CSE Customer Success Engineering

Apache Spark for Azure Synapse Guidance

This document outlines best practices guidance for developing Spark applications with Azure Synapse Analytics. It is composed of four sections:

- Reading Data – reading data into Spark
- Writing Data – writing data out of Spark
- Developing Code – developing optimized Spark code
- Production Readiness – best practices for scalability, reproducibility and monitoring

Reading Data

Whether you are reading in data from an ADLS Gen2 data lake, an Azure Synapse Dedicated SQL pool, or other databases in Azure there are several important steps to take to optimize reading data into Apache Spark for Synapse.

Fast Connectors

Typically for reading data, ODBC or JDBC connectors are used which read data in serially. Microsoft has developed connectors to greatly improve read performance by reading in parallel. This is especially recommended when reading large datasets from Synapse SQL where JDBC would force all the data to be read from the Synapse Control node to the Spark driver and negatively impact Synapse SQL performance. When possible, use these connectors: Synapse SQL, Cosmos DB, Synapse Link, Azure SQL/SQL Server.

File Types

Spark can read various file types including but not limited to Parquet, CSV, JSON and Text Files. More information on the supported file types available can be found here.

The recommended file type to use when working with big data is Parquet. Parquet is an open-source columnar file format that provides several optimizations compared to rowstore formats including:

- Compression with algorithms like snappy and gzip to reduce file size.
- Predicate pushdown which leverages metadata to skip reading non-relevant data.
- Support for file partitioning to limit the data that needs to be scanned.
- Better performance when compared to other file formats.

Delta Lake is an open-source storage layer that builds on top of Parquet to provide the following additional capabilities:

- ACID transactions and support for DML commands (SQL Data Manipulation Language which includes INSERT, UPDATE, DELETE, MERGE).
- Time travel and audit history via the transaction log to allow you to revert to earlier versions and have a full audit trail.
- Schema enforcement and schema evolution.

File Sizes

In addition to recommend file types, there are recommended file sizes. When writing files, it is best to keep file sizes between 1GB and 2GB for optimal performance. This can be achieved by utilizing the writing strategies discussed later in the document. Attaining this file size depends on the overall size of the dataset, in some cases this may not be achievable.

Pre-defined Schema vs. Inferred Schema

When reading data from a source, Spark can infer the schema of the incoming data automatically. This is not guaranteed to have a 100% match to the source datatypes and the datatypes used in the dataframe, potentially causing issues for downstream systems. While this will work for small sets of data, this can lead to performance issues when reading large sets of data. To improve performance, it is always best to define the schema prior to reading the data. If the data source will be re-used frequently it will be best to save the schema for reuse.

Here are examples of an inferred schema vs. pre-defined schema:

Inferred Schema:

```
df = spark.read.parquet(/data/dataset.parquet)
```

Pre-Defined Schema:

```
from pyspark.sql.types import *

schema = StructType(

        [

            StructField("name", StringType(), True),

            StructField("age", IntegerType(), True),

            StructField("paid_in_full", BooleanType(), True),

            StructField("birthday", DateType(), True),

        ]
```

```
      )

df = spark.read.schema(schema).parquet(/data/dataset.parquet)
```

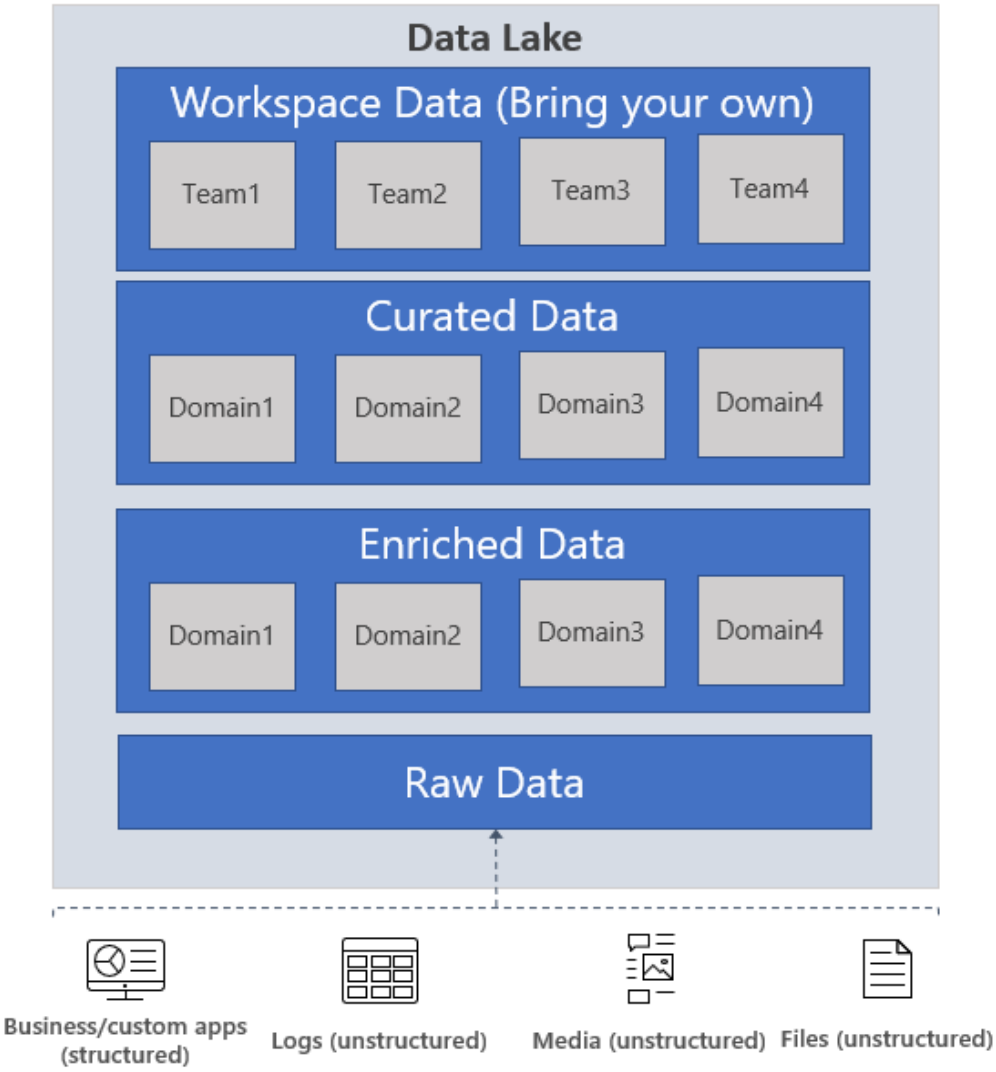
Utilize ADLS Gen2 Storage

When storing your data to be utilized with Spark always use ADLS Gen2 Storage (hierarchical namespace enabled) and **NOT** Blob storage. ADLS Gen2 storage utilizes an optimized driver that is specifically built for big data analytics. In addition, ADLS Gen2 allows access to stored data via the same methods as accessing a Hadoop Distributed File System (HDFS).

Data Lake Organization

There are several ways to organize your data lake. Organization should be based on the business requirements of the organization. A common pattern is to segment your data into different zones due to the lifecycle of the data.

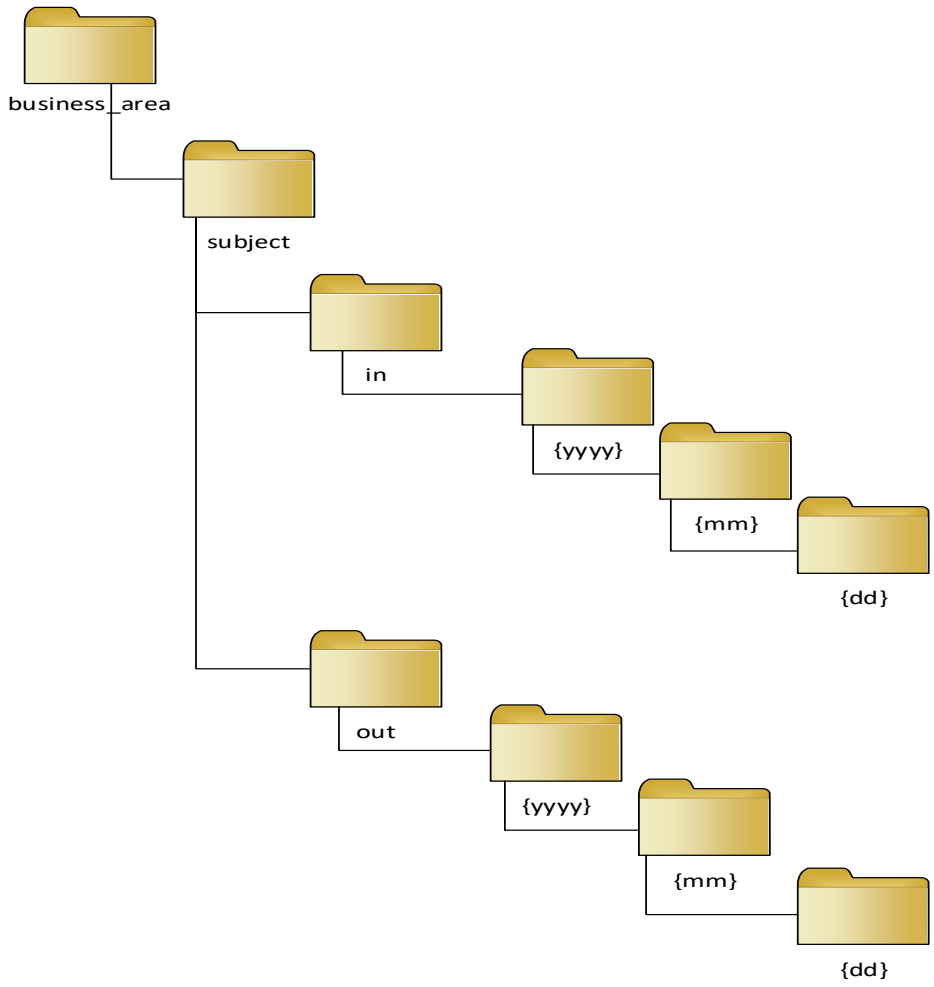
- **Raw Data:** This is data that is inserted directly from the source systems in its original format.
- **Enriched Data:** The zone contains data received from the "Raw" zone that has been cleaned of all erroneous data, modified, and enriched (with other sources) to be used by downstream systems.
- **Curated Data:** This zone is where the data has been aggregated and is ready to be served to users via reporting services, dedicated API or other down-stream systems. In some scenarios, Curated Data might be separated into Clear and Hashed sub-areas applying necessary masking in the latter for general or common user groups.
- **Workspace Data:** This is a zone that contains data that was ingested by each individual team/data consumers (data scientists, business analysts and others) that will be used in conjunction to the data initially ingested by the data engineering to provide greater value to their specific teams.



Perform Partition Elimination by Partitioning Folders

With distributed data, when storing data within each data lake zone, it is recommended to use a partitioned folder structure. This technique helps us improve data manageability and query performance. Partitioned data is a folder structure that enables us faster search for specific data entries by partition pruning/elimination when querying the data. Using this technique we avoid extensive listing of file metadata. For example, utilizing the folder structure below, partitioning by day, you can run a spark query to limit the amount of data that is searched.

Folder Partitioning Structure



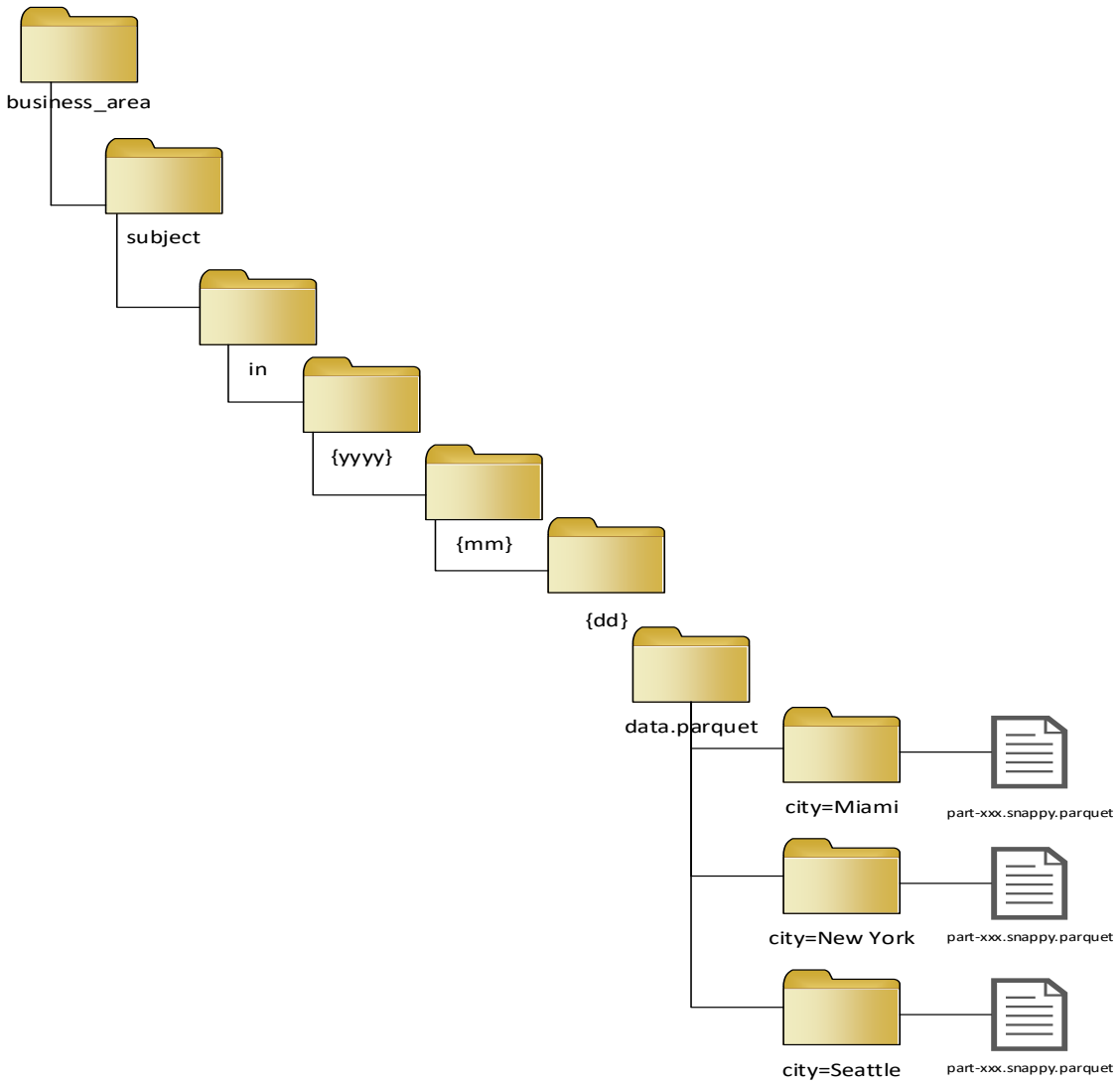
Read Data in Partitioned Folder

```
df = spark.read.parquet("/businessArea/subject/in/2021/03/01/data.parquet")
```

Partition Elimination by Partitioning Within Files

In addition to configuring a folder structure to partition your data at a folder level, you should also partition the data within the file. This limits the amount of data needed to be scan within a file when reading/querying the file. For example, look at the folder structure below, whereby partitioning data by city, you can run a spark query on specific city, or list of cities and the partition technique would help us avoid full scan of the data, that is both expensive and inefficient. The partition technique effectively speeds up your overall operation.

File Partitioning Structure



Reading A Partitioned File

Below are two examples of how to read data that has been partitioned at the file level. The first example shows how to read the partitioned data of the file directly from storage. The second example reads in the entire file. Once read, a tmp table is created and a "WHERE" statement is executed on the partitioned column. Example 1 shows how the file partitioning is done on the initial read from storage. Example 2 the file partition is occurring as a result of the WHERE clauses in the spark SQL statement. Both accomplish the same result.

Example 1:

```
df = spark.read.parquet("/businessArea/subject/out/2021/03/01/data.parquet/ci

ty=Miami")
```

Example 2:

```
df = spark.read.parquet("/businessArea/subject/out/2021/03/01/data.parquet")

df.createOrReplaceTempView("DataTable")

city = spark.sql("SELECT * FROM DataTable WHERE city = 'Miami'")
```

Writing Data

Just as with reading files, there are several optimization techniques to improve performance when writing data out to ADLS Gen 2 storage or Azure databases.

Writing Partitions

When writing files, you should also utilize partitioning. This will improve performance on the write. This also provides benefits for the future read operations to utilize specific partitions, as described prior, to improve performance. Here is an example of how to partition your writes based on a specific table column, refer to file partitioning structure above:

```
df.write.partitionBy("city").parquet("/businessArea/subject/out/2021/03/01/data.parquet")
```

* When utilizing partitioning with Delta Lake files, ensure that the column has at least 1GB of data to be partitioned. See Delta Partitioning for more information.

Avoid Writing Too Many/Too Few Files

When writing files to storage it is good to ensure that there are not too many files written. A rule of thumb is for your files to be 1-2GB in size. One task equates to one file being written out. With many tasks there will be many files generated with small data size. When utilizing any cloud storage system, such as ADLS Gen2, having many small files will cause high I/O on the storage system resulting in throttling. To resolve this, use coalesce to reduce your partitions to a smaller size. This will allow for files written to be larger.

In some cases, you may want to utilize a larger number of partitions to execute the job quicker. This will increase parallelism by utilizing all the executors in your cluster. As a result, you will create more files of a smaller size as described prior. A rule of thumb is for you to have 3-4x as many files as the number of cores in your pool. After a period, you should run a compaction process to merge the files into larger ones. When working with Delta, you will see many small files created. You can find an example of the compaction process here.

Another way to increase performance of smaller files is to implement partitionBy() as mentioned prior. This would require you to know the dataset however it will allow your files to be structured in a larger format depending on the partition chosen.

When working with partitions utilize the function `df.rdd.getNumPartitions` to view the number of partitions currently being occupied by data.

Fast Connectors

Just as with reading data, writing data to database systems it is best to utilize connectors that allow parallel writes. Your generic ODBC or JDBC connectors read data in serial manner, resulting in slow writes. When possible, use these connectors to optimize your writes to their respective locations: Synapse SQL, Cosmos DB, Synapse Link, Azure SQL/SQL Server.

Only Write What You Need

When writing data ensure you are only writing data that is needed. Writing more data than needed will increase execution time as well as increase data to be read by downstream systems.

Developing Code

Once you have optimized reading and writing data with Spark, it is useful to follow these guidelines for the code that makes up the core logic of your Spark application.

Use Dataframes/Datasets over RDDs

When working with data in Spark, always use Dataframes or Datasets over RDDs. Just as with RDDs, Dataframes are immutable. However, Dataframes and datasets organizes data in a columnar format. This gives a structure to the data, ultimately allowing for high level abstraction. The Dataframe also utilizes the Catalyst Optimizer improving performance of your Spark operations.

Avoid UDFs

Conventional UDFs operate serially one by one. It is best to implement needed functionality with built-in functions (i.e. spark.sql.functions). If UDFs must be used utilize them in this order:

Built-in Functions > Scala/Java UDFs > Pandas UDFs > Python UDFs

Both Scala UDFs and Pandas UDFs are vectorized. This allows computations to operate over a set of data.

Turn on Adaptive Query Execution (AQE)

Adaptive Query Execution (AQE), introduced in Spark 3.0, allows for Spark to re-optimize the query plan during execution. This allows for optimizations with joins, shuffling, and partition

sizing. By default, this functionality is turned off. To turn this on set the following spark config to "true":

```
spark.conf.set("spark.sql.adapative.enabled", true)
```

Increase Broadcast Hash Join Size

Broadcast Hash Join is the fastest join operation when completing SQL operations in Spark. This operation copies the dataframe/dataset to each executor when the `spark.sql.autoBroadcastJoinThreshold` is greater than the size of the dataframe/dataset. The default threshold size is 25MB in Synapse. To improve performance increase threshold to 100MB by setting the following spark configuration.

```
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 100*1024*1024)
```

When increasing the broadcast threshold monitor the following:

- Dataframe is not bigger than the driver/executor available working memory
- Dataframe is not bigger than `spark.driver.maxResultSize`

Cache Repetitive Work

When performing operations in Spark you may find that your SQL plan is reusing datasets to complete the operations. When this occurs, `df.cache()` or `df.persist(level)` the dataframe to memory. This will allow processes to reuse the data instead of recomputing it each time. Both operations perform the same function, persist provides the ability to choose the memory level at which to store the information. Be mindful of caching as caching too frequently or caching large datasets will consume more memory and can lead to disk spillage. Always run df.unpersist() after caching to remove data from cache.

Avoid Expensive Operations

There are some operations that need be avoided if possible due to the cost of performance when executing. These are:

- collect() – brings all data to the driver, possibly causing out-of-memory errors.
- count() – this forces Spark to iterate one by one over the entire dataset.
- distinctCount() – this gives the count of all distinct items in the dataset. Instead, use approxDistinctCount(). This will provide the approximate distinct count with a 5% margin of error. If distinct is needed, perform dropDuplicates() on data that is in the correct sorted order and before you join or group by.
- repartition() – this is very useful when looking to resolve data skew or when needing to increase the number of partitions. However, use with caution, executing this command

will force a full shuffle of data. Utilizing coalesce() instead will shrink the number of partitions without forcing a full shuffle.
groupByKey() – this is used to combine data on each partition before sending over the network. To accomplish the same with a less expensive operation, use reduceByKey() or combineKey()/aggegateKey().

Utilize Koalas instead of Pandas

Pandas operate on a single executor and does not utilize the full parallelism of Spark. To resolve this and still have functionality like Pandas, use Koalas. Koalas is an API that allows you to simplify exploratory data analysis by applying Pandas dataframe functions over distributed Spark dataframes. Kolas 1.0 has 80% coverage of Pandas functions.

There may be certain scenarios where a user needs to convert a Spark Dataframe to a Pandas Dataframe such as ML scenarios that are not supported in Spark ML/MLlib. For these scenarios, Apache Arrow is enabled by default in Synapse to optimize this conversion.

Implement Hyperspace Indexes

Hyperspace Indexes provide the ability to implement indexes on various datasets including CSV, JSON, Parquet, and Delta Lake to enable data skipping. This indexing method can accelerate queries up by 11x. Once indexes are implemented, they will be leveraged automatically without any updates to your current code. Hyperspace indexes are most useful under the following conditions:

- There are queries that have filters on predicates that are highly selective. i.e. Matching 100 rows from a pool of 100 million rows.
- There is a heavy shuffle due to a join. i.e. Joining a 100GB dataset with a 10GB dataset.
- Delta Lake files with a large number of transaction files (similar to Databricks Z-Order)

Production Readiness

How to Size Clusters

When working with Apache Spark for Azure Synapse there are five different Spark pool sizes that can be used.

Size	vCore	Memory
Small	4	32 GB
Medium	8	64 GB
Large	16	128 GB
XLarge	32	256 GB
XXLarge	64	512 GB
XXXLarge (Isolated Compute)	80	504 GB

Isolated Compute

Isolated Compute gives added security to your Spark environment by dedicating physical compute resources to a single user. This option is ideal for workloads that require greater isolation. Isolated Compute is only offered with the XXXLarge compute size and in the following regions: East US, West US 2, South Central US, US Gov Arizona, US Gov Virgina.

How to Size Clusters

If you are unsure what size cluster to utilize, start with a Medium Spark pool with three nodes and the ability to scale to ten nodes. After executing your spark jobs, view the Apache Spark history server to check performance. There are a few areas to monitor.

- **Disk Spillage** – If there are jobs where data spilled to disk, consider increasing the size of your Spark instance.
- **Number of Tasks vs. Number of Executors** – Utilizing larger clusters can cause high Garbage Collection (GC). To prevent this, it is best to have less than 5 tasks executing per core when running Spark jobs. If you have exceeded this, consider scaling up your Spark instance.
- **Executor Usage –** In the Diagnostics tab, the Executor Usage Graph will show you the executor usage over time. If executor usage is very low, consider scaling down your Spark instance.

Dynamic Allocation

Dynamic allocation is an Apache spark mechanism where the VMs allocated for executers are adjusted automatically, it allows you to specify the range by providing it with a min and max of executors for your Spark job to use. Leveraging this feature provides control over how the Spark job auto-scale up and down based on the demand of the job. In addition, it prevents exhaustion of executors on the pool. Here is an example of how you can configure it directly from Apache Spark notebooks in Synapse studio:

```
%%configure -f
{
 "conf" : {
       "spark.dynamicAllocation.maxExecutors" : "4",
       "spark.dynamicAllocation.enabled": "true",
       "spark.dynamicAllocation.minExecutors": "2"
 }
}
```

With the given configuration here, your job automatically starts with 2 executors, if it requires only 2 executors, it will only use those. When the job requires more, it will scale up to 4 executors (1 driver, 4 executors) maximum. When the job does not need the executors, then it will decommission the executors and if it does not need the node, it will free up the node.

Note: The maxExecutors attribute will reserve the # of executors. For example, if you only use 2, it will reserve 4 and you won't be able to start up a second session. It's best to increase the number of nodes in your pool to compensate for the maxExecutors. When you combine both features, you can improve your performance while also saving on cost.

Library Package Management

Synapse Spark pools support both .whl and .jar packages. When choosing to retrieve the package from the external repository at the start of the cluster for .whl packages, ensure you have chosen the correct version to be used in the requirements.txt file. This prevents any issues from the package owner updating the library at any moment and adversely affects your notebook execution.

Monitoring

Configure Azure Log Analytics or Prometheus and Grafana for monitoring of your application metrics and logs. Both provide the ability to visualize the metrics and logs from your Spark Applications. Utilizing these monitoring tools will allow you to dive deep into your application's operations and debug any potential issues. This also provides an easy way to view metrics over a period. A tutorial for implementing Log Analytics can be found here and a tutorial for Prometheus and Grafana implementation can be found here.

In addition, consider utilizing custom logging for code exceptions for your Spark applications. To implement this, leverage Log4j for Scala applications and Opencensus for Python applications. Once applied, you will have the ability to track exceptions that occur from an application perspective when your Spark jobs are running.