Apache Spark

Apache Spark is a framework for distributed streaming, data-processing, and model training. It provides high-level APIs in Java, Scala, Python, and R. Computation is defined on the 'driver' process, the driver's 'SparkContext' is responsible for managing the Spark application and connects to the cluster manager which allocates resources, distributes the data across the worker machines, and launches executor processes on worker nodes. Each worker node runs a JVM which executes Scala operations on the data.
Spark has a lazy execution model, computation is only executed when a result is needed. There are two kinds of Spark operations: transformations (e.g. map/reduce operations), which are lazy and map one dataset to another, and actions (e.g. writing to file), which are eager and execute the plan to return the desired result. This allows Spark to optimise the execution plan, for example by only reading relevant columns/partitions to the result. Catalyst is a query optimizer that can optimise the execution plan based on the data and the operations performed on it. Furthermore, Spark has Adaptive Query Execution (AQE) which is runtime execution that optimises the operations Spark performs. For example, if at runtime Spark identifies one side of a join is far smaller, it will perform a broadcast join instead of a shuffle/sort merge join.

The main abstraction in Spark is the Resilient Distributed Dataset **RDD**. RDDs are an immutable collection of elements distributed across nodes of a cluster, they are created by either parallelizing a collection stored locally on the driver or by reading in from external storage, such as a Hadoop Distributed Filesystem or AWS S3. RDDs are fault-tolerant; they are split into partitions and distributed across the worker nodes. Should a node fail, Spark knows what partitions of the original Dataframe created the lost partitions and can recompute them via the defined computation graph (called the lineage).

In Spark, variables in functions distributed to each worker node, there are two forms of shared variables between all worker nodes: Broadcast variables (cached in the memory of workers) and accumulators which can be incremented by all workers.

Spark SQL

While Spark offers a general model to perform arbitrary transformations on a distributed dataset, Spark SQL is a Spark module for structured data processing. Spark SQL builds upon the RDD with higher-level APIs like DataFrames and Datasets that offer better performance through optimizations like predicate pushdown and columnar storage.

PySpark

Pyspark is the Python interface to Spark. It is built on top of Py4J, a library that allows Python to interact with Java objects. PySpark provides a Pythonic interface to Spark's RDDs and DataFrames, and allows Python users to use Spark's powerful features without having to learn Scala. In PySpark each worker process also runs a Python interpreter, which allows user defined functions ('UDF's) to be executed on the worker nodes. This requires serialsiing the data from the JVM to the Python interpreter and back, which is costly. To minimise this overhead, PySpark uses Apache Arrow for efficient data transfer between Python and the JVM and provides UDF (User Defined Function) optimizations like vectorized operations.

Optimisations

Spark spins up a JVM per cpu core inside the worker pod. The combined memory available to the executor is specified in: `spark.executor.memory`. Executor JVM memory is used for much of the work done when using the dataframes API, and it also includes the actual data that loaded in memory. To reduce IO (read/write time), it is therefore important to consider memory per core. The overall runtime of many jobs can be reduced by reducing the number of cpu cores (and increasing the proportion of the dataset that can sit on each core). Increasing the number of partitions and reducing number of cores, are very common fixes for performance issues.

Sometimes it is useful to cache intermediate results, usually this is done in two scenarios:

- The combined transformation is too complicated/expensive/heavy to be performed "at once" with Spark, or Spark gets confused and makes mistakes when trying to optimize it
- To use an intermediate output for multiple things, e.g. Cache the result of reading in and processing a large initial dataset before using it downstream in different operations

Generally, `.cache()` is sufficient, which stores the dataframe in memory on the worker nodes. This can be heavy on Spark workers and if the workers die, the cached partitions are lost. Alternatively, `.persist()` exists and the storage location (e.g. disk, memory, or off-heap memory) can be specified. Finally, the dataframe can be saved to a file or database, although this breaks lineage, so Spark cannot optimise the whole computation graph as well.

Calling `.explode()` on a column will increase the dataframe's size, if Spark runs out of memory it will attempt to write intermediate results to disk, which means this operation is hugely costly.

When reducing the number of partitions (logical divisions of your data), use `coalesce` over `repartition` as it does not shuffle the data, it just logically combines partitions, while `repartition` shuffles the data across the worker nodes (which is roughly as expensive as a sort).

Pandas UDFs

PySpark executes Python UDFs row-by-row. Pandas UDFs, introduced in Spark 2.3, offer a faster (parallelised) alternative. Each worker loads the partitions into a Pandas Dataframe/Series and executes the UDF using vectorised operations.
Spark implements two kinds of Pandas UDF:

1. Scalar Pandas UDFs: Essentially a map, define a function that takes in a Pandas series and returns a Pandas series object of the same size (one-to-one).
2. Grouped Map Pandas UDFs: Apply a function (e.g. an aggregation) to all values sharing the same key. Grouped map Pandas UDFs splits a Spark DataFrame into groups specified by the groupby operator, and applies a user-defined function (pandas.DataFrame $\rightarrow$ pandas.DataFrame) to each group, combines and returns the results as a new Spark DataFrame. No column pruning is done before these, so it is worth a `.select` on relevant columns first.

Avoid using Pandas in Pandas UDFs. This sounds counter-intuitive however it can offer magnitudes of speedups. The general strategy is to use pd.DataFrame.values or pd.Series.values to get the data as an array-like object and then use something very fast like itertools or numpy to transform the data, inside the Pandas UDF.
Use grouped map UDFs to do partial UDFs. In grouped map UDFs, each group must be able to fit in memory, which can make large aggregations difficult, to circumvent this:

1. Run some pre-processing to figure out how many batches are needed to comfortably fit each batch in worker memory.
2. Add a random integer to each row (the batch number).
3. GroupBy the batch number and apply the grouped map UDF to the batch.
4. Apply the grouped map UDF again to the result of each batch.

Pandas UDFs perform much better than row-at-a-time UDFs across the board, ranging from 3x to over 100x.

RAPIDS, Nvidia GPU Acceleration for Spark

RAPIDS are a collection of Nvidia GPU accelerated libraries that mimic the API of open-source Python data processing and machine learning libraries, for example cuDF for Pandas and cuML for scikit-learn.

RAPIDs GPU accelerator for PySpark hijacks the execution of Scala operations on the CPUs of the worker nodes and executes CUDA on the GPUs instead, with no code changes required in defining the computation graph in the driver process.
It simply requires two JARs to be installed on the worker nodes:

1. cuDF Jar: This is a GPU-accelerated DataFrame library (written in C++/CUDA, with a Java wrapper). It provides low-level, columnar data processing on the GPU for Spark.
2. RAPIDS accelerator for PySpark JAR: A plugin that integrates with Spark SQL/Catalyst. It rewrites Spark's execution plans to use GPU (cuDF) operations where supported, delegating suitable Spark SQL/DataFrame operations to the GPU via cuDF.