Mirai Solutions
smarter analytics - better decisions

Scale Your Analytics - Leveraging Apache Spark in Python and R

A practical introduction & guide for data science practitioners.

Martin Studer

September 17, 2024

Who Am I

Martin Studer

Executive Partner & Senior Software Engineer @ Mirai Solutions

Education: Computer Science @ ETH Zurich

Mirai Solutions

Mirai Solutions

- Boutique consulting company based in Zurich
- Specialized in data science and data engineering
- Main clients in the re-/insurance field
- Visit us
  - https://mirai-solutions.ch
  - https://github.com/miraisolutions/
  - https://linkedin.com/company/mirai-solutions-gmbh

MiraiLabs | Data Science Workshops

- **MiraiLabs** is a series of data science workshops aimed at professionals
- Designed to benefit people who work with data or models on a daily basis and would like to expand and strengthen their skill set
- Join the MiraiLabs group on meetup if you would like to be notified about future events

License

Disclaimer

This presentation includes references to various third-party products and organizations. Please note that we are not affiliated with, sponsored by, or endorsed by any of these entities. All trademarks, service marks, trade names, and logos mentioned in this presentation are the property of their respective owners. These references are provided for informational purposes only to illustrate concepts and the solutions we utilize or encounter in our work.

Agenda

1. Motivation
2. Scaling
3. Apache Spark
4. Drinks & Snacks 🥪 🥂

Motivating Example

Motivating Example - Dataset (1)

- Dataset: Stack Exchange Data Dump by Stack Exchange Inc., available under CC-BY-SA 4.0 license
- Anonymized dump of all user-contributed content on the Stack Exchange network
- Focus on Stack Overflow network ("Q&A for programmers")
- Data tables as XML files zipped via 7-zip using bzip2 compression
- Schema information: see this meta post
- XML files pre-processed and stored as parquet files: notebook
  - Apache Parquet was designed for efficient data storage and retrieval
  - Many of the frameworks/libraries in this example would have struggled to efficiently handle the large XML files

Motivating Example - Dataset (2)

- Main files / tables of interest
  - **Posts***: provides information about posts, i.e. questions, answers, wiki entries, moderator nominations, etc. (~193 GiB XML, ~35 GiB parquet)
  - **Tags***: linking tag names to wiki and excerpt texts in the posts table (~5.5 MiB XML, ~1.5 MiB parquet)
  - **Users**: provides user information including reputation and number of up and down votes (~10.5 GiB XML, ~1.5 GiB parquet)
  - **Comments**: comments added to posts (~51.5 GiB XML, ~10.5 GiB parquet)
  - **Votes**: votes on posts, incl. if a post was accepted, marked as spam, deleted, etc.; also includes whether a bounty has been started on a post (~43 GiB XML, ~2 GiB parquet)
- Question: What are the top 5 highest scoring questions related to Apache Spark?

* Relevant for this initial example

Motivating Example - Execution Environment

- Azure Databricks (Databricks 14.3 LTS, based on Apache Spark 3.5.0)
- VMs: Azure Eadsv5-series; offer a combination of vCPU, memory and temporary storage that is ideal for in-memory analytics workloads
- Data stored on Azure Data Lake Storage Gen2

Motivating Example - Python/pyarrow/pandas

- Notebook
- 1x `Standard_E16ads_v5` (16 cores, 128 GB memory):

  ⚠️ takes ~10.5m in total to execute; pre-filtering rows & columns is essential!

  ❌ fails without pre-filtering on import

  ```
  Fatal error: The Python kernel is unresponsive.
  The Python process exited with exit code 137 (SIGKILL: Killed).
  This may have been caused by an OOM error. Check your command's memory usage.
  ```

- 1x `Standard_E32ads_v5` (32 cores, 256 GB memory):

  ⚠️ takes ~10m in total to execute

- Does not scale to multiple cores and beyond a single machine without drop-in replacement libraries such as e.g. modin and dask

Motivating Example - R/arrow/dplyr

- Notebook
- 1x `Standard_E16ads_v5` (16 cores, 128 GB memory):

  ⚠️ takes ~11m in total to execute
- 1x `Standard_E32ads_v5` (32 cores, 256 GB memory):

  ⚠️ takes ~11m in total to execute
- Does not scale to multiple cores and beyond a single machine

Motivating Example - Python/polars

- Notebook
- 1x `Standard_E16ads_v5` (16 cores, 128 GB memory):

  ✔ takes ~4m in total to execute
- 1x `Standard_E32ads_v5` (32 cores, 256 GB memory):

  ✔ takes ~2.5m in total to execute
- Scales well to multiple cores but does not scale beyond a single machine

Motivating Example - Python/pyspark

- Notebook
- Databricks default configuration
- 1x `Standard_E16ads_v5` (16 cores, 128 GB memory):

  ✔ ~2m in total to execute
- 1x `Standard_E32ads_v5` (32 cores, 256 GB memory):

  ✔ ~1m in total to execute
- 1x `Standard_F4s_v2` (4 cores, 8 GB memory; driver) + 4x `Standard_E8ads_v5` (8 cores, 64 GB memory; worker)

  ✔ ~1-1.5m in total to execute
- Scales to multiple cores and to multiple machines

Motivating Example - R/SparkR

- Notebook
- Databricks default configuration
- 1x `Standard_E16ads_v5` (16 cores, 128 GB memory):

  ✔ ~2m in total to execute
- 1x `Standard_E32ads_v5` (32 cores, 256 GB memory):

  ✔ ~1m in total to execute
- 1x `Standard_F4s_v2` (4 cores, 8 GB memory; driver) + 4x `Standard_E8ads_v5` (8 cores, 64 GB memory; worker)

  ✔ ~1-1.5m in total to execute
- Scales to multiple cores and to multiple machines

Motivating Example - Python/pyspark with Photon

The following timings are based on Databricks 14.3 LTS with Photon acceleration enabled. Databricks Photon is an optimized query engine designed to accelerate Apache Spark SQL workloads.

- 1x `Standard_E32ads_v5` (32 cores, 256 GB memory) or 1x `Standard_E4ads_v5` (4 cores, 32 GB memory; driver) + 4x `Standard_E8ads_v5` (8 cores, 64 GB memory; worker)

  ✔ $\lesssim$ 1m in total to execute

Scaling

Scaling - Scaling Up vs Scaling Out

- Scaling up aka vertical scaling
  - Increase the capacity of a single machine by adding more or improved resources (CPUs/cores, RAM, storage, ...)
  - Optimize code efficiency
    - use efficient algorithms & data structures
    - leverage vectorized operations
    - use parallelization
    - perform non-blocking operations
    - use specialized hardware if available (e.g. GPUs)
    - ...
- Scaling out aka horizontal scaling
  - Add more machines to a network of interconnected systems to handle increased load
  - *Distributed computing* is the methodological framework that enables multiple computers to work together on a common task

Scaling - Scaling Up with Python

- Multi-threading: use Python's threading module; restricted use due to global interpreter lock (GIL); mostly beneficial for I/O-bound tasks
- Multi-processing: use Python's multiprocessing module; each sub-process has its own Python interpreter and memory
- NumPy and SciPy: highly optimized operations for numerical and scientific computing (partially implemented in C; some operations leveraging SIMD instructions)
- Cython: superset of Python that allows to add static typing to your Python code and compile it to C
- PyPy: an alternative interpreter to CPython with a JIT compiler
- Numba: JIT compiler that translates a subset of Python and NumPy code into machine code
- Parallel computing libraries: joblib, polars, dask, modin, ray, vaex, ...
- duckdb: in-process analytical database
- GPU acceleration: RAPIDS (cuDF), TensorFlow, PyTorch, ...

Scaling - Scaling Up with R

- Parallel computing & HPC libraries:
  - data.table, parallel, foreach, doParallel, future, furrr, parallely, …
  - polars (rapidly evolving, not on CRAN yet)
  - See CRAN task view high-performance and parallel computing
- Interfacing with compiled code and other languages: Rcpp, inline, reticulate, …
- Specialized and optimized libraries: see CRAN task views such as e.g. numerical mathematics
- duckdb: in-process analytical database
- GPU acceleration: torch, OpenCL, tensorflow, keras, …

Scaling - Scaling Out with Python

- Apache Spark: unified engine for large-scale data analytics
- Dask / Coiled: library for parallel computing designed to integrate seamlessly with NumPy, pandas and scikit-learn; supports a distributed scheduler to execute tasks on a distributed cluster
- Ray: unified distributed computing framework with a focus on ML & AI workloads
- Apache Beam: unified programming model for batch and streaming data processing pipelines; supports several backends, including Apache Spark
- Apache Flink: distributed processing engine for stateful computations over unbounded and bounded data streams
- h2o: Python interface for the H2O scalable machine learning platform
- Celery: distributed task queue framework that can be used for task scheduling and distributed computing (lower-level)

Scaling - Scaling Out with R

- Apache Spark: unified engine for large-scale data analytics
- Parallel computing libraries: parallel, snow, Rmpi, future, parallely, … (lower-level)
- RHadoop: a collection of five R packages that allow users to manage and analyze data with Hadoop; ⚠️ seems no longer maintained
- h2o: R interface for the H2O scalable machine learning platform

Scaling - Scaling Up or Scaling Out? (1)

Opt for scaling up when ...

- you have a significant investment in single-threaded applications or tasks that do not parallelize well
- your data and processing needs are growing but remain within the capabilities of (high-end) single-machine systems (possibly also by leveraging out-of-memory algorithms or by leveraging specialized frameworks/libraries)
- short-term performance improvements are needed
- infrastructure simplicity and ease of management are priorities

Scaling - Scaling Up or Scaling Out? (2)

Opt for scaling out when ...

- you anticipate rapid or continuous growth in data volume and/or processing needs that a single machine would struggle to handle
- your tasks are inherently parallel or can (easily) be distributed across multiple nodes
- you require high availability, fault tolerance, and redundancy, which are more naturally achieved through distributed systems
- cost efficiency and flexibility in scaling according to demand are crucial

NOTE: Scaling out does not exclude scaling up!

Apache Spark

Apache Spark - What Is It and Why Use It? (1)

- Open-source, general-purpose, distributed data processing engine for data engineering, data science and machine learning
    - Multi-language: supports **Python**, **R**, Java, Scala, **SQL**
    - Unified: single, integrated computing solution for a wide range of data processing tasks and workloads such as batch processing, interactive queries, real-time streaming analytics, machine learning - all within the same core engine and with a consistent set of APIs
- Designed to scale from a single server to thousands of machines
- Built-in fault tolerance: the system automatically recovers from failures in individual nodes, ensuring that processing can continue without data loss
- Large and active community that contributes to its continued development and which provides extensive documentation, resources and forums for support

Apache Spark - What Is It and Why Use It? (2)

- Integrates with a wide range of data sources and storage systems
- Rich ecosystem
  - Open-source storage layers and open table formats: Delta Lake, Apache Hudi, Apache Iceberg; see also Delta UniForm
  - Large-scale spatial data processing: Apache Sedona
  - GPU acceleration: RAPIDS accelerator
- Availability of cloud offerings, supporting services and commercial platforms
  - Reduces operational complexity
  - Microsoft Azure
    - Apache Spark in Azure Synapse Analytics
    - Azure Databricks
  - Amazon AWS
    - Apache Spark on Amazon EMR
    - Databricks on AWS
  - Google Cloud
    - Dataproc
    - Databricks on Google Cloud

Apache Spark - Components (1)

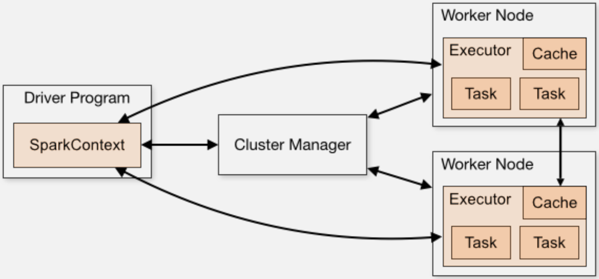


Figure 1: Apache Spark [1]

- Spark is based on a coordinator/worker architecture. It has one central coordinator (*driver*) that communicates with many distributed *workers* (*executors*).

[1] https://spark.apache.org/docs/latest/cluster-overview.html

Apache Spark - Components (2)

- The driver runs the user program and is responsible for ...
    - Requesting resources (executors) from the cluster manager
    - Converting the user program into *tasks*
    - Scheduling these tasks on executors (workers)
    - Managing and monitoring the executors across the cluster
- Executors are processes that run individual tasks and are responsible for ...
    - Executing the code assigned to them by the driver and returning results to the driver
    - Providing storage (cache) to store frequently accessed data

Apache Spark - APIs

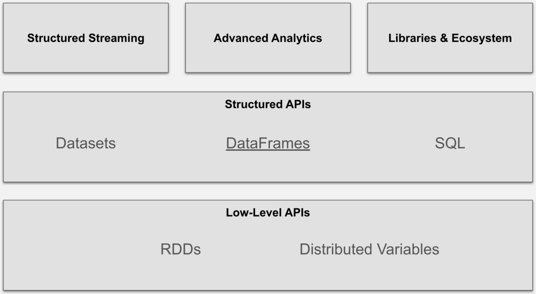
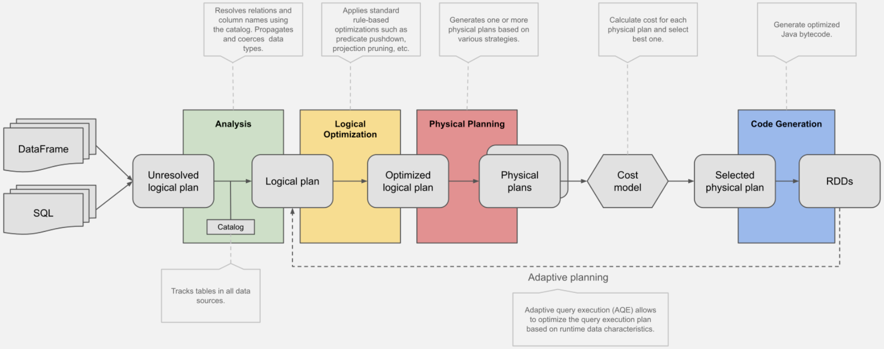


Figure 2: Low-Level vs Structured APIs

- Python & R language APIs: use the powerful structured data frame APIs
  - Python: pyspark (official)
  - R: SparkR (official, but planned to be deprecated in Spark 4.0 in favour of sparklyr), sparklyr (dplyr-compatible; by posit)
  - SQL: seamlessly use SQL queries in your Python & R programs

Apache Spark - Query Planning, Optimization & Execution



Resolves relations and column names using the catalog. Propagates and coerces data types.
Applies standard rule-based optimizations such as predicate pushdown, projection pruning, etc.
Generates one or more physical plans based on various strategies.
Calculate cost for each physical plan and select best one.
Generate optimized Java bytecode.
DataFrame
SQL
Analysis
Unresolved logical plan
Catalog
Logical plan
Logical Optimization
Optimized logical plan
Physical Planning
Physical plans
Cost model
Selected physical plan
Code Generation
RDDs
Tracks tables in all data sources.
Adaptive planning
Adaptive query execution (AQE) allows to optimize the query execution plan based on runtime data characteristics.

Figure 3: Query Planning, Optimization & Execution

Apache Spark - Transformations vs Actions (1)

There are two types of operations in Spark: *transformations* and *actions.*

Transformations

- An operation that transforms one or more input data frames (RDDs) to another one.
- They are *lazy*, i.e. they do not compute the result right away. Rather, Spark maintains a record of operations that have been requested, and which are only executed when an action is performed. This “record of operations” is referred to as the data lineage and is represented by a directed acyclic graph (DAG).
- It’s this laziness which allows Spark to optimize the execution plan (e.g. by rearranging operations to minimize data shuffling or combining operations when possible).

Apache Spark - Transformations vs Actions (2)

Actions

- An operation that triggers a computation or a sequence of computations and produces an output, such as a result which is returned to the driver program or a data frame which gets written to permanent storage.
- When an action is called Spark *materializes* the result, i.e. it computes the lineage graph, optimizes the execution plan and executes the transformations.

Apache Spark - Developing Applications Locally (1)

Make sure you have an installation of Java:

- Preferably Java 17 for Spark 3.5.x
- `JAVA_HOME` environment variable may need to be set accordingly

Apache Spark - Developing Applications Locally (2)

Using Python/pyspark

- Install pyspark from PyPI (`pip install pyspark`) or conda (`conda install -c conda-forge pyspark`)
- Create a Spark session using

```
from pyspark import SparkConf
from pyspark.sql import SparkSession

spark_config = (
   SparkConf()
   .setAppName("scale-your-analytics")
   .setMaster("local[*]")
   .set("spark.driver.memory", "2g")
)
spark = SparkSession.builder.config(conf=spark_config).getOrCreate()
```

Apache Spark - Developing Applications Locally (3)

Using R/SparkR

- Install SparkR from github using e.g. `devtools::install_github('apache/spark@v3.5.1', subdir='R/pkg')`
- Create a Spark session using

```
require(SparkR)

spark_config <- list(
 "spark.driver.memory" = "2g"
)
spark <- sparkR.session(
 master = "local[*]",
 appName = "scale-your-analytics",
 sparkConfig = spark_config
)
```

Apache Spark - Developing Applications Locally (4)

Using R/sparklyr

- Install sparklyr using `install.packages("sparklyr")` and install a suitable version of Spark using e.g. `sparklyr::spark_install(version = "3.5.1")`
- Create a Spark session using

```r
require(sparklyr)

spark_config <- spark_config()
spark_config$spark.driver.memory <- "2g"

spark <- spark_connect(
 master = "local[*]",
 app_name = "scale-your-analytics",
 config = spark_config
)
```

Apache Spark - Displaying Data (1)

Using Python/pyspark

Use the `show()` method to display a data frame:

```
df.show()
```

By default, `show()` displays the first 20 rows only. You can customize the number of rows displayed and the truncation of long strings with the following parameters:

```
df.show(n=100, truncate=False)
```

Apache Spark - Displaying Data (2)

Using R/SparkR

Use the `showDF()` method to display a data frame:

```r
showDF(df)
```

By default, `showDF()` displays the first 20 rows only. You can customize the number of rows displayed and the truncation of long strings with the following parameters:

```r
showDF(df, n = 100, truncate = FALSE)
```

Apache Spark - Displaying Data (3)

Using R/sparklyr

Use the `print()` method to display a data frame:

```
print(df)
```

By default, `print()` displays the first 10 rows only (see printing tibbles). You can customize the number of rows displayed using:

```
print(df, n = 100)
```

To control the width of the generated text output use the `width` argument:

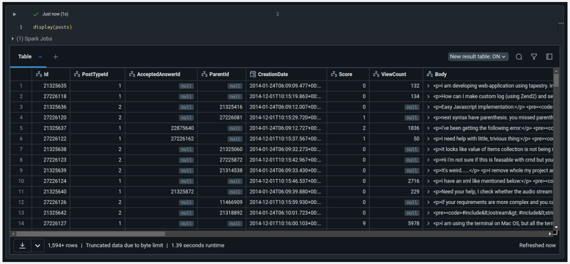
```
print(df, n = 100, width = 150)
```

Apache Spark - Displaying Data (4)

Using Databricks Notebooks

In Databricks notebooks you can also leverage the `display()` method/function to render a sortable and filterable grid control:

- Using Python/pyspark: `df.display()` or `display(df)`
- Using R/SparkR/sparklyr: `display(df)`



Just now (1s)

```
display(posts)
```

(1) Spark Jobs

Table

New result table: ON

	Id	PostTypeId	AcceptedAnswerId	ParentId	CreationDate	Score	ViewCount	Body
1	21325635	1	null	null	2014-01-24T06:09:09.477+00:...	0	132	<p>I am developing web-application using tapestry. In
2	27226118	1	null	null	2014-12-01T10:15:19.863+00:...	0	134	<p>How can I make custom log (using Zend2) and se
3	21325636	2	null	21325416	2014-01-24T06:09:12.007+00:...	0	null	<p>Easy Javascript implementation:</p> <pre><code
4	27226120	2	null	27226081	2014-12-01T10:15:29.720+00:...	1	null	<p>next syntax have parenthesis. you missed parenth
5	21325637	1	22875640	null	2014-01-24T06:09:12.727+00:...	2	1836	<p>I've been getting the following error:</p> <pre><co
6	27226122	1	27226162	null	2014-12-01T10:15:37.567+00:...	1	50	<p>I need help with little, trivious thing:</p> <pre><co
7	21325638	2	null	21325060	2014-01-24T06:09:32.273+00:...	0	null	<p>It looks like value of items collection is not being r
8	27226123	2	null	27225872	2014-12-01T10:15:42.967+00:...	0	null	<p>Hi i'm not sure if this is feasable with cmd but you
9	21325639	2	null	21314538	2014-01-24T06:09:33.430+00:...	0	null	<p>It's weird......</p> <p>I remove whole my project ar
10	27226124	1	null	null	2014-12-01T10:15:46.537+00:...	0	2716	<p>I have an xml like mentioned below:</p> <pre><co
11	21325640	1	21325872	null	2014-01-24T06:09:39.880+00:...	0	229	<p>Need your help, I check whether the audio stream
12	27226126	2	null	11466909	2014-12-01T10:15:59.930+00:...	0	null	<p>If your requirements are more complex and you ca
13	21325642	2	null	21318892	2014-01-24T06:10:01.723+00:...	0	null	<pre><code>#include<iostream> #include<stri
14	27226127	1	null	null	2014-12-01T10:16:00.103+00:...	9	5978	<p>I am using the terminal on Mac OS, but all the term

1,594+ rows | Truncated data due to byte limit | 1.39 seconds runtime

Refreshed now

Figure 4: Databricks notebook table control

Apache Spark - Concepts in Action

- Dataset
  - Chess games from kaggle (License: CC0)
  - 6.25 million chess games played on lichess during July of 2016
- Code: See this gist
- Concepts explained
  - Drivers & workers → Executors tab in Spark UI
  - DAG → SQL/DataFrame tab in Spark UI
  - Transformations vs actions → Jobs & Stages tabs of Spark UI
  - Query planning & optimization → SQL/DataFrame tab of Spark UI; note the filter pushdown!
  - Partitions, jobs, stages, tasks, shuffles → Jobs & Stages tabs of Spark UI
  - Persisting/caching intermediate results → Storage tab of Spark UI

Apache Spark - Partitions, Jobs, Stages, Tasks, Shuffles (1)

Partition

- A chunk of a dataset.
- Data frames (RDDs) are divided into logical partitions, which are then processed in parallel across the cluster.
- The partitioning scheme defines how the data is distributed across the cluster and can be configured for optimization.
- The number of partitions can significantly affect the performance of Spark applications, as more partitions mean more parallelism but also potentially more overhead from shuffling data across the cluster.

Job

- A (parallel) computation triggered in response to an action (e.g. `count()` or `show()` in the previous example).
- A job can consist of multiple stages.

Apache Spark - Partitions, Jobs, Stages, Tasks, Shuffles (2)

Task

- The smallest unit of work in Spark which corresponds to a set of data processing steps on a single partition.
- Each task will be sent to one executor on a worker node for execution.
- The number of tasks in a stage equals the number of partitions in the RDD or data frame being operated on.
- The effective parallelism is determined by `min(executors * cores, partitions)`.

Stage

- A set of *tasks* that can be executed in parallel (one task per partition).
- Stages are separated by *shuffles*, which are redistributions of data.

Apache Spark - Partitions, Jobs, Stages, Tasks, Shuffles (3)

Shuffle

- Occurs when data needs to be redistributed across different partitions and nodes.
- This typically happens as the result of certain wide transformations (like `groupBy()` and `join()`) that require data from different partitions to be combined.
- Shuffling is a costly operation because it involves network I/O and data (de-)serialization.
- Optimizing the shuffle process can lead to significant performance improvements.

Apache Spark - Partitions, Jobs, Stages, Tasks, Shuffles (4)

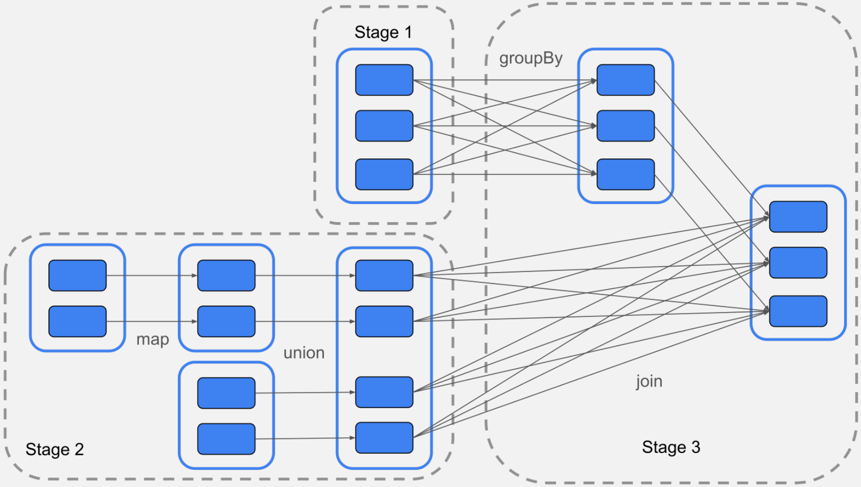


Figure 5: Spark Shuffles and Stages Example

Apache Spark - Persisting Intermediate Results (1)

- Persisting (or caching) is a fundamental concept in Spark to store the results of computations across operations. When a data frame is persisted/cached, Spark keeps the partitions around on the cluster for much faster access the next time it is being used.
- A data frame can be persisted using the `persist()` or `cache()` methods, the latter being a special case of the former.
- `persist()` allows to specify different storage levels, which controls how (serialized vs deserialized) and where (memory vs disk) the data is stored. As such, they allow trading off between memory usage and CPU efficiency.
- Choosing the right storage level depends on your application's requirements for speed and data size.

Apache Spark - Persisting Intermediate Results (2)

See previous chess games example:

```
played_classical_openings = (
    chess_games
    .groupby("Event", "Opening")
    .agg(sf.count("*").alias("Count"))
    .filter(sf.col("Event") == "Classical")
    .persist()  # without persist here, group-by-agg would be executed twice below!
)

print(played_classical_openings.count())  # 1st action
played_classical_openings.sort("Count", ascending=False).show()  # 2nd action
```

Apache Spark - Partitioning & Shuffles

- Shuffling data across the network is costly and can therefore have serious negative performance impacts.
- We can reduce the amount of shuffling needed by *partitioning* the data accordingly.
  - Hash partitioning (`repartition()`)
    - hash-partition the data according to a set of key columns
    - partition assignment is according to `hash(<key_columns>) % <target_number_of_partitions>`
    - can lead to uneven data distribution (aka data skew) if some keys are more frequent than others
  - Range partitioning (`repartitionByRange()`)
    - range-partition the data according to a set of key columns
    - data is sorted according to key columns and split to reach a target number of partitions
    - ideal for operations that require sorted data
- Example: See this gist

Apache Spark - Example

- Dataset: Stack Exchange Data Dump by Stack Exchange Inc., available under CC-BY-SA 4.0 license
- Focus on Stack Overflow network (“Q&A for programmers”)
- Schema information: see this meta post
- Notebook

Apache Spark - Driver & Worker Node Sizing (1)

The type and size of driver and worker nodes one should choose for a Spark cluster very much depends on the workload. The following are some high-level guidelines.

Driver Node

- Usually the driver node does not need as much CPU and memory resources as the worker nodes unless you are `collect`ing large amounts of data to the driver or doing significant driver-side processing (should be avoided if possible).
- Applications with a huge number of tasks and complex DAGs will likely benefit from more CPUs and memory.
- Tip: Start with a driver with 4-8 cores and 8-32GB of memory. Scale up as needed.

Apache Spark - Driver & Worker Node Sizing (2)

Worker Nodes

- Worker nodes need enough memory to process tasks.
- The memory requirement depends on the nature of the tasks and the size of the data partitions (see also `spark.sql.shuffle.partitions`).
- Make sure there's enough memory to handle the data processing without causing excessive garbage collection or spilling to disk, which can slow down processing.
- The number of (total) CPUs impacts how many tasks can be processed in parallel. More CPUs can lead to better parallelism and faster data processing.

Apache Spark - Driver & Worker Node Sizing (3)

- Tip:
  - Start by deciding on a memory-to-core ratio: how much memory do I need per task & data partition. Example with Azure machine sizes:
    - High CPU-to-memory ratio: compute optimized
    - Balanced memory-to-CPU ratio: general purpose
    - High memory-to-CPU ratio: memory optimized
  - Start with workers with 16 cores / 64GB (general purpose) or 16 cores / 128 GB (memory optimized) and choose as many workers as needed
  - Scale up/down and out from there

Apache Spark - Configuration

- Configuring Spark properly is crucial for achieving optimal performance
- The "right" configuration depends on the specific workload (data sizes involved and complexity of processing tasks) and machine/cluster size
- When using a managed Spark service such as Databricks, many of the core configuration settings have appropriate values or sensible defaults
- One of the most important settings that you would want to set depending on your workload is `spark.sql.shuffle.partitions` which configures the number of partitions to use when shuffling data for joins or aggregations (default is 200)

Apache Spark - Configuration: Shuffle Partitions (1)

- Default is 200
- More partitions typically lead to smaller data per partition, which can be beneficial for parallelism but might increase the overhead due to a larger number of tasks
- Considerations
  - Data size
    - For large datasets, increasing the number of partitions can help by distributing the workload more evenly across the cluster.
    - For smaller datasets, reducing the number of partitions can prevent too many small tasks, which can cause overhead and degrade performance.
  - Cluster size: A good starting point is to have 2-3 tasks per CPU core in the cluster.

Apache Spark - Configuration: Shuffle Partitions (2)

Tuning approach

1. Start with the default value and monitor the application's performance.
2. Adjust the number of partitions based on the performance and resource utilization. Incrementally increase or decrease the number and observe the impact.

Apache Spark - Configuration: Adaptive Query Execution

- Adaptive Query Execution (AQE) is an optimization technique in Spark that makes use of the runtime statistics to choose the most efficient query execution plan
- Enabled by default since Spark 3.2.0 (Databricks 7.3 LTS)
- Three major features:
  - coalescing post-shuffle partitions
  - converting sort-merge join to broadcast join
  - skew join optimization
- Databricks only: Spark AQE has a feature called auto-optimize shuffle (`spark.sql.shuffle.partitions=auto`) which can automatically find the right number of shuffle partitions

Apache Spark - Checkpointing (1)

Checkpointing an RDD/DataFrame means ...

- truncating the lineage of an RDD/DataFrame and saving the fully materialized state to reliable (distributed) storage such as HDFS
- any future transformations or actions will use the checkpointed data as the starting point

`checkpoint()` is eager by default and will block until the data has been fully stored on reliable storage. For performance reasons, it typically makes sense to `persist()` the data *before* calling `checkpoint()` to allow faster access to data partitions in subsequent operations.

Apache Spark - Checkpointing (2)

Why would you want to checkpoint?

- Fault tolerance: If part of a computation fails, Spark can recover from the checkpointed state, rather than needing to recompute the data from the beginning. This can significantly reduce the computation time in case of failures, especially for long-running jobs.
- Improves performance
    - Free up resources by discarding previous stages
    - Less work for the query planner & optimizer (especially for very deep computation graphs)
- Particularly useful in iterative algorithms

Apache Spark - Advanced Concepts (1)

- Window functions: operate on a group of rows (referred to as a window) and calculate a return value for each row based on the group of rows (*)
- User-defined functions (UDFs)
    - Extend Spark's built-in capabilities
    - Problem: UDFs are opaque to the query optimizer
    - Two types
        - Scalar UDFs: operate on a single row
        - Pandas UDFs aka Vectorized UDFs: Arrow is used to transfer data to and from Spark; Pandas is used to operate on the data, which allows for vectorized operations

* See alternative implementation of `open_bounty_votes` in this notebook

Apache Spark - Advanced Concepts (2)

- Pandas API on Spark
  - A pandas-equivalent API on Spark
  - A possible migration path from Python/pandas to Spark
- Delta Lake: open-source storage framework
  - ACID transactions
  - Streaming and batch unification
  - Schema enforcement
  - Time travel
  - Upserts and delete

Apache Spark - Don’ts (1)

Don’t ...

- unnecessarily perform actions, i.e. materialize the data (e.g. using `collect()`, `count()`, etc.)
- forget to `persist()` intermediate results that would be expensive to recompute
- forget to `checkpoint()` large computation graphs
- manually loop if you can perform the same operation in a vectorized manner using built-in functions

Apache Spark - Don’ts (2)

Don’t ...

- unnecessarily use UDFs
    1. prefer native Spark capabilities using built-in functions - consult the language-specific API reference!
    2. fall back to Pandas UDFs if necessary
    3. use Scalar UDFs as a last resort
- use notebooks to develop/manage large codebases/data pipelines; develop appropriate Python or R packages with clear interfaces and use them in smaller notebooks instead
    - much easier to manage, especially in teams
    - easier to test using language-specific tooling and best-practice frameworks
    - much easier to integrate in DevOps pipelines

Apache Spark - Dos

Do ...

- avoid data shuffling by partitioning data accordingly
- make use of delta tables and its various optimizations such as auto-compaction, optimized writes, data skipping etc.
- typically prefer long (many rows) over wide (many columns) format
    - long format typically comes with some redundancy (replication of values in a column) which makes it much better suited for (massive) parallelization
    - wide format, especially if very wide (100s of columns), can lead to issues with the query planner and optimizer since it needs to keep track of many columns
- leverage window functions
- write unit & integration tests for your data pipelines and applications!

Thank you!