SQL Analytics Pools Data Loading Scenarios and Best Practices

(Azure Synapse Analytics In-A-Day Lab 01)

Contents

Overview

This module will go over the internals of loading data with PolyBase and provide a deeper understanding of factors that can impact loading performance.

To simplify this experience, the entire lab is conducted using SSMS and T-SQL. In the real world, you would automate everything that you see in this lab with a tool like Azure Data Factory, SSIS, or another 3rd party tool; however, the learnings presented here are still valuable as you move into production scenarios.

Pre-requisites:

- Existing SQL Analytics Pool
- Existing Azure Storage Instance with datasets
- Azure subscription

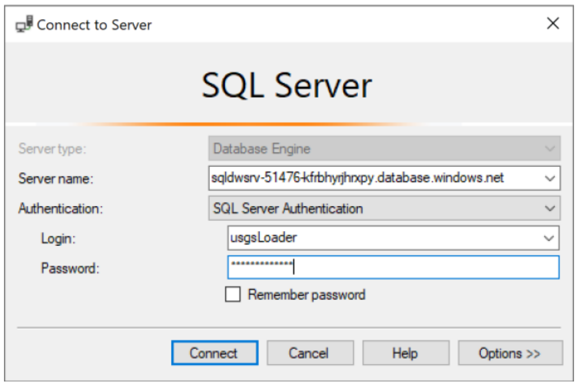
- SQL USER in database with CONTROL Permissions

Connect to your SQL Data Warehouse as loading user

Before you start loading data into your SQL Data Warehouse, you need to login into the DW with your loading user. We advise having a separate user for loading because it gives you greater control over security, resource allocation, and workload management by separating loading jobs from analytical queries.

In SQL Server Management Studio:

1. Log in as 'usgsLoader'

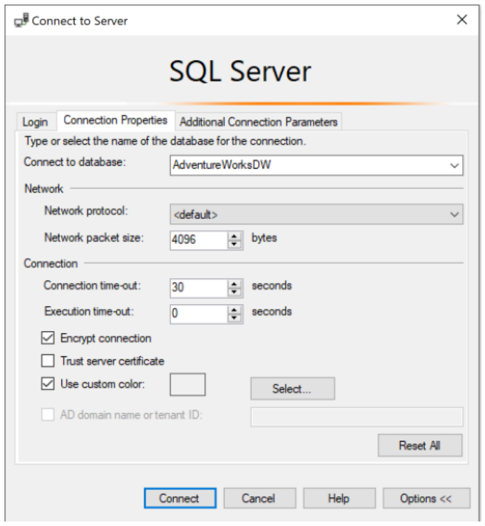


Connect to Server

SQL Server

Server type: Database Engine
Server name: sqldwsrv-51476-kfrbhyrjhrxpy.database.windows.net
Authentication: SQL Server Authentication
Login: usgsLoader
Password: *************
☐ Remember password

2. Click on 'Options', change the database name to 'AdventureWorksDW' and click 'Connect'.



Connect to Server

SQL Server

Type or select the name of the database for the connection.
Connect to database: AdventureWorksDW
Network
Network protocol: <default>
Network packet size: 4096 bytes
Connection
Connection time-out: 30 seconds
Execution time-out: 0 seconds
☑ Encrypt connection
☐ Trust server certificate
☑ Use custom color:
☐ AD domain name or tenant ID:

Create your loading objects

When loading data into SQL DW it is a best practice to use PolyBase. PolyBase allows you to load files (Delimited Text, ORC, RC, and Parquet) from Azure Blob Storage, Azure Data Lake Store and Azure Data Lake Store Gen2.

To use PolyBase there are four objects that must be created:

- **Database Scoped Credential** – Credential used to authenticate to External Data Source.
- **External Data Source** – Describes where the data is located at a coarse grain.
- **External File** – Describes how the files are written.
- **External Table** – Describes the fine grain location and how the data is written in the file.

Let's take a look at the code!

Select New Query and verify that AdventureWorksDW is the selected database.

Execute the following TSQLs as admin user (sqladmin)

```
CREATE MASTER KEY
GO

CREATE DATABASE SCOPED CREDENTIAL Ready2020
WITH IDENTITY = 'mas',
SECRET = '<your storage account key>'
GO

CREATE EXTERNAL DATA SOURCE Ready_store
WITH
(
TYPE = HADOOP,
LOCATION = N'wasbs://usgs@<your storage account name>.blob.core.windows.net/',
CREDENTIAL = Ready2020
)
GO


CREATE EXTERNAL FILE FORMAT TextFileFormat_Ready
WITH
(
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS
    (
        FIELD_TERMINATOR = '|'
    )
)
GO
```

The next thing that we need to do is create an external table over the files so that we can access the data from SQL DW. We can do this with the objects that we created above. Think of an External Table as a pointer to the files on the storage system.

Execute the following TSQL to create an external table:

```
CREATE SCHEMA staging;
GO


CREATE EXTERNAL TABLE [staging].[STG_factWeatherMeasurements_text]
(
       [SiteName] [varchar](52) NOT NULL,
       [ReadinType] [varchar](52) NOT NULL,
       [ReadingTypeID] [real] NOT NULL,
       [ReadingValue] [real] NOT NULL,
       [ReadingTimestamp] [datetime2](7) NOT NULL,
       [ReadingUnit] [varchar](52) NOT NULL,
       [fpscode] [int] NOT NULL
)
WITH (
DATA_SOURCE = [Ready_store],
LOCATION = N'loading/factWeatherMeasurements/Text_files',
FILE_FORMAT = [TextFileFormat_Ready],
REJECT_TYPE = VALUE,
REJECT_VALUE = 0)
```

Impact of file format on loading

Choosing the right file format for loading is often an overlooked choice when customers are designing their load jobs into SQL DW.

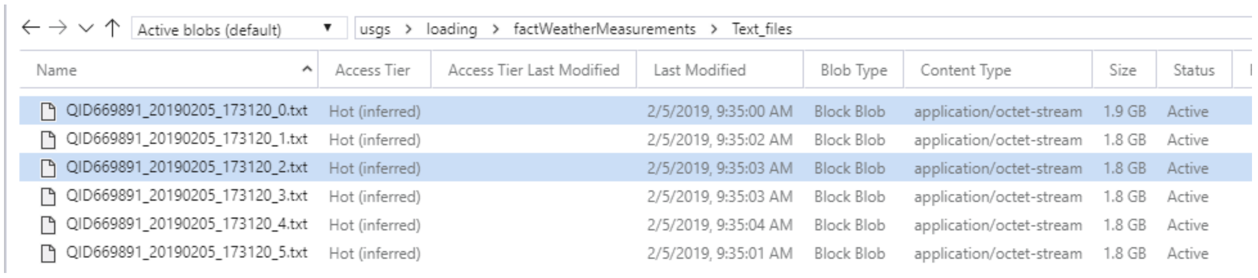
By far, the most popular format used to load data into SQL DW is the delimited text file. It is human readable, generic and the default output of many on-prem systems. However, the delimited text file can cause loading issues because it is essentially a collection of strings which are open to interpretation by the loading tool. Additionally, several delimited text generation tools do not enforce data quality during writes, which means you when the data is written is can no longer be interpreted as the same data. For example, the difference between a null value and an empty string is subtle in a delimited text file. If quotes are turned off, then all empty strings will be interpreted as a null! This is seemingly innocent form of data loss that can have large consequences.

Parquet is another very popular format that is gaining popularity in the big data ecosystem because of its binary data representation, strongly typed data, and compression capabilities. This format is popularized by the spark and hive ecosystem and we are seeing the increase in adoption as the Modern Data Warehouse Architecture continues to grow in popularity.

For this section of the lab, the same data is stored on Azure Storage in Delimited Text, Parquet and Gzip Compressed Delimited text. We are going to load each in turn and look at storage size and load performance.

Delimited Text

The data below was generated via a PolyBase export of 134,333,511 records. As you can see below, PolyBase tries to evenly distribute the data across multiple files. This is a best practice when writing data because it effectively parallelizes the DW resources to load each file simultaneously.



Active blobs (default) | usgs > loading > factWeatherMeasurements > Text_files

Name	Access Tier	Access Tier Last Modified	Last Modified	Blob Type	Content Type	Size	Status
QID669891_20190205_173120_0.txt	Hot (inferred)		2/5/2019, 9:35:00 AM	Block Blob	application/octet-stream	1.9 GB	Active
QID669891_20190205_173120_1.txt	Hot (inferred)		2/5/2019, 9:35:02 AM	Block Blob	application/octet-stream	1.8 GB	Active
QID669891_20190205_173120_2.txt	Hot (inferred)		2/5/2019, 9:35:03 AM	Block Blob	application/octet-stream	1.8 GB	Active
QID669891_20190205_173120_3.txt	Hot (inferred)		2/5/2019, 9:35:03 AM	Block Blob	application/octet-stream	1.8 GB	Active
QID669891_20190205_173120_4.txt	Hot (inferred)		2/5/2019, 9:35:04 AM	Block Blob	application/octet-stream	1.8 GB	Active
QID669891_20190205_173120_5.txt	Hot (inferred)		2/5/2019, 9:35:01 AM	Block Blob	application/octet-stream	1.8 GB	Active

One should note that these are 1.8GB files because the files are uncompressed. This is raw data size which we will see change as we change file formats and compressions later in the lab.

This statement points to the directory location of the files. PolyBase will traverse all files in a directory and all subdirectories (recursive directory traversal) up to 33k files.

The next step is to run a CREATE TABLE AS SELECT (CTAS) to create a new table from the external data. Let's load some data!

```
CREATE TABLE [staging].[STG_text_load]
WITH
(
DISTRIBUTION = ROUND_ROBIN,
HEAP
)
AS
SELECT * FROM [staging].[STG_factWeatherMeasurements_text] option(label =
'STG_text_load')
```

While this is running you can view the steps with the Dynamic Management Views (DMVs)

Open another Query windows as admin user and execute

```
SELECT s.*
FROM
sys.dm_pdw_exec_requests r
JOIN
Sys.dm_pdw_request_steps s
ON r.request_id = s.request_id
WHERE r.[label] = 'STG_text_load'
```

You'll probably notice that the operation is in the HadoopRoundRobinOperation operation_type for quite a while. This is the operation the reads the files from the data source into SQL DW. Notice that this step has a DMS location_type. DMS stands for Data Movement Service. This component of the DW architecture is responsible for data movement between nodes and data movement from outside sources into SQL DW.

We can see more detailed information about what is happening in this step with the following.

```
SELECT ew.*
FROM[sys].[dm_pdw_dms_external_work] ew
```

```
JOIN sys.dm_pdw_exec_requests r
ON r.request_id = ew.request_id
JOIN Sys.dm_pdw_request_steps s
ON r.request_id = s.request_id
WHERE r.[label] = 'STG_text_load'
ORDER BY input_name, read_location
```

This is particularly interesting because it shows how we effectively load uncompressed Delimited Text files in parallel. For large files >512 MB, we split the file into multiple chunks so it can be read more efficiently.

Parquet

For Parquet, let's kick off the code and then talk about what's happening while the data loads.

Execute the following statements as **admin user (sqladmin)**

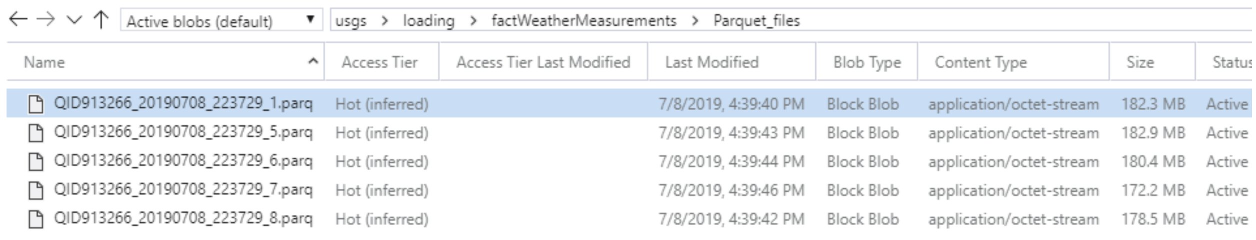
```
CREATE EXTERNAL FILE FORMAT [Parquet] WITH (FORMAT_TYPE = PARQUET)
GO

CREATE EXTERNAL TABLE [staging].[STG_factWeatherMeasurements_parquet]
(
       [SiteName] [varchar](52) NOT NULL,
       [ReadinType] [varchar](52) NOT NULL,
       [ReadingTypeID] [real] NOT NULL,
       [ReadingValue] [real] NOT NULL,
       [ReadingTimestamp] [datetime2](7) NOT NULL,
       [ReadingUnit] [varchar](52) NOT NULL,
       [fpscode] [int] NOT NULL
)
WITH (
DATA_SOURCE = [Ready_store],
LOCATION = N'loading/factWeatherMeasurements/Parquet_files',
FILE_FORMAT = [Parquet],
REJECT_TYPE = VALUE,
REJECT_VALUE = 0)
GO
```

Execute the following CTAS as **load user** (**usgsLoader**)

```
CREATE TABLE [staging].[STG_parquet_load]
WITH
(
DISTRIBUTION = ROUND_ROBIN,
HEAP
)
AS
SELECT *
FROM [staging].[STG_factWeatherMeasurements_parquet]
OPTION (label = 'STG_parquet_load')
GO
```

The first thing that we did above was create a new External File Format named Parquet. Notice that this didn't require a lot of information because the Parquet file format is predefined unlike delimited text. This helps keep data consistent when it moves around an open big data ecosystem.

If we look at the files on the storage layer, we can see that the data size is much smaller than delimited text files. This is because Parquet is a columnar format and has compression built into the file format.



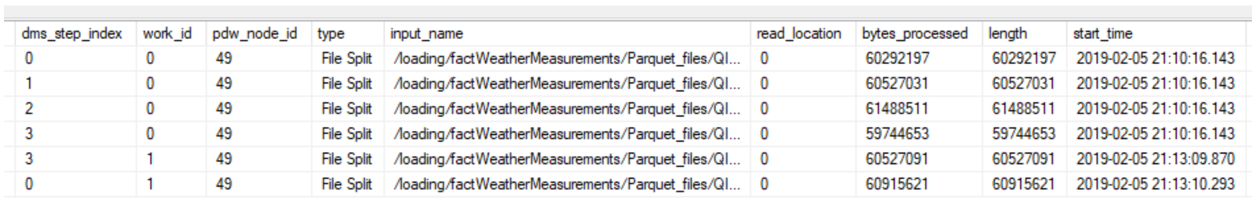
Active blobs (default) | usgs > loading > factWeatherMeasurements > Parquet_files

Name	Access Tier	Access Tier Last Modified	Last Modified	Blob Type	Content Type	Size	Statu
QID913266_20190708_223729_1.parq	Hot (inferred)		7/8/2019, 4:39:40 PM	Block Blob	application/octet-stream	182.3 MB	Active
QID913266_20190708_223729_5.parq	Hot (inferred)		7/8/2019, 4:39:43 PM	Block Blob	application/octet-stream	182.9 MB	Active
QID913266_20190708_223729_6.parq	Hot (inferred)		7/8/2019, 4:39:44 PM	Block Blob	application/octet-stream	180.4 MB	Active
QID913266_20190708_223729_7.parq	Hot (inferred)		7/8/2019, 4:39:46 PM	Block Blob	application/octet-stream	172.2 MB	Active
QID913266_20190708_223729_8.parq	Hot (inferred)		7/8/2019, 4:39:42 PM	Block Blob	application/octet-stream	178.5 MB	Active

Let's take a look at the DMVs again to see what happens with Parquet load.

```
SELECT ew.*
FROM[sys].[dm_pdw_dms_external_work] ew
JOIN sys.dm_pdw_exec_requests r
ON r.request_id = ew.request_id
WHERE r.[label] = 'STG_parquet_load'
ORDER BY  start_time ASC, dms_step_index
```

Parquet is not split-able today in SQL DW. This is shown by each file only appearing once.
If there are more files than available external readers, you will see work_id values >0. This means that the reader read one file through to its entirety then went back and started reading the next one.

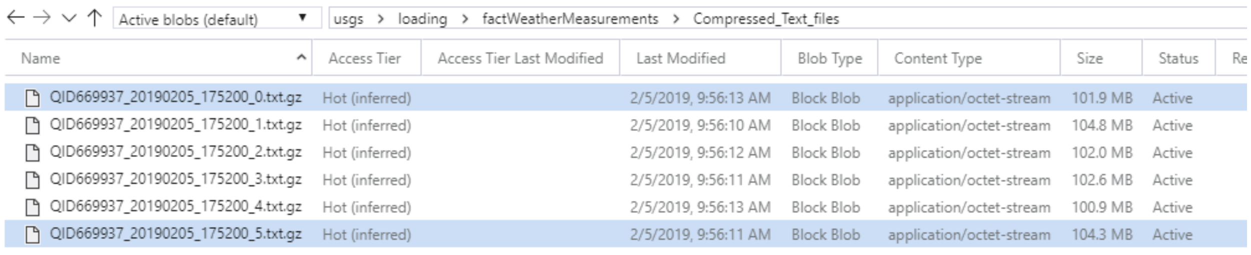


dms_step_index	work_id	pdw_node_id	type	input_name	read_location	bytes_processed	length	start_time
0	0	49	File Split	/loading/factWeatherMeasurements/Parquet_files/QI...	0	60292197	60292197	2019-02-05 21:10:16.143
1	0	49	File Split	/loading/factWeatherMeasurements/Parquet_files/QI...	0	60527031	60527031	2019-02-05 21:10:16.143
2	0	49	File Split	/loading/factWeatherMeasurements/Parquet_files/QI...	0	61488511	61488511	2019-02-05 21:10:16.143
3	0	49	File Split	/loading/factWeatherMeasurements/Parquet_files/QI...	0	59744653	59744653	2019-02-05 21:10:16.143
3	1	49	File Split	/loading/factWeatherMeasurements/Parquet_files/QI...	0	60527091	60527091	2019-02-05 21:13:09.870
0	1	49	File Split	/loading/factWeatherMeasurements/Parquet_files/QI...	0	60915621	60915621	2019-02-05 21:13:10.293

GZIP Compressed Delimited Text

Again, let's go ahead and execute the following statements before discussing:

```
CREATE EXTERNAL FILE FORMAT compressed_TextFileFormat_Ready
WITH
(
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS
    (
        FIELD_TERMINATOR = '|'
    ), DATA_COMPRESSION = N'org.apache.hadoop.io.compress.GzipCodec'
)
GO

CREATE EXTERNAL TABLE [staging].[STG_factWeatherMeasurements_CompressedText]
(
       [SiteName] [varchar](52) NOT NULL,
       [ReadinType] [varchar](52) NOT NULL,
       [ReadingTypeID] [real] NOT NULL,
        [ReadingValue] [real] NOT NULL,
        [ReadingTimestamp] [datetime2](7) NOT NULL,
        [ReadingUnit] [varchar](52) NOT NULL,
        [fpscode] [int] NOT NULL
)
WITH
(
DATA_SOURCE = [Ready_store],
LOCATION = N'loading/factWeatherMeasurements/Compressed_Text_files',
FILE_FORMAT = [compressed_TextFileFormat_Ready],
REJECT_TYPE = VALUE,
REJECT_VALUE = 0
)
GO

CREATE TABLE [staging].[STG_compressed_text_load]
WITH
(
DISTRIBUTION = ROUND_ROBIN,
HEAP
)
AS
SELECT *
FROM [staging].[STG_factWeatherMeasurements_CompressedText]
OPTION (label = 'STG_compressed_text_load')
GO
```


Active blobs (default) | usgs > loading > factWeatherMeasurements > Compressed_Text_files

Name	Access Tier	Access Tier Last Modified	Last Modified	Blob Type	Content Type	Size	Status	Re
QID669937_20190205_175200_0.txt.gz	Hot (inferred)		2/5/2019, 9:56:13 AM	Block Blob	application/octet-stream	101.9 MB	Active	
QID669937_20190205_175200_1.txt.gz	Hot (inferred)		2/5/2019, 9:56:10 AM	Block Blob	application/octet-stream	104.8 MB	Active	
QID669937_20190205_175200_2.txt.gz	Hot (inferred)		2/5/2019, 9:56:12 AM	Block Blob	application/octet-stream	102.0 MB	Active	
QID669937_20190205_175200_3.txt.gz	Hot (inferred)		2/5/2019, 9:56:11 AM	Block Blob	application/octet-stream	102.6 MB	Active	
QID669937_20190205_175200_4.txt.gz	Hot (inferred)		2/5/2019, 9:56:13 AM	Block Blob	application/octet-stream	100.9 MB	Active	
QID669937_20190205_175200_5.txt.gz	Hot (inferred)		2/5/2019, 9:56:11 AM	Block Blob	application/octet-stream	104.3 MB	Active	

Again, you will notice that SQL DW does not split GZIP compressed files. This is because the compression algorithm itself is not split-able. More on this problem later!

Let's take a moment and compare performance of the three file formats that we used. (NOTE: Wait until the CTAS is completed before you execute the statement below)

```
SELECT AVG(total_elapsed_time) AS [avg_loadTime_ms], [label]
FROM sys.dm_pdw_exec_requests
WHERE [label] IS NOT NULL
AND [label] <> 'System'
AND Status = 'Completed'
GROUP BY [label]
```

avg_loadTime_ms	label
59734	STG_compressed_text_load
95579	STG_parquet_load
71561	STG_text_load

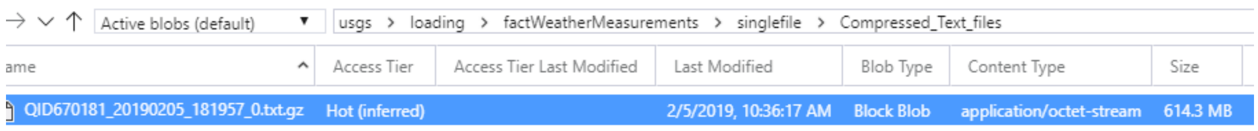
It's clear that for this particular table and data volume that compression helped. Gzip compression was the fastest, then uncompressed text files, and finally parquet.

While you may be tempted to make all loads GZIP compressed we need to look at the edge cases that appear. Both Gzip and Parquet files are not splittable in SQL DW. If there are not a enough files to utilize all the file readers or if the files are not all roughly the same size load performance will severely impact. Additionally, we have found that for wider tables with lots of strings, the CPU overhead of converting the strings slows down the overall performance.

Impact of Single File Compression

Let's take the same data and try to load it from a single compressed Gzip file.

```
CREATE EXTERNAL TABLE [staging].[STG_factWeatherMeasurements_CompressedText_single_file]
(
       [SiteName] [varchar](52) NOT NULL,
       [ReadinType] [varchar](52) NOT NULL,
       [ReadingTypeID] [real] NOT NULL,
       [ReadingValue] [real] NOT NULL,
       [ReadingTimestamp] [datetime2](7) NOT NULL,
       [ReadingUnit] [varchar](52) NOT NULL,
       [fpscode] [int] NOT NULL
)
WITH
(
DATA_SOURCE = [Ready_store],
LOCATION = N'loading/factWeatherMeasurements/singlefile/Compressed_Text_files',
FILE_FORMAT = [compressed_TextFileFormat_Ready],
REJECT_TYPE = VALUE,
REJECT_VALUE = 0)
GO


CREATE TABLE [staging].[STG_CompressedText_single_file]
WITH
(
DISTRIBUTION = ROUND_ROBIN, HEAP
)
AS
SELECT *
FROM [staging].[STG_factWeatherMeasurements_CompressedText_single_file]
OPTION(label = 'STG_single_compressed_load')
GO
```



Active blobs (default) | usgs > loading > factWeatherMeasurements > singlefile > Compressed_Text_files

ame	Access Tier	Access Tier Last Modified	Last Modified	Blob Type	Content Type	Size
QID670181_20190205_181957_0.txt.gz	Hot (inferred)		2/5/2019, 10:36:17 AM	Block Blob	application/octet-stream	614.3 MB

You've probably notice that this is taking quite a bit a time. While the load is running, we would like for you to think about the following questions and discuss with the person next to you.

Challenge Questions:

Using what you have learned above, how would you explain to a customer the behavior of SQL DW on this load?

Would scaling the DW to DWU30000c increase load performance in this case? Why or Why not?

What would you do if a customer loaded a 150 GB compressed file to Azure Storage?

How long would that 150 GB load take?

Impact of Table Distribution on loading

So far, we have run Create Table as Select (CTAS) statements with Round_Robin distributions and Heap indexes. This is the most performant index and distribution that you can use to load data into SQL DW. Fixing file format issues as describe above or scaling the DW are the only ways to improve performance. However, there are ways to *degrade* performance off this standard.

The most common problem is data skew. This occurs when the cardinality of the distribution column is low or skewed to a certain set of values. The canonical case is a key that has a lot of NULL or 0 values.

Let's suppose that we wanted to load that same data again but hash the data on the readingUnit column

```
CREATE TABLE [staging].[STG_Hash_ReadingUnit]
WITH
(
DISTRIBUTION = HASH(ReadingUnit),
HEAP
)
AS
SELECT *
FROM [staging].[STG_factWeatherMeasurements_CompressedText]
OPTION (label = 'STG_Hash_ReadingUnit')
```

While this is running, run the following in another query window:

```
Select *
FROM sys.dm_pdw_dms_workers dw
JOIN sys.dm_pdw_exec_requests r
ON r.request_id = dw.request_id
WHERE r.[label] = 'STG_Hash_ReadingUnit'
```

This shows how the data is being read from files, how that data is being converted to SQL data type format and how it is being written to SQL DW distributions. The steps are the following:

- EXTERNAL_READER = Each line is an individual thread reading data from files
- HASH_CONVERTER = Thread responsible for converting the data generated by the EXTERNAL_READER
- WRITER = Thread responsible for writing to the correct distribution based on sink table definition.

Notice how many writers are sitting around with NULL (or 0 if query is completed) bytes_processed. This is because of the table skew! Table skew occurs when certain distributions receive more data than others. Distributions receive data based on the hash value in the table definition.

If we did a simple check on how many rows per readingUnit, we would see that there are only 17 unique readingUnit values. When writing, these 17 distributions receive all of the load causing a bottleneck. Now, some skew is natural, but this is extreme! The table would be better suited to a ROUND_ROBIN distribution.

RowCount	readingUnit
6658050	T
47216	D
11394	A
134061	B
10871562	7
36655	M
9902661	W
75345491	0
16481	6
126562	S
615070	H
13002585	N
9373275	U
755951	K
91195	Z
293697	R
7051605	X

Let’s take a look again at the DMVs and compare our load speeds

Execute the following query as admin user (wait until the previous load is completed):

```
Select avg(total_elapsed_time) as [avg_loadTime_ms], [label]
FROM sys.dm_pdw_exec_requests
where [label] is not null
and [label] <> 'System'
and Status = 'Completed'
GROUP BY [label] order by 1 desc
```

avg_loadTime_ms	label
437907	STG_Hash_ReadingUnit
95579	STG_parquet_load
71561	STG_text_load
60742	STG_compressed_text_load
20593	STG_single_compressed_load

As you can see, table structure had an order of magnitude impact on my loading performance!

Impact of CTAS vs Insert into Select

Depending on your ELT pattern, you can either use a CTAS to create a brand-new table or an INSERT INTO SELECT from external table to append records to an existing table. Both methods have their purposes, but the two syntaxes are not functionally equivalent.

Let's take a look and see the performance differences. Since we have already done several CTAS statements lets focus on INSERT INTO SELECT and compare.

```
CREATE TABLE [staging].[STG_factWeatherMeasurements_Insert_Into]
(
	[SiteName] [varchar](52) NOT NULL,
	[ReadinType] [varchar](52) NOT NULL,
	[ReadingTypeID] [real] NOT NULL,
	[ReadingValue] [real] NOT NULL,
	[ReadingTimestamp] [datetime2](7) NOT NULL,
	[ReadingUnit] [varchar](52) NOT NULL,
	[fpscode] [int] NOT NULL
)
WITH
(
	DISTRIBUTION = ROUND_ROBIN,
	HEAP
)
GO

INSERT INTO [staging].[STG_factWeatherMeasurements_Insert_Into] SELECT  * FROM
[staging].[STG_factWeatherMeasurements_text] option (label = 'STG_insertInto')
GO
```

Wait until the above inserts are completed and execute the following query:

```
SELECT avg(total_elapsed_time) as [avg_loadTime_ms], [label]
FROM sys.dm_pdw_exec_requests
WHERE [label] is not null
and [label] in( 'STG_insertInto' ,'STG_text_load')
and Status = 'Completed'
GROUP BY [label] ORDER BY 1 DESC
```

As you can see CTAS is slightly faster (~10%) than writing directly into a table via insert into select. CTAS is a fully parallelize, minimally logged operation while Insert INTO has normal logging enabled. This functional difference accounts for the difference in load speeds.

Closing

You've reached the end of the DW loading lab. Hopefully, you are now comfortable with the concepts behind loading data into SQL DW and better understand some of the levers you can utilize to optimize your data loads

Side Bar: Simplified loading experience with COPY Command (Public Preview Now)

There has been substantial feedback from customers and the field that loading data into SQL DW is too difficult with the syntax above. In the near future, the service will have a new loading syntax (in additional to External Tables) COPY INTO. The COPY INTO functionality will provide a single statement to define data location, file format, authentication and destination table. In addition to changing the syntax, the copy command will provide capabilities to handle other common issues with loading delimited text files today including: NullValue, CarriageReturns in strings, etc.

```
CREATE TABLE [staging].[STG_factWeatherMeasurements_Copy_Into]
(
	[SiteName] [varchar](52) NOT NULL,
	[ReadinType] [varchar](52) NOT NULL,
	[ReadingTypeID] [real] NOT NULL,
	[ReadingValue] [real] NOT NULL,
	[ReadingTimestamp] [datetime2](7) NOT NULL,
	[ReadingUnit] [varchar](52) NOT NULL,
	[fpscode] [varchar](52) NOT NULL
)
WITH
(
	DISTRIBUTION = ROUND_ROBIN,
	HEAP
)
GO


COPY INTO [staging].[STG_factWeatherMeasurements_Copy_Into]
FROM ' https://<your storage
account>.blob.core.windows.net/usgs/loading/factWeatherMeasurements/Text_files/*.txt'
WITH
(
	FILE_TYPE = 'CSV',
	FIELDTERMINATOR = '|',
	ROWTERMINATOR = '0x0A',
	CREDENTIAL = (IDENTITY = 'Storage Account Key', Secret = '<your account key>')
);
```