An Introduction to Apache Spark

Performance Tuning

Different parts of Spark Job to optimize

- Code-level design choices (e.g., RDDs versus DataFrames)
- Data at rest
- Joins
- Aggregations
- Data in flight
- Individual application properties
- Inside of the Java Virtual Machine (JVM) of an executor
- Worker nodes
- Cluster and deployment properties

Implement monitoring and job history tracking

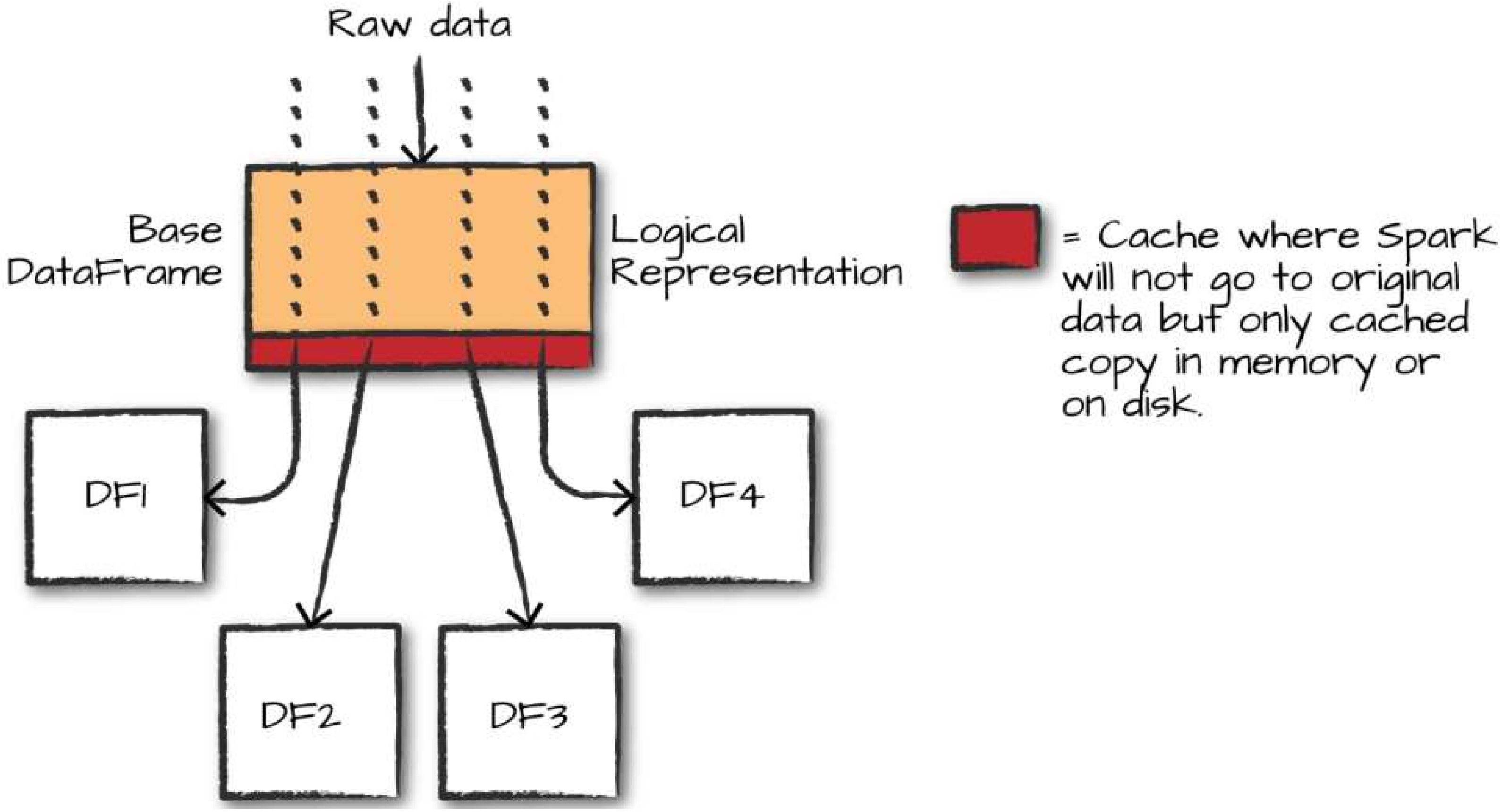
- To figure out how to improve performance
- To know whether you're really improving job performance.

Indirect Performance Enhancements

- Design Choices
  - Scala versus Java versus Python versus R
    - Depending on the use case
- Data at Rest
  - Making sure that you’re storing your data for effective reads later on is absolutely essential to successful big data projects.
- Caching
  - Will place a DataFrame, table, or RDD into temporary storage across the executors in your cluster, and make subsequent reads faster

Caching

- We can avoid having to recompute the original DataFrame (i.e., load and parse the CSV file) many times by adding a line to cache



Caching

- DF1 = spark.read.format("csv")\
- .option("inferSchema", "true")\
- .option("header", "true")\
- .load("/data/flight-data/csv/2015-summary.csv")
- DF2 = DF1.groupBy("DEST_COUNTRY_NAME").count().collect()
- DF3 = DF1.groupBy("ORIGIN_COUNTRY_NAME").count().collect()
- DF4 = DF1.groupBy("count").count().collect()

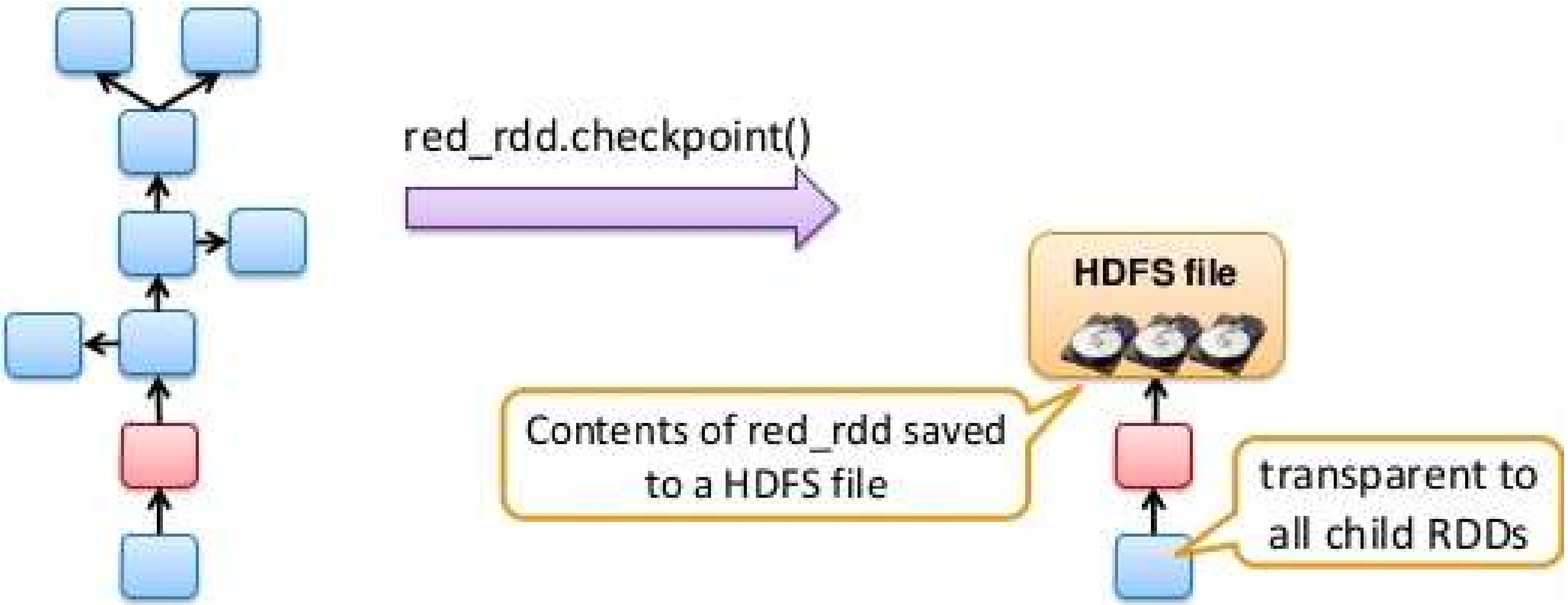
- DF1.cache()
- DF1.count()

Caching

- Now that the data is cached, the commands will be faster, as we can see by running the following code:
  - DF2 = DF1.groupBy("DEST_COUNTRY_NAME").count().collect()
  - DF3 = DF1.groupBy("ORIGIN_COUNTRY_NAME").count().collect()
  - DF4 = DF1.groupBy("count").count().collect()

What is Check-pointing?

- Saving RDD to HDFS to prevent RDD graph from growing too large
- Will persist the transformed RDD or DataFrame forever.



When should I cache or checkpoint?

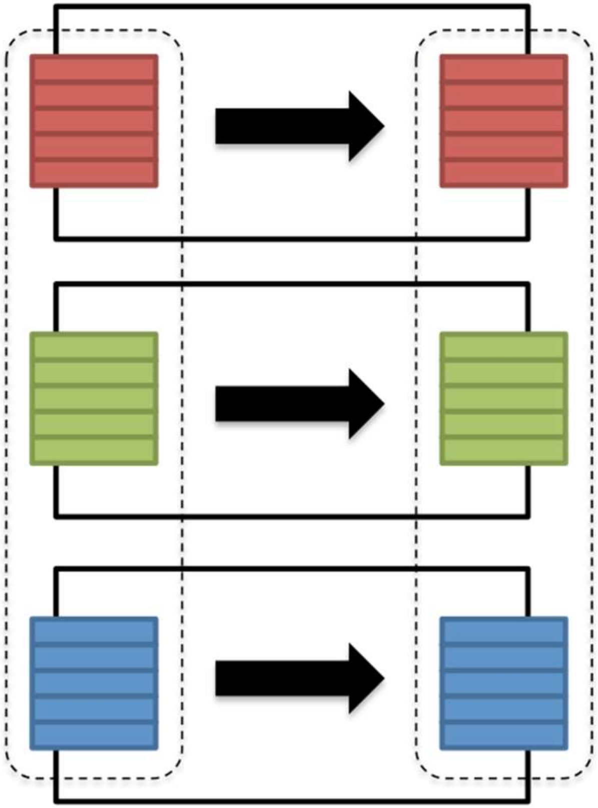
- Determine if the results of a set of transformations can be reused for a very long time or not
  - If the answer is yes, use **checkpointing**
  - If the answer is no, use **caching**

- Example when checkpointing would be preferred:
  - Crunching a RDD or DataFrame of taxes for a previous year
  - They are unlikely to change once calculated so it would be much better to checkpoint and save them forever

Suffling

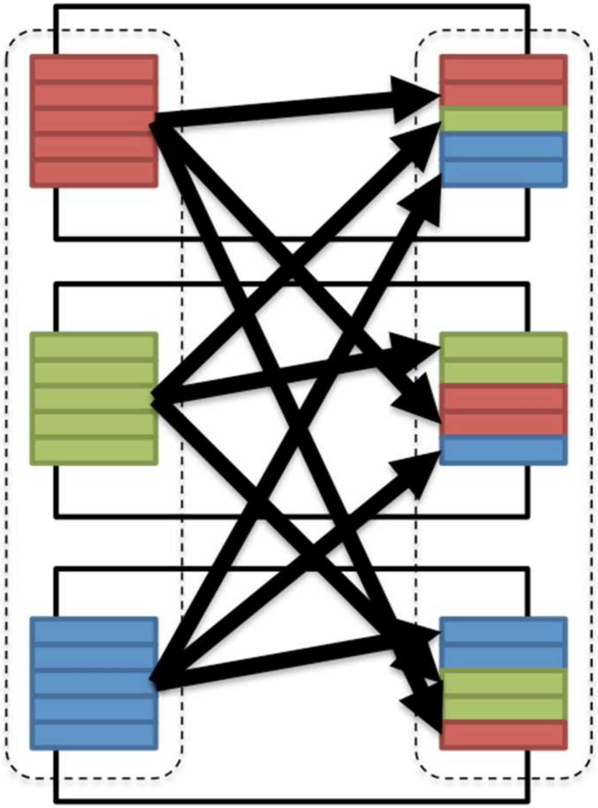
Narrow transformation

- Input and output stays in same partition
- No data movement is needed



Wide transformation

- Input from other partitions are required
- Data shuffling is needed before processing



Minimizing Shuffling for Increased Performance

- Shuffle is an expensive operation
- Each shuffling generates a new stage.
- Here are some tips to reduce shuffle:
  - Use the Spark UI to study the plan to look for opportunity to reduce the shuffle
  - Use the built in aggregateByKey() operator instead of writing your own aggregations.
  - Filter input earlier in the program rather than later.
  - repartition, join, cogroup, and any of the *By or *ByKey transformations can result in shuffles.

Thanks