joins.pdf

About different join strategies to Join Parquet File 1 with 10000 events/sec and File 2 which is a static reference table

Section 1 : Pre-processing

Before discussing the join solution in the next section, this section is on pre-processing

-(**preprocessing** assumed completed) Check for null values / empty string in File 1 for item_name, replace with string "UNKNOWN"

- (**preprocessing** assumed completed) Check for null values in File 1 **geo-location^**, list down other rows that has the same video_camera_oid, but geo-location is not null and replace null with these other rows' geo-location (assuming video-camera-oid is globally unique)

[Sample Error handling]

^if all the geo-location rows for this video_camera_oid is null, replace File 1 geo-location with "-1", and

add a new row in File 2 with :

geographical_location_oid as "-1" and

geographical_location as *geo-loc description is undefined for the camera-id " ${video-camera-id}"*, As an example the description of this geo-location field would be "geo-loc description is undefined for camera-id 2234"

-Analyse the 10k events/sec in File 1, for data skew (e.g. in 1 geo location), aggregate by item_name first

[if there is skew, tune Spark at SparkSession builder first,

leveraging on Spark Adaptive Query Execution (**AQE**), pls see explanation in **Annex A]** from

val sparkS=SparkSession.builder().config(sparkContext.getConf).getOrCreate()

to

```
val sparkS=SparkSession.builder()
  .config(sparkContext.getConf)
  .config("spark.sql.adaptive.enabled", "true") //ensure that it is on (by default it is)
  .config("spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes", "512MB")
.  getOrCreate()
```

Finally, add a **salt** to the detection event dataset(File 1) w.r.t. key field to split this data skew to 10 parts, create a new salt column (holding this value) and used as the join-key. An example to illustration is :

The **100k** rows in File1 of geographical_location_oid of values “120798”
is now **split into 10 parts** with salt column holding values ranging from
“120798-0”, “120798-1”, and so on till “120798-9”

Each reference table rows (File2) are also **replicated** to hold these values, in the salt column key added, for matching (increasing the 10k rows 10 times, i.e. becomes 100k rows). The .join is then done with this salt column key

(Please see code snippets in **Annex B**)

Section 2 : Join Strategy

Solution for join :

A. For the current geo-location reference table, File 2 (considered small with 10k rows and 2 columns), detection-event File 1 is much larger

The quickest method is to use Spark RDD .join() which Spark Adaptive Query Execution (AQE) defaults to broadcast join automatically, when within the threshold of 10MB. Spark will broadcast this reference table (dataset B, i.e. File 2) across all the executors

We can increase the size for the auto-broadcast-join from default threshold of 10MB to for e.g. to 100MB where appropriate:

*spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 100 * 1024 * 1024)*

We can also encourage the broadcast join with the pseduo code

*val joinedDf = File1largerDf.join(**broadcast**(File2smallerDf), "join_key")*

B. For future proofing the solution (taking into consideration for the 2nd part of the assessment, a proposal is to use dataset joinWidth method, after creating custom partition, while aiming to reduce shuffling and increase processing speed

-Re-partition File1 (i.e. dataset1) by geo-location^ (called ***repartitionedFile1***)
-Re-partition File2 (i.e. dataset2) by geo-location (called ***repartitionedFile2***)
(using spark Dataset *repartition* method)
-joinWith on geo-location (note that inner join can be used as geo-location null values are replaced above (*assume completed successfully during pre-processing*)

*^full name of field is called **geographical_location_oid***
*scala case class for File1 is called **Events**, and for File 2 is **GeoLoc**)*

To reduce shuffle and speed up processing further
-Get the number of executors used, and stored in variable *numOfExecutor*

Create a custom partitioner using the sparksession (variable called *sparkS*), *key-by* geo-location (variable called *geographical_location_oid)*
(*variable sparkS, after reading the RDD from the parquet files*), overriding *numPartitions* and *getPartition* method in *Partitioner* class, code snippets below

```
.keyBy(_.geographical_location_oid).partitionBy (new Partitioner {
        override def numPartitions : Int = numOfExecutor
        ... )...
override def getPartition (key: Any): Int = {
        key.hashCode % numOfExecutor }
        ...}...)
```

*Val result: Dataset [(**Events**, **GeoLoc**)]*
*= **repartitionedFile1**.joinWith (**repartitionedFile2**,*
Events("geographical_location_oid")===GeoLoc("geographical_location_oid"),"inner")

This join strategy adopts a pre-partitioned approach using *joinWith* operator of Dataset instead of relying on the behaviour of Spark AQE for *Broadcast-Join* or the *Sort-Merge-Join* during runtime via .join method.

In summary, the Spark join strategies are as follows:

Strategy for Equi-joins i.e. the joins are done on the exact matches of the join-key.

1. **Broadcast-Join (aka Broadcast-Hash-Join)**, where broadcast variables are created. This is good for small reference table dataset (File 2) as broadcast variable. This entire reference table dataset is broadcasted so that all the partitions have access and can perform a local hash join. This strategy is **suitable** and fast for joining a large detection-events table (File1) with a **small reference table** (File2), noting that the Spark autoBroadcastJoinThreshold is 10MB, if not set.

This may not be suitable when the File 2 reference table grows(or it can be foreseen that it will grow), during production ops deployment, as it costed resources issues in the executers when the File 2 expands in future, as more cameras are deployed (i.e. future proofing the solution).

2. **Sort-Merge-Join**, where during join, Spark will shuffle the data on the same join-key to the same partition ensuring data of matching join-key resides together. A sort based on the join-key inside the same partition is done, before these sorted partitions from both sides are merged with records from same matching join-key joined.

This is done when Spark AQE cannot use broadcast join, due to both datasets (File1 and File2) are large. As the datasets (File1 & File2) are repartitioned, i.e. shuffled based on the join key, ensuring that matching keys reside in the same partition, plus the data within each partition is sorted based on the join key, leading to **slower runtime performing the shuffle and sort**. This is **suitable for large datasets that are not sorted**, since it performs the sorting at runtime. This is also good for large resilient joins that would not fail due to memory limits during runtime. The join key must be sortable as well.

3. **Storage-Partitioned-Join**. The pre-requisite for this is during the write storage process there is a check to ensure the tables are appropriately partitioned and sorted, i.e. tables involved in the join are already bucketed and sorted by the join-key during storage. Spark directly merges the corresponding buckets/partitions, eliminating the need for a runtime shuffle and sort. This requires careful pre-planning and data preparation **during the write-process** to ensure the tables are appropriately partitioned (pre-bucketed) and sorted by the join key.

This is **suitable** for situations where the use-case requires **frequent joining** of the datasets, and hence justify for the time spent to plan and optimize to have the pre-organized datasets during their initial storage. This avoid/minimize shuffle & sort during execution, hence **faster runtime**.

4. **Shuffle-Hash-Join**

This is used when the smaller reference table (File2) is not small enough to broadcast, and the large detection-events table (File1) remains bigger than File2 by **at least three times**.

Spark **redistributes** both datasets so that all rows with the same join-key end up residing on the same partition (shuffling). Spark takes the smaller of the two redistributed datasets (File2) and builds a lookup table (hash table) in its memory. During the join, the larger dataset (File1) uses the hash table to find the matching rows. This is **faster than Sort-Merge-Join at runtime** because it **avoids the sorting step as well**. This is **suitable** for the situation of File1 and File2 having a difference of at least three times in size.

Spark allows a preference to set
sparkSession.conf.set("spark.sql.join.preferSortMergeJoin", "false")

However, setting the above does not guarantee a Shuffle-Hash-Join. Spark will only choose it if the smaller dataset (File2), where the hash table is built from, is at least three times smaller than the other dataset (File1). Another consideration is that the join keys are not sortable.

The **above join strategies are for equi-joins**

The strategies for non-equi joins are below:

5a Broadcast-Nested-Loop-Join

This join uses the comparison operators other than equals, e.g. "less than" (<), "greater than" (>), or "not equal to" (!=), Spark need to perform a brute-force check/comparison of every row against every other row, a process called a nested-loop join, which is **slow and inefficient**, especially for larger datasets.

The smaller dataset (File2) is broadcast to all partitions (i.e. must be small enough to broadcast), but because it cannot use the hash table (as in the case of Broadcast-Join), for each row in the larger dataset (File1), it has to perform a nested loop over the entire broadcasted File2, hence the name.

This is **suitable** for cases where the join is complex, such that Spark cannot use hash or sort-based **strategies**.

5b Cartesian-Product-Join

This join is a full permutation where every row of detection-event-dataset(File1) is paired with every row of reference-table-dataset(File2). For e.g., if File1 has 10,000 rows and File2 has 1000) rows, the result dataset after the join would be 10,000,000 rows (10k multiply by 1k)

This is **suitable** when there is no join condition specified, or when a cross-join is explicitly set, and for small datasets. Larger datasets will have to take note of the consideration of potential out-of-memory issues and network resource usage. This is because this join shuffles and replicates partitions from both datasets to form a complete pairing of all rows.

Annex A: Leverage on Spark Adaptive Query Execution (AQE)

```
val sparkS=SparkSession.builder()
  .config(sparkContext.getConf)
  .config("spark.sql.adaptive.enabled", "true") //ensure that it is on (by default it is)
  .config("spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes", "512MB")
.  getOrCreate()
```

skewedPartitionThresholdInBytes can be set as “1GB” if required (even larger dataset for production ops)

The above two settings aim to leverage on Spark Adaptive Query Execution (AQE) that dynamically re-optimizes the query plan at runtime based on the actual data shuffle statistics, taking into consideration of the join-key (or composite-join-key).

After the initial shuffle, AQE detected a partition is skewed (having significantly more data than the median).

AQE will then split this large partition into smaller sub-partitions and replicate the necessary rows from the smaller side of the join to each of those sub-partitions. This allows the skewed key to be processed in parallel across multiple tasks instead of bottlenecking in a single one.

Annex B: **Adding Salt to handle data skew, after spark tuning to AQE**

```
// Add Salt to the skewed dataset, a salt range of 0-9, the value of the key field ("geo-loc-value")
// becomes (" geo-loc-value-0"), (" geo-loc-value-1"), and so on till (" geo-loc-value-9")
val saltedDetectionEvents = detectionEvents.withColumn("salt", (rand() * 10).cast("int"))
val saltedDetectionEventsWithKey = saltedDetectionEvents.withColumn(
  "salted_key",
  concat(col("geographical_location_oid"), lit("-"), col("salt"))
)

// Add to the smaller reference table all the salt values, i.e. postfix “-0”, “-1”, and so on till “-9”
// this reference table size will increase by 10 times
// i.e. each geo-loc is replicated 10 times with postfix  -0 to -9
// i.e. becomes (" geo-loc-value-0"), (" geo-loc-value-1"), and so on till (" geo-loc-value-9")
// This is called the “salted_key” which will be the join key
val saltRange = (0 until 10).toList
```

```
val replicatedReferenceTable = referenceTable.crossJoin(
  sparkS.createDataset(saltRange).withColumnRenamed("value", "salt")
)
val replicatedReferenceTableWithKey = replicatedReferenceTable.withColumn(
  "salted_key",
  concat(col(" eographical_location_oid "), lit("_"), col("salt"))
)

// Thereafter, perform join on salted key, so that load is spread across executors when doing .join
val joinedDF = saltedDetectionEventsWithKey.join(
  replicatedReferenceTableWithKey,
  "salted_key"
)
 ....
// Finally, aggregate the result and drop the salt column
val finalResult = joinedDF.drop("salt").groupBy(...).agg(...)
```