

## 5 Caching and Query Performance

### 5.1 Lab Introduction

#### 5.1.1 Purpose

The purpose of this lab is to introduce you to the three types of caching Snowflake employs and how you can use the Query Profile to determine if your query is making the best use of caching.

If you're a data analyst, you'll learn skills critical to your role. If you're in a role other than data analyst, you'll learn how order of ingestion impacts clustering and query performance. All roles will learn how caching enables performant queries without an index.

#### 5.1.2 What you'll learn

- How to access and navigate the Query Profile.
- The differences between metadata cache, query result cache and data cache.
- How to understand when and why the query result cache is being used or not.
- How to determine if partition pruning is efficient, and if not, how to improve it.
- How to determine if spillage is taking place.
- How to use EXPLAIN to determine how to improve your queries.

#### 5.1.3 How to complete this lab

In order to complete this lab, you can key the SQL commands below directly into a worksheet. It is not recommended that you cut and paste from the workbook pdf as that sometimes results in errors.

You can also use the code file for this lab that was provided at the start of the class. You would simply need to open it in TextEdit (mac) or Notepad (Windows), and then copy and paste the code directly into a worksheet.

#### 5.1.4 Scenario

Like traditional relational database management systems, Snowflake employs caching to help you get the query results you want as fast as possible. Snowflake cache is turned on by default and it works for you in the background without you having to do anything. However, if you aren't sure if you're writing queries that leverage caching in the most efficient way possible, you can use the Query Profile to determine how caching is impacting your queries.

Let's imagine that you're a data analyst at Snowbear Air and that you've just learned that Snowflake has different types of caching. You have been working on a few queries that you think could be running faster, but you're not sure. You've decided to become familiar with the Query Profile and plan to use it to see how caching is impacting your queries.

Let's get started!

5.1.5 Create a new folder and call it Caching and Query Performance.

5.1.6 Create a new worksheet inside the folder and call it Working with Cache and the Query Profile.

## 5.2 Accessing and Navigating the Query Profile

In this section you're going to learn how to access, navigate and use the Query Profile. This will prepare you for analyzing query performance and caching in this lab.

5.2.1 Set the worksheet context as follows:

```
USE ROLE TRAINING_ROLE;  
USE WAREHOUSE LEARNER_WH;  
USE DATABASE SNOWBEARAIR_DB;  
USE SCHEMA PROMO_CATALOG_SALES;
```

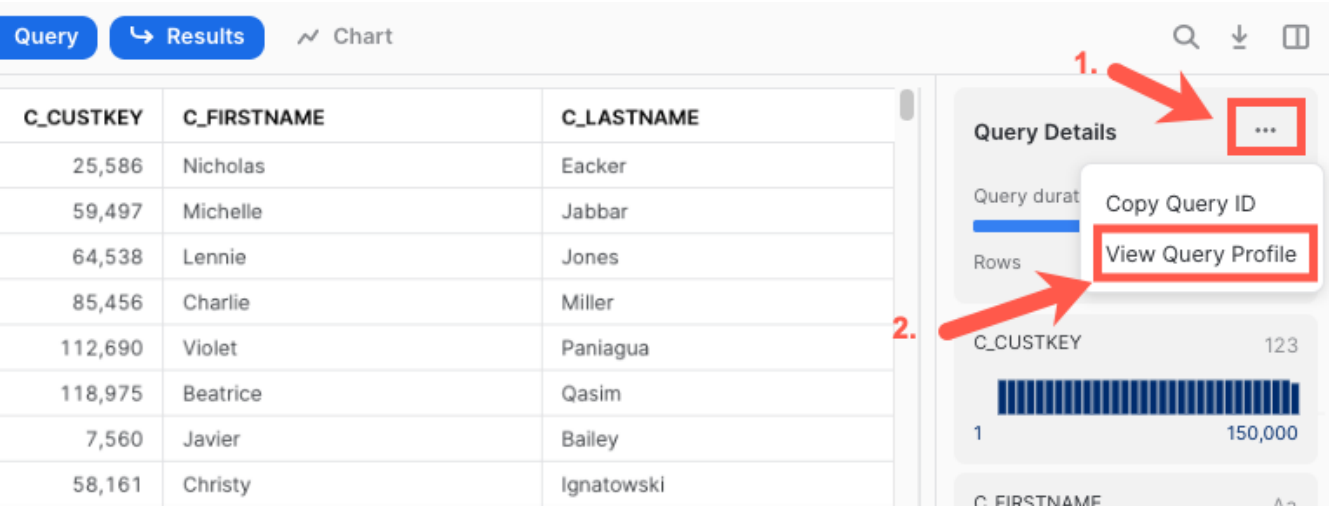
## 5.3 Execute the following simple query in your worksheet.

We need query results to view in the profile. Run the statement below.

```
SELECT DISTINCT  
    c_custkey  
    , c_firstname  
    , c_lastname  
  
FROM  
    customer c;
```

### 5.3.1 Access the Query Profile

Now let's view the profile. Click the ellipses shown in the screenshot. When the dialog box appears, click "View Query Profile". The Query Profile will open in a new tab.



**Figure 25:** Accessing the Query Profile

The query profile will appear as shown below:

< Query History

Query - 019cfd8a-0502-1892-0000-7a21003a113e

+

-

View Query

Node 3

Result [3]

10%

C.C\_CUSTKEY,C.C\_FIRSTNAME,C.C\_LA...

150k

Node 2

Aggregate [1]

10%

150k

Node 1

TableScan [2]

10%

SNOWBEARAIR\_DB.PROMO\_CATALOG\_...

Query Details

Status	Success
Duration	1.3s
Rows	150,000
<a href="#">View more</a>	

Most Expensive Nodes (3 of 3)

Result [3]	(1ms) 10.0%
TableScan [2]	(1ms) 10.0%
Aggregate [1]	(1ms) 10.0%

Profile Overview (Finished)

<div></div>	
Total Execution Time	(1.2s) 100.0%
Processing	20.0%
Remote Disk I/O	10.0%
Initialization	70.0%

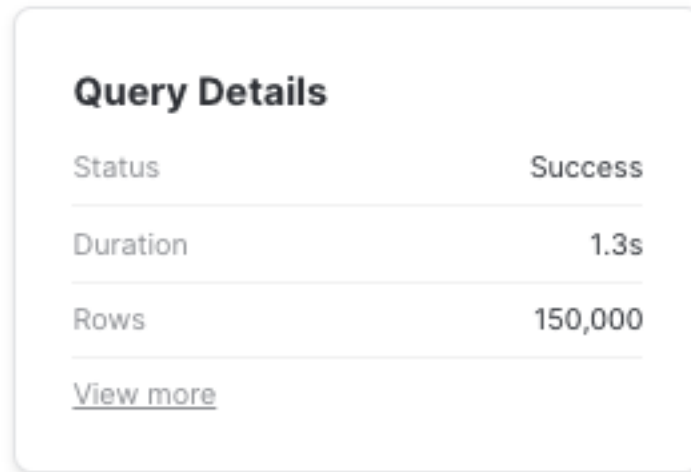
Statistics

Scan progress	100.00%
Bytes scanned	1.39MB
Percentage scanned from cache	0.00%
Bytes written to result	1.13MB
Partitions scanned	1
Partitions total	1

Figure 26: Query Profile

Note that there are four panels that show specific aspects of execution.

- The Query Details panel shows the status of execution, the overall execution duration and the number of rows returned. We can see that this query successfully executed in 1.3 seconds (your results may vary slightly) and returned 150,000 rows. By looking at this you can see if the query succeeded and if it ran within the time frame you were hoping for.

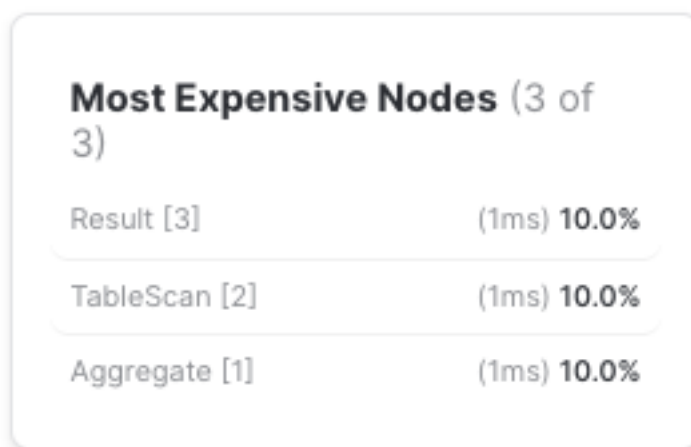


The Query Details panel is a light gray box with rounded corners. It has a title 'Query Details' in bold. Below the title is a table with two columns. The first column lists metrics: Status, Duration, and Rows. The second column shows the corresponding values: Success, 1.3s, and 150,000. At the bottom of the panel is a link labeled 'View more'.

Query Details	
Status	Success
Duration	1.3s
Rows	150,000
<a href="#">View more</a>	

**Figure 27:** Query Details Panel

- The Most Expensive Nodes panel shows the nodes that took the longest to execute. In this case the three nodes all took the same amount of time, so they are all shown. But if one node took a long time to execute, this panel lets you identify and analyze it with the purpose of making it run more efficiently.



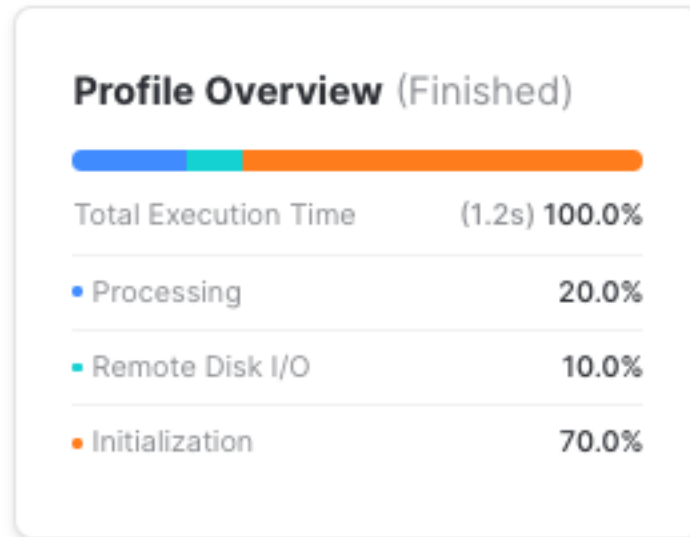
The Most Expensive Nodes panel is a light gray box with rounded corners. It has a title 'Most Expensive Nodes (3 of 3)' in bold. Below the title is a table with two columns. The first column lists execution stages: Result [3], TableScan [2], and Aggregate [1]. The second column shows the time taken and percentage of total time: (1ms) 10.0% for each stage.

Most Expensive Nodes (3 of 3)	
Result [3]	(1ms) 10.0%
TableScan [2]	(1ms) 10.0%
Aggregate [1]	(1ms) 10.0%

**Figure 28:** Most Expensive Nodes Panel

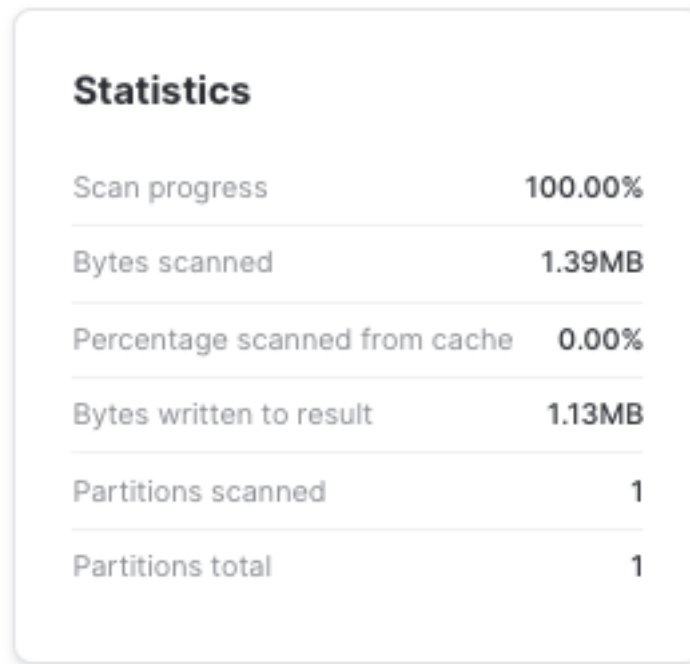
- The Profile Overview Panel displays the percentage of execution time that went towards the following:
  - Processing — The percentage of time spent on data processing by the CPU.
  - Remote Disk IO — The percentage of time when the processing was blocked by remote disk access.
  - Initialization — The percentage of time spent setting up the query processing.

Note that 70% of the execution time was spent setting up the query (Initialization), 10% was blocked by remote disk access (Remote Disk I/O), and 20% was spent on processing data for the query by the CPU (Processing). Most of the execution time (which was just slightly over 1 second) was spent in Initialization due to the fact we were running a very simple query against a small amount of data. If we had been running a more complex query against more data and the execution time was not satisfactory, finding a high disk I/O would tell us we need to do something to reduce that, such as filtering.



**Figure 29:** Profile Overview Panel

- The Statistics panel shows values related to the scanning of partitions. The critical fields are the number of partitions scanned and the number of total partitions. These figures tell you if partition pruning is efficient or not. If the partitions scanned are a far smaller number than the total partitions, you're getting good pruning and should be getting a satisfactory execution time. If the numbers are similar or identical in cases where you expect pruning to take place, you need to revisit and revise your query.



The image shows a 'Statistics' panel with a light gray background and rounded corners. It contains a table with six rows of query execution statistics. The title 'Statistics' is at the top left in bold. Each row has a label on the left and a value on the right, separated by a horizontal line.

Statistics	
Scan progress	100.00%
Bytes scanned	1.39MB
Percentage scanned from cache	0.00%
Bytes written to result	1.13MB
Partitions scanned	1
Partitions total	1

**Figure 30:** Statistics Panel

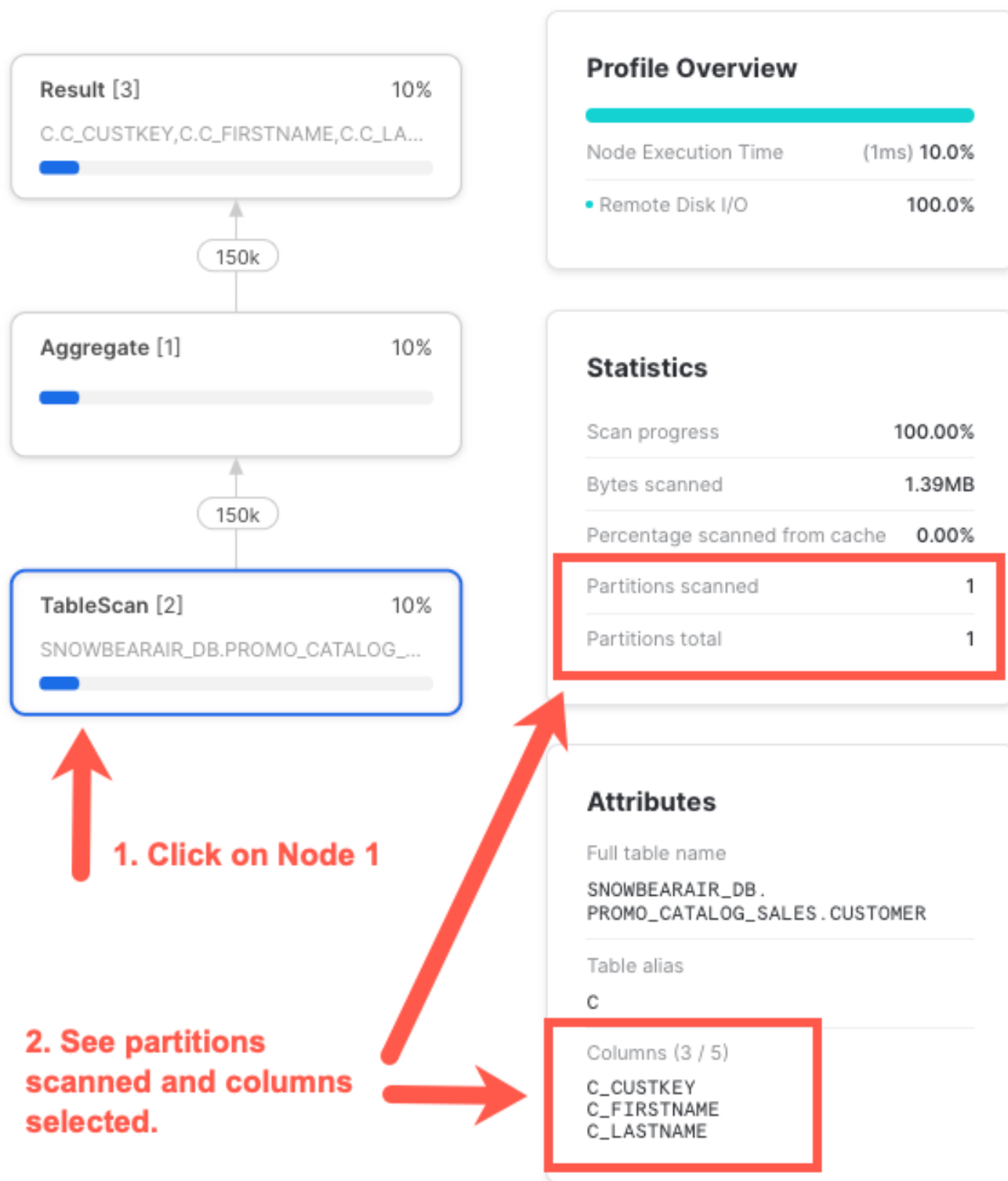
Note that in the center there is a graphical representation in the form of a tree. Floating your cursor in the area around the tree causes it to become a four-way arrow cursor. If you click, hold and drag, you can move the tree around within the pane.

This tree is called an Operator Tree and it illustrates the various processes that were executed and the order in which they were executed. These processes are represented by rectangles called nodes, which are points in the query execution process where a specific type of processing took place. Each of these nodes is clickable.

#### 5.3.2 Click on Node 1 (TableScan[2]).

This node shows statistics related to the scan of the customer table. As we saw in a previous step, the Statistics panel shows that one partition was scanned out of one total partitions evaluated. The query was very simple and the data set isn't very large, so this was to be expected.

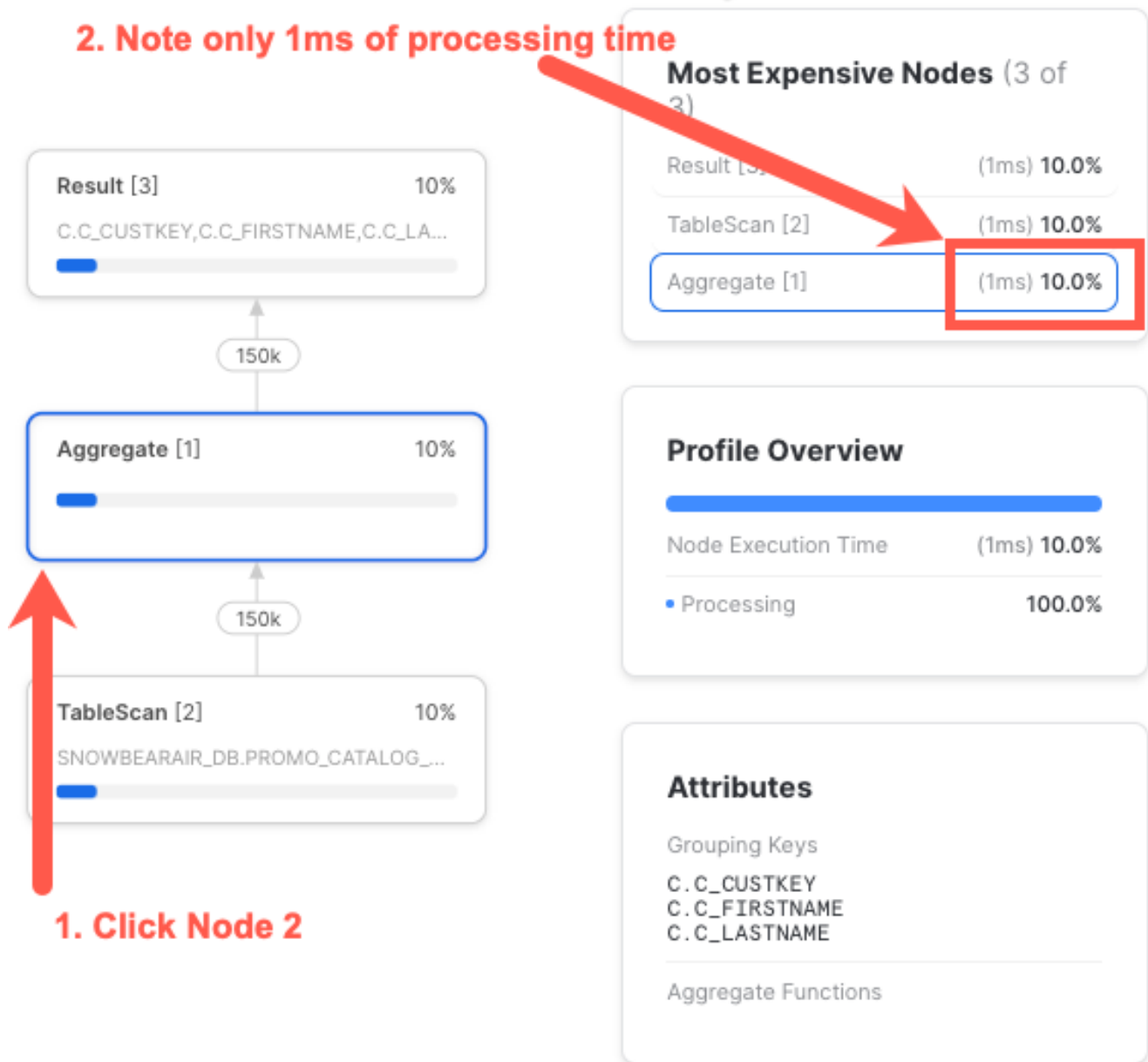
Note that an Attributes panel appears that shows the columns selected during processing of this node's query.

**Figure 31:** Node 1 - Table Scan



### 5.3.3 Click on Node 2 (Aggregate[1]).

This node shows the statistics related to the aggregation of the data. Since the query didn't call for aggregation or have a GROUP BY clause, it processed in one millisecond.

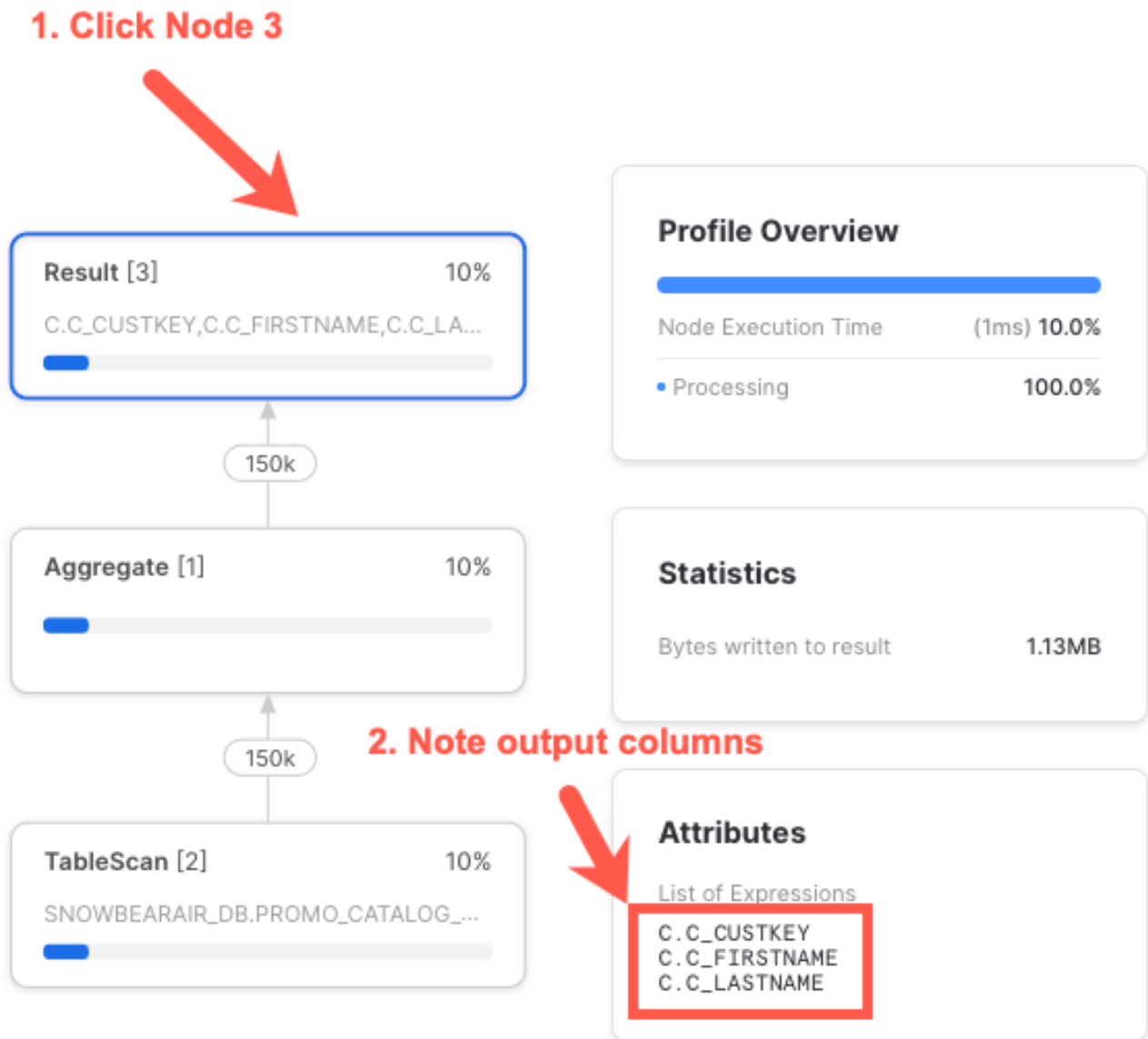


**Figure 32:** Node 2 - Aggregate

**NOTE:** You should also get similar results, it's possible they may differ slightly.

### 5.3.4 Click on Node 3 (Result[3]).

This node shows the columns that were in the output.



**Figure 33:** Node 3 - Result

Note that an Attributes panel appears that shows the columns produced by the processing of this node.

#### 5.3.5 Close the tab

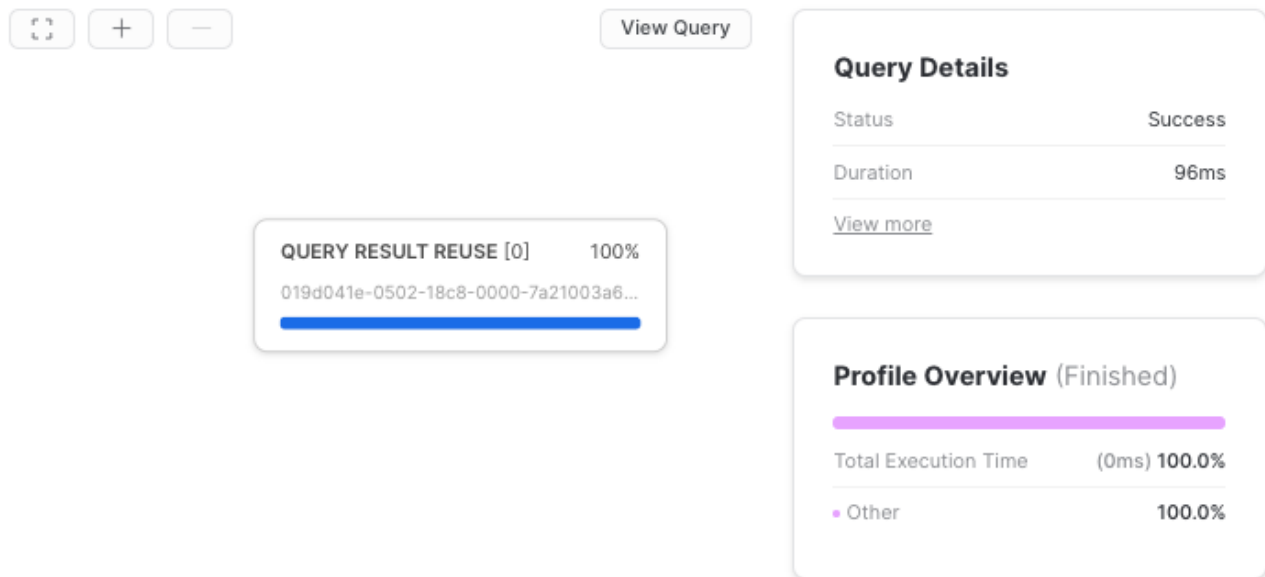
You'll remember that the Query Profile opened up in a new tab. While we could navigate back to our folder, that would leave us with two tabs open. Close this tab and return to the tab with your worksheet.

### 5.3.6 Go back to the tab with your worksheet and run your query again

Once you've run the query again, you should notice that it ran in a few milliseconds, much faster than before. Now let's look at the Query Profile again to see what happened.

< Query History

## Query - 019d041e-0502-18c8-0000-7a21003a604e



**Figure 34:** Query Result Cache

As you can see, the query result cache gave us the same exact result in just a few milliseconds because it was serving us the same results from the query result cache in the cloud services layer. This is a ready-to-go feature that you don't have to think about. You can just run your query a second time and get the same results again if needed.

## 5.4 Metadata Cache

Metadata cache is simple. When data is written into Snowflake partitions, the MAX, MIN, COUNT and other values are stored in the metadata cache in the Cloud Services layer. This means that when your query needs these values, rather than scanning the table and calculating the values, it simply pulls them from the metadata cache. This makes your query run much faster. Let's try it out!

### 5.4.1 Scenario

Let's imagine you've been asked to analyze part and supplier data. We're going to use the PARTSUPP table in our database called SNOWFLAKE\_SAMPLE\_DATA because it provides enough data for this exercise.

### 5.4.2 Set the context.

Note we're setting `USE_CACHED_RESULT = FALSE` in order to avoid using cached results.

```
USE ROLE TRAINING_ROLE;  
USE WAREHOUSE LEARNER_WH;  
USE DATABASE SNOWFLAKE_SAMPLE_DATA;  
USE SCHEMA TPCH_SF10;  
  
ALTER SESSION SET USE_CACHED_RESULT = FALSE;
```

### 5.4.3 Run the following SQL statement:

```
SELECT  
    MIN(ps_partkey)  
    , MAX(ps_partkey)  
  
FROM  
    PARTSUPP;
```

Now check the Query Profile. You should see a single node that says “METADATA-BASED RESULT”. This is because the query profile simply went to cache to get the data you needed rather than scanning a table. So, there was no disk I/O at all.

## 5.5 Data Warehouse Cache

Like any other database system, Snowflake caches data from queries you run so it can be accessed later by other queries. This cache is saved to disk in the warehouse. Let's take a look at how it works. Once again, let's assume you've been asked to analyze part and supplier data.

### 5.5.1 Clear your cache:

```
ALTER SESSION SET USE_CACHED_RESULT = FALSE;  
ALTER WAREHOUSE LEARNER_WH SUSPEND;  
ALTER WAREHOUSE LEARNER_WH RESUME;
```

### 5.5.2 Run the SQL statement below.

Let's start by selecting two columns, `ps_suppkey` and `ps_availqty` with a `WHERE` clause selects only part of the dataset. This will cache the data for the two columns plus the column in the `WHERE` clause.

```
SELECT  
    ps_partkey  
    , ps_availqty  
  
FROM
```

```
PARTSUPP  
  
WHERE  
    ps_partkey > 1000000;
```

### 5.5.3 Look at Percentage Scanned From Cache under Total Statistics in the Query Profile

You should see that the percentage scanned from cache is 0.00%. This is because we just ran the query for the first time.

### 5.5.4 Run the query again

```
SELECT  
    ps_partkey  
    , ps_availqty  
  
FROM  
    PARTSUPP  
  
WHERE  
    ps_partkey > 1000000;
```

### 5.5.5 Look at percentage scanned from cache under total statistics in the Query Profile

You should see that the percentage scanned from cache is 100.00%. This is because the query was able to get 100% of the data it needed from the warehouse cache. This results in faster performance than a query that does a lot of disk I/O.

### 5.5.6 Add columns, run the query, and check the Query Profile.

```
SELECT  
    ps_partkey  
    , ps_suppkey  
    , ps_availqty  
    , PS_supplycost  
    , ps_comment  
  
FROM  
    PARTSUPP  
  
WHERE  
    ps_partkey > 1000000;
```

When you check the percentage scanned from cache, it should be less than 100%. This is because we added columns that weren't fetched previously, so some disk I/O must occur in order to fetch the data in those columns.

## 5.6 Partition Pruning

Partition pruning is a process by which Snowflake eliminates partitions from a table scan based on the partition's metadata. What this means is that fewer partitions are involved in filtering and joining, which gives you faster performance.

Tables will have a natural clustering based on order of ingestion. For example, if a set of customer data is sorted on a unique identifier like customer ID prior to ingestion into the CUSTOMER table, then the rows in the table will be naturally ordered in that way. Thus filtering on that field will result in more efficient pruning than if the table wasn't clustered in an intentional way. Cluster keys can also be created on a field in a table. They will override the natural clustering and can be used in WHERE clause predicates to get more efficient pruning.

In short, if the data in a table has been clustered on a particular column, either naturally or intentionally, knowing which column that is and including it in WHERE clause predicates will result in more partitions being pruned and thus faster query performance.

Let's look at an example. We're going to be using a different and larger dataset than our PROMO\_CATALOG\_SALES dataset so we can leverage partition pruning. Let's set the context, set our warehouse size to xsmall, and clear our data cache.

### 5.6.1 Set the context, warehouse size and clear your cache.

```
USE ROLE TRAINING_ROLE;  
USE WAREHOUSE LEARNER_WH;  
USE DATABASE SNOWFLAKE_SAMPLE_DATA;  
USE SCHEMA TPCDS_SF10TCL;  
  
ALTER WAREHOUSE LEARNER_WH SET WAREHOUSE_SIZE = 'XSMALL';  
  
ALTER SESSION SET USE_CACHED_RESULT=FALSE;  
  
ALTER WAREHOUSE LEARNER_WH SUSPEND;  
ALTER WAREHOUSE LEARNER_WH RESUME;
```

Remember if your warehouse is already suspended, either by timeout or command, you will receive the following error message. "Invalid state. Warehouse LEARNER\_WH cannot be suspended".

### 5.6.2 Execute a query with partition pruning

Let's imagine that the Snowbear Air marketing team has asked you for a list of customer addresses via a join on the CUSTOMER and CUSTOMER\_ADDRESS tables. The data in the CUSTOMER table has been partitioned on C\_CUSTOMER\_SK, which is a unique identifier for each customer. The WHERE clause filters on both C\_CUSTOMER\_SK and on C\_LAST\_NAME. Normally the predicate with LIKE would result in a full table scan. Execute the query below and check the Query Profile to see what happens.

```
SELECT  
    C_CUSTOMER_SK
```

```
    , (CA_STREET_NUMBER || ' ' || CA_STREET_NAME) AS CUST_ADDRESS
    , CA_CITY
    , CA_STATE

FROM
    CUSTOMER
  INNER JOIN CUSTOMER_ADDRESS ON C_CUSTOMER_ID = CA_ADDRESS_ID

WHERE
    C_CUSTOMER_SK between 100000 and 600000
  AND
    C_LAST_NAME LIKE '%John%'

ORDER BY
    CA_CITY
  , CA_STATE;
```

If you check the nodes for each table, you'll see that the CUSTOMER and CUSTOMER\_ADDRESS tables have just over 500 total partitions.

The Query Profile tells us that the query ran in a few seconds and only about half of the partitions were scanned. So this query ran faster than it would have otherwise because partition pruning worked for us.

Now let's run a query without the C\_CUSTOMER\_SK field in the WHERE clause predicate and see what happens.

### 5.6.3 Execute a query without partition pruning and check the Query Profile

```
SELECT
    C_CUSTOMER_SK
    , (CA_STREET_NUMBER || ' ' || CA_STREET_NAME) AS CUST_ADDRESS
    , CA_CITY
    , CA_STATE

FROM
    CUSTOMER
  INNER JOIN CUSTOMER_ADDRESS ON C_CUSTOMER_ID = CA_ADDRESS_ID

WHERE
    C_LAST_NAME = 'Johnson'

ORDER BY
    CA_CITY
  , CA_STATE;
```

The Query Profile tells us that this query took longer to run and all partitions were scanned. This is because the data in the CUSTOMER table is not clustered on the C\_LAST\_NAME column, and the natural clustering for that column was not such that we could get good pruning.

Now let's consider what happens when a table's data is clustered such that it enables efficient pruning. For example, let's imagine that the data ingested was sorted by customer ID, and the IDs run from 1 to 1,000,000. If data is ingested incrementally after the initial load and the new IDs run from 1,000,001 to 2,000,000 but they are not sorted by customer ID, the natural clustering for the table overall will drift and the pruning may become less

efficient. At that time clustering keys can be added to the table in order to get better pruning and thus greater query efficiency. Although as a data analyst you will not be adding clustering keys to tables, the takeaway is that understanding how your table is clustered can help you write more efficient queries. Your DBA (Snowflake Administrator) is someone you can work with to understand how your tables are clustered.

## 5.7 Determine If Spillage Is Taking Place

Now let's determine if spillage is taking place in one of our queries. Spillage means that because an operation cannot fit completely in memory, data is spilled to disk within the warehouse. Operations that incur spillage are slower than memory access and can greatly slow down query execution. Thus, you may need to be able to identify and rectify spillage.

Let's imagine that Snowbear Air wants to determine the average list price, average sales price and average quantity for both male and female buyers in the year 2000 for the months January through October.

Rather than use our PROMO\_CATALOG\_SALES database for this scenario, we're going to use another database that has enough data to create spillage. The structure and content of the data is less important than the fact that we can generate and resolve a spillage issue.

### 5.7.1 Set the context and resize the warehouse.

```
USE ROLE TRAINING_ROLE;  
USE WAREHOUSE LEARNER_WH;  
USE DATABASE SNOWFLAKE_SAMPLE_DATA;  
USE SCHEMA TPCDS_SF10TCL;  
  
ALTER WAREHOUSE LEARNER_WH SET WAREHOUSE_SIZE = 'XSMALL';
```

### 5.7.2 Clear the cache.

Run the SQL statements below to set USE\_CACHED\_RESULT to false, and suspend and resume your warehouse to clear any cache. This will ensure we get spillage when we run our query.

```
ALTER SESSION SET USE_CACHED_RESULT=FALSE;  
  
ALTER WAREHOUSE LEARNER_WH SUSPEND;  
ALTER WAREHOUSE LEARNER_WH RESUME;
```

### 5.7.3 Run a query that generates spillage.

Note that the query below has a nested query. The nested query determines the average list price, average sales price and average quantity per gender type and order number. The outer query then aggregates those values by gender.

Run the query below. It should take between 4-5 minutes to run.



```
SELECT
    cd_gender
  , AVG(lp) average_list_price
  , AVG(sp) average_sales_price
  , AVG(qu) average_quantity
FROM
  (
    SELECT
      cd_gender
      , cs_order_number
      , AVG(cs_list_price) lp
      , AVG(cs_sales_price) sp
      , AVG(cs_quantity) qu

    FROM
      catalog_sales
      , date_dim
      , customer_demographics

    WHERE
      cs_sold_date_sk = d_date_sk
      AND
      cs_bill_cdemo_sk = cd_demo_sk
      AND
      d_year IN (2000)
      AND
      d_moy IN (1,2,3,4,5,6,7,8,9,10)

    GROUP BY
      cd_gender
      , cs_order_number
  ) inner_query
GROUP BY
  cd_gender;
```

#### 5.7.4 View the results.

For female buyers you should see something very similar to the following figures:

- average\_list\_price: 100.995691505527
- average\_sales\_price: 50.494185840967
- average\_quantity: 50.497579311044

For male buyers you should see something very similar to the following figures:

- average\_list\_price: 100.992076976143
- average\_sales\_price: 50.491772005658
- average\_quantity: 50.499846880459

### 5.7.5 Check out the Query Profile.

Go to the Query Profile. As you will see at the bottom of the Statistics, gigabytes of data were spilled to local storage. Let's rectify this issue by rewriting our query.

If you look back at the query you just ran, you'll see that the outer query is not really necessary. All you need to do is remove the `cs_order_number` column from the nested query and then run it.

### 5.7.6 Run the modified nested query.

Let's run the query:

```
SELECT
    cd_gender
  , AVG(cs_list_price) lp
  , AVG(cs_sales_price) sp
  , AVG(cs_quantity) qu
FROM
    catalog_sales
  , date_dim
  , customer_demographics
WHERE
    cs_sold_date_sk = d_date_sk
  AND
    cs_bill_cdemo_sk = cd_demo_sk
  AND
    d_year IN (2000)
  AND
    d_moy IN (1,2,3,4,5,6,7,8,9,10)
GROUP BY
    cd_gender;
```

### 5.7.7 Check your results.

The query should have run in 1-3 minutes (your results may vary). Compare your results to the ones you got previously. They may be slightly different past the hundreds place to the right of the decimal, but that is due to the differences in rounding between the original query and the modified nested query. So, in essence you got the same results only in half the time.

### 5.7.8 Check the Query Profile.

As you will see at the bottom of the Statistics, there is no longer a spillage entry. This means that you resolved your spillage issue by simply rewriting your query so that it was more efficient.

## 5.8 Review the EXPLAIN Plan.

Now let's compare the EXPLAIN plans from both of the queries we just ran in order to see how they are different.

### 5.8.1 Use EXPLAIN to see the plan for the first query.

```
EXPLAIN
SELECT
    cd_gender
  , AVG(lp) average_list_price
  , AVG(sp) average_sales_price
  , AVG(qu) average_quantity
FROM
  (
    SELECT
      cd_gender
      , cs_order_number
      , AVG(cs_list_price) lp
      , AVG(cs_sales_price) sp
      , AVG(cs_quantity) qu

    FROM
      catalog_sales
      , date_dim
      , customer_demographics

    WHERE
      cs_sold_date_sk = d_date_sk
      AND
      cs_bill_cdemo_sk = cd_demo_sk
      AND
      d_year IN (2000)
      AND
      d_moy IN (1,2,3,4,5,6,7,8,9,10)

    GROUP BY
      cd_gender
      , cs_order_number
  ) inner_query
GROUP BY
  cd_gender;
```

### 5.8.2 Click on the Operation header to sort the rows

Note that there are 12 rows that correspond to the execution nodes that you would see in the Query Profile. Also note that two of the rows are aggregate rows. The one below executes the averaging of the list price, sales price and quantity:

```

    aggExprs: [SUM((SUM(CATALOG_SALES.CS_LIST_PRICE)) / (COUNT(CATALOG_SALES.CS_LIST_PRICE)))
, COUNT((SUM(CATALOG_SALES.CS_LIST_PRICE)) / (COUNT(CATALOG_SALES.CS_LIST_PRICE)))
, SUM((SUM(CATALOG_SALES.CS_SALES_PRICE)) / (COUNT(CATALOG_SALES.CS_SALES_PRICE)))
, COUNT((SUM(CATALOG_SALES.CS_SALES_PRICE)) / (COUNT(CATALOG_SALES.CS_SALES_PRICE)))
, SUM((SUM(CATALOG_SALES.CS_QUANTITY)) / (COUNT(CATALOG_SALES.CS_QUANTITY)))
, COUNT((SUM(CATALOG_SALES.CS_QUANTITY)) / (COUNT(CATALOG_SALES.CS_QUANTITY)))])
, groupKeys: [CUSTOMER_DEMOGRAPHICS.CD_GENDER]

```

**Figure 35:** Aggregate Row Expression

5.8.3 Run the EXPLAIN statement for the second query.

```

EXPLAIN
SELECT
    cd_gender
,   AVG(cs_list_price) lp
,   AVG(cs_sales_price) sp
,   AVG(cs_quantity) qu

FROM
    catalog_sales
,   date_dim
,   customer_demographics

WHERE
    cs_sold_date_sk = d_date_sk
AND
    cs_bill_cdemo_sk = cd_demo_sk
AND
    d_year IN (2000)
AND
    d_moy IN (1,2,3,4,5,6,7,8,9,10)

GROUP BY
    cd_gender;

```

Notice now that this plan is identical to the first one except that there is one aggregate row fewer than in the previous explain plan (for a total of 11 rows). Specifically, the one shown in the previous step in this lab is the one that is gone because we removed the outer query. Making that change alone was enough to cut query time by more than half.

5.8.4 Change your warehouse size to XSmall.

```

ALTER WAREHOUSE LEARNER_WH
    SET WAREHOUSE_SIZE = 'XSmall';

ALTER WAREHOUSE LEARNER_WH SUSPEND;

```

## 5.9 Summary

Writing efficient queries is an art that takes a solid understanding of how Snowflake caching and query pruning impact query performance. While it's impossible to show you every single scenario, you should know that getting proficient at using tools like the Query Profile and the EXPLAIN plan will help you better understand how caching is impacting your query performance. This in turn will allow you to write better queries that will enable you to achieve a shorter run time.

### 5.10 Key takeaways

- Snowflake employs caching to help you get the query results you want as fast as possible.
- Cache is turned on by default and it works for you in the background without you having to do anything.
- The Query Profile is a useful tool for understanding how caching and partition pruning are impacting your queries.
- As you add or remove columns to/from a SELECT clause or a WHERE clause, your percentage scanned from cache value could go up or down.
- If your query only requests MIN or MAX values, that will come from metadata cache in the Cloud Services layer rather than from disk I/O, which results in fast performance.
- Query Result cache is invoked when you run the exact same query twice.
- Data cache resides in the warehouse and it stores data from queries you run for a temporary period of time.
- Including the column on which the data in the table is clustered in a WHERE clause predicate can leverage partition pruning to get improved query performance.
- Natural clustering can drift over time such that clustering keys may need to be added to a table in order to restore querying efficiency.
- Using EXPLAIN can give you insight into how Snowflake will execute your query. You can use it to identify and remove bottlenecks in your query so you can resolve them and get better efficiency.