DATA VIRTUALITY MASTERCLASS

Topic: Performance Optimization

@ Data Virtuality GmbH 2021

What to expect from this session?

This track will tell you how to Optimize the Performance of Data Virtuality:

- Query Plans
- Materialization and in-Memory Caching
- Performance Optimization of Queries / Views / Joins
- Data Source Settings
- DV Server Instance Specifications (RAM, vCPUs, IOPS)

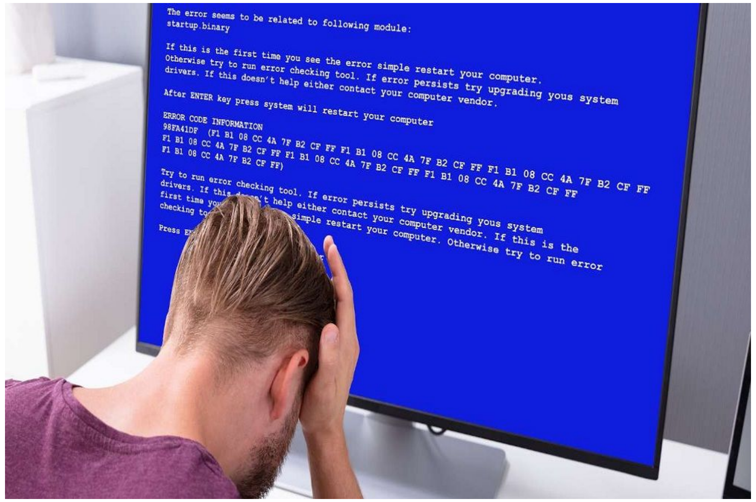
@ Data Virtuality GmbH 2020

Soft Limits

Soft Limits

Data Virtuality comes with pre-configured soft limits

- 20 x Concurrently Active Queries
- 15 x Concurrently Running Jobs
- These limits can be reconfigured.
- If the limits are **breached**, the system will **write** the corresponding messages in the **server.log**.
    - **Degraded/Unresponsive Behaviour** is noticed when queries and jobs are **queued** or **buffer overflows**.



Reading and Setting Soft Limits

--read concurrent job count value exec "SYSADMIN.executeCli"("script" => '/subsystem=teiid/:read-attribute(name=quartz)');;	**--increase concurrent job count to 30** exec "SYSADMIN.executeCli"("script" => '/subsystem=teiid/:write-attribute(name=quartz,value=[("org.quartz.threadPool.threadCount" => "30"),("org.quartz.jobStore.class" => "com.datavirtuality.dv.core.scheduler.DVJobStore"),("com.datavirtuality.quartz.DVJobStore.misfireThreshold" => "3600000")])');;
--read the number of concurrent queries and threads EXEC "SYSADMIN.executeCli"("script" => '/subsystem=teiid:read-attribute(name=max-active-plans)');; EXEC "SYSADMIN.executeCli"("script" => '/subsystem=teiid:read-attribute(name=max-threads)');;	**--increase concurrent queries to 40** EXEC "SYSADMIN.executeCli"("script" => '/subsystem=teiid:write-attribute(name=max-active-plans,value="40")');; EXEC "SYSADMIN.executeCli"("script" => '/subsystem=teiid:write-attribute(name=max-threads,value="128")');;

Threading Options

- Need to edit the following XML file
  - %dvDir%/standalone/configuration/dvserver-standalone.xml
- Please note that when increasing the value of *max-active-plans*, it is also necessary to increase the value of *max-threads* value accordingly.
- The calculation of (*max-threads* / *max_active_plans*) * 2 will indicate the number of max-threads that are available to be used for processing each user query.

Official Documentation

[Threading Options (datavirtuality.com)]

Considerations for Statistics Gathering

Considerations for Statistics Gathering

Data Virtuality's optimization subsystem can work with cost based optimization.

As a best practice, we recommend gathering statistics when tables with more than **100 million rows** are involved, both for the 100 million rows tables as well as the small tables they are joined to. This way, **dependent joins** will be created when **joining** very **large tables** with very **small tables**.

For all other cases, it is sufficient to use **Merge Joins** so that the additional **computational effort of gathering statistics can be saved**.

More on the different types of joins later in the presentation.

Understanding Buffers

Understanding Buffers

What are Buffers?

1. Data Virtuality uses buffers for storing data during internal operations.
   a. The buffers can be used for storing two types of data:
      i. **Temporary Data**: needed for operations during query execution (SORT, ORDER BY, etc)
      ii. **Large Data:** coming from data sources, for example CLOBs, BLOBs, XML documents etc.
2. Buffers can reside in main memory (outside of Java Heap space) **or** on disk.
   a. Data Movement is transparent for the user and is decided based on various factors such as **size of data**, **Queries** running in parallel, Available **RAM**.
   b. The size of the **disk buffer** is per **default** set to **50GB** but is user configurable.
3. EXEC SYSADMIN.executeCli (script => '/subsystem=teiid/:write-attribute(name=buffer-service-max-buffer-space,value=102400)');;
4. **Example**: The above command sets the Disk Buffer size to 100GB, restart services to apply setting.

Understanding Buffers

The buffer usage overall and per query can be monitored using the performance monitoring tool.

The key to understanding the performance impact of buffer configuration is the following:

1. In most cases, **using buffers** by queries is a **consequence of not being able to push down** the operations to the source.
    a. **Optimizing Pushdowns** may result in a **Performance Improvement**.
2. In most cases, memory buffers operate much quicker than disk buffers. (**SSD vs HDD**) (**IOPS**)
3. **More RAM** is always an option to increase the performance.

More on Pushdowns later on in the Presentation

XML Parsing Optimization

Memory usage and Streaming XML processing

1) **Unlike** other data types, **XML** processing is always done in the **Java Heap** and not in **buffers**.
2) This is the **limitation** of the Java **XML parsing library** used by Data Virtuality.
3) The XML library inside Data Virtuality is able to automatically switch between two parsing modes to provide better performance.

    a) **Streaming Mode:** Low memory footprint for simple XML parsing.
    b) **Full Mode:** Entire XML documents are loaded into Java Heap space in memory.
        i) To avoid Poor Performance or Out-Of-Memory crash, please provide enough Java Heap memory to Data Virtuality server.

For more detailed discussion on Eligible (i.e. the ones which support streaming) and Ineligible XML operations, please see the following link.

Using the Monitoring Tool

Using the Monitoring Tool

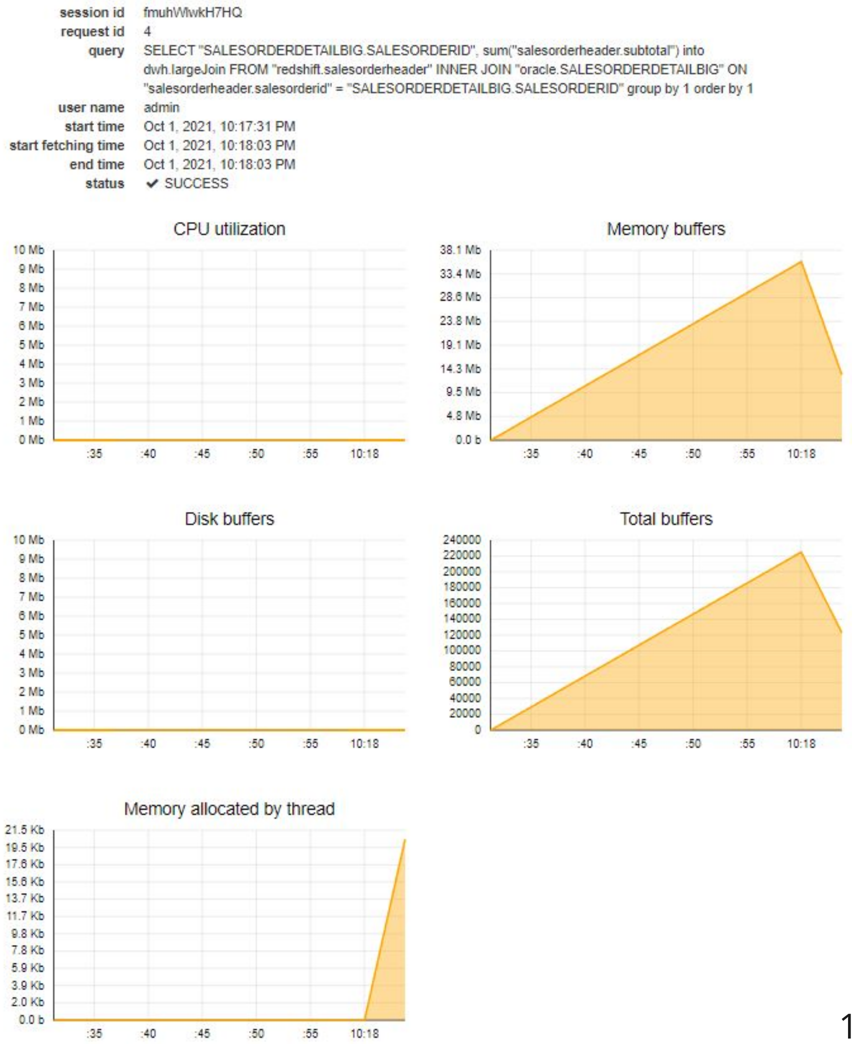
Performance of single queries can be **analyzed** using the **web based monitoring tool**. If a query is selected, the following information about the consumed resources of DV Server is shown:

- **CPU Utilization** - percentage of CPU utilized to execute the query.
- **Memory buffers** - amount of memory that is stored in buffers.
- **Disk buffers** - the amount of memory that can not be stored in memory and needs to be spilled to disk.
- **Total buffers** - the total number of buffers
- **Memory allocated by thread** - in contrast to memory buffers that store query results, memory allocated by thread depicts the RAM usage of the process itself

Using the Monitoring Tool / Example

In the below example, a query was joining two disparate tables and wrote the results to a third database system.

1) CPU usage is very low, this is due to the fact that the query could be completely pushed down, no calculation needed to be performed in DV Server's query engine.
2) Memory buffer usage goes up, as results are stored in memory before writing them to the target database.
3) Disk buffers are not used, as memory buffers suffice in storing intermediate results.
4) The total number of buffers runs in parallel to the memory buffers, as expected.
5) Almost no memory is allocated for executing the query, as all queries can be pushed down.



Working with Query Plans

Accessing Query Plans

There are three ways to see the query plans in the Data Virtuality Studio:

1. In the SQL Editor of Data Virtuality Studio, the query plan can be accessed by clicking on the “Show Query Plan” icon: 

The query plan will be displayed for the query that currently contains the cursor.

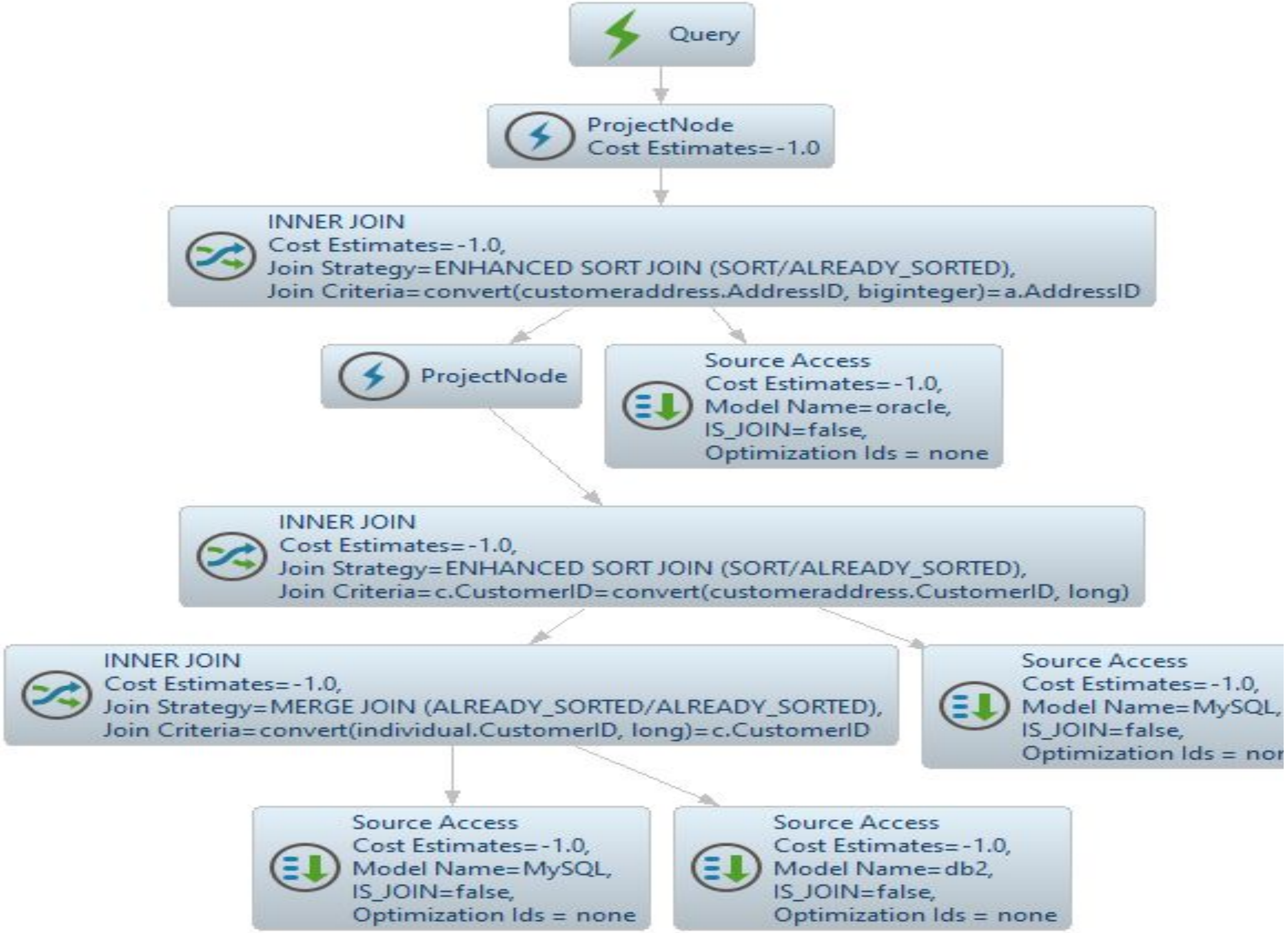
2. It is also possible to investigate the query plan for the selected SQL if its text is marked before clicking “Show Query Plan”.
3. Query plans are also visible in the “Show Query Plan” Tab of DV Studio by selecting a query and clicking “Show Query Plan”. Please note that the query plans are archived along with the queries., so that even if you change a view, it will not affect the plans of former queries.

Reading Query Plans

A query plan is read **from bottom to top**.

In most cases, the source access will be happening first, along the path processing on the Data Virtuality side and enriching data, with a final query node on top of the plan, indicating the result is returned to the user.

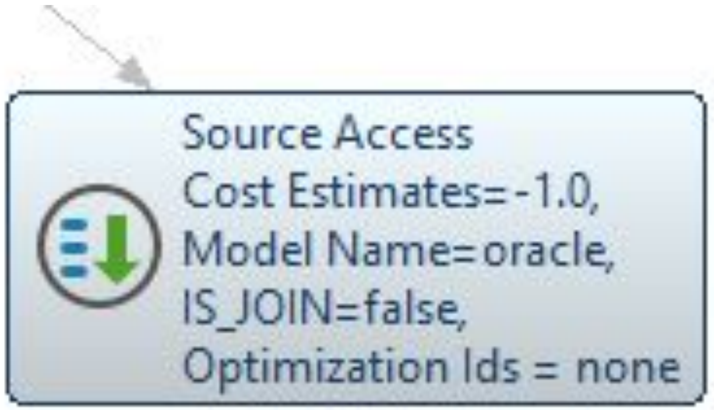
Reading Query Plans: Example



Pushdown

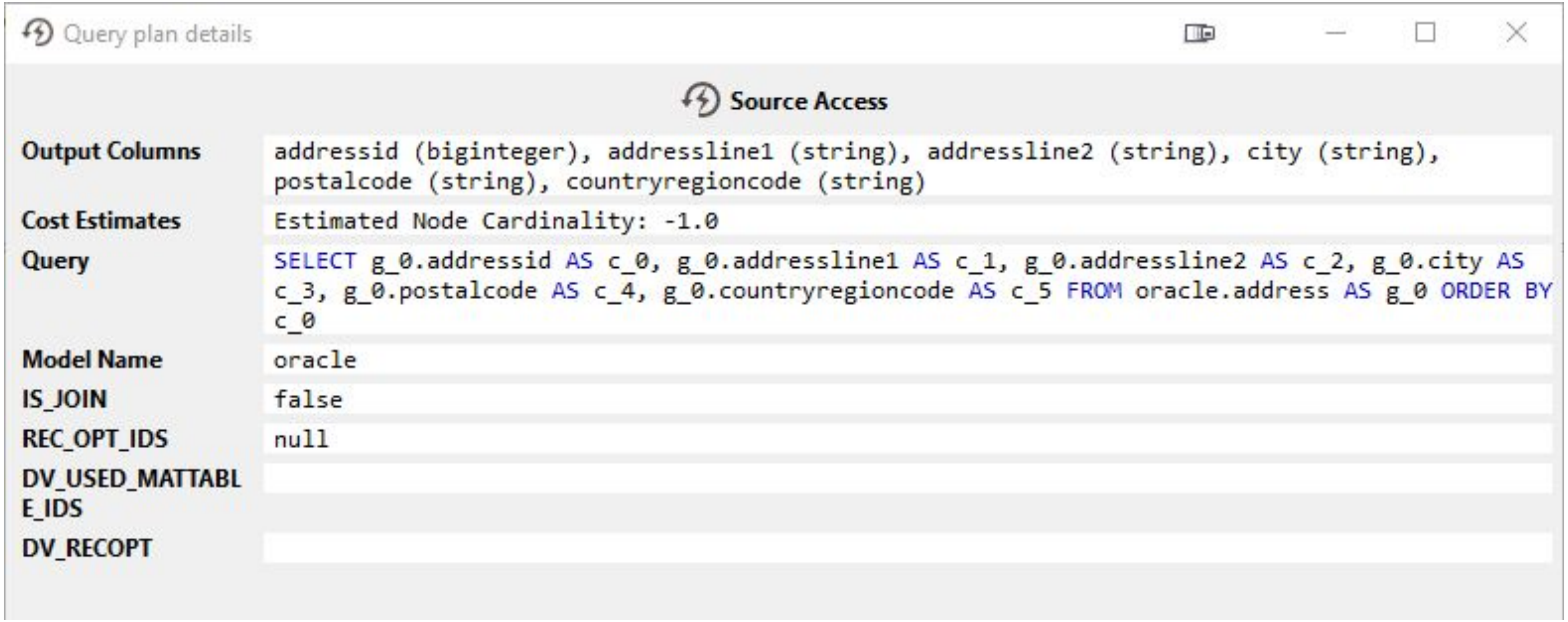
What is Pushdown?

- It’s an optimization technique that moves the processing of data closer to the data source.
- The DV Server will spend Less time on transfering data and processing it.
- Pushdown is available for functions, criteria, joins and aggregations.
- The set of available pushdowns differs across data sources, being limited by the driver and translator capabilities.
- To identify a pushdown in an query plan, it is generally indicated by a “Source Access” node:



Pushdown Cont'd

When double clicking the previous node, additional information will be displayed, including the query that DV was sending to the source (in DV SQL, not the actually executed query on the data source side)



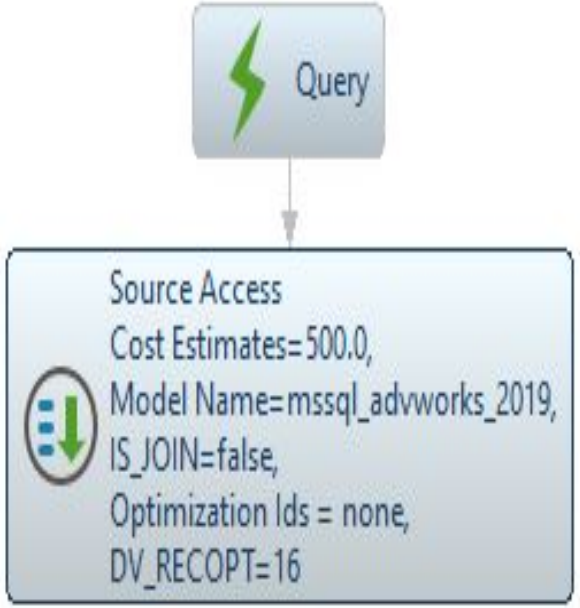
Aggregation Pushdown

Aggregation Pushdown

In this example, the query is performing a “GROUP BY” and a “SUM”.

```
SELECT
    "ProductID"
    ,sum ("UnitPrice")
FROM
"mssql_advworks_2019.AdventureWorks2019.Sales.SalesOrderDetail"
group by
    "ProductID" LIMIT 500;;
```

Since MSSQL supports both operations, the entire query is pushed down to the data source. This can be verified by inspecting the query plan.



Aggregate Pushdown Cont’d

Query plan details

Source Access

Output Columns	`ProductID (integer), expr2 (bigdecimal)`
Cost Estimates	`Estimated Node Cardinality: 500.0`
Query	`SELECT g_0.ProductID AS c_0, SUM(g_0.UnitPrice) AS c_1 FROM mssql_advworks_2019.AdventureWorks2019.Sales.SalesOrderDetail AS g_0 GROUP BY g_0.ProductID LIMIT 500`
Model Name	`mssql_advworks_2019`
IS_JOIN	`false`
REC_OPT_IDS	`null`
DV_USED_MATTABLE_IDS	
DV_RECOPT	`16`
DV_EXT0	

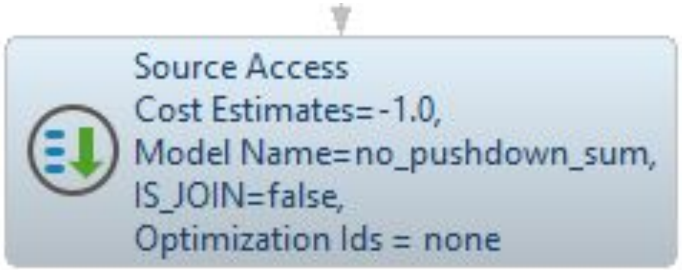
Aggregate Pushdown / Unsupported Data Source

In the next example, the query is performing a similar operation of “GROUP BY”, “SUM”, and “ORDER BY”, but ***against a different data source which does not support these operations natively***.

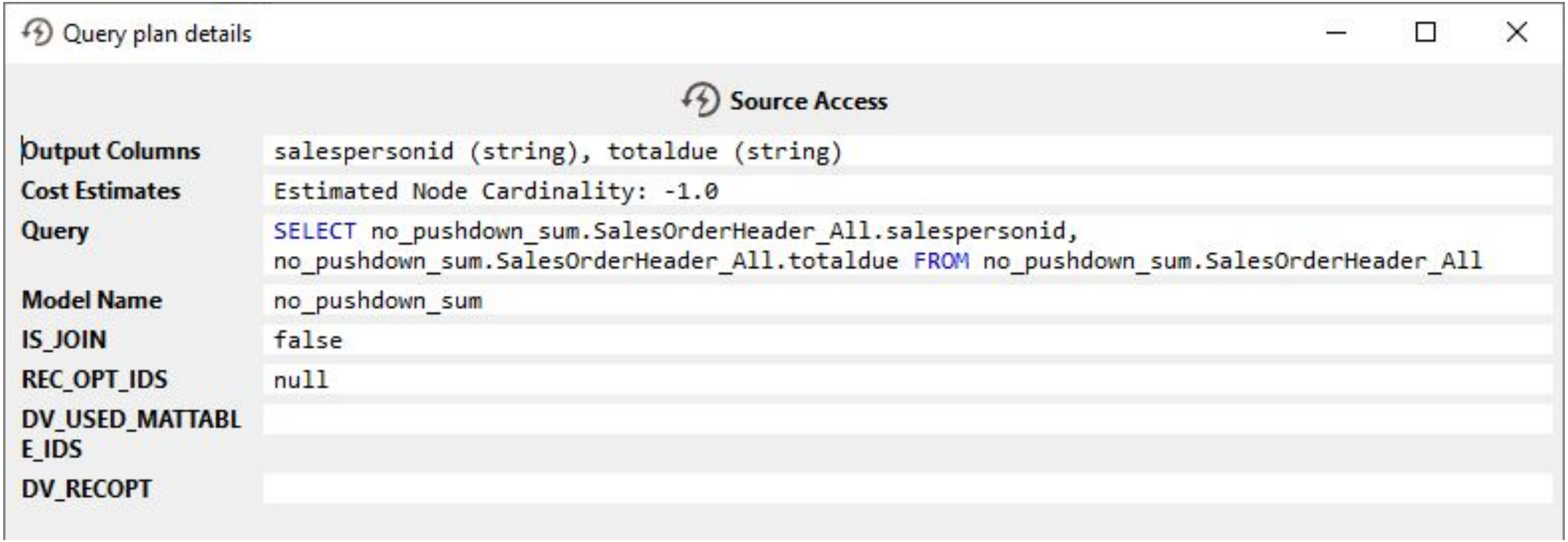
```
SELECT
    "salespersonid"
    ,sum(cast("totaldue" as float)) as "totaldue"
FROM
    "no_pushdown_sum.SalesOrderHeader_All"
group by
    "salespersonid"
order by
    "salespersonid" LIMIT 500;;
```

Aggregate Pushdown / Unsupported Data Source

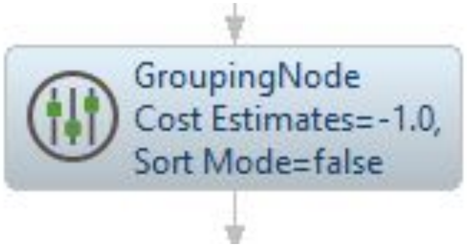
1 ->



Because the data source does not implement “GROUP BY” nor “SUM”, the first step is to read all of the data into Data Virtuality.



2 ->



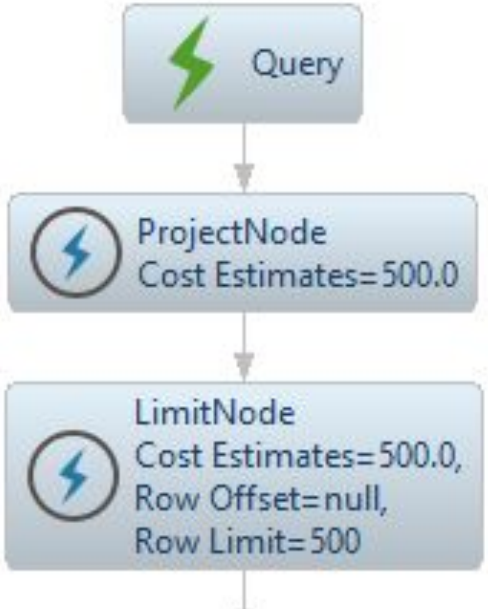
Data Virtuality performs the grouping and aggregation in memory.

Query plan details

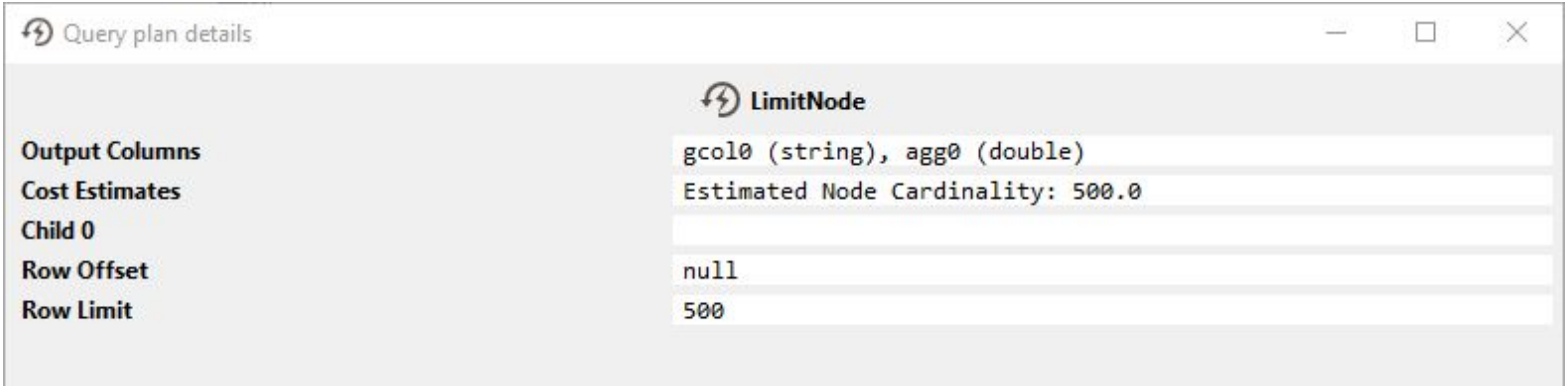
GroupingNode

Output Columns	gcol0 (string), agg0 (double)
Cost Estimates	Estimated Node Cardinality: -1.0
Child 0	
Grouping Columns	no_pushdown_sum.SalesOrderHeader_All.salespersonid
Sort Mode	false

3 ->



Data Virtuality implements the “LIMIT 500” part of the query.



Function Pushdown

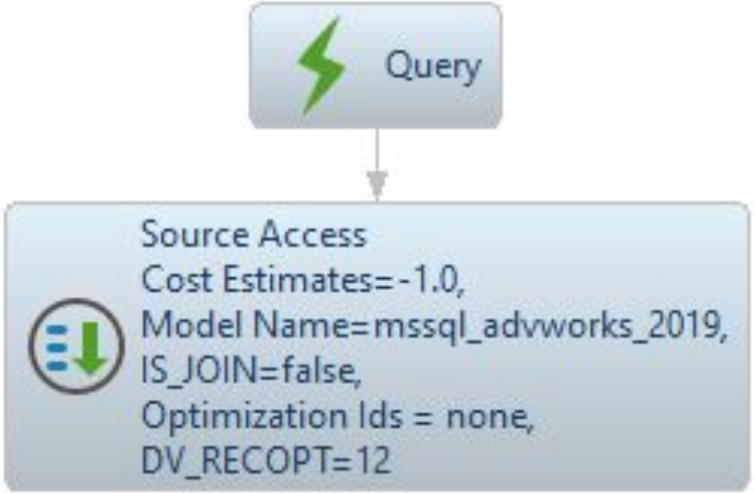
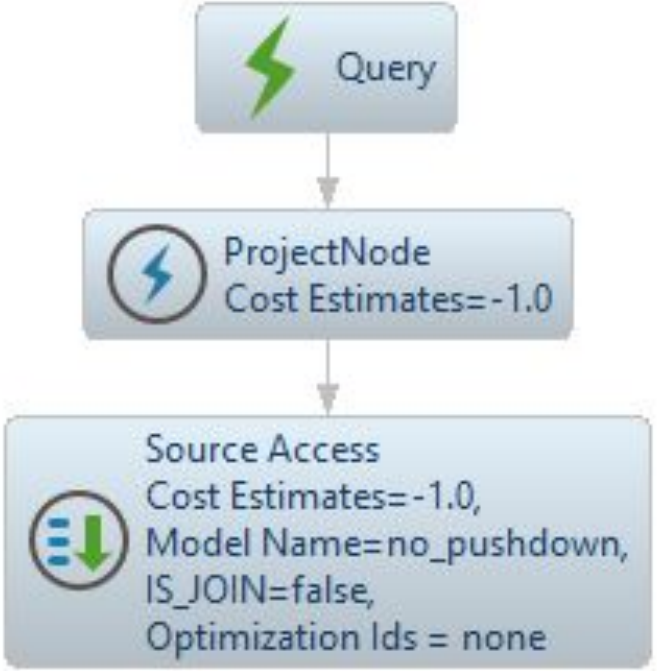
Function Pushdown

The following examples contrast the query plans when a function cannot be pushed down. In the right column, an extra step appears in the query plan. This extra step appears because the data source does not support the MD5 function. Data Virtuality compensated for the lack of MD5 functionality by implementing it internally.

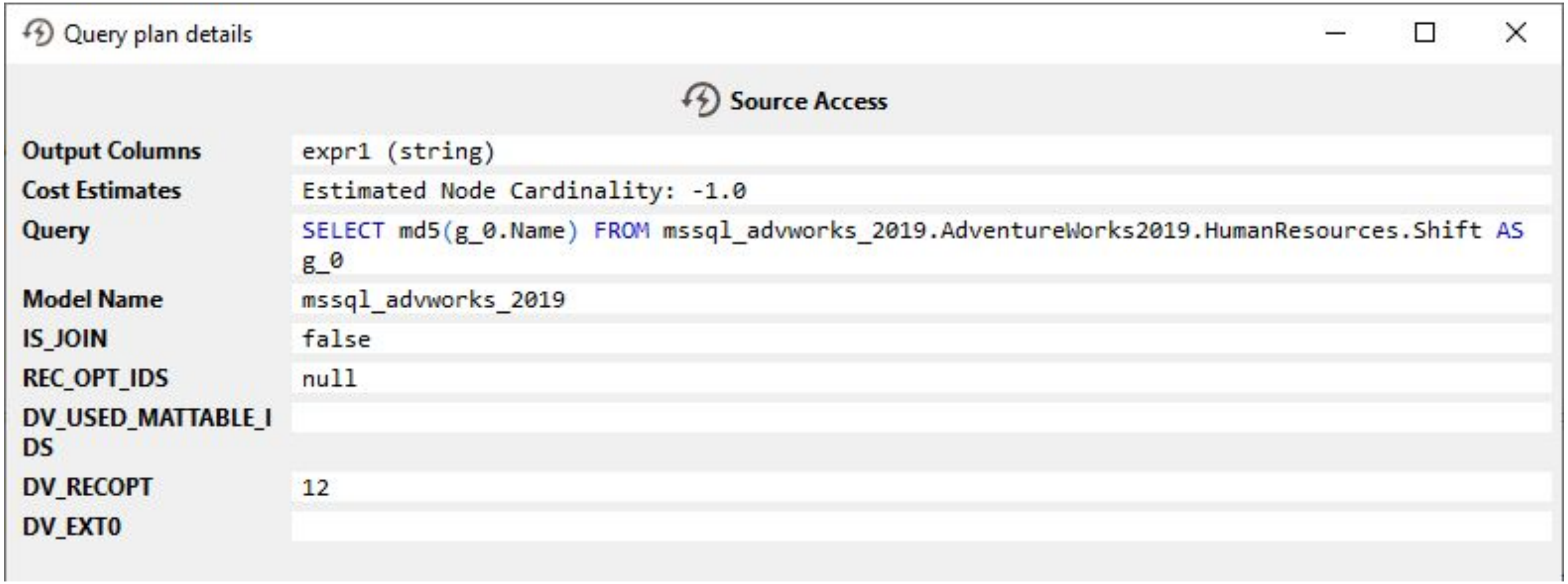
Pushdown Supported	**Pushdown NOT Supported**

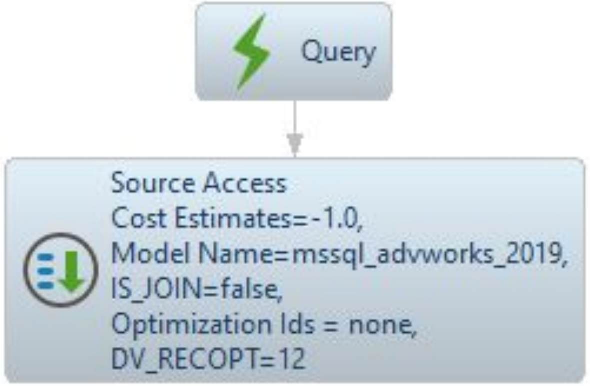
SELECT
  md5("Name")
FROM
 	
"mssql_advworks_2019.AdventureWorks2019.
HumanResources.Shift"
;;
	SELECT
	  md5("salespersonid")
	FROM
	  "no_pushdown.SalesOrderHeader_ALL"
	;;

Function Pushdown

Pushdown Supported	**Pushdown NOT Supported**
	

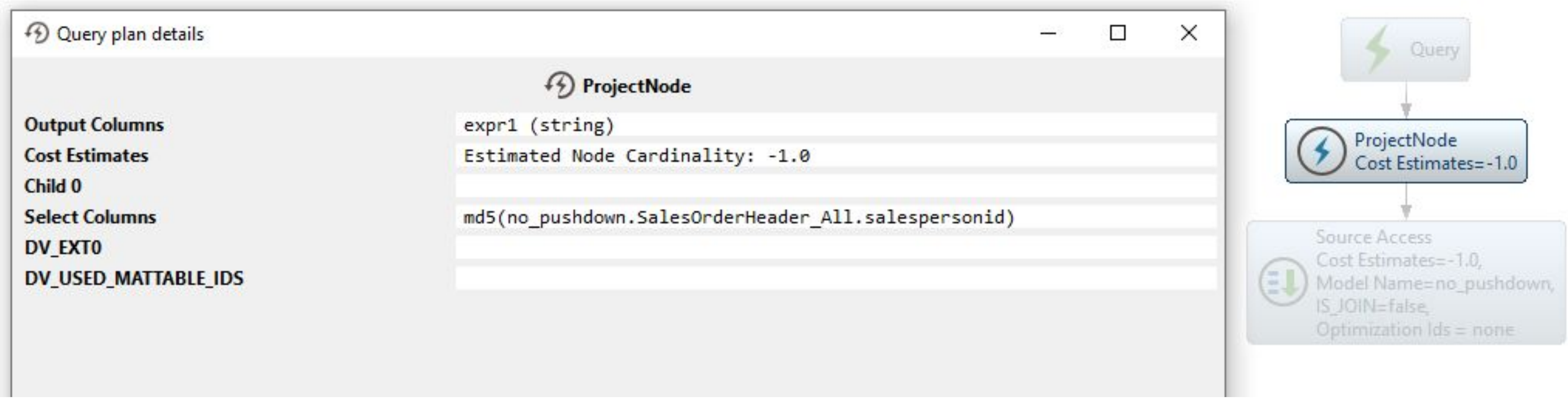
Pushdown Supported





SELECT md5(g_0.Name) FROM mssql_advworks_2019.AdventureWorks2019.HumanResources.Shift AS g_0

Pushdown NOT Supported STEP 1



md5(no_pushdown.SalesOrderHeader_All.salespersonid)

Pushdown NOT Supported STEP 2


```sql
SELECT no_pushdown.SalesOrderHeader_All.salespersonid
FROM no_pushdown.SalesOrderHeader_All
```

Criteria Pushdown

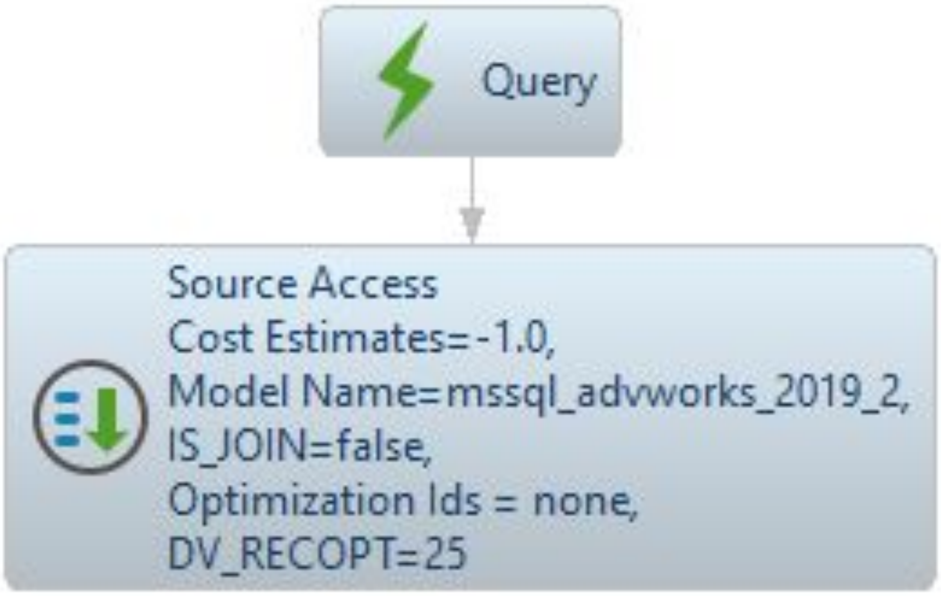
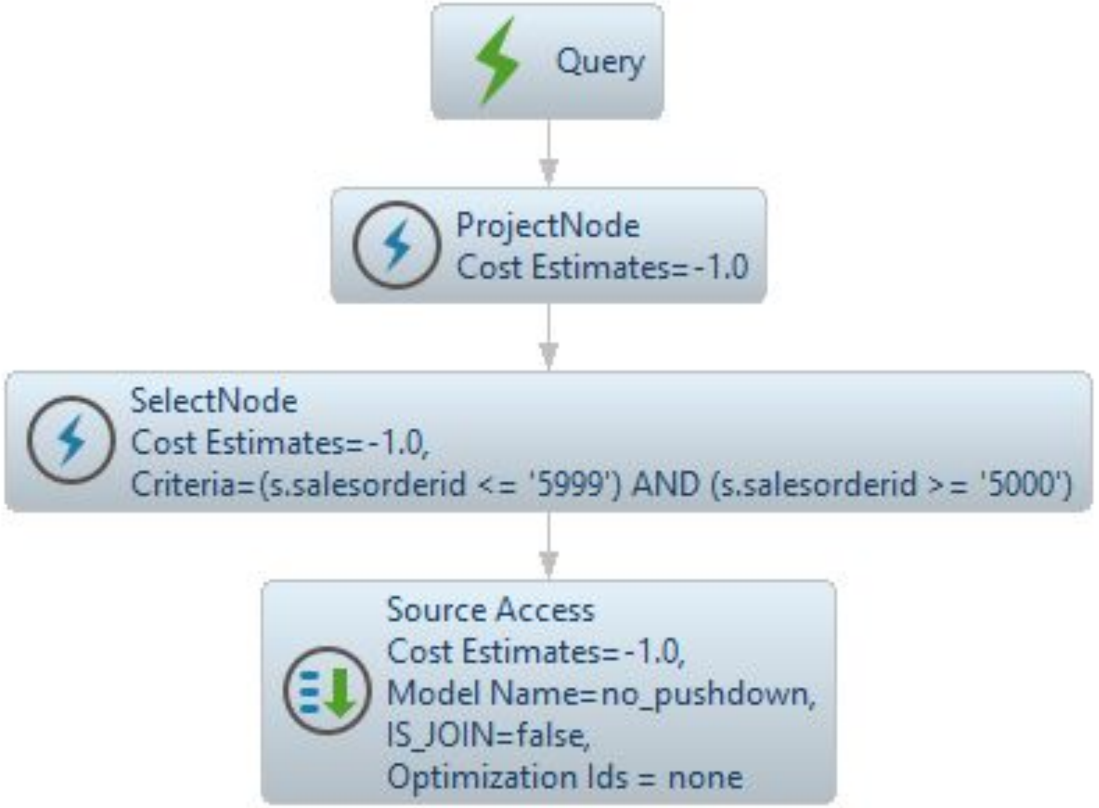
Criteria Pushdown

If the data source **supports** filtering, the filtering criteria is **sent** to the data source **reducing** the **data returned** to Data Virtuality. The simplest **example** of criteria pushdown is the inclusion of the **WHERE** clause in SQL. In cases where the data source does not support "**WHERE**" clauses. Data Virtuality will **retrieve** the data set and filter the results in memory.

Pushdown supported	Pushdown NOT supported

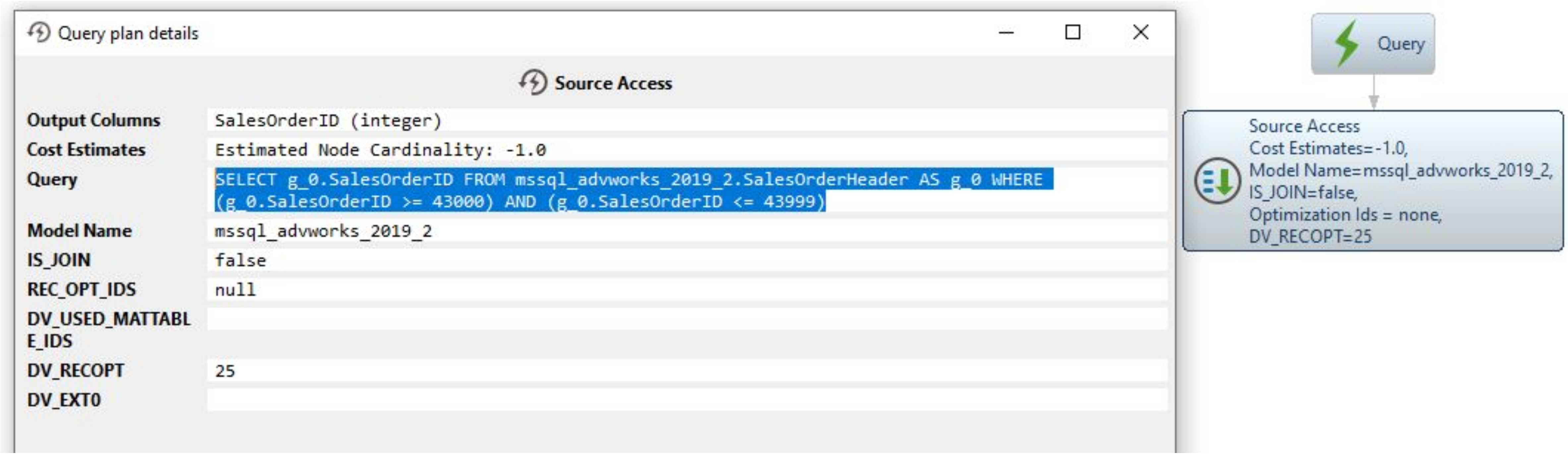
SELECT
   s.SalesOrderId
FROM
   "mssql_advworks_2019_2.SalesOrderHeader" s
where
      s.SalesOrderId between 43000 and 43999
;;
	SELECT
	   s.SalesOrderId
	FROM
	   "no_pushdown.SalesOrderHeader_ALL" s
	where
	      s.SalesOrderId between 5000 and 5999
	;;

Criteria Pushdown

Pushdown Supported	Pushdown NOT Supported
	

Criteria Pushdown

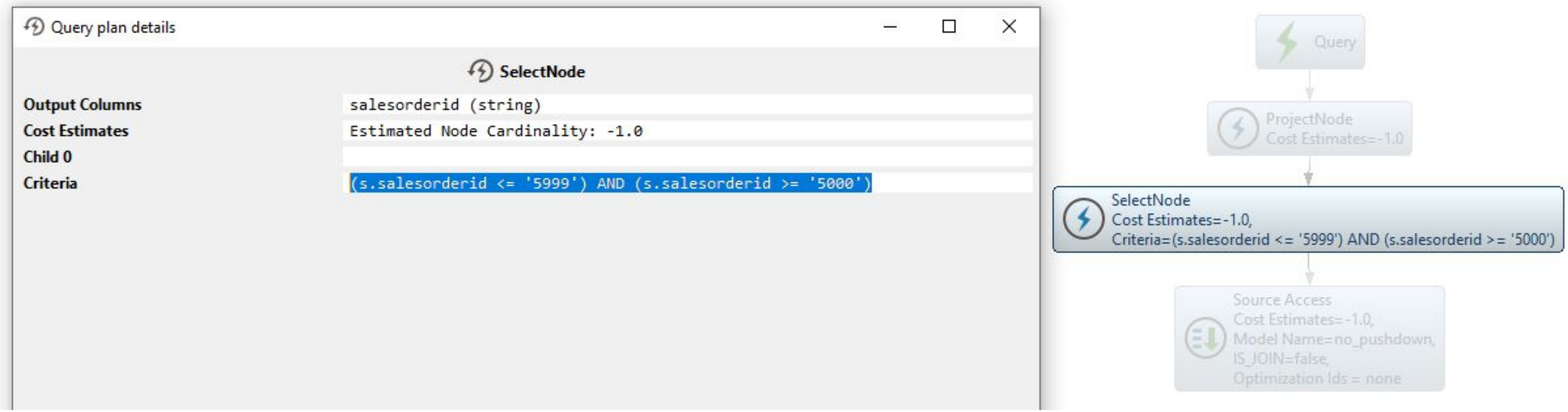
Supported Pushdown



SELECT g_0.SalesOrderID FROM mssql_advworks_2019_2.SalesOrderHeader AS g_0 WHERE (g_0.SalesOrderID >= 43000) AND (g_0.SalesOrderID <= 43999)

Criteria Pushdown

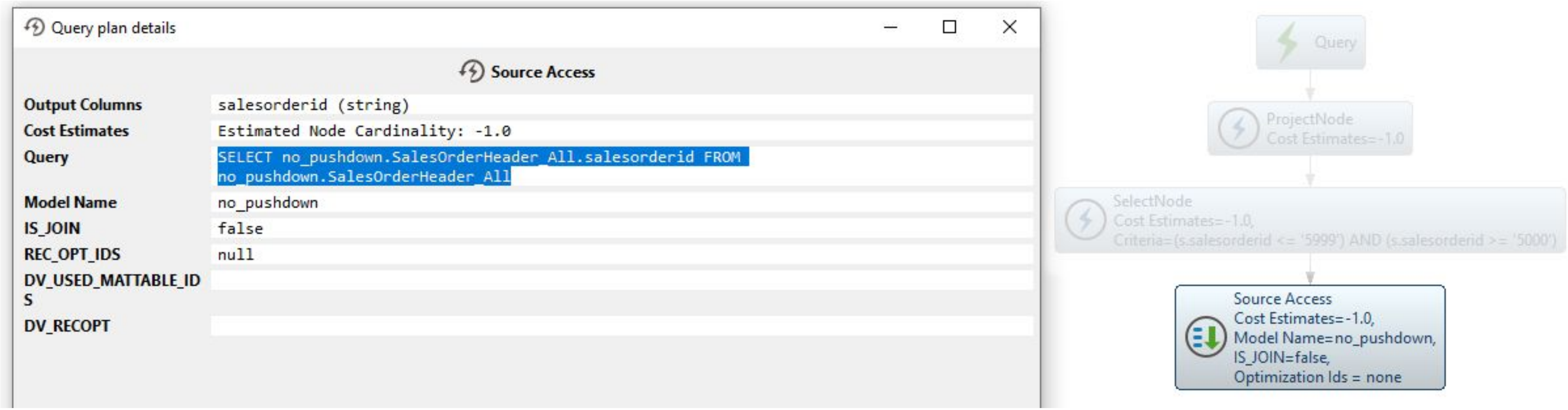
Pushdown NOT Supported STEP #1



(s.salesorderid <= '5999') AND (s.salesorderid >= '5000')

Criteria Pushdown

Pushdown NOT Supported STEP #2



SELECT no_pushdown.SalesOrderHeader_All.salesorderid
FROM no_pushdown.SalesOrderHeader_All

Join Pushdown

Join Pushdown

When possible, a SQL query using joins is pushed down to the data source and the query is evaluated by the data source. There are several criteria that must be met in order to push the joins to the data source.

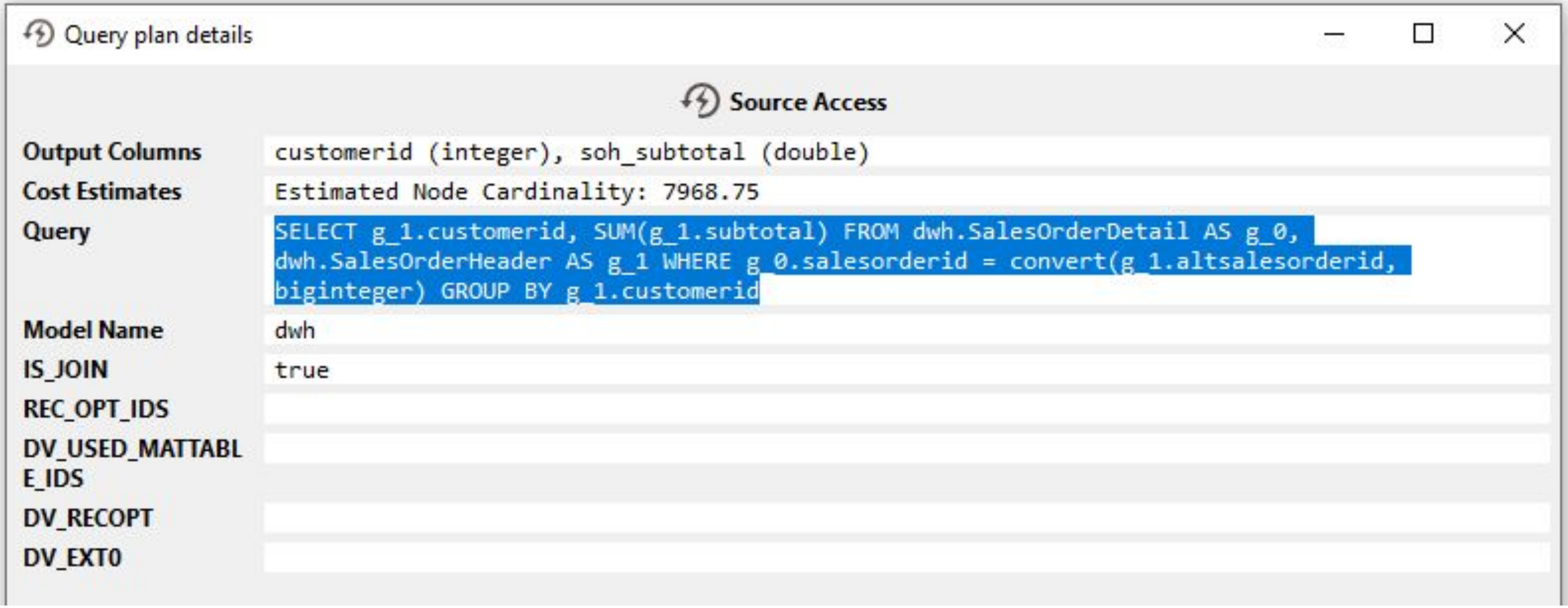
1. The tables in the join must be from the same data source.
2. The data source driver must support joins.

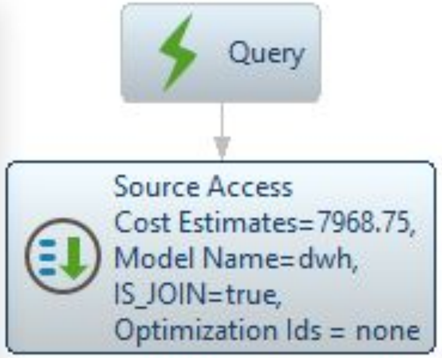
Join Pushdown Example / Same Data Source

The following example shows two joins from the same data source.

```
select
    soh.customerid,
    sum(soh.subtotal) as soh_subtotal
from
    dwh.SalesOrderDetail sod
    join dwh.SalesOrderHeader soh
        on sod.salesorderid = soh.altsalesorderid
group by
    soh.customerid
;;
```

Join Pushdown Example / Same Data Source





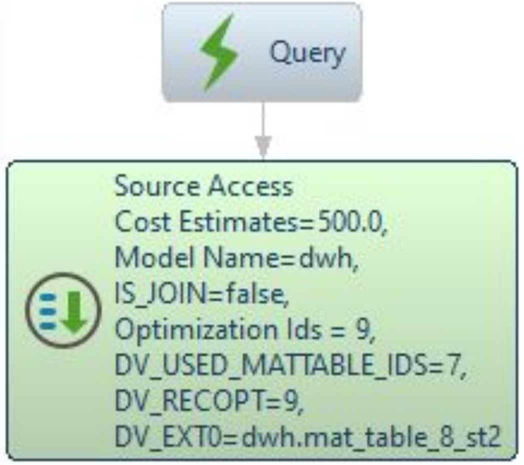

```sql
SELECT g_1.customerid, SUM(g_1.subtotal) FROM dwh.SalesOrderDetail AS g_0, dwh.SalesOrderHeader
AS g_1 WHERE g_0.salesorderid = convert(g_1.altsalesorderid, biginteger) GROUP BY g_1.customerid
```

Join Pushdown Example / Different Data Sources / DWH

Query plan details

Source Access

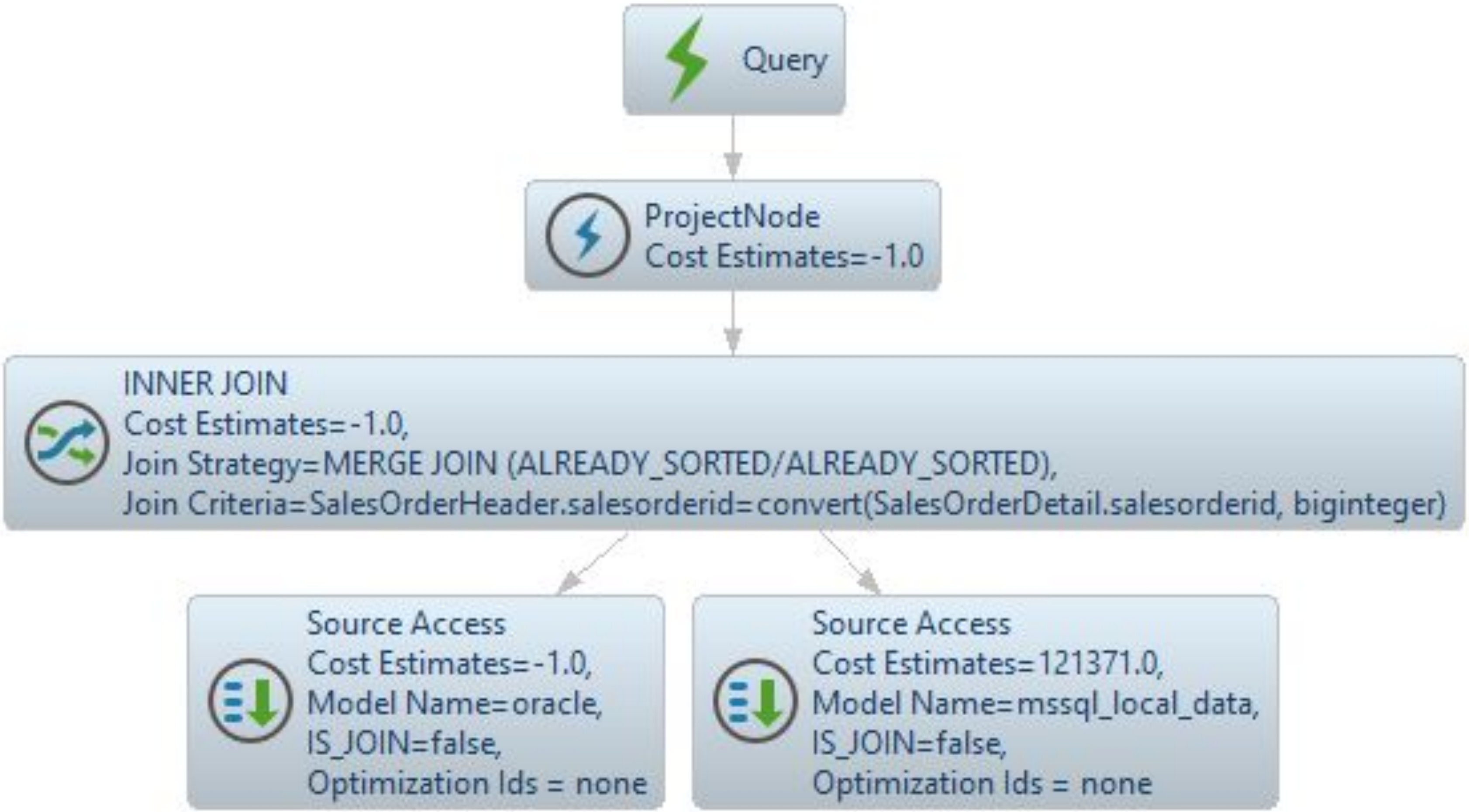
Output Columns	salesorderid (biginteger)
Cost Estimates	Estimated Node Cardinality: 500.0
Query	SELECT g_0.salesorderid AS c_0 FROM dwh.mat_table_8_st2 AS g_0 LIMIT 500
Model Name	dwh
IS_JOIN	false
REC_OPT_IDS	9
DV_USED_MATTABLE_IDS	7
DV_RECOPT	9
DV_EXT0	dwh.mat_table_8_st2
Replication date of mat_table_8_st1	2021-09-30 12:01:28.792




```
CREATE VIEW "views.SalesOrdersPlusHeaders_oracle_mssql_mat" AS
SELECT * FROM "oracle.SalesOrderHeader"
INNER JOIN "mssql_local_data.SalesOrderDetail"
ON "SalesOrderHeader.salesorderid" = "SalesOrderDetail.salesorderid";;
SELECT "SALESORDERID" FROM "views.SalesOrdersPlusHeaders_oracle_mssql_mat" LIMIT 500;;
```

To Opt-Out USE
OPTION $PREFER_DWH NEVER

Join Pushdown Example / Different Data Sources / DWH NEVER / (Federated Query)



SELECT "SALESORDERID" FROM

"views.SalesOrdersPlusHeaders

_oracle_mssql_mat"

OPTION $PREFER_DWH NEVER

;;

https://documentation.datavirtuality.com/24/reference-guide/sql-clauses/option-clause

Optimizing Data Types for Pushdown

Optimizing data types for Pushdown

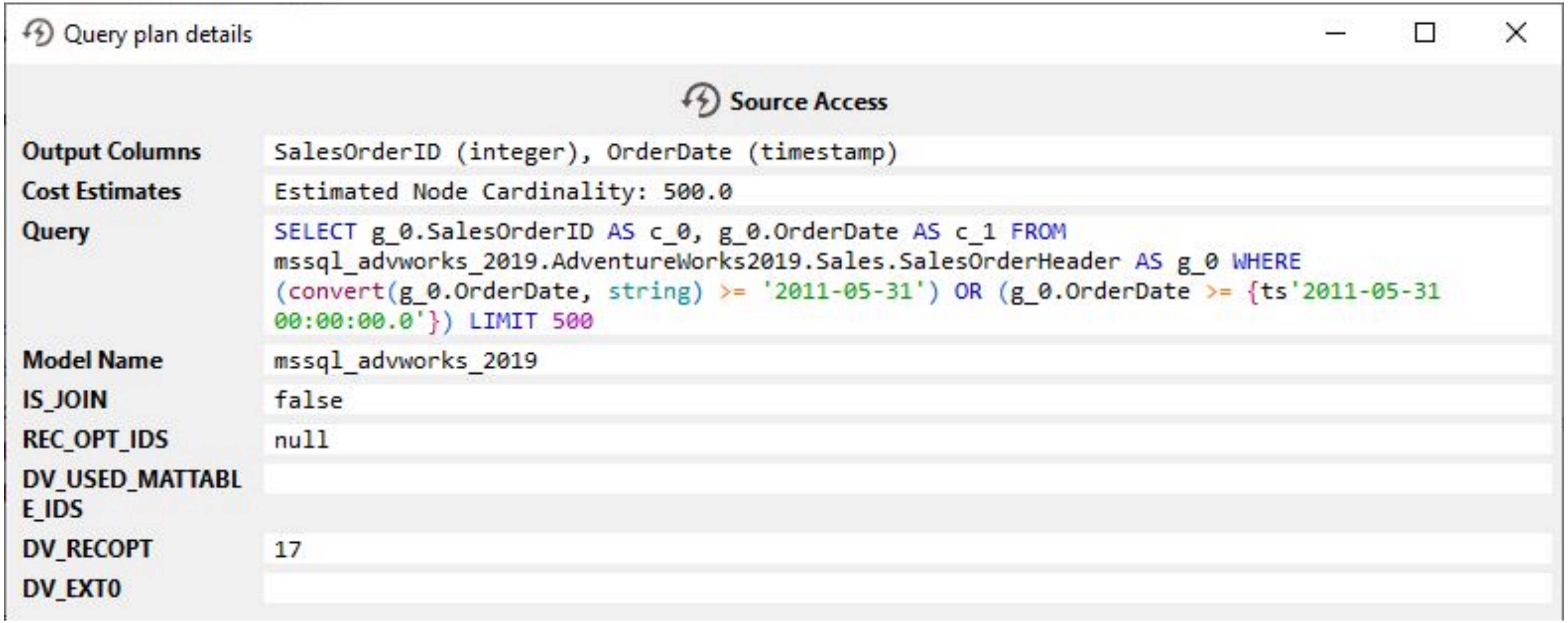
When joining or filtering on time based criteria, it is advisable to use literals instead of passing the parameters as string. This makes sure that the source system does not have to convert these parameters.

Examples:
Date: {d 'yyyy-mm-dd'}
Time: {t 'hh-mm-ss'}
Timestamp: {ts 'yyyy-mm-dd hh:mm:ss.[fff...]'}
See our [documentation on Literals]

In this example, we are going to look at a scenario where a simple date comparison is negatively impacting performance.

```
SELECT
    "SalesOrderID"
    ,"OrderDate"
FROM
    "mssql_advworks_2019.AdventureWorks2019.Sales.SalesOrderHeader"
where
    OrderDate >= '2011-05-31'
    or OrderDate >= {d '2011-05-31' } LIMIT 500;;
```

Optimizing data types for Pushdown Cont’d



OrderDate >= '2011-05-31' was translated to (convert(g_0.OrderDate, string) >= '2011-05-31')

OrderDate >= {d '2011-05-31' } was translated to (g_0.OrderDate >= {ts'2011-05-31 00:00:00.0'})

Optimizing data types for Pushdown Cont'd

Using **MSSQL Server Profiler** we can see the actual query received by **MSSQL**	
SELECT TOP 500 g_0."SalesOrderID" AS "c_0", g_0."OrderDate" AS "c_1" FROM "AdventureWorks2019"."Sales"."SalesOrderHeader" g_0 WHERE (**convert(varchar(34), g_0."OrderDate", 21) >= N'2011-05-31' COLLATE Latin1_General_CS_AS)** OR	Every row in the table must be converted from the binary storage format to a human readable string and then compared against the date string. The conversion from date to string also means that any indexes on "OrderDate" are ignored. This conversion severely hurts performance and should be avoided.
g_0."OrderDate" >= CAST('2011-05-31 00:00:00.0' AS DATETIME2)'	The string is converted once into a date time value and then compared against the column. If there are any indexes on the "OrderDate" column, this comparison can take advantage of the index and as such its very efficient.

Limits of Pushdown

Limits of Pushdown

There are **limiting factors** to the ability of pushing down queries. If there are **multiple data sources connected** on the Data Virtuality Platform, a **query spanning multiple databases can not be pushed down completely**, so that some operations will have to be executed in DV's memory (or to be more exact in the buffer space which will be discussed below)

As an example, two different databases on the same MSSQL server can be joined when querying it directly. If each database is represented by a different connection, the **performance will be mitigated** by the fact that DV interprets them as different sources. A way to work around this behavior is to **create a view on the MSSQL server** that joins them already on the MSSQL server side.

Distributed Joins

Difference to Joins in the Same Database

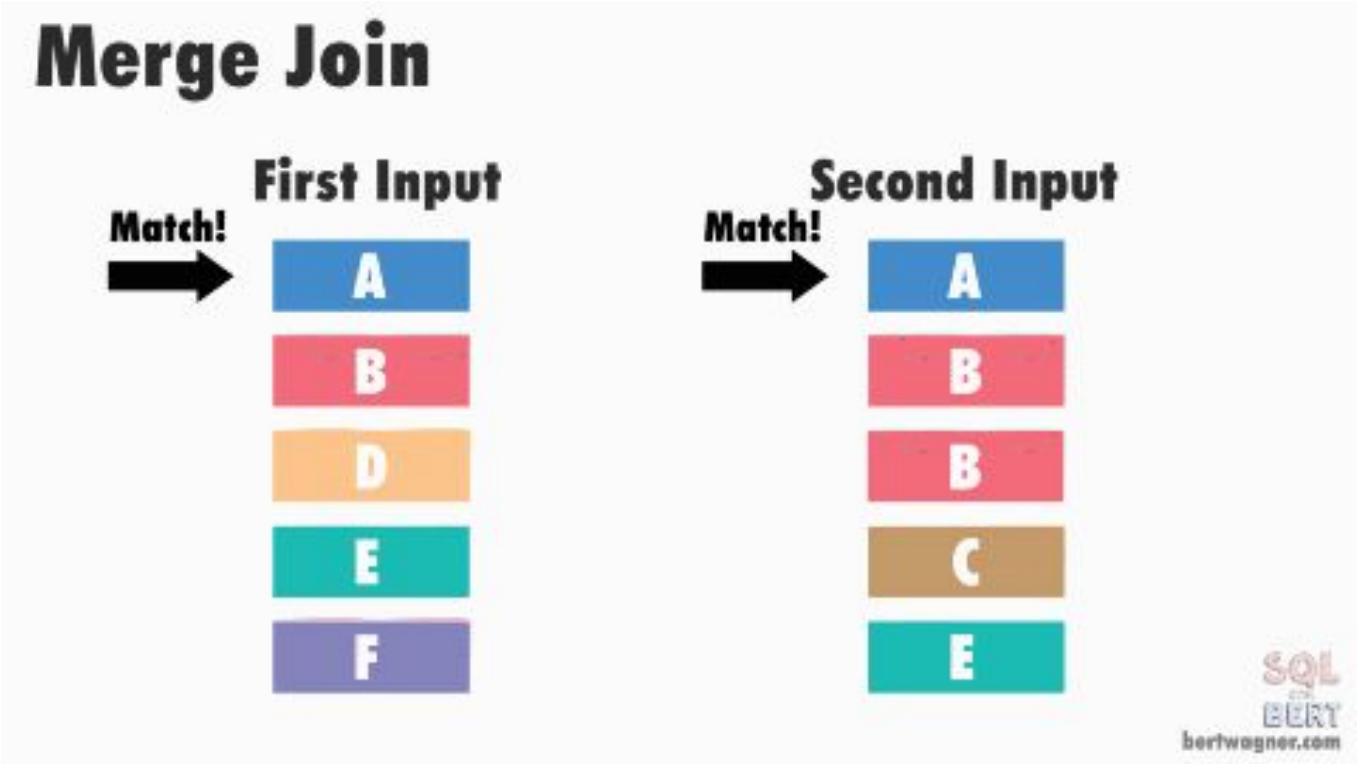
Distributed joins work differently from joins in a single database.

For example, joining two tables from the same database are **executed directly** on the RDBMS having local access to both tables. In a **federated join**, the joined tables are on different databases and can not be joined on the source system. Data Virtuality Platform tries to join disparate data sources as efficiently as possible.

Types of Distributed Joins / Merge Join

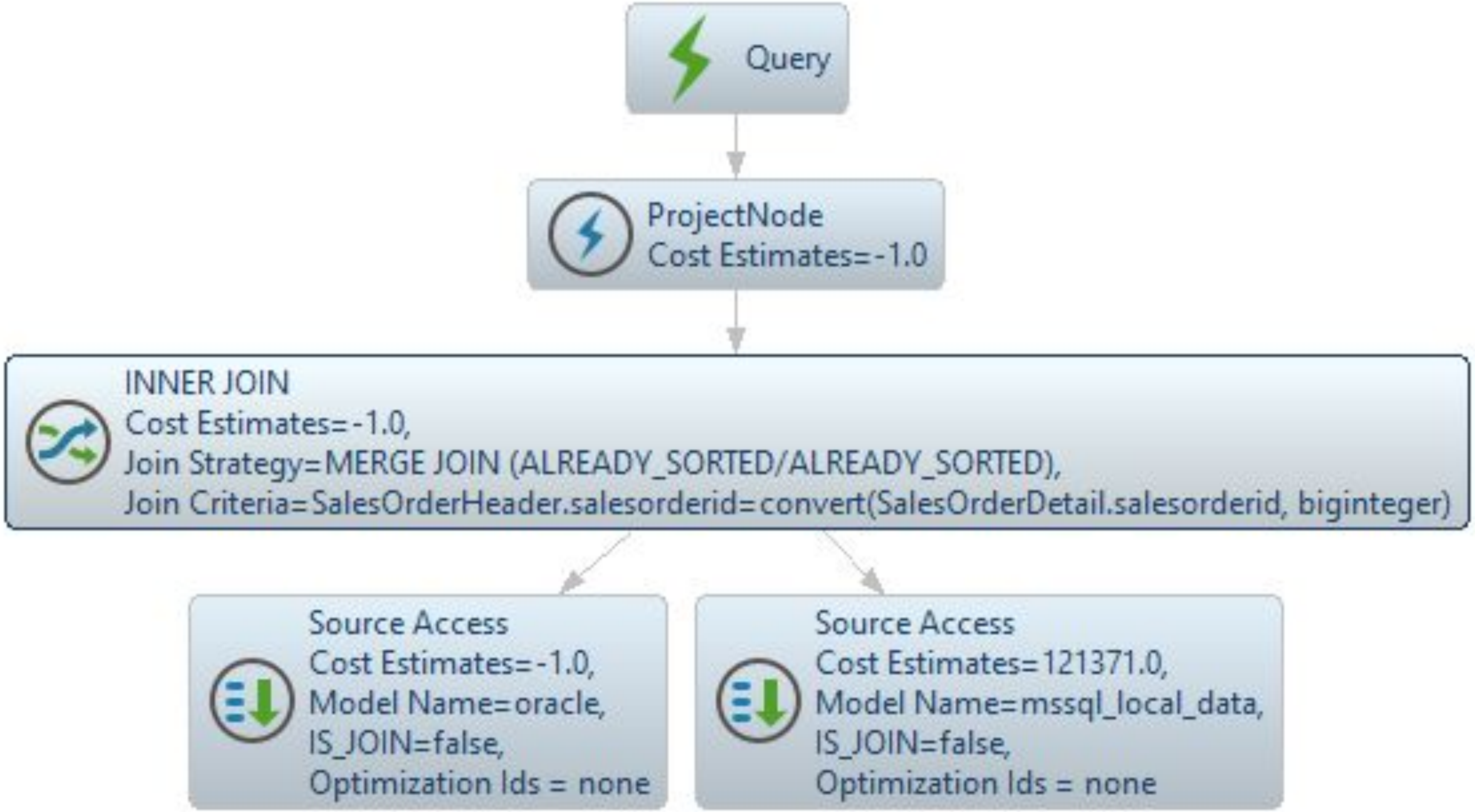
A merge join is considered one of the fastest methods for joining, it does require both inputs to be sorted. The visualization to the right shows how the comparison is done.

More information can be found on https://documentation.datavirtuality.com/24/reference-guide/query-processing/processing



Wagner, Bert. "Visualizing Merge Join Internals and Understanding Their Implications." *Data with Bert Atom*, [https://bertwagner.com/posts/visualizing-merge-join-internals-and-understanding-their-implications/.](https://bertwagner.com/posts/visualizing-merge-join-internals-and-understanding-their-implications/)

Types of Distributed Joins / Merge Join / Query Plan



Types of Distributed Joins / Merge Join / Query Plan Details

Query plan details

INNER JOIN

Output Columns	SALESORDERID (biginteger)
Cost Estimates	Estimated Node Cardinality: -1.0
Child 0	
Child 1	
Join Strategy	MERGE JOIN (ALREADY_SORTED/ALREADY_SORTED)
Join Type	INNER JOIN
Join Criteria	SalesOrderHeader.salesorderid=convert(SalesOrderDetail.salesorderid, biginteger)

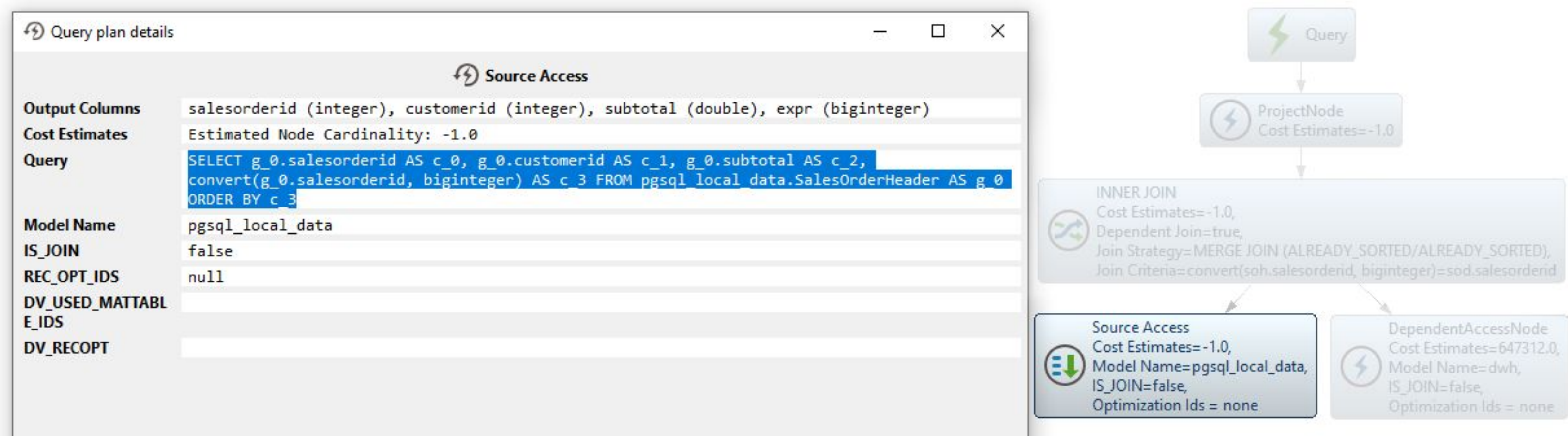
Types of Distributed Joins / Dependent Join / Query Plan

When joining a small and large table across multiple data sources. A dependent join is a great strategy to reduce the amount of rows returned from the larger table. Consider the query below.

```
select
	soh.customerid,
	soh.subtotal
from
	dwh.SalesOrderDetailBig sod
	join
pgsql_local_data.SalesOrderHeader soh
		on sod.salesorderid =
soh.salesorderid
OPTION MAKEDEP
dwh.SalesOrderDetailBig
;;
```

Types of Distributed Joins / Dependent Join / Query Plan Details

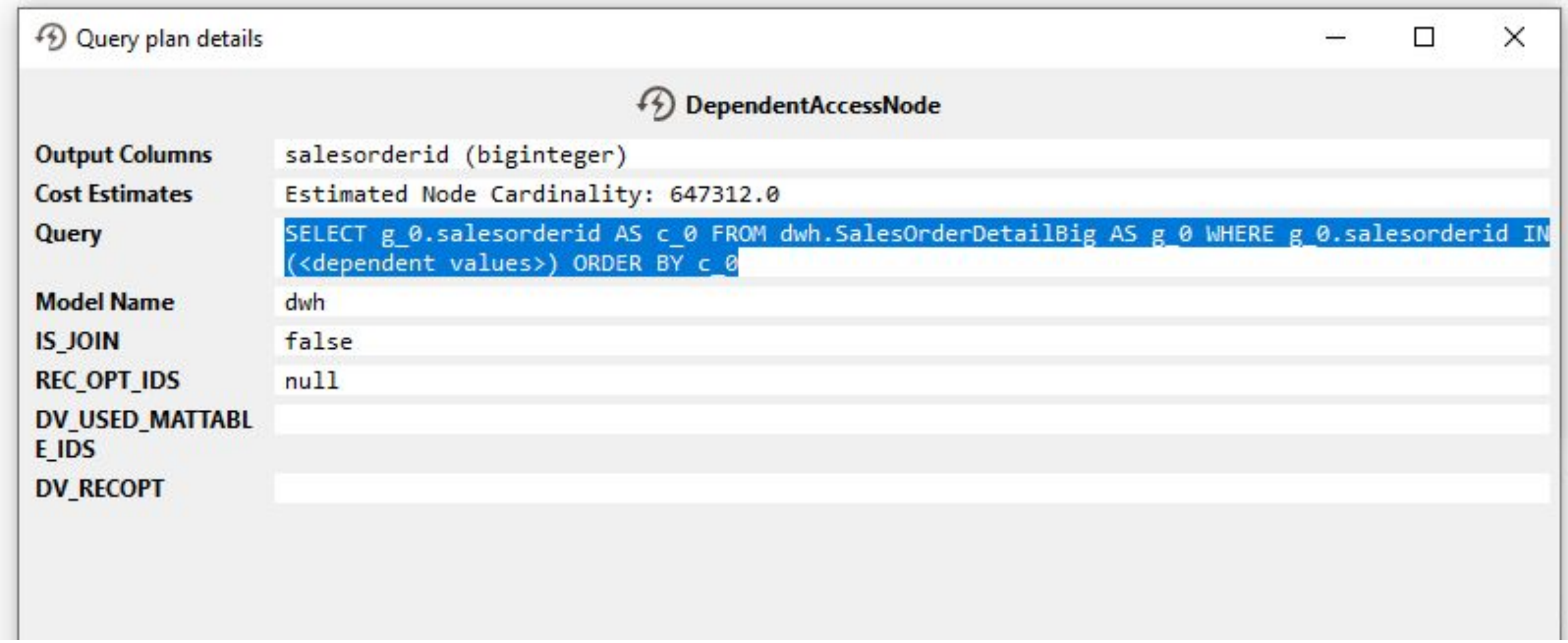
Data Virtuality server first retrieves the rows from the smaller table.



SELECT g_0.salesorderid AS c_0, g_0.customerid AS c_1, g_0.subtotal AS c_2, convert(g_0.salesorderid, biginteger) AS c_3 FROM pgsql_local_data.SalesOrderHeader AS g_0 ORDER BY c_3

Types of Distributed Joins / Dependent Join / Query Plan Details

Next the salesorderid values from the smaller table are used to generate a “WHERE IN” clause to limit the data from the larger table. This can greatly reduce the memory and CPU usage required to process the query.

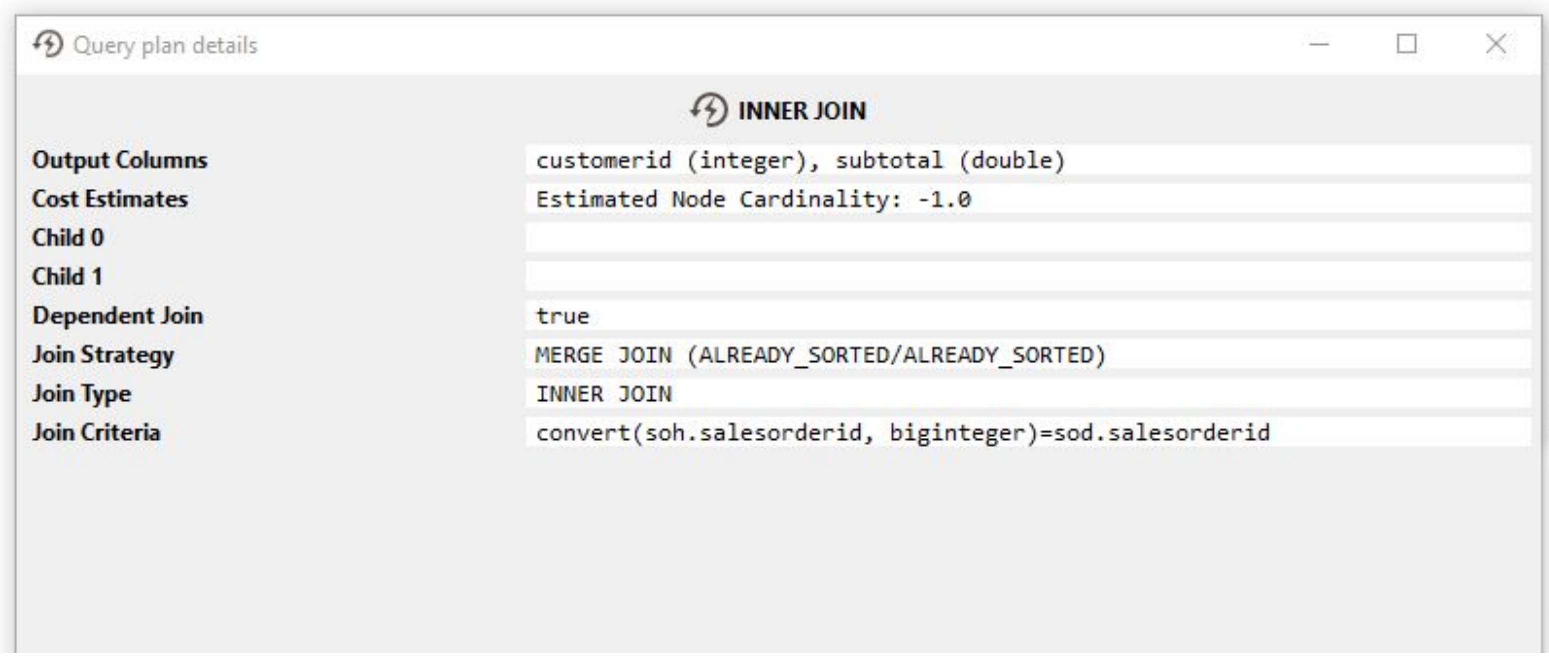


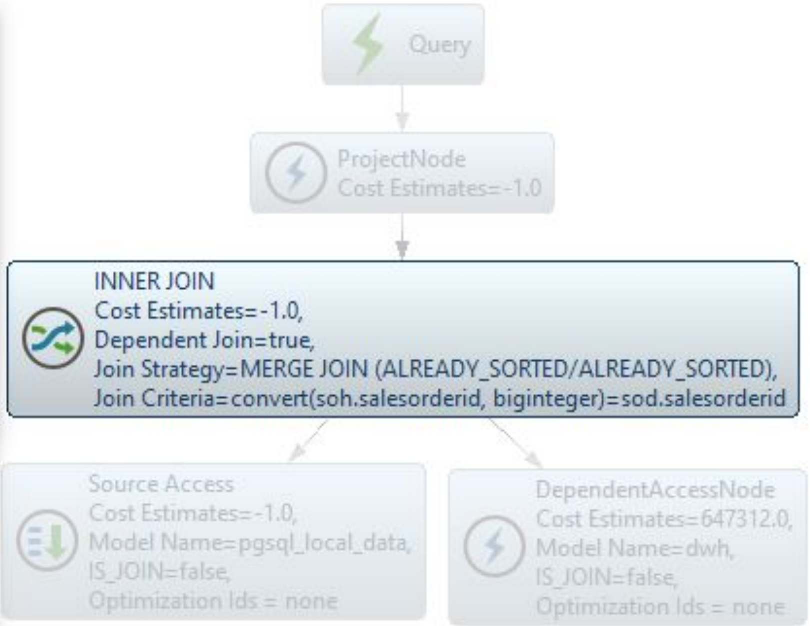
Note the icon

SELECT g_0.salesorderid AS c_0 FROM dwh.SalesOrderDetailBig AS g_0 WHERE g_0.salesorderid IN (<dependent values>) ORDER BY c_0

Types of Distributed Joins / Dependent Join / Query Plan Details

Then a merge join is performed to join both result sets.





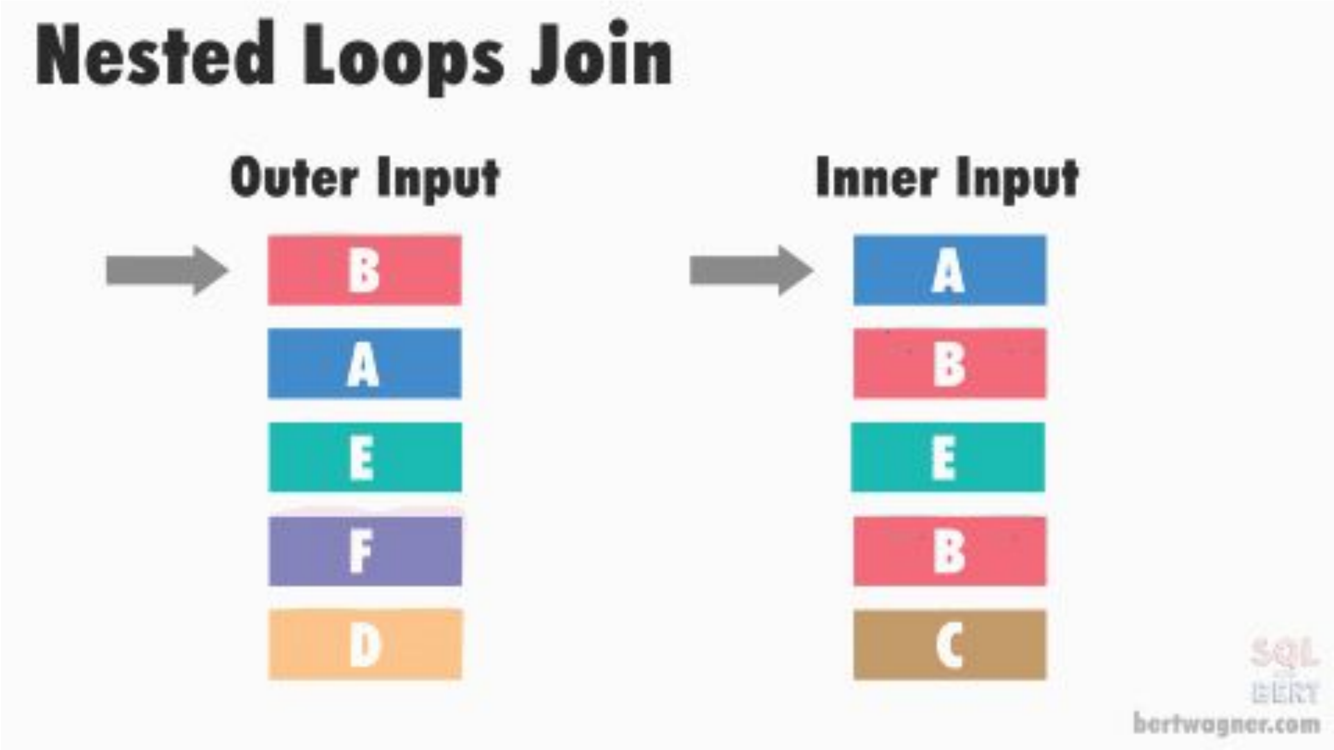
Advice:

The benefits of splitting a query into multiple ones and executing it on one of the sources is efficient **ONLY** for large tables. When using smaller tables, a Merge Join is almost always preferable to a Dependent Join.

Types of Distributed Joins / Nested Loop

Nested loop joins are typically considered slow because of the amount of comparisons that must be done. In the diagram below, both tables have 5 rows and this will result in a total of 25 comparisons (5 x 5). With larger tables this can result in a very large number of comparisons.

This can result in intense CPU and memory usage.



Wagner, Bert. "Visualizing Merge Join Internals and Understanding Their Implications." *Data with Bert Atom*, [https://bertwagner.com/posts/visualizing-merge-join-internals-and-understanding-their-implications/.](https://bertwagner.com/posts/visualizing-merge-join-internals-and-understanding-their-implications/)

Types of Distributed Joins / Nested Loop / Query Plan

Nested loops are used when a cross join is specified.

```
select sod.salesorderid ,soh.salesorderid
from dwh.SalesOrderDetail sod
cross join pgsql_local_data.SalesOrderHeader
soh
OPTION $ALLOW_CARTESIAN ALWAYS;;
```

Note the use of the **OPTION $ALLOW_CARTESIAN ALWAYS**.

By default, cross joins (aka cartesian joins) are prohibited to prevent users from accidentally creating a cross join and impacting performance.

Types of Distributed Joins / Nested Loop / Query Plan Details

Query plan details

CROSS JOIN

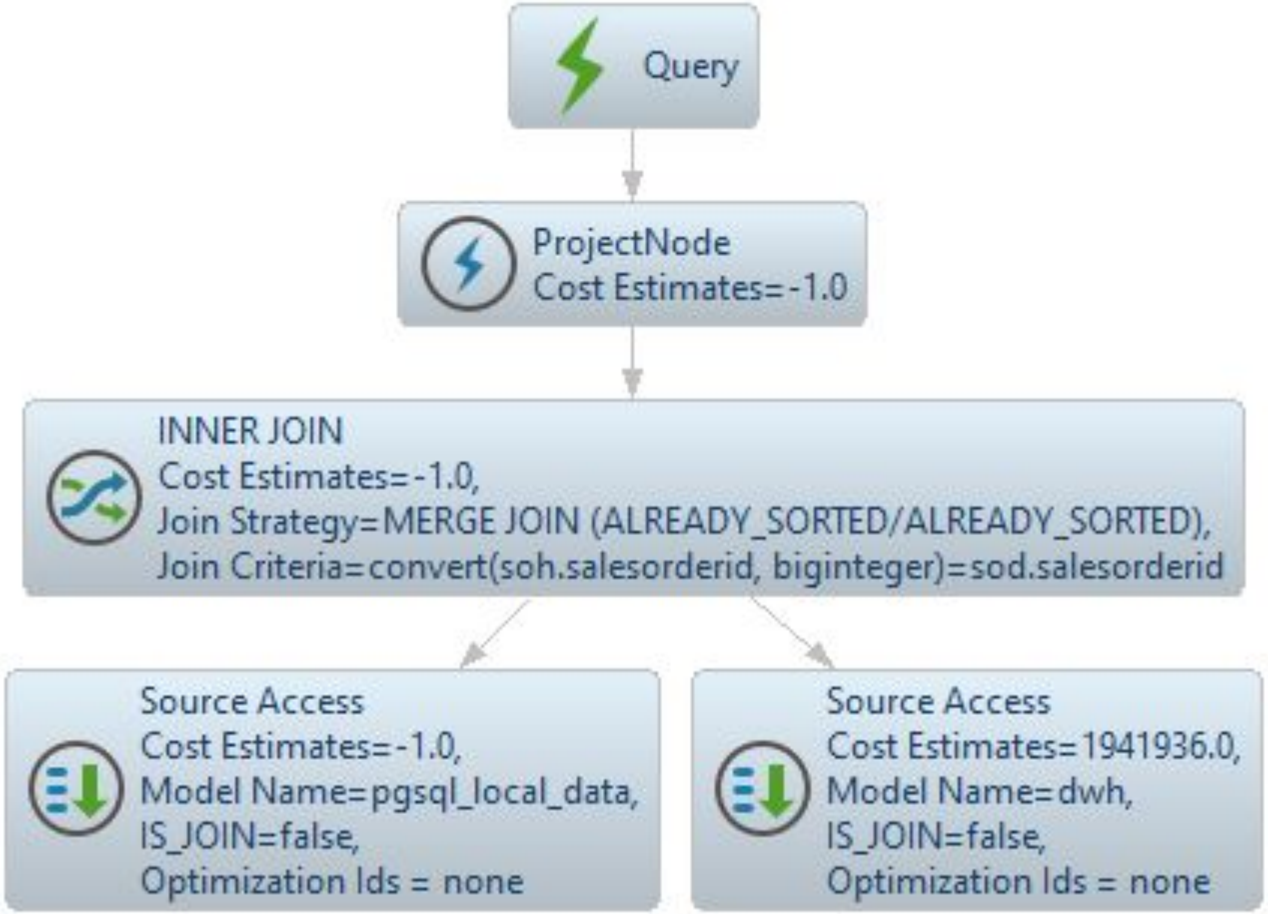
Output Columns	salesorderid (biginteger), salesorderid (integer)
Cost Estimates	Estimated Node Cardinality: -1.0
Child 0	
Child 1	
Join Strategy	NESTED LOOP JOIN
Join Type	CROSS JOIN
Join Criteria	

Forcing Specific Join Types

Forcing Specific Join Types / MAKENOTDEP

This option can be used to prevent a table to be used as a dependent join. The example below shows how the joins are forced to be independent.

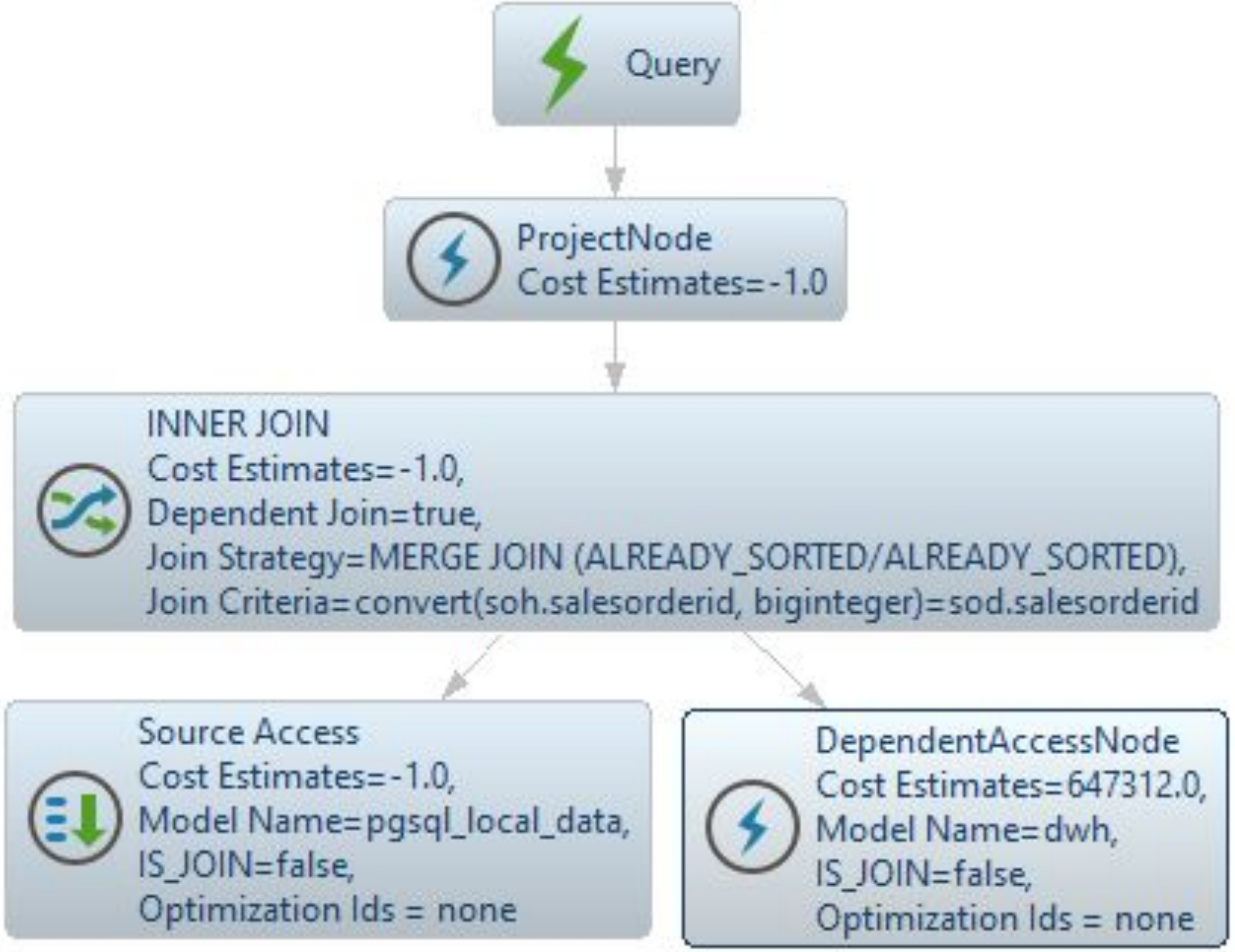
```
select
	soh.customerid,
	soh.subtotal
from
	dwh.SalesOrderDetailBig sod
	join /*+ MAKENOTDEP */
pgsql_local_data.SalesOrderHeader soh
	on sod.salesorderid =
soh.salesorderid
;;
```



Forcing Specific Join Types / MAKEDEP

This option forces the table to query planner to use a dependent join to reduce the amount of data retrieved.

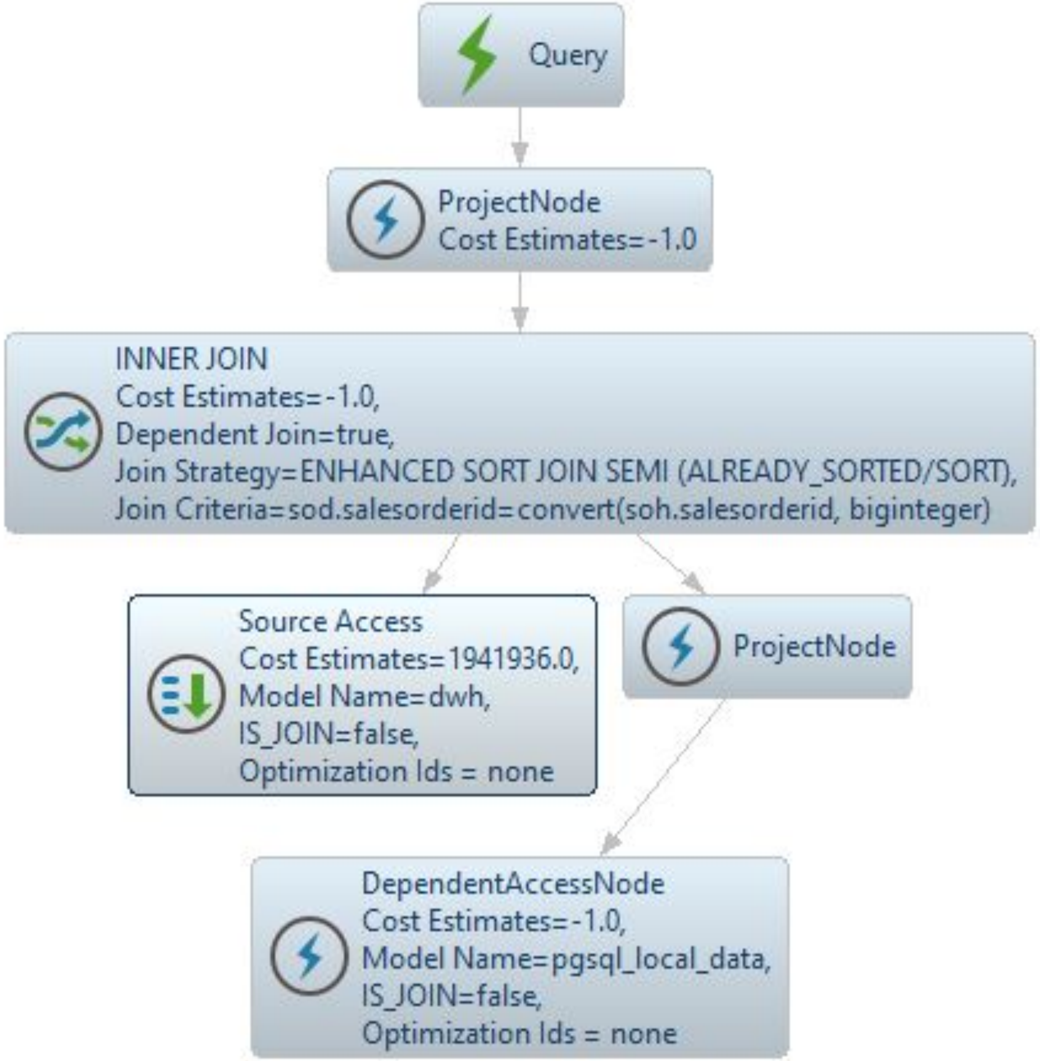
```
select
	soh.customerid,
	soh.subtotal
from
	dwh.SalesOrderDetailBig sod
	join
pgsql_local_data.SalesOrderHeader soh
		on sod.salesorderid =
soh.salesorderid
OPTION MAKEDEP
dwh.SalesOrderDetailBig
;;
```



Forcing Specific Join Types / MAKEIND

In this example, we use the **MAKEIND** and **MAKEDEP** options to reverse the dependency. Now SalesOrderDetailBig will be queried first and those rows will be used to filter the SalesOrderHeader.

```
select
	soh.customerid,
	soh.subtotal
from
	/*+ MAKEIND */
dwh.SalesOrderDetailBig sod
	join /*+ MAKEDEP */
pgsql_local_data.SalesOrderHeader soh
		on sod.salesorderid =
soh.salesorderid
;;
```



Optional Joins

Optional Joins

If a join is marked as optional, the optimizer will omit the query to the joined table if no fields from it are selected. Using the hint, the performance of queries may significantly be improved. This is especially the case with data models where views contain many tables, but the query patterns are varying.

Example for an optional join:

```
select a.column1, b.column2 from sa.a, /*+ optional */ sb.b WHERE a.key =
b.key
```

One **drawback** is that counting the result set size will have unexpected results, which is due to the fact that not all tables are taken into consideration for counting.

See our documentation for optional joins.

Subqueries: Correlated vs Uncorrelated

Subqueries: Correlated vs Uncorrelated

Correlated and uncorrelated are some common types of subqueries. In many cases -- although not always -- the same results can be achieved with correlated and with uncorrelated subquery. However, the uncorrelated subquery will typically have superior performance compared to the correlated one. Here are some examples of equivalent correlated and uncorrelated subqueries:

Correlated

```
SELECT * FROM source1.table1 a WHERE a.id IN (SELECT id FROM source2.table2
WHERE date>NOW()  and a.id=id)
```

Uncorrelated

```
SELECT * FROM source1.table1 a JOIN (SELECT id FROM source2.table2 WHERE
date>NOW()) b ON a.id=b.id
```

The uncorrelated query will execute much faster since Data Virtuality can use optimized JOIN operations using just two source queries while in the case of correlated Data Virtuality is forced to use a Cartesian Product which is computationally intensive.

Any feedback / questions?

@ Data Virtuality GmbH 2021

Thank you!

Please feel free to contact us at:
presales@datavirtuality.com

or

visit us at:
datavirtuality.com

@ Data Virtuality GmbH 2021