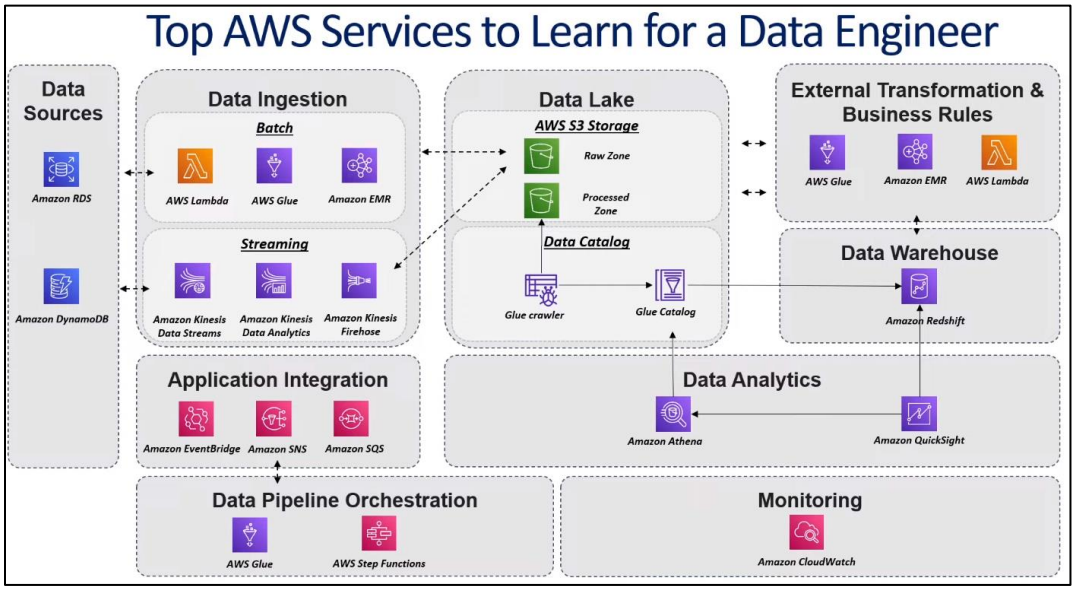
Contents

AWS for Data Engineers – ETL/ELT - pipelines

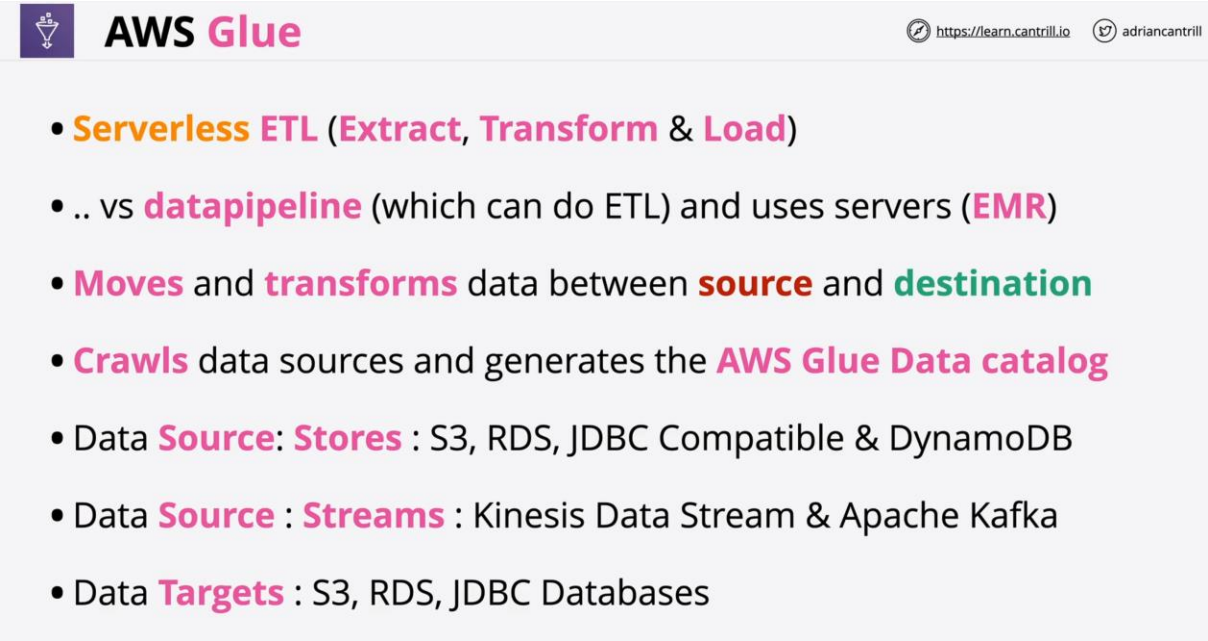


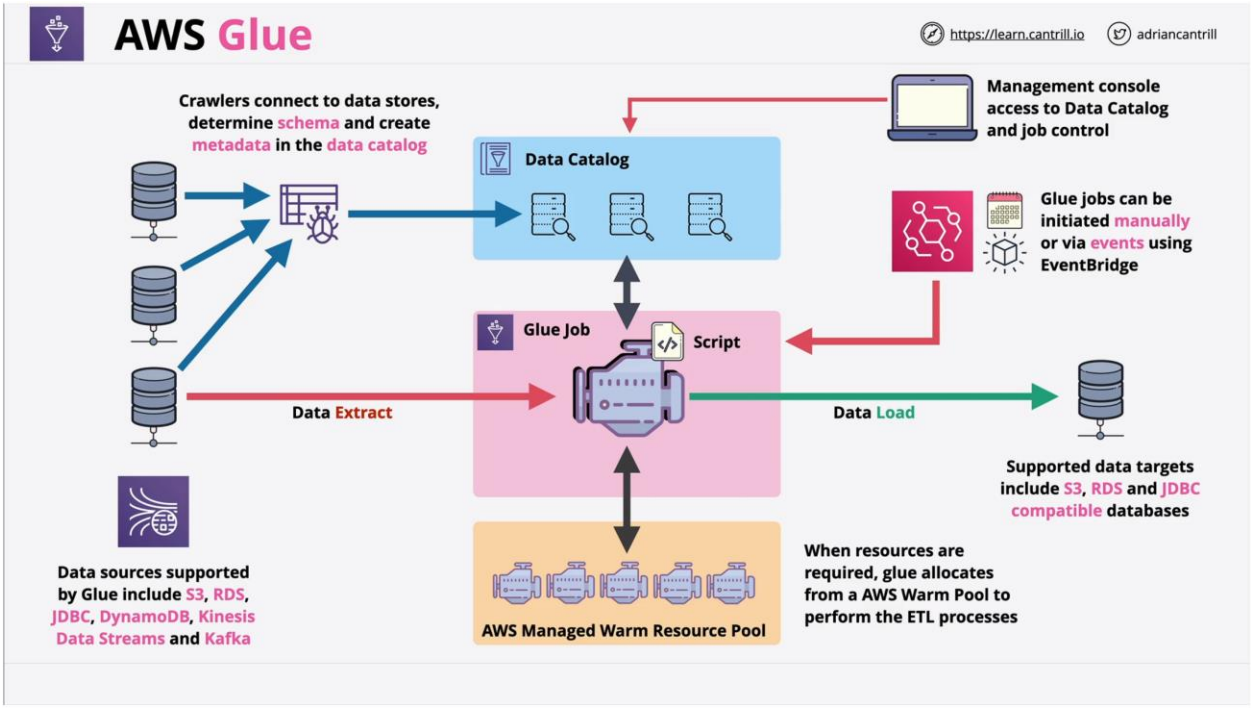
AWS Glue

All chapter about Glue and features:

https://learning.oreilly.com/library/view/serverless-etl-and/9781800564985/B16741_02_ePub.xhtml#_idParaDest-40

AWS Glue is a fully managed extract, transform, and load (ETL) service that makes it easy for customers to prepare and load their data for analytics. You can create and run an ETL job with a few clicks in the AWS Management Console. You simply point AWS Glue to your data stored on AWS, and AWS Glue discovers your data and stores the associated metadata (e.g. table definition and schema) in the AWS Glue Data Catalog. Once cataloged, your data is immediately searchable, queryable, and available for ETL.





AWS Glue - Data Catalog

https://learn.cantrill.io adriancantrill

- **Persistent metadata** about data sources in region
- **One catalog** per **region** per **account**
- **Avoids data silos** ..
- Amazon Athena, Redshift Spectrum, EMR & AWS Lake Formation all use Data Catalog
- .. configure **crawlers** for data sources

Spark

- Spark performance tuning
- shuffle spark / core spark architecture
- memory management
- brodcast spark
- spark casching
- function for imporoving Spark
- joins in Spark
- Glue sdk
- Table schema verions aws
- avro and parquet
- Datasets vs dataframe Spark
- pushdown - spark
- normal form - tabele - bazy danych and denormalization
- Redshift

Database pruning is an optimization process used to avoid reading files that do not contain the data that you are searching for. You can skip sets of partition files if your query has a filter on a particular partition column. In Apache Spark, dynamic partition pruning is a capability that combines both logical and physical optimizations to find the dimensional filter, ensures that the filter executes only once on the dimension side, and then applies the filter directly to the scan of the table which speeds up queries and prevents reading unnecessary data.

https://learning.oreilly.com/library/view/the-azure-data/9781484282335/html/522396_1_En_12_Chapter.xhtml#:-:text=Dynamic%20Partition%20Pruning

Partitioning is a database process that divides very large tables into multiple smaller parts to enable queries to run faster because they would only need to access a portion of the data. These partitioning strategies support the maintenance of large tables and improve their performance. Database table partitioning has been a mature feature of traditional SQL Server infrastructure appliances.

https://learning.oreilly.com/library/view/the-azure-data/9781484282335/html/522396_1_En_12_Chapter.xhtml#:-:text=Partitions

Strategies for designing partitions. The three typical strategies are horizontal, vertical, and functional partition designs. Horizontal partitioning, also known as “sharding,” splits partitions into separate data stores which hold subsets of the data with the same schema. With vertical partitioning, each partition contains a subset of the fields for items in the data store split by usage patterns, such as splitting frequently accessed data in one partition and less frequently accessed data in another. Functional partitioning aggregates data by how it is used in each bounded context by splitting, for example, finance and marketing data

Z-Ordering and Data Skipping - There are a few available optimization commands within Databricks that can be used to speed up queries and make them more efficient. Seeing that Z-Ordering and Data Skipping are optimization features that are available within Databricks

https://learning.oreilly.com/library/view/the-azure-data/9781484282335/html/522396_1_En_13_Chapter.xhtml#:-:text=13.%20Z-Ordering%20and%20Data%20Skipping

Z-Ordering is a method used by Apache Spark to combine related information in the same files. This is automatically used by Delta Lake on Databricks data-skipping algorithms to dramatically reduce the amount of data that needs to be read. The OPTIMIZE command can achieve this compaction on its own without Z-Ordering which compacts small files into larger ones to improve the speed of reading queries from the table. This bin-packing process will produce evenly balanced data files when compared to size on disk but may not necessarily optimize on a specific column, which could be useful for scenarios where the column is used to filter queries.

Adding Z-Ordering to the OPTIMIZE command allows us to specify the column to compact and optimize on, which will impact querying speeds if the specified column is in a Where clause and has high cardinality. Additionally, Data Skipping is an automatic feature of the optimize command and works well when combined with Z-Ordering.

https://learning.oreilly.com/library/view/the-azure-data/9781484282335/html/522396_1_En_13_Chapter.xhtml#:-:text=Z-Ordering

Data Skipping - One result from optimizing your data is that Databricks puts in practice a concept called Data Skipping. Data Skipping does not need to be configured and is applied automatically when data is written into a Delta Table and is most effective when combined with Z-Ordering.

https://learning.oreilly.com/library/view/the-azure-data/9781484282335/html/522396_1_En_13_Chapter.xhtml#:-:text=Data%20Skipping

Adaptive Query Execution - Databricks has solved this with its Adaptive Query Execution (AQE) feature that is available with Spark 3.0 and higher.

Adaptive Query Execution (AQE) is a query re-optimization process that occurs when a query is executed. Since Databricks collects the most updated statistics at the end of a query stage which includes shuffle and broadcast exchange operations, it can optimize and improve the physical strategy. In the newer versions of Databricks which includes runtime 7.3 and above, AQE is automatically enabled by default and applies to non-streaming queries which contain at least one join, aggregate, or window operation.

AQE is capable of dynamically changing sorting merge joins into broadcast hash joins, combining partitions into reasonably sized partitions after shuffle exchanges, and splitting skewed tasks into evenly sized tasks. AQE queries contain what is called an AdaptiveSparkPlan node which contains the initial and final plans. The query plans will change once AQE optimizations take effect.

The AQE framework possesses the ability to dynamically coalesce shuffle partitions, dynamically switch join strategies, and dynamically optimize skew joins.

https://learning.oreilly.com/library/view/the-azure-data/9781484282335/html/522396_1_En_14_Chapter.xhtml#:-:text=How%20It%20Works

Bloom Filter Indexes - Within the Data Lakehouse, there have been limited methods of applying indexes to Delta Tables. Bloom Filter Indexes are space-efficient data structures that enable Data Skipping on chosen columns. It operates by stating that data is definitively not in the file, or that it is probably in the file, with a defined false positive probability (FPP). Bloom Filter Indexes are critical to building highly performant Delta Lakehouses. In this chapter, we will address the question of how to get started with Bloom Filter Indexes.

A Bloom Filter Index is intended to optimize query performance and will enable Data Skipping on a column. A data file can have a single index file associated with it. An index file is an uncompressed single row parquet file stored in the _delta_index subdirectory. Without a Bloom Filter Index, Databricks will always read the entire data file; therefore, the advantage of the Bloom Filter Index that has been created is that Databricks first checks the index file and then the corresponding data file is only read if there is a potential filter match.

The false positive probability (fpp) and numItems tuning options, which will be included in the code to create the Bloom Filter Index in subsequent sections, can be defined at the table or column level. The default fpp is 0.1 and must be larger than 0 or smaller than 1. This configuration option defines the acceptable tolerance level of having to read more data than you should, which is 10% as the default value. This could have the implication of taking longer time to write new incoming data because a fairly complex tree of indices will need to be rebuilt every time. A lower fpp will use more bits per element, resulting in greater accuracy with a potential negative impact on performance and cost.

The fpp can be tuned based on your acceptance criteria to balance the cost to maintain with the possibility of reading more data that you might need to. The numItems option defines the number of distinct items a file can contain, and the default is one million items. This option depends on a number of factors including volume of data. In the following examples, we have used 5% of the total records within a table as the numItems option. The balance between the fpp and numItems must be carefully adjusted and tuned to prevent performance issues. Sometimes, this tuning process can include a number of different factors, thus resulting in an iterative trial-and-error process to find the optimal balance.

Typically, a Bloom Filter Index is applied to an empty table, but can also be used in conjunction with the Z-Order command to rebuild the Bloom Filter Index on tables that frequently have new data being inserted. While this index is enabled by default, through the following example, you will also learn how to enable or disable this filter manually, as needed. A Bloom Filter can be dropped once it is created by using the DROP command written in SQL Syntax.

https://learning.oreilly.com/library/view/the-azure-data/9781484282335/html/522396_1_En_15_Chapter.xhtml#:-:text=How%20a%20Bloom%20Filter%20Index%20Works

Hyperspace

While Spark offers tremendous value in the advanced analytics and big data spaces, there are currently a few known limitations around indexing with Spark when compared to the best-in-class SQL Server indexing systems and processes. While Spark isn't great at b-tree indexing and single record lookups, Spark partitioning attempts to address some of these indexing limitations. However, when users query the data with a different search predicate than what was partitioned, this will result in a full scan of the data along with in-memory filtering on the Spark cluster, which is quite inefficient.

Similar to a SQL Server non-clustered index, Hyperspace will create an index across a specified data frame, create a separate optimized and reorganized data store for the columns that are being indexed, and can include additional columns in the optimized and reorganized data store, much like a non-clustered SQL Server index. Since Hyperspace is open source and readily available out of the box in the form API with multi-language support, it can also be used within Databricks notebooks. In this chapter, we will focus on creating a dataset in a Synapse Analytics workspace with a Hyperspace Index added on it to compare a query using Hyperspace indexed vs. non-indexed tables to observe performance optimizations within the Lakehouse.

Schema Evolution

Since every data frame in Apache Spark contains a schema, when it is written to a Delta Lake in delta format, the schema is saved in JSON format in the transaction logs. This allows for a few neat capabilities and features such as schema validation, to ensure quality data by rejecting writes to a table that do not match the table's schema and schema evolution, which allows users to easily change a table's current schema to accommodate data that may be changing over time. It is commonly used when performing an append or overwrite operation to automatically adapt the schema to include one or more new columns. In this chapter, you will explore schema evolution capabilities and limitations with regular parquet format and explore schema evolution features and capabilities through delta format with appends and overwrites. If you are interested in following along with this example, simply create a new Python Databricks notebook and ensure that your ADLS gen2 account is mounted and that you have a running cluster.

https://learning.oreilly.com/library/view/the-azure-data/9781484282335/html/522396_1_En_7_Chapter.xhtml#:-:text=Schema%20Evolution

Row-based versus columnar storage

Databases physically store data in one of two ways, either in a row-based manner or in a columnar fashion. Each has its own advantages and disadvantages, depending on its use case. In row-based storage, all the values are stored together, and in columnar storage, all the values of a column are stored contiguously on a physical storage medium, as shown in the following screenshot:

Row-based Storage

uid	last_name	first_name	birth_month
1	Asimov	Isaac	January
2	Brown	Dan	June
3	Burroughs	Edgar Rice	September

Columnar Storage

uid	1	2	3
last_name	Asimov	Isaac	January
first_name	Brown	Dan	June
birth_month	Burroughs	Edgar Rice	September

Figure 12.1 – Row-based versus columnar storage

As depicted in the previous screenshot, in row-based storage, an entire row with all its column values is stored together on physical storage. This makes it easier to find an individual row and retrieve all its columns from storage in a fast and efficient manner. Columnar storage, on the other hand, stores all the values of an individual column contiguously on physical storage, which makes retrieving an individual column fast and efficient.

Row-based storage is more popular with transactional systems, where quickly retrieving an individual transactional record or a row is more important. On the other hand, analytical systems typically deal with aggregates of rows and only need to retrieve a few columns per query. Thus, it is more efficient to choose columnar storage while designing analytical systems. Columnar storage also offers a better data compression ratio, thus making optimal use of available storage space when storing huge amounts of historical data. Analytical storage systems including **data warehouses** and **data lakes** prefer columnar storage over row-based storage. Popular big data file formats such as **Parquet** and **Optimized Row Columnar** (**ORC**) are also columnar.

The ease of use and ubiquity of SQL has led the creators of many non-relational data processing frameworks such as Hadoop and Apache Spark to adopt subsets or variations of SQL in creating Hadoop Hive and Spark SQL. We will explore Spark SQL in detail in the following section.

https://learning.oreilly.com/library/view/essential-pyspark-for/9781800568877/B16736_12_Final_JM_ePub.xhtml#:-:text=Row-based%20versus%20columnar%20storage

Catalyst optimizer –

A **SQL query optimizer** in an RDBMS is a process that determines the most efficient way for a given SQL query to process data stored in a database. The SQL optimizer tries to generate the most optimal execution for a given SQL query. The optimizer typically generates multiple query execution plans and chooses the optimal one among them. It typically takes into consideration factors such as the **central processing unit** (**CPU**), **input/output** (**I/O**), and any available statistics on the tables being queried to choose the most optimal query execution plan. The optimizer based on the chosen query execution plan chooses to re-order, merge, and process a query in any order that yields the optimal results.

The Spark SQL engine also comes equipped with a query optimizer named **Catalyst**. **Catalyst** is based on **functional programming** concepts, like the rest of Spark's code base, and uses Scala's programming language features to build a robust and extensible query optimizer. Spark's Catalyst optimizer generates an optimal execution plan for a given Spark SQL query by following a series of steps, as depicted in the following diagram:

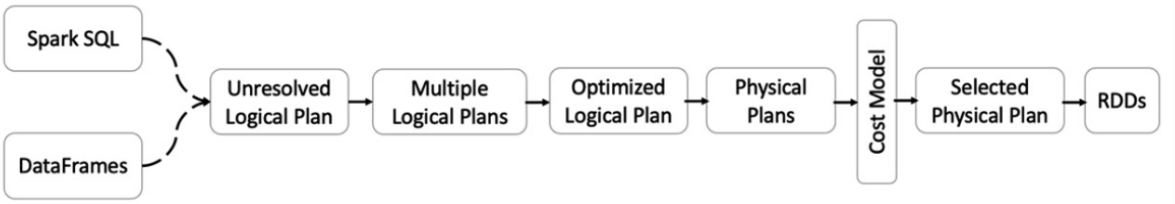


Figure 12.3 – Spark's Catalyst optimizer

https://learning.oreilly.com/library/view/essential-pyspark-for/9781800568877/B16736_12_Final_JM_ePub.xhtml#:-:text=Catalyst%20optimizer

Memory Management Overview –

Memory usage in Spark largely falls under one of two categories: execution and storage. Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations, while storage memory refers to that used for caching and propagating internal data across the cluster. In Spark, execution and storage share a unified region (M). When no execution memory is used, storage can acquire all the available memory and vice versa. Execution may evict storage if necessary, but only until total storage memory usage falls under a certain threshold (R). In other words, R describes a subregion within M where cached blocks are never evicted. Storage may not evict execution due to complexities in implementation.

This design ensures several desirable properties. First, applications that do not use caching can use the entire space for execution, obviating unnecessary disk spills. Second, applications that do use caching can reserve a minimum storage space (R) where their data blocks are immune to being evicted. Lastly, this approach provides reasonable out-of-the-box performance for a variety of workloads without requiring user expertise of how memory is divided internally.

https://spark.apache.org/docs/latest/tuning.html#memory-management-overview

Shuffle operations - Certain operations within Spark trigger an event known as the shuffle. The shuffle is Spark's mechanism for re-distributing data so that it's grouped differently across partitions. This typically involves copying data across executors and machines, making the shuffle a complex and costly operation.

To understand what happens during the shuffle, we can consider the example of the reduceByKey operation. The reduceByKey operation generates a new RDD where all values for a single key are combined into a tuple - the key and the result of executing a reduce function against all values associated with that key. The challenge is that not all values for a single key necessarily reside on the same partition, or even the same machine, but they must be co-located to compute the result.

In Spark, data is generally not distributed across partitions to be in the necessary place for a specific operation. During computations, a single task will operate on a single partition - thus, to organize all the data for a single reduceByKey reduce task to execute, Spark needs to perform an all-to-all operation. It must read from all partitions to find all the values for all keys, and then bring together values across partitions to compute the final result for each key - this is called the **shuffle**.

https://spark.apache.org/docs/latest/rdd-programming-guide.html#shuffle-operations

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage. The data broadcasted this way is cached in serialized form and deserialized before running each task. This means that explicitly creating broadcast variables is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

https://spark.apache.org/docs/latest/rdd-programming-guide.html#broadcast-variables

Spark clusters are broadly categorized into all-purpose and job clusters. All-purpose clusters can be used for several purposes such as ad hoc analysis, data engineering development, and data exploration. On the other hand, job clusters are spawned to perform a particular workload (an ETL process and more) and are automatically terminated as soon as the job finishes.

Regarding cluster modes, there are three types available for all-purpose clusters:

- **Standard**: This mode is the most frequently used by users. They work well to process big data in parallel but are not well suited for sharing with a large number of users concurrently.
- **Single Node**: Here, a cluster with only a driver and no workers is spawned. A single-node cluster is used for smaller workloads or use cases, wherein data needs to be processed in a non-distributed fashion.
- **High Concurrency**: As the name suggests, a high concurrency cluster is ideal for use among many users at the same time. It is ideal for ad hoc analytical use cases. It is also recommended to enable autoscaling when using a high concurrency cluster.

https://learning.oreilly.com/library/view/optimizing-databricks-workloads/9781801819077/B17782_04_Final_ASB_eBook.xhtml#:-:text=Understanding%20cluster%20types

CSV and Parquet

Advantages of CSV files:

- CSV is the most common file type among data scientists and users.
- They are human-readable, as data is not encoded before storing. They are also easy to edit.
- Parsing CSV files is very easy, and they can be read by almost any text editor.

Advantages of Parquet files:

- Parquet files are compressed using various compression algorithms, which is why they consume less space.
- Being a columnar storage type, Parquet files are very efficient when reading and querying data.
- The file carries the schema with itself and is partitioned in nature.
- With Parquet files, Spark scans much less data than with CSV files. This leads to a reduction in costs.

Disadvantages of CSV files:

- They cannot be partitioned, and being a row-based storage type, they are not very efficient for reading and querying data.
- In the majority of use cases, when using CSV with Spark, the entire dataset needs to be scanned for working with the data.

Disadvantages of Parquet files:

- Parquet files are not human-readable.

https://learning.oreilly.com/library/view/optimizing-databricks-workloads/9781801819077/B17782_05_Final_ASB_eBook.xhtml#:-:text=Learning%20to%20differentiate%20CSV%20and%20Parquet

column predicate pushdown

Column predicate pushdown is an optimization technique where we filter down to the level of the data source to reduce the amount of data getting scanned. This greatly enhances jobs, as Spark only reads the data that is needed for operations. For example, if we are reading from a Postgres database, we can push down a filter to the database to ensure that Spark only reads the required data. The same can be applied to Parquet and delta files as well. While writing Parquet and delta files to the storage account, we can partition them by one or more columns. And while reading, we can push down a filter to read only the required partitions.

https://learning.oreilly.com/library/view/optimizing-databricks-workloads/9781801819077/B17782_05_Final_ASB_eBook.xhtml#:-:text=Learning%20column%20predicate%20pushdown

partitioning strategies in Spark

Apart from this, there are also two very useful functions to manage Spark partitions. These functions are used to change the number of Spark partitions of a DataFrame:

- **repartition()**: This function is used to increase or decrease the number of Spark partitions for a DataFrame. It induces a shuffle when called and is often very helpful to remove skew from a DataFrame. The syntax is **dataframe.repartition(number)**. Here, **number** designates the new partition count of the DataFrame. The **repartition** function leads to roughly equally sized partitions.
- **coalesce()**: This function is used to decrease the number of partitions and is extremely helpful when the partition count needs to be drastically reduced. Also, note that it does not lead to shuffling. The syntax is **dataframe.coalesce(number)**. Here, **number** designates the new partition count of the DataFrame. The **coalesce** function can often lead to skew in partitions.

https://learning.oreilly.com/library/view/optimizing-databricks-workloads/9781801819077/B17782_05_Final_ASB_eBook.xhtml#:-:text=Learning%20partitioning%20strategies%20in%20Spark

broadcast joins

In **ETL** operations, we need to perform join operations between new data and lookup tables or historical tables. In such scenarios, a join operation is performed between a large DataFrame (millions of records) and a small DataFrame (hundreds of records). A standard join between a large and small DataFrame incurs a shuffle between the worker nodes of the cluster. This happens because all the matching data needs to be shuffled to every node of the cluster. While this process is computationally expensive, it also leads to performance bottlenecks due to network overheads on account of shuffling. Here, **broadcast joins** come to the rescue! With the help of broadcast joins, Spark duplicates the smaller DataFrame on every node of the cluster, thereby avoiding the cost of shuffling the large DataFrame.

In the case of a standard join, the partitions of both the DataFrames need to shuffle across worker nodes or executors so that matching records based on the join condition can be joined. In the case of a broadcast join, Spark sends a copy of the smaller DataFrame to each node or executor so that it can be joined with the respective partition of the larger DataFrame.

https://learning.oreilly.com/library/view/optimizing-databricks-workloads/9781801819077/B17782_07_Final_ASB_eBook.xhtml#:-:text=Learning%20about%20broadcast%20joins

shuffle partitions

Every time Spark performs a wide transformation or aggregations, shuffling of data across the nodes occurs. And during these shuffle operations, Spark, by default, changes the partitions of the DataFrame. For example, when creating a DataFrame, it may have 10 partitions, but as soon as the shuffle occurs, Spark may change the partitions of the DataFrame to 200. These are what we call the shuffle partitions.

https://learning.oreilly.com/library/view/optimizing-databricks-workloads/9781801819077/B17782_07_Final_ASB_eBook.xhtml#:-:text=Understanding%20shuffle%20partitions

caching in Spark

Every time we perform an action on a Spark DataFrame, Spark has to re-read the data from the source, run jobs, and provide an output as the result. This may not be a performance bottleneck when reading data for the first time, but if a certain DataFrame needs to be queried repeatedly, Spark will have to re-compute it every time. In such scenarios, Spark caching proves to be highly useful. Spark *caching* means that we store data in the cluster's memory. As we already know, Spark has memory divided for cached DataFrames and performing operations. Every time a DataFrame is cached in memory, it is stored in the cluster's memory, and Spark does not have to re-read it from the source in order to perform computations on the same DataFrame.

https://learning.oreilly.com/library/view/optimizing-databricks-workloads/9781801819077/B17782_07_Final_ASB_eBook.xhtml#:-:text=Understanding%20caching%20in%20Spark

Learning about AQE

We already know how Spark works under the hood. Whenever we execute transformations, Spark prepares a plan, and as soon as an action is called, it performs those transformations. Now, it's time to expand that knowledge. Let's dive deeper into Spark's query execution mechanism.

Every time a query is executed by Spark, it is done with the help of the following four plans:

- **Parsed Logical Plan**: Spark prepares a *Parsed Logical Plan*, where it checks the metadata (table name, column names, and more) to confirm whether the respective entities exist or not.
- **Analyzed Logical Plan**: Spark accepts the Parsed Logical Plan and converts it into what is called the *Analyzed Logical Plan*. This is then sent to Spark's catalyst optimizer, which is an advanced query optimizer for Spark.
- **Optimized Logical Plan**: The catalyst optimizer applies further optimizations and comes up with the final logical plan, called the *Optimized Logical Plan*.
- **Physical Plan**: The *Physical Plan* specifies how the Optimized Logical Plan is going to be executed on the cluster.

Apart from the catalyst optimizer, there is another framework in Spark called the **cost-based optimizer** (**CBO**). The CBO collects statistics on data, such as the number of distinct values, row counts, null values, and more, to help Spark come up with a better Physical Plan. AQE is another optimization technique that speeds up query execution based on runtime statistics. It does this with the help of the following three features:

- **Dynamically coalescing shuffle partitions**
- **Dynamically switching join strategies**
- **Dynamically optimizing skew joins**

Let's discuss these in detail.

Dynamically coalescing shuffle partitions

When dealing with very large datasets, shuffle has a huge impact on performance. It is an expensive operation that requires data to be moved across nodes so that it can be re-distributed as required by the downstream operations. But two types of issues can occur:

- If the number of partitions is less, then their size will be larger, and this can lead to data spillage during the shuffle. This can slow down Spark jobs.
- If the number of partitions is more, then there could be a chance that the partitions would be small in size, leading to a greater number of tasks. This can put more burden on Spark's task scheduler.

To solve these problems, we can set a relatively large number of shuffle partitions and then coalesce any adjacent small partitions at runtime. This can be achieved with the help of AQE, as it automatically coalesces small partitions at runtime.

Dynamically switching join strategies

With the help of AQE, Spark can switch join strategies at runtime if they are found to be inefficient. Spark supports various join strategies but usually, the *broadcast hash join* (also called the *broadcast join*) is often considered to be the most performant if one side of the join is small enough to fit in the memory of every node.

Dynamically optimizing skew joins

Data skew occurs when data is unevenly distributed across the partitions of the DataFrame. It has the potential to downgrade query performance. With the help of AQE, Spark can automatically detect data skew while joins are created. After detection, it splits the larger of those partitions into smaller sub-partitions that are joined to the corresponding partition on the other side of the join. This ensures that the Spark job does not get stuck due to a single enormously large partition.

https://learning.oreilly.com/library/view/optimizing-databricks-workloads/9781801819077/B17782_07_Final_ASB_eBook.xhtml#:-:text=Learning%20about%20AQE

Spark caching and Spark checkpointing

Caching can be used to increase performance. Caching will persist the dataframe in memory, or disk, or a combination of memory and disk. Caching will also save the lineage of the data. Saving the lineage is useful only if you need to rebuild your dataset from scratch, which will happen if one of the nodes of your cluster fails.

Spark offers two methods for caching: `cache()` and `persist()`. They work the same, except that `persist()` enables you to specify the storage level you wish to use. When using an argument, `cache()` is a synonym for `persist(StorageLevel.MEMORY_ONLY)`.
https://learning.oreilly.com/library/view/spark-in-action/9781617295522/OEBPS/Text/16.xhtml#:-:text=Spark%20caching

Checkpoints are another way to increase Spark performance. In this subsection, you'll learn what checkpointing is, what kind of checkpointing you can perform, and how it differs from caching.

The `checkpoint()` method will truncate the DAG (or logical plan) and save the content of the dataframe to disk. The dataframe is saved in the checkpoint directory.

https://learning.oreilly.com/library/view/spark-in-action/9781617295522/OEBPS/Text/16.xhtml#:-:text=Spark%20checkpointing