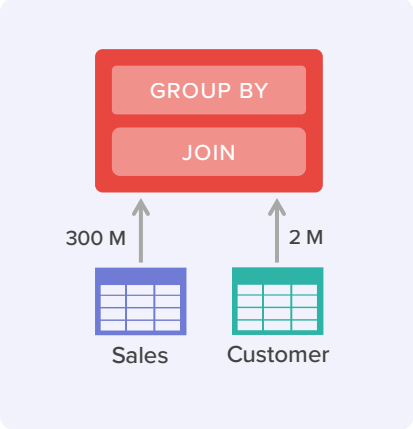
The result is an execution plan with one or several "execution branches," often executed in parallel, to reach each of the sources and retrieve a partial dataset. The final datasets are composed by the upper processing stages of the data virtualization engine using the technique chosen by the optimizer and sent to the client tool.

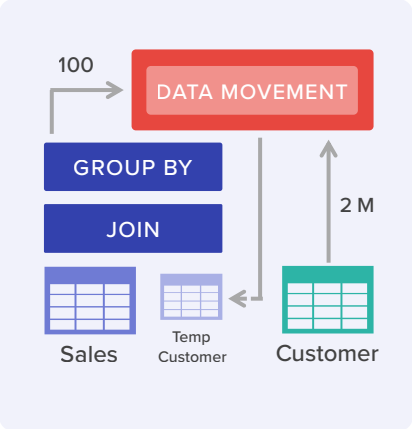
To illustrate these effects, let's use a simple query as an example:

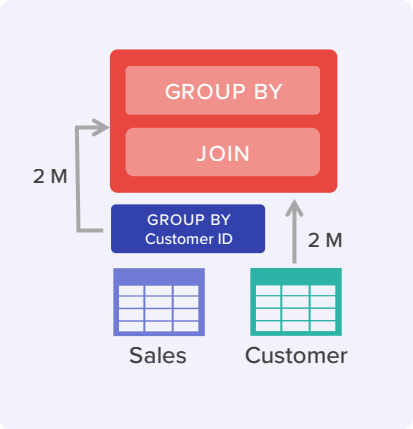
```
SELECT c.country, SUM(s.amount)
FROM psql.customer c JOIN sf.sales s
ON c.id = s.customer_id
GROUP BY c.country
```

This query calculates the total sales for each country. For context, let's say that the table Customer is in PostgreSQL and has 2 million rows, and the table Sales is in Snowflake and has 300 million rows. How can it process that query?

The optimizer will generate multiple combinations of algorithms and query rewriting rules, estimating data volumes involved in each step, and choosing the most optimal one based on estimated cost. For example, here are three candidate plans that the optimizer could generate:







Candidate #1
Naive Strategy

Candidate #2
Data Movement

Candidate #3
Partial Aggregation Pushdown

Let's look at these candidates in detail:

Candidate #1 is the most naive and a direct translation of the SQL query. Data is pulled from both sales and customer tables, joined in memory by the data virtualization engine using one of its available techniques (e.g. merge or hash join), and then aggregated by country to generate the final output. The weakest point in this strategy is that a lot of data needs to be moved (302 million rows) and processed in the engine to produce a relatively small result of a few hundred rows.

Candidate #2 uses a completely different technique, a multi-step execution. In the first step, the customer data is retrieved from PostgreSQL and moved to a temporary table in Snowflake. In a second stage, with all data available in Snowflake, the entire query processing is pushed down to Snowflake and sent to the consumer. As we can see, data movement has been significantly reduced to a mere two million rows, and all processing has been pushed down to Snowflake to take advantage of its processing power.

Candidate #3 goes back to a single-step execution, but instead of a naive plan, it rewrites the query to a more efficient form. To maximize query push-down, the aggregation is split into two steps: first by customer ID and second by country. Although this may seem counterintuitive, it significantly reduces network traffic and processing in the data virtualization engine, while leveraging the MPP capabilities of an engine like Snowflake.

How does this affect execution times?

Execution Plan	Time
Candidate #1	57 min 2 sec
Candidate #2	10.26 sec
Candidate #3	15.23 sec

As seen here, a bad plan may take an hour, whereas an optimal plan just takes a few seconds. In this case, candidate #2 would be the optimal one, and is the one chosen by the optimizer to run the query.

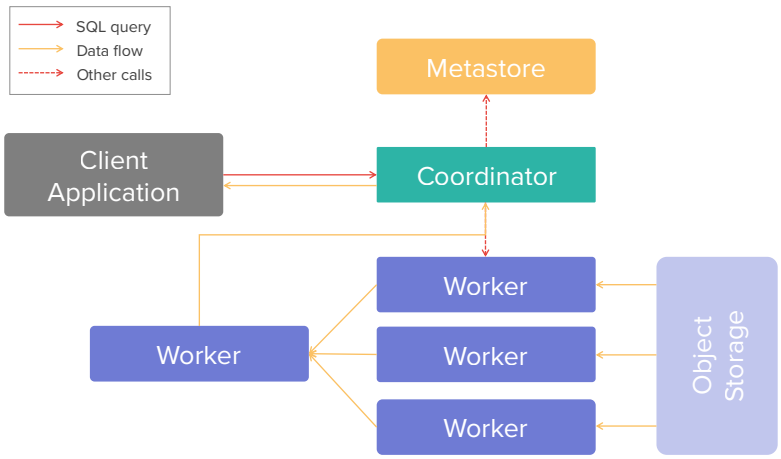
This example just scratches the surface of the problem. For example, we're not touching on topics like join algorithms or table order. Also note that we are not using any caching or aggregate-aware acceleration, just real-time execution (see the section "Other acceleration techniques" below for more details on those options). But it's already easy to see that an advanced and complex optimizer is critical to produce results in a timely manner.

DATA LAKE ENGINES WITH DATA VIRTUALIZATION EXTENSIONS

Modern data lake engines are massively parallel processing (MPP) systems in which the execution engine has been decoupled from storage. Originally created for the Hadoop ecosystem, they quickly evolved to adapt to cloud environments and different flavors of object storage, such as Amazon Web Services (AWS)'s S3 and Microsoft Azure's ADLS. The idea of separating processing and storage also opened the door for these engines to incorporate connectors to other systems, like external relational databases or noSQL platforms.

Data lake MPP engines are deployed in multi-node clusters and normally have three types of nodes:

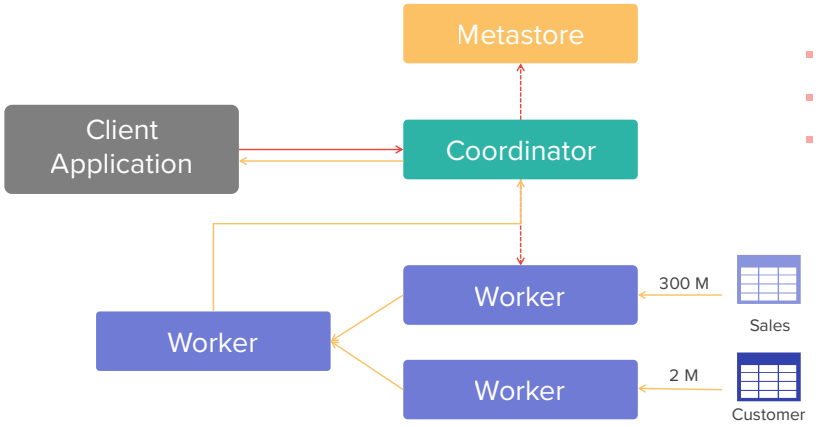
- **Coordinator node.** It is responsible for parsing queries, creating execution plans, and managing worker nodes. The coordinator is also responsible for returning the final result-set to the client.
- **Metastore.** It stores metadata information of the tables, including columns, data types, partitions, and statistics.
- **Worker nodes.** These are responsible for executing the query plan and processing data. Workers are also in charge of fetching data, normally from object storage, and can communicate with each other.



Additionally, as seen in the picture above, datasets are often stored in the object storage (a distributed file system, often in the cloud like S3 in AWS, ADLS in Azure, etc.) in the form of files in formats specialized for analytics, like Parquet, ORC, and more recently, table formats like Delta or Iceberg.

In terms of the execution pipeline, when the coordinator receives a query it parses it, maps it to the information in the metastore, and uses its optimizer to create a distributed query plan. The query plan is formed as a hierarchy of tasks that will run on the worker nodes. Each task operates on a split of data, that is, a portion of a larger dataset. This enables the execution to be parallelized. For example, you can divide the task of accessing a large Parquet file into multiple workers that access and process different portions of the file. The coordinator keeps track of what task is executed where.

So far, we've seen how execution works with data in the lake (e.g. Parquet files in object storage), but how does this architecture apply to external data sources? Consider the PostgreSQL and Snowflake datasets we used as an example in the previous section. In this case, the coordinator would instantiate tasks to retrieve those datasets, via JDBC, to a worker node. The worker nodes will retrieve the dataset and from there on, will use the MPP engine to process it as it does when data comes from object storage.



However, unlike access to Parquet files in object storage, which can be split into multiple workers, access to external data is channeled via JDBC connections that map one worker to each table. This limits the level of parallelism that can be achieved, and as seen in the figure above, creates a scenario similar to Candidate #1 described in the section above, and it shares the same problems discussed there with regards to data transfer. Furthermore, the optimizers of data lake engines do not offer the variety of advanced techniques for federation that a specialized data virtualization engine includes, and that we've seen in the section above. Their focus is to provide a powerful architecture for processing data in the lake (e.g. parquet, ORC, Iceberg, etc.) and the federation capabilities simply reuse that same processing pipeline.

Most data lake engines also use this same execution model even when all data is in the same data source, as the connectors do not have advanced SQL dialect conversion logic, mapping of complex data types and other information about data source logic, like for example, data collation (the way in which a data source orders non numerical values). This means that data lake engines will perform scans of each table and use their own engine to process queries from external sources. This approach can have some advantages (e.g. to reduce the workload of legacy sources), but it also has significant issues. For example, it is not able to leverage internal structures of the source to accelerate query processing, like indexes, partitions, caches, etc.

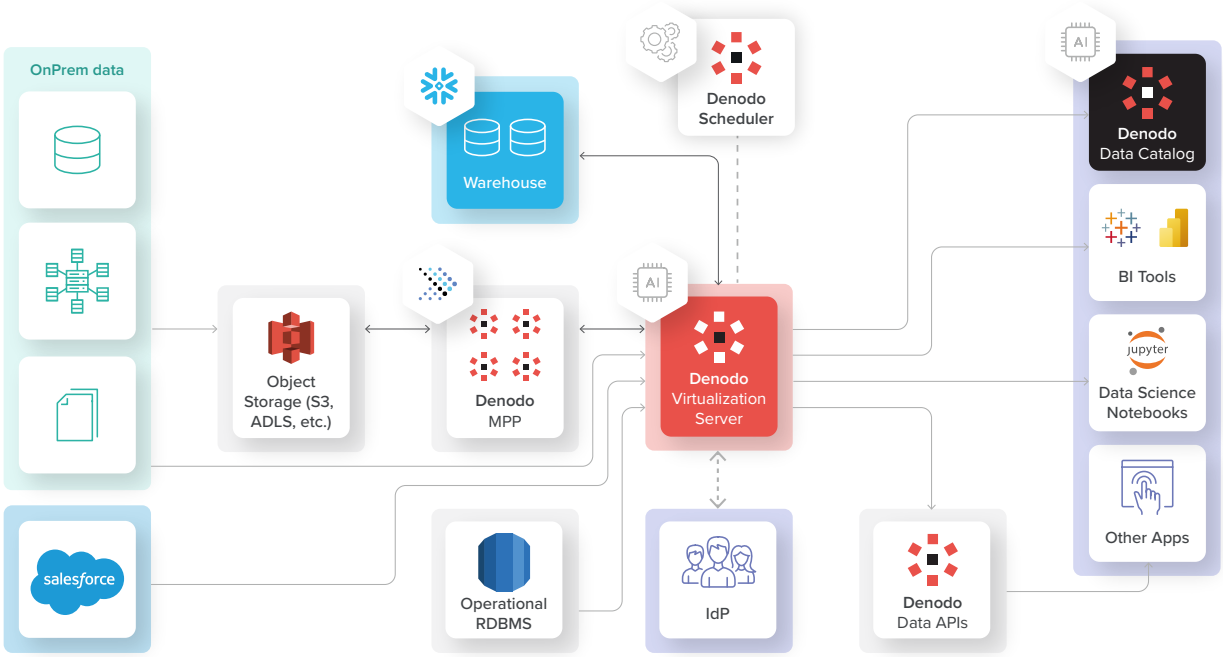
Finally, it's also worth noting that in data lake-centric ecosystems, it will be common to see queries that mix data lake content with external systems. In those cases, the engine will use parallel access to data lake files and individual access to external content, in a mix of the two scenarios described above.

HYBRID APPROACHES

Currently, we can see that many vendors are crossing the lines across architectures, and it is common to see hybrid offerings that blend concepts from both of these approaches. In fact, the coordinator node of a data lake engine and an instance of a specialized virtualization server have much in common.

Data lake engines are often able to push down simple predicates, like filters in a WHERE clause. Some data lake vendors advertise better push-down capabilities as part of their enterprise offerings, which expand on what their open-source versions offer. These enable the engine to push down some additional operations like joins. That said, in general, data lake engines have more limited capabilities for push-down, in part due to limited SQL dialect translation capabilities, and in part due to the lack of advanced rewriting techniques as we mentioned in the previous section.

On the other hand, you will also see several specialized data virtualization vendors including MPP engines in their offering. This enables those engines to provide better integration in data lake scenarios and also take advantage of parallel processing in the upper stages of processing in multi-source queries.



Denodo, for example, has embedded Presto, an open-source data lake engine, as a component in its architecture. It can be used for data lake processing as well as for accelerating execution in federated scenarios.

OTHER ACCELERATION TECHNIQUES

Although the focus of this analysis is on real-time queries, many vendors in this space offer other acceleration capabilities to increase performance.

Caching

Caching is offered by almost all vendors. Caching enables the engine to reuse results that have been previously calculated. Most vendors also offer capabilities to update the cached content and keep it current, often with increments, to avoid a full refresh whenever possible. The Denodo Platform contains a specialized component to orchestrate cache loads, among other things: the Denodo Scheduler.

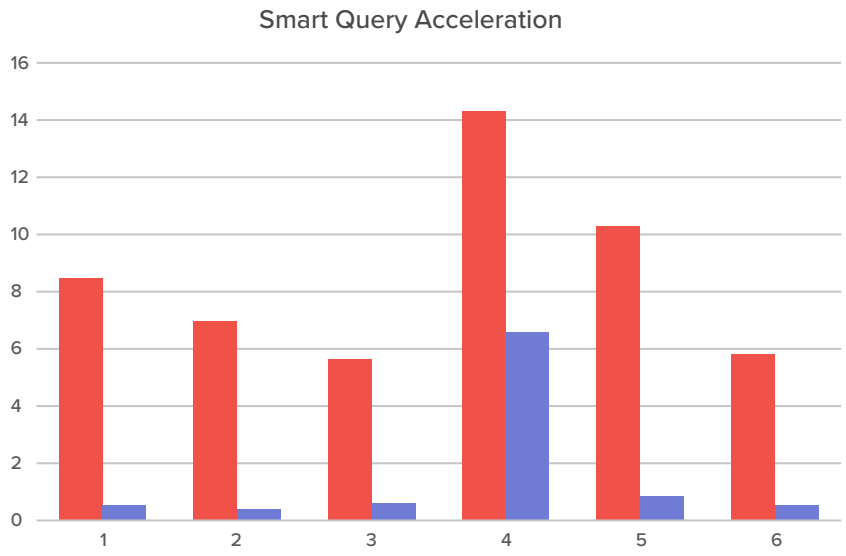
When the optimizer detects a "cache hit," which can be total or partial, it redirects the engine to the cache system to retrieve that data instead of recalculating it from the original tables. There is often a configurable time-to-live (TTL) parameter that tells the system how long the cached calculation is still valid to avoid returning stale data.

Aggregate-Aware Acceleration

Additionally, a few vendors, including Denodo, offer aggregate-aware acceleration. This technique takes advantage of the smaller datasets produced by aggregations and the fact that analytical queries often aggregate raw data to produce final results. Think, for example, of the query "total sales by country" described in the section above. Aggregate-aware acceleration is based on the idea that intermediate, pre-aggregated results can be used as the basis for calculating the final results much more quickly than using the original tables. There are two key benefits that make this technique incredibly useful in analytics:

- It offers great acceleration factors, very often 10x, 20x, or even 50x faster than the original queries. At the same time, it often provides a much higher number of "hits" than a traditional cache. That is, a single acceleration aggregate can be used by hundreds of different analytical queries.
- It requires no input from the end user. The execution engine automatically detects the queries that can be accelerated and rewrites them to use the precalculated aggregates.

The implementation of this technique is quite complex, and not many data virtualization providers offer this capability, but as seen in the chart below, its benefits are significant.



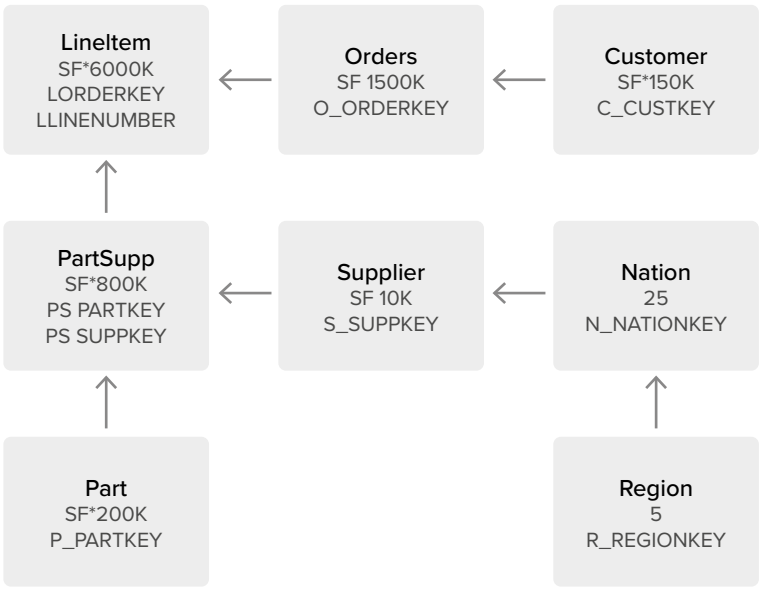
Effects of aggregate-aware query acceleration using the Denodo Platform. All queries leverage a single acceleration structure created using AI-based recommendations. Execution times are in seconds.

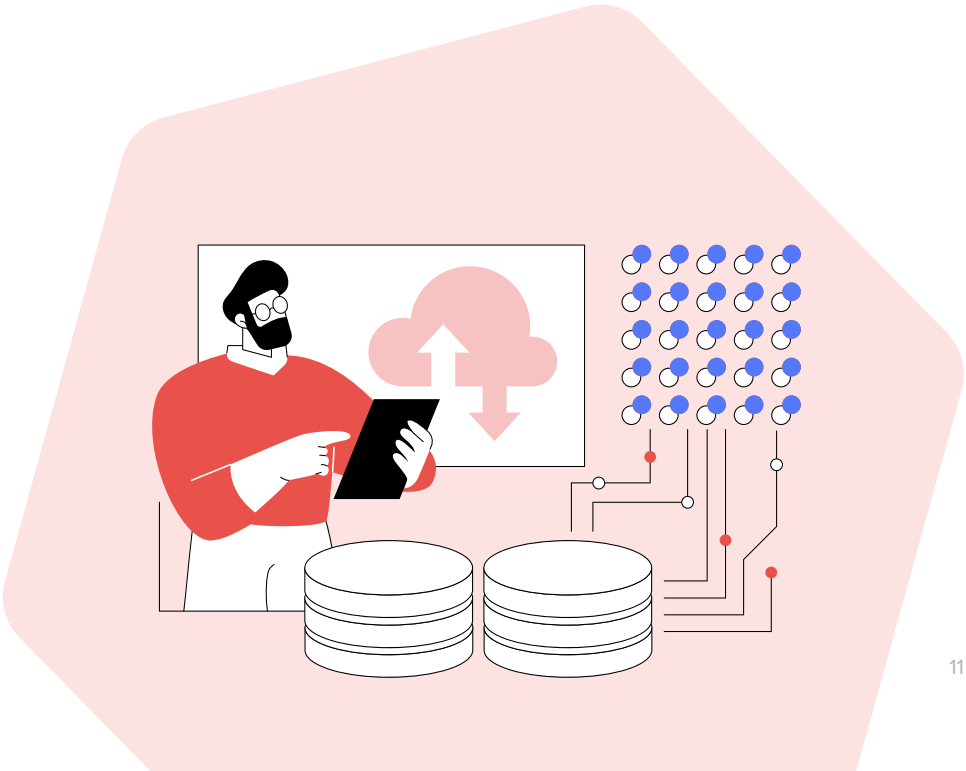
Finally, for this technique to work effectively, it is critical to choose skillfully what data should be pre-calculated and pre-aggregated, a decision that is not trivial. In the past, this required the intervention of a seasoned DBA who would analyze the reports and dashboards that required performance improvements to detect common patterns and transform them into aggregates. Fortunately, this is an area in which AI can help, and vendors like Denodo offer AI-assisted wizards that analyze usage to produce customized recommendations. A more in-depth explanation of this functionality can be found in this article, while the AI-based recommendation engine is described in this one.

Benchmarks

To test these different architectures, we have designed a scenario based on TPC-H v3.0.1, a benchmark from the Transaction Processing Council (TPC). We chose the version with a scale factor of 100, in which the largest table (Lineltem) has 600 million rows. The use of TPC-H is ubiquitous among data management vendors and represents real-life analytic scenarios reasonably well.

TABLE	ROWS (SF 100)
Customer	15,000,000
Orders	150,000,000
Lineltem	600,037,902
PartSuppr	80,000,000
Part	20,000,000
Supplier	1,000,000
Nation	25
Region	5





BENCHMARK QUERIES

To keep the run time and costs of running the benchmarks within reason, we've focused on the first 10 queries (of a total of 22) of TPC-H.

Query #	Business Question
1	This query reports the amount of business that was billed, shipped, and returned.
2	This query finds which supplier should be selected to place an order for a given part in a given region.
3	This query retrieves the 10 unshipped orders with the highest value.
4	This query determines how well the order priority system is working and gives an assessment of customer satisfaction.
5	This query lists the revenue volume done through local suppliers.
6	This query quantifies the amount of revenue increase that would have resulted from eliminating certain company- wide discounts in a given percentage range in a given year. Asking this type of "what if'' query can be used to look for ways to increase revenues.
7	This query determines the value of goods shipped between certain nations to help in the re-negotiation of shipping contracts.
8	This query determines how the market share of a given nation within a given region has changed over two years for a given part type.
9	This query determines how much profit is made on a given line of parts, broken out by supplier nation and year.
10	The query identifies customers who might be having problems with the parts that are shipped to them.

The corresponding SQL for those queries is available in the technical specification of TPC-H available online in the TCP website.

The same TPC-H datasets have been made available in different sources and formats, detailed in each scenario below, in order to represent a distributed data landscape: AWS Redshift, PostgreSQL, and Parquet files in AWS S3.

ENVIRONMENT SPECIFICATIONS

All the different data sources tested in this benchmark are deployed in the same region in AWS with the following configuration:

- Redshift: dc2.8xlarge - 4 nodes
- PostgreSQL: m5.2xlarge
- Object Storage data (S3) in Parquet format

For the data lake engine, we used the latest version (available at the time of writing this paper, February 2024) of one of the leading open-source vendors in the data lake space. The engine was also deployed in the same region and VPC in AWS with the following specs:

- R5.4xlarge nodes
  - 16 cores per node
  - 128GB RAM per node
- 20 Worker nodes
- Max 140GB heap per worker
- Metadata in Hive Metastore

For the Denodo virtualization server, we used the following configuration:

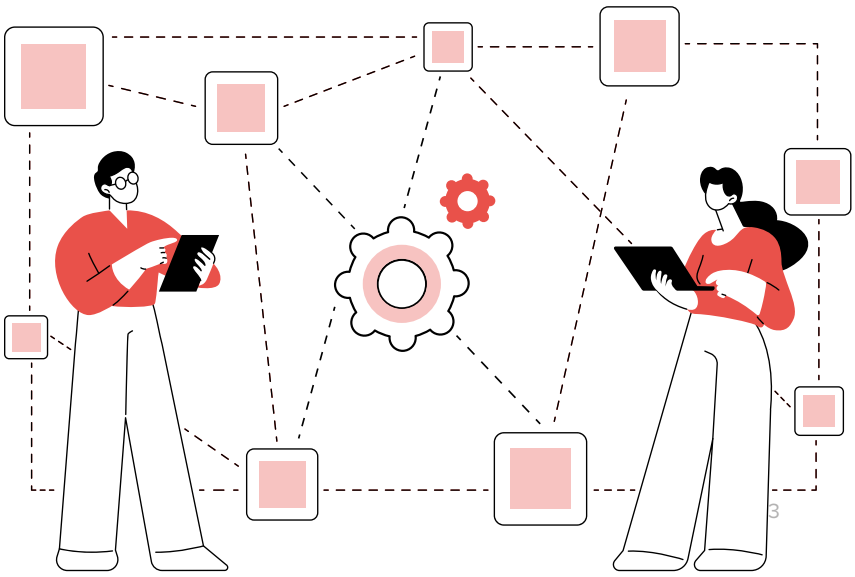
- M4.2xlarge server
  - 8 cores
  - 32 GB RAM
- 1 virtualization server
- Max 8 GB heap

No caching mechanisms were used in these tests for any of the data sources involved. No caching nor query acceleration mechanisms were used in the virtual layer either.

BENCHMARK SCENARIOS

To represent a distributed scenario, we have placed TPC-H tables in multiple systems. This opens the door for multi-source variations of this benchmark that we have detailed in each scenario.

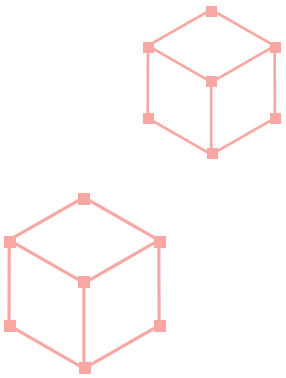
To avoid outliers or the influence of external factors, each query was run three times, and the numbers used in these tables are the average of those execution times.



Scenario #1: Access to an External Source

In this scenario, we want to simulate a query that requires data that is fully stored in an external system. Let's say, for example, that you need to run a query with data that comes from your data warehouse. For this scenario, we used AWS Redshift as the data source.

Query #	Denodo	Data Lake Vendor
1	1123.67	7477205
2	849.33	140670
3	1272.00	670518
4	9205.33	3080784
5	430.33	1382472
6	397.33	24604.5
7	5897.67	1191421
8	1361.33	1526313
9	3417.33	2529649
10	2570.67	484988
TOTAL	**26524.99**	**18508624.5**



Execution times in milliseconds.

Both engines finished all tests successfully. However, it's easy to see that the specialized architecture of the Denodo Platform can leverage access to an external EDW like Redshift orders of magnitude faster than the data lake engine. As explained in the section above, the data lake strategy of using a worker for each table requires the movement of large data volumes over the network, so even with 20 nodes processing the query in parallel, the execution times are significantly higher than the proper push-down of the queries that the Denodo Platform performs.

As a summary:

DENODO

TOTAL execution time

26.52 seconds

DATA LAKE VENDOR

TOTAL execution time

5 Hours 8 Minutes 28 Seconds

As an additional note, some data lake vendors offer enterprise connectors in their commercial offerings with better push-down logic than their open-source version used in these tests. Although this improves their performance, it is still orders of magnitude slower than Denodo. In the section "Comparing Query Execution in Data Virtualization Architectures" you can find a detailed explanation of the reasons behind these differences.

Scenario #2: Federating Two External Sources

For the second scenario, we separated the datasets into two sources, to simulate a federated scenario. The large tables remain in AWS Redshift, but some tables have been moved to PostgreSQL. The distribution is as follows:

- AWS Redshift: Supplier, PartSuppr, LineItem, Orders
- PostgreSQL: Customer, Part, Nation, Region

Note: queries #1, #4 and #6 do not involve multi-source federation, and therefore are not part of this test, since they would be identical to the data already shown in scenario #1.

Query #	Denodo	Data Lake Vendor
2	6756	7747522
3	17927.25	140670
5	4005.25	670518
7	20549.25	3328448
8	19205.5	1382472
9	20478	29161
10	110142.25	1327585
TOTAL	**199063.5**	**8835578**

Execution times in milliseconds.

Again, the versatility of the Denodo Platform overcomes the data lake approach. We can see that the execution times in the data lake are very similar to the single-source scenario #1, since the execution plan applied by the lake engine is very similar, based on mapping workers to data source tables. Interestingly, a few of the queries are faster in this federated test. Given that the execution plan in the data lake engine is almost identical, this can be explained by the reduced workload at the sources since it is now split in two different systems.

As a summary:

DENODO

TOTAL execution time
3 Minutes 19 Seconds

DATA LAKE VENDOR

TOTAL execution time
2 Hours 27 Minutes 15 Seconds

Scenario #3: Federating the Data Lake and a Small External Source

Since data lake engines are highly focused on scenarios in which the data lake is a central piece of the ecosystem, it seemed fair to also test scenarios in which part of the datasets are in the data lake. This and the following test represent that scenario.

For access to the data lake with the Denodo Platform, we have used the embedded MPP engine, based on Presto, included in the Denodo Platform. The specs of the cluster and workers are identical to those of the other data lake vendor. In this particular case, we are placing the large fact tables in the data lake, whereas the smaller dimensions are in PostgreSQL. The distribution of tables is as follows:

- Data Lake: Supplier, PartSuppr, LineItem, Orders
- PostgreSQL: Customer, Part, Nation, Region

Query #	Denodo	Data Lake Vendor
2	8122	5473.75
3	10591.75	7070.25
5	6929.25	23262.5
7	10251	8024
8	15711	12532.5
9	20478	7951.75
10	77191	69454.25
TOTAL	**149274**	**133769**

Execution times in milliseconds.

Both vendors successfully completed all the tests. This scenario is the sweet spot of the data lake, as it can parallelize most of the processing given the parallel access to the fact tables. The Denodo Platform's executions showcase a hybrid execution plan, as it can leverage its MPP for processing and federation when needed, providing very similar results.

As a summary:

DENODO

TOTAL execution time

2 Minutes 29 Seconds

DATA LAKE VENDOR

TOTAL execution time

2 Minutes 13 Seconds

Scenario #4: Federating the Data Lake and a Large External Source

As a variation on the scenario above, we've modified the distribution of the tables, so that the large tables are outside the data lake. This represents, for instance, scenarios in which large tables in an EDW need to be crossed with the data lake.

The distribution of tables is as follows:

- Data Lake: Customer, Part, Nation, Region
- AWS Redshift: Supplier, PartSuppr, Lineltem, Orders

Query #	Denodo	Data Lake Vendor
2	244304	268837
3	15746	1275039
5	41227	2318577
7	12960.5	1297888
*8	16325.25	2774511
*9	31007.5	6537800
*10	14736.25	559807
TOTAL	**376306.5**	**15032459**

Execution times in milliseconds.

We can see, once again, that the flexible options that the Denodo Platform provides adapt to this scenario better. For example, following data gravity, the Denodo Platform can execute on-the-fly data movement from the data lake to the EDW and other advanced techniques that the data lake is not capable of.

As a summary:

DENODO

TOTAL execution time
6 Minutes 16 Seconds

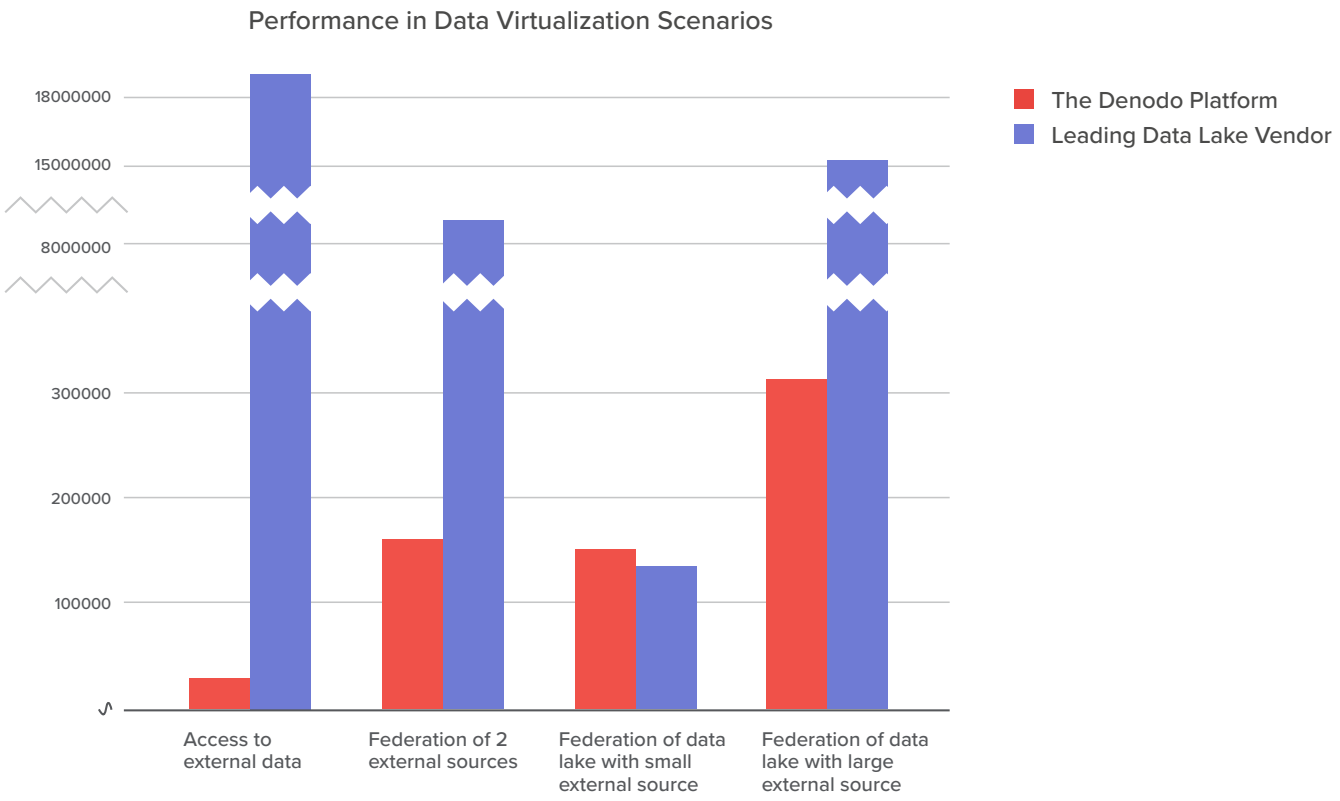
DATA LAKE VENDOR

TOTAL execution time
4 Hours 10 Minutes 32 Seconds

Conclusion

Data lake engines are designed to provide ample processing muscle for analytical queries with large data volumes at scale. However, as shown in these tests, their processing style is not a good fit for a data delivery layer in a distributed landscape. Their multi-source query capabilities work well when a small portion of the data is outside of the lake, as shown in scenario #3, but do not provide a viable solution for most other scenarios.

Scenario	The Denodo Platform	Leading Data Lake Vendor
Access to external data	26.52 seconds	5 Hours 8 Minutes 28 Seconds
Federation of 2 external sources	3 Minutes 19 Seconds	2 Hours 27 Minutes 15 Seconds
Federation of data lake with small external source	2 Minutes 29 Seconds	2 Minutes 13 Seconds
Federation of data lake with large external source	6 Minutes 16 Seconds	4 Hours 10 Minutes 32 Seconds



Modern, specialized data virtualization solutions like the Denodo Platform, which works as a sort of "meta-coordinator" across multiple sources, provide a **robust, performant, and better designed solution to serve as a global access point for the entire corporate data landscape**. Its MPP engine, based on Presto, brings that performance also to data-lake-centric scenarios. As a final note, any choice of an architecture that enables a data fabric must also consider other capabilities in areas like data governance, modeling, security, and self-service. A more thorough analysis of those areas can be found in the paper entitled Logical Data Fabric - A whitepaper.

denodo

Visit www.denodo.com | Email info@denodo.com | Discover community.denodo.com