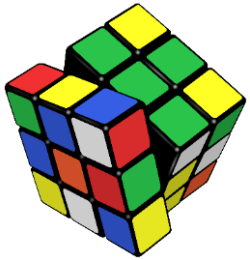
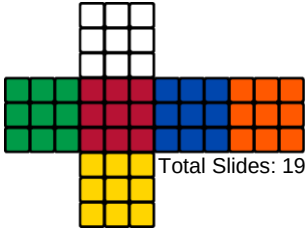


Big Data Engineering For Analytics

Spark Best Practices

Dr LIU Fan
(isslf@nus.edu.sg)
NUS-ISS



Write Tests !!

- Scala unit testing framework
  - ➢Scalatest

    ScalaTest is the most flexible and most popular testing tool in the Scala ecosystem

    https://www.scalatest.org/
  - ➢Scalacheck

    ScalaCheck is a library written in Scala and used for automated property-based testing of Scala or Java programs.

    https://www.scalacheck.org
- Python and Scala Spark testing base

  https://github.com/holdenk/spark-testing-base

Partitioning


```scala
def writeData(inputDf: DataFrame,
              outputFilePath: String,
              format: String,
              partitionCol: String,
              mode: SaveMode = SaveMode.Overwrite)
             (implicit spark: SparkSession): Unit = {
  inputDf
    .write
    .format(format)
    .mode(mode)
    .partitionBy(partitionCol)
    .save(outputFilePath)
}
```

Partitioning

- Choice of partition
  - ➢ Business logic
    - Reduce the working dataset size. E.g.: filter your data by skipping the partitions if they do not meet your condition. A properly selected condition can significant speed up reading and retrieval of the necessary data
    - Repartition before multiple joins. In order to join data, Spark needs data with the same condition on the same partition. This is done by shuffling the data. How to avoid shuffle:
      - ❑ Both dataframe have a common partitioner
      - ❑ One of the dataframe is smaller enough to fit into the memory.
  - ➢ Data
    - Data skew. The join key is not evenly distributed among the partitions.
    - How to solve it:
      - ❑ Repartition data on a more evenly distributed key.
      - ❑ Broadcast the smaller dataframe if possible
      - ❑ Use an additional random key for better distribution
      - ❑ Iterative broadcast join
  - ➢ Environment
    - Cluster core count, memory per executor, etc..

Job tuning

- Cache (with appropriate persistence)
- Data Seriazation
  - JavA serialization: By default, Spark serializes objects using Java's ObjectOutputStream framework, and can work with any class you create that implements java.io.Serializable. Java serialization is flexible but often quite slow, and leads to large serialized formats for many classes.
  - Kryo serialization: Spark can also use the Kryo library to serialize objects more quickly. Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all Serializable types and requires you to *register* the classes you'll use in the program in advance for best performance.
  - We can switch to **Kryo** by initializing our job with *SparkConf* and calling:
  - `conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`
- Shuffle partitions – Configures the number of partitions to use when shuffling data for joins or aggregation
  - `conf.set("spark.sql.shuffle.partitions", "25")` – configures the number of partitions that are used when shuffling data for joins or aggregations.
  - `conf.set("spark.default.parallelism", "25")` – set the number of partitions in RDDs.Note that it only works for raw RDD and is ignored when working with dataframes.
- Broadcast Join
  - `conf.set("spark.sql.autoBroadcastJoinThreshold", "52428800")`
- Use higher level abstractions (Dataframe and Datasets than RDD)
- Use FAIR scheduler

Shuffle partitions

- Shuffle partitions are partitions that are used at data shuffle for wide transformation. The parameter that controls the parallelism that results from a shuffle is `spark.sql.shuffle.partitions`
- How to choose the number of shuffle partitions:
  - On one hand, when you have too much data and too few partitions, it causes fewer tasks to be processed in excutors, it will increase the load on each individual executor and often leads to memeory error.
  - On the other hand, when you dealing with less amount of data, you should typically reduce the shuffle partitions otherwise you will end up with many partitioned files with a fewer number of records in each partition, which results in running many tasks with lesser data to process.
  - Getting a right size of the shuffle partition is always tricky and takes many runs with different value to achieve the optimized number. This is one of the key property to look for when you have performance issues on Spark jobs.

Broadcast join

- Broadcast join can be very efficient for joins between a large table with relatively small tables
- Broadcast joins are easier to run on a cluster.
- Spark can “broadcast” a small DataFrame by sending all the data in that small DataFrame to all nodes in the cluster. After the small DataFrame is broadcasted, Spark can perform a join without shuffling any of the data in the large DataFrame.

FAIR scheduler

- Any action in Spark is a Job
- Jobs are scheduled in FIFO pool – it means first defined job will get the priority for all available resources.
- By default, Spark's internal scheduler runs jobs in FIFO fashion
- The FAIR scheduler supports the grouping of jobs into pools. It also allows setting different scheduling options (e.g. weight) for each pool.
- FAIR scheduler mode is a good way to optimize the execution time of multiple jobs inside one Apache Spark program. Unlike FIFO mode, it shares the resources between tasks and therefore, do not penalize short jobs by the resources lock caused by the long-running jobs

FAIR scheduler

- Both FIFO and FAIR are configurable.
- The scheduling method is set in *spark.scheduler.mode*
- The pools are defined with *sparkContext.setLocalProperty*

1

fair-scheduler.xml

```xml
<?xml version="1.0"?>

<allocations>
    <pool name="fair_pool">
        <schedulingMode>FAIR</schedulingMode>
        <weight>2</weight>
        <minShare>4</minShare>
    </pool>
</allocations>
```

2

```
conf.
...
.set("spark.scheduler.mode", "FAIR")
.set("spark.scheduler.allocation.file", "/Users/nus1/fair-scheduler.xml")
```

3

```
spark.sparkContext.setLocalProperty("spark.scheduler.pool", "fair_pool")
```

Watch your UI

- Duration - Percentile
- Number of executors running - YARN may not allocate enough resources
- Check out fraction cached
- Look are DAG in SQL tab

Spark Jobs (?)

User: isslf
Total Uptime: 8 s
Scheduling Mode: FIFO
Completed Jobs: 3

▸ Event Timeline

▾ Completed Jobs (3)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2021/01/21 15:16:49	84 ms	1/1	1/1
1	save at NativeMethodAccessorImpl.java:0 save at NativeMethodAccessorImpl.java:0	2021/01/21 15:16:48	0.8 s	1/1	1/1
0	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/01/21 15:16:47	0.3 s	1/1	1/1

Data formats

- Use correct formats for the job - Row/Columnar
- Use Arrow when using Pandas with Spark

UDFs

- UDFs are common but avoid (try your best) to use the UDFs inside Spark `(org.apache.spark.sql.functions._)`

```
def replaceCharUdf(map: Map[String, String]) = udf(
  (source: String) => {
    map.iterator.foldLeft(source) { case (src, (find, replace)) =>
      src.replace(find, replace)
    }
  }
)

val generateUUID = udf(
  () => {
    s"${UUID.randomUUID().toString}_${System.nanoTime()}"
  }
)
```

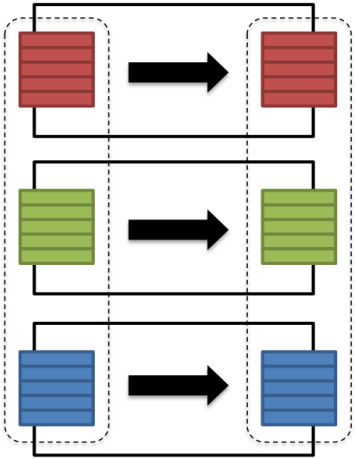
UDFs


```
import edu.nus.bd.config.PipelineConfig.DataColumn
import edu.nus.bd.ingest.models.ErrorModels.DataError
import edu.nus.bd.ingest.stages.base.DataStage
import org.apache.spark.sql.functions._
import org.apache.spark.sql.{DataFrame, Dataset, Encoder, SparkSession}
import edu.nus.bd.ingest.StageConstants._
import edu.nus.bd.ingest.UDFs.generateUUID


class AddRowKeyStage(dataCols: List[DataColumn])
                    (implicit spark: SparkSession, encoder: Encoder[DataError])
  extends DataStage[DataFrame] {

  override val stage: String = getClass.getSimpleName

  def apply(errors: Dataset[DataError], data: DataFrame): (Dataset[DataError], DataFrame) = {
    val colOrder = RowKey +: dataCols.map(_.name)
    val withRowKeyDf = data.withColumn(RowKey, generateUUID()).cache()
    val returnDf = withRowKeyDf.select(colOrder.map(col): _*)
    (spark.emptyDataset[DataError], returnDf)
  }
}
```

Shuffles are expensive

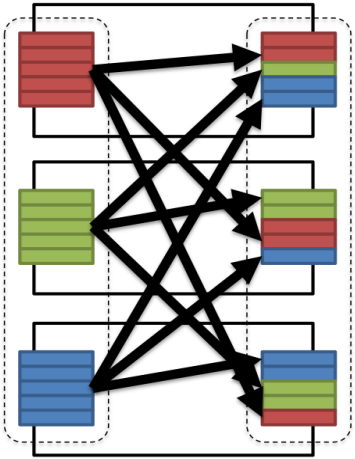
Narrow transformation

- 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



Source : https://blog.knoldus.com/rdds-in-apache-spark/

Narrow and Wide transformations

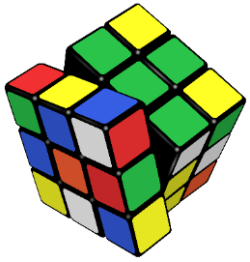
Narrow transformation	Wide Transformation
Map	GroupByKey
FlatMap	ReduceByKey
Filter	Join
Sample	Distinct
Union	Repartition

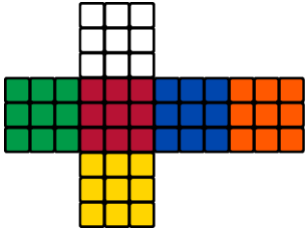
References

- Spark Definitive Guide, Bill Chambers et al
- Cloudera - Tuning your Spark job
- https://atlas.apache.org/#/
- Spline: Spark Lineage, not only for the Banking Industry, Scherbaum, Jan, 2018 IEEE International Conference on Big Data and Smart Computing (BigComp)
- https://sparkbyexamples.com/spark/spark-performance-tuning/

Recommended books

- Spark: The Definitive Guide
- Learning Spark: Lightning-Fast Data Analytics
- Spark in Action, 2nd Edition
- Machine Learning with Apache Spark Quick Start Guide
- Data Analytics with Spark Using Python
- Apache Spark Quick Start Guide





Appendix

Review: Apache Spark Work Implamentation Overview

Cluster	A cluster is a group of JVMs (nodes) connected by the network, each of which runs Spark, either in Driver or Worker roles
Driver	The **Driver** is one of the nodes in the **Cluster.** It plays the role of a master node in the Spark cluster
Executor	**Executors** are JVMs that run on Worker nodes. These are the JVMs that actually run **Tasks** on data **Partitions**
Job	A Job is a sequence of Stages, triggered by an Action such as *.count(), foreachRdd(), collect(), read()* or *write()*
Stages	A **Stage** is a sequence of **Tasks** that can all be run together, **in parallel**, without a shuffle. For example: using *.read* to read a file from disk, then runnning *.map* and *.filter* can all be done without a shuffle, so it can fit in a single stage.
Task	A **Task** is a single operation (*.map* or *.filter*) applied to a single **Partition.** Each **Task** is executed as a single thread in an **Executor**! If your dataset has 2 **Partitions**, an operation such as a filter() will trigger 2 **Tasks**, one for each **Partition**. The number of Tasks in a Stage also depends upon the number of Partitions your dataset have.
Shuffle	A **Shuffle** refers to an operation where data is *re-partitioned* across a **Cluster**. *join* and any operation that ends with *ByKey* will trigger a **Shuffle**
Partition	A **Partition** is a logical chunk of your RDD/Dataset