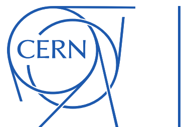
Spark Course - Introduction to Apache Spark APIs Building on DataFrames

Luca Canali
CERN IT, Data Analytics and Spark Service

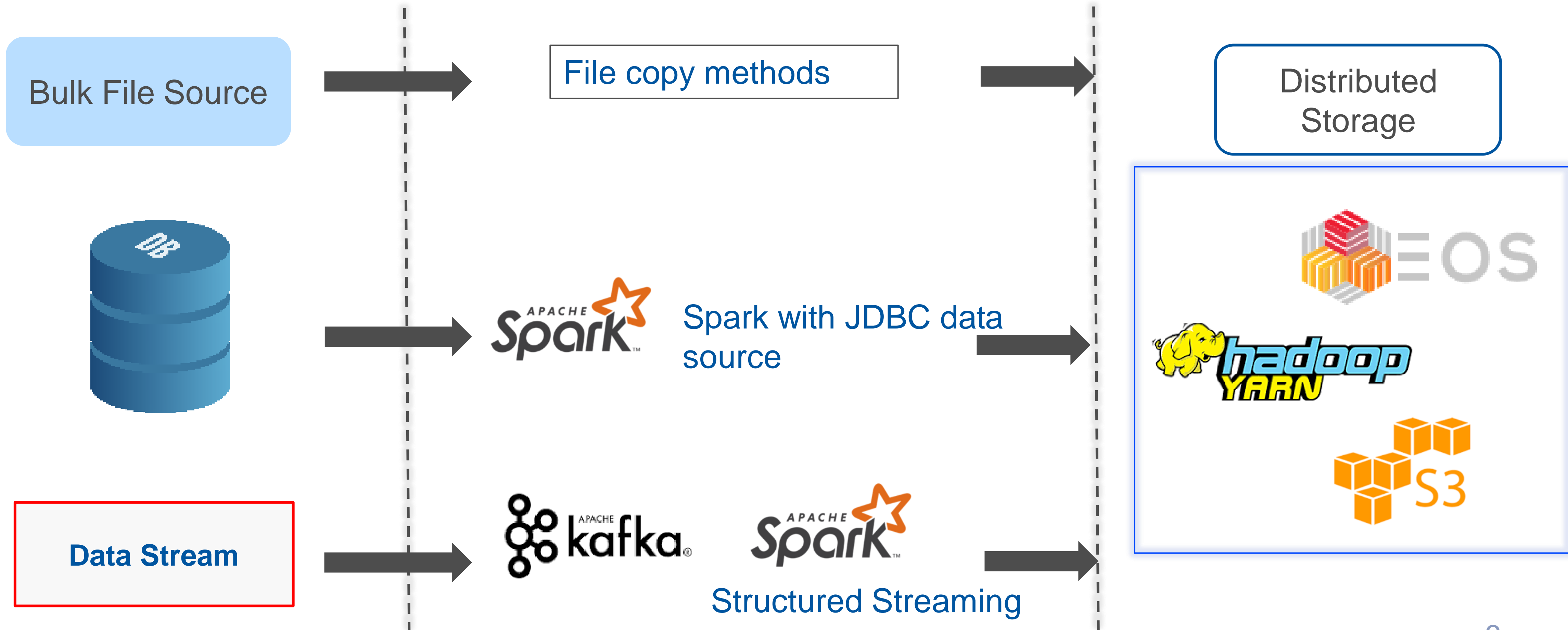


Build a Data Platform with Spark

- What Spark (SQL and DataFrame API) do well:
  - Provide powerful abstractions and rich language(s)
    - Both for data preparation and analytics
  - Run SQL at scale using distributed computing
    - Runs on clusters (YARN/Hadoop, Kubernetes, etc)
  - Integration with a large ecosystem
    - Can use for many file formats: Parquet, csv, (ROOT), …
    - Storage systems: HDFS, S3, (EOS), ...
    - External systems: databases, elastic search, streaming, etc
    - Table formats and transactions: Delta, Iceberg, Hudi

Data Ingestion - Examples

- Data pipelines into “data lakes”



Spark DataFrame Reader and Writer and the Apache Parquet Data Format

CERN

DataFrame Reader API

- Spark can process many file-based data formats
  - The DataFrame reader ingests from files or folders
    - All files are read mapped into the DataFrame (table)
  - Example of DataFrame reader ingesting Apache Parquet

```
>>> df = spark.read.format("parquet").load("filename")

>>> df = spark.read.parquet("PATH_and_directoryname")
```

CERN

Reading Partitioned Data

- Partition Discovery
  - Partitions defined through the filesystem folder structure
    - Naming convention: <partition_col=value>

```
# Filesystem folder structure
table_name_folder
└── part_col=1
    ├── part-xxxxxx-id1.snappy.parquet
└── part_col=2
    ├── part-xxxxxx-id2.snappy.parquet

>>> df = spark.read.parquet("table_name_folder")

>>> df.printSchema() # will show part_col as a table column
```

CERN

DataFrame Writer

- Use “df.write.parquet” to write in Parquet format
  - Use coalesce if you want to reduce the number of output partitions
    - beware that it also affects/reduces the number of concurrent writer tasks
  - The output is a structure of nested folders representing partitions

```
df.coalesce(N_partitions)
.write
.partitionBy("colPartition1", "colOptionalSubPart") # partitioning column(s)
.parquet("filePathandName")

# Options
.repartition(col("colPartition1"), col("colOptionalSubPartition") # compact to 1 file
per partition
.option("compression","zstd") # the default compression algorithm is snappy
.mode("overwrite") # overwrite if the file/directory exists
```

Advantages of the Parquet Format

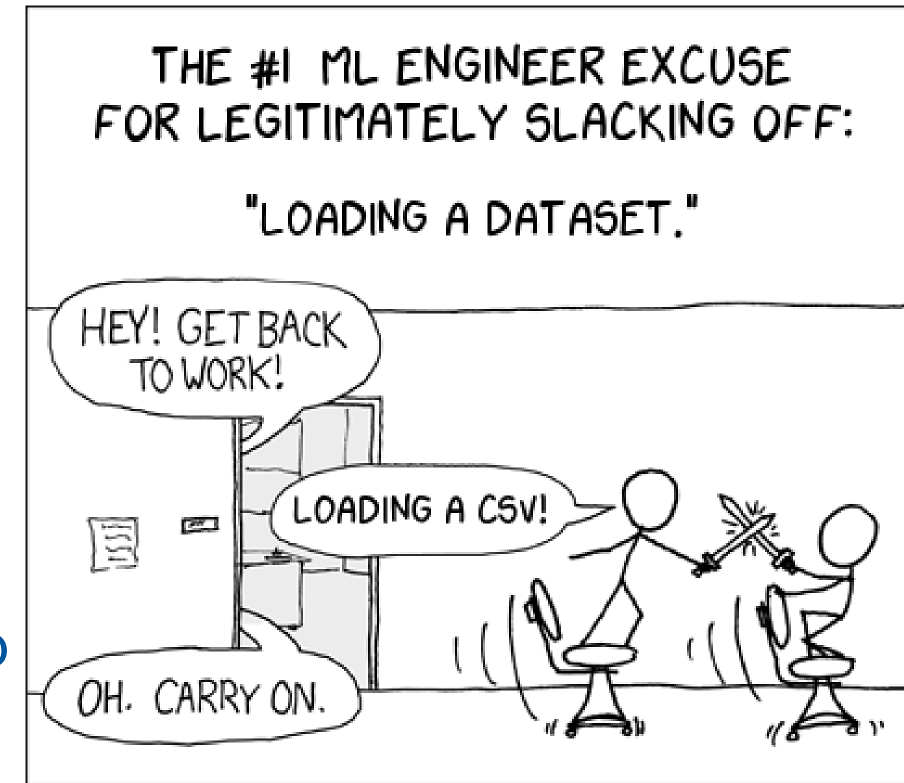
- Data is stored and accessed by column
  - Optimized when your query needs to read only a few columns
- Data encoding
  - Example: run length encoding to more efficiently store data repetition
- Compression also available
  - Default is snappy compression: lightweight and good compression
  - Also available: Zstandard, gzip, etc
- Data is stored with its schema
  - Schema evolution is also supported

CERN

Advantages of the Parquet Format

- Spark is optimized for Parquet
  - Integrates Hadoop Parquet-MR
  - In addition, Spark has a custom vectorized Parquet reader, for performance

- Filter pushdown and use of metadata
  - Filters can be pushed down to the Parquet level
  - Use of min/max and count values per row group and per page
    - used to skip reading data for improved performance
    - work best when data is sorted on the filter column
    - recent versions also support bloom filters



From an XKCD comic

CERN

Key Learning Points

- Spark ecosystem builds on DataFrames
  - Spark can run SQL at scale, integrating with clusters, storage systems
- Choosing the data format is key
  - For many data analysis workloads columnar data formats as Apache Parquet are a good fit

CERN