TF-IDF with Spark – Optimization Report

Import Original Dataset to HDFS

The original dataset is 9 GB total over 72 arrow files, each 128 MB. Since there is no easy way to import the data from source location to HDFS, I installed local Spark in the source server then use the *ingest_dataset.py* script to transfer the source files to HDFS. The script imported Huggingface's dataset library and read each file sequentially then transform them into Spark Dataframe that is then saved into HDFS in parquet format. The dataset becomes only about 5 GB in total.

Cluster Resource

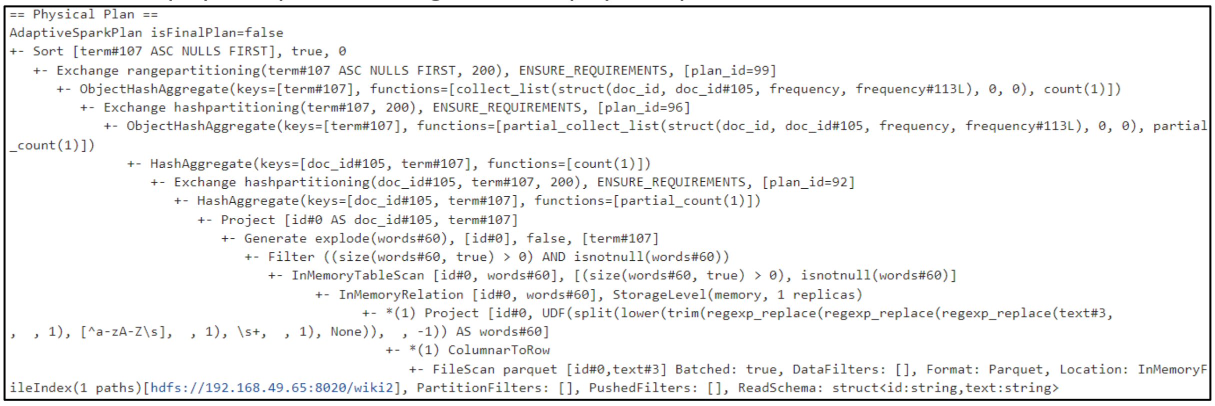
The Spark Cluster that is used is run on 3 VMs, 1 for master and 2 for worker. The master has 2 GB memory meanwhile each worker has 4 GB memory each. All 3 VMs have 2 cores each. 2 GB executor memory is used on each worker with 2 cores being used.

Original Solution

For starters, I implemented the TF-IDF DAG in the most natural way I can think of, with below as the remaining of the DAG after the data is cleaned:

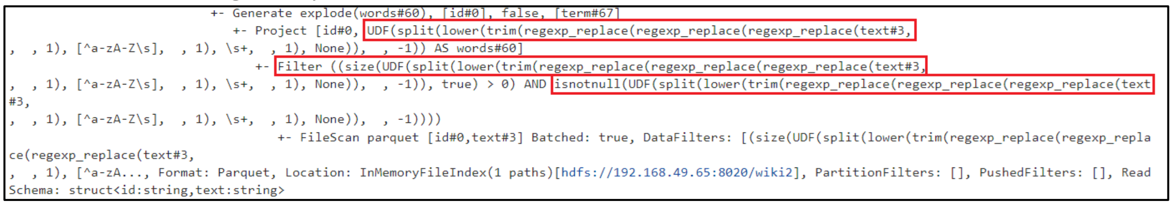
```
df_cached = df_rm_stopword.select('id', 'words').persist(StorageLevel.MEMORY_ONLY)

df_explode = df_cached.select(
       col("id").alias('doc_id'), explode(col("words")).alias('term'))

df_term_doc = df_explode.groupBy("doc_id", "term")
       .agg(count("*").alias("frequency"))

df_term_doc.createOrReplaceTempView("v_word_count")

df_termdoc = spark.sql(f"""
with cte as (
SELECT term, COLLECT_LIST(STRUCT(doc_id, frequency)) AS term_frequency_array,
log({df_count} / count(*)) / log(10) as idf
FROM v_word_count
GROUP BY term
)
SELECT term, TRANSFORM(term_frequency_array, x -> STRUCT(x.doc_id as doc_id,
x.frequency * idf as tfidf)) as term_tfidf_array
FROM cte
""")

df_termdoc_ordered = df_termdoc.sort("term")
```

Below is the physical plan that is generated physical plan:



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [term#107 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(term#107 ASC NULLS FIRST, 200), ENSURE_REQUIREMENTS, [plan_id=99]
      +- ObjectHashAggregate(keys=[term#107], functions=[collect_list(struct(doc_id, doc_id#105, frequency, frequency#113L), 0, 0), count(1)])
         +- Exchange hashpartitioning(term#107, 200), ENSURE_REQUIREMENTS, [plan_id=96]
            +- ObjectHashAggregate(keys=[term#107], functions=[partial_collect_list(struct(doc_id, doc_id#105, frequency, frequency#113L), 0, 0), partial
_count(1)])
               +- HashAggregate(keys=[doc_id#105, term#107], functions=[count(1)])
                  +- Exchange hashpartitioning(doc_id#105, term#107, 200), ENSURE_REQUIREMENTS, [plan_id=92]
                     +- HashAggregate(keys=[doc_id#105, term#107], functions=[partial_count(1)])
                        +- Project [id#0 AS doc_id#105, term#107]
                           +- Generate explode(words#60), [id#0], false, [term#107]
                              +- Filter ((size(words#60, true) > 0) AND isnotnull(words#60))
                                 +- InMemoryTableScan [id#0, words#60], [(size(words#60, true) > 0), isnotnull(words#60)]
                                       +- InMemoryRelation [id#0, words#60], StorageLevel(memory, 1 replicas)
                                             +- *(1) Project [id#0, UDF(split(lower(trim(regexp_replace(regexp_replace(regexp_replace(text#3,
,  , 1), [^a-zA-Z\s],  , 1), \s+,  , 1), None)),  , -1)) AS words#60]
                                                +- *(1) ColumnarToRow
                                                   +- FileScan parquet [id#0,text#3] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryF
ileIndex(1 paths)[hdfs://192.168.49.65:8020/wiki2], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:string,text:string>
```

The execution is separated to 5 stages:

1. Scan HDFS > Clean > Cache > Explode > PartialGroupBy (doc_id, term):
   In this stage, cache is crucial because the optimizer implements some filters that made the cleaning functions got executed multiple times. Below is what the physical plan looks like before caching is implemented:
   

```
                              +- Generate explode(words#60), [id#0], false, [term#67]
                                 +- Project [id#0, UDF(split(lower(trim(regexp_replace(regexp_replace(regexp_replace(text#3,
,  , 1), [^a-zA-Z\s],  , 1), \s+,  , 1), None)),  , -1)) AS words#60]
                                    +- Filter ((size(UDF(split(lower(trim(regexp_replace(regexp_replace(regexp_replace(text#3,
,  , 1), [^a-zA-Z\s],  , 1), \s+,  , 1), None)),  , -1)), true) > 0) AND isnotnull(UDF(split(lower(trim(regexp_replace(regexp_replace(regexp_replace(text
#3,
,  , 1), [^a-zA-Z\s],  , 1), \s+,  , 1), None)),  , -1))))
                                       +- FileScan parquet [id#0,text#3] Batched: true, DataFilters: [(size(UDF(split(lower(trim(regexp_replace(regexp_repla
ce(regexp_replace(text#3,
,  , 1), [^a-zA..., Format: Parquet, Location: InMemoryFileIndex(1 paths)[hdfs://192.168.49.65:8020/wiki2], PartitionFilters: [], PushedFilters: [], Read
Schema: struct<id:string,text:string>
```

   This results in the first stage running for 22 minutes, meanwhile caching reduces this stage into running for only 12 minutes. Implementing the cache in this way didn't cause out of memory error because Spark implemented LRU for in memory caching, which means when the execution memory needs to occupy some memory space that is currently used for caching, the first cache to be removed will be the least recently used one, therefore the older partition in the cache will always be the first one to be removed. I also set spark.memory.storageFraction configuration to 0.05 to make execution memory be able to evict almost everything from the cached RDD if needed since all the old cache won't be needed anyway.
2. Read Shuffle > GroupBy (doc_id, term) > PartialGroupBy (term):
   This stage took about 5 minutes to complete.
3. Read Shuffle > GroupBy (term) > Sample Sort:
   Spark sort task will have 2 stages, the first one is to sample the data on each partition to build the rule of which key goes to which partition. To calculate this, spark need to perform the whole operation in the current stage, or to the cached result of it.
4. Read Shuffle > GroupBy (term):
   Spark will now perform the GroupBy again, but this time, worker will know to which shuffle partition it will write the output. Both stage 3 & 4 take similar amount of time, with 4th task being a bit longer, but combined they average around 5 minutes.
5. Read Shuffle > Sort > Action:
   This task just sort each shuffle partition, then perform action on the result Dataframe. In this experiment in particular my action is always write all output partition to HDFS. This stage averages in 1.5 minute.

All the stages took about 23.5 minutes to complete.

Inefficiency of the Original Solution

The biggest issue that I find in the first solution is the explode + group by operation that is performed because the group by caused shuffle, meanwhile when performing term count on each document, the whole document is already within the same partition, which means shuffling shouldn't be needed. One additional issue is explode operation made the Dataframe a lot bigger, the total input read is 5 GB while the shuffle write after the explode is 7.8 GB. To be fair, the other optimization that I did here also have about the same or even more shuffle write size, but all of them have at least 1 less shuffle operation.

Python UDF

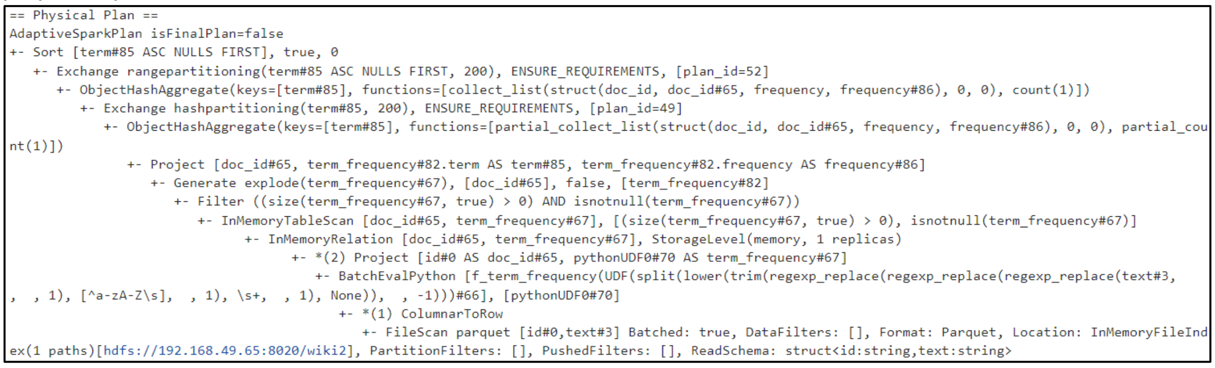
To prevent the unnecessary shuffle operation due to the explode + group by operation, I tried to implement a UDF:

```
def f_term_frequency(words_list):
    word_count = len(words_list)
    word_count_dict = {}
    for word in words_list:
        if word in word_count_dict:
            word_count_dict[word] += 1
        else:
            word_count_dict[word] = 1
    return [{"term": word, "frequency": count / word_count} for word, count in
word_count_dict.items()]
```

The result after the UDF operation is also cached to avoid multiple calculations:

```
udf_term_frequency = udf(f_term_frequency, ArrayType(StructType([
    StructField("term", StringType()),
    StructField("frequency", DoubleType())
])))

df_term_frequency = df_rm_stopword.select(col("id").alias("doc_id"),
udf_term_frequency("words").alias("term_frequency")).persist(StorageLevel.MEMORY_ON
LY)
```

The function above is then followed by explode then group by term, which skips the group by term & doc_id operation. This results in this solution having 1 less shuffle. Below is the physical plan:



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [term#85 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(term#85 ASC NULLS FIRST, 200), ENSURE_REQUIREMENTS, [plan_id=52]
      +- ObjectHashAggregate(keys=[term#85], functions=[collect_list(struct(doc_id, doc_id#65, frequency, frequency#86), 0, 0), count(1)])
         +- Exchange hashpartitioning(term#85, 200), ENSURE_REQUIREMENTS, [plan_id=49]
            +- ObjectHashAggregate(keys=[term#85], functions=[partial_collect_list(struct(doc_id, doc_id#65, frequency, frequency#86), 0, 0), partial_cou
nt(1)])
               +- Project [doc_id#65, term_frequency#82.term AS term#85, term_frequency#82.frequency AS frequency#86]
                  +- Generate explode(term_frequency#67), [doc_id#65], false, [term_frequency#82]
                     +- Filter ((size(term_frequency#67, true) > 0) AND isnotnull(term_frequency#67))
                        +- InMemoryTableScan [doc_id#65, term_frequency#67], [(size(term_frequency#67, true) > 0), isnotnull(term_frequency#67)]
                              +- InMemoryRelation [doc_id#65, term_frequency#67], StorageLevel(memory, 1 replicas)
                                    +- *(2) Project [id#0 AS doc_id#65, pythonUDF0#70 AS term_frequency#67]
                                       +- BatchEvalPython [f_term_frequency(UDF(split(lower(trim(regexp_replace(regexp_replace(regexp_replace(text#3,
,  , 1), [^a-zA-Z\s],  , 1), \s+,  , 1), None)),  , -1)))#66], [pythonUDF0#70]
                                          +- *(1) ColumnarToRow
                                             +- FileScan parquet [id#0,text#3] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileInd
ex(1 paths)[hdfs://192.168.49.65:8020/wiki2], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:string,text:string>
```

From the above physical plan, this solution combines stage 1 & 2 of the original solution: Scan HDFS > Clean > TF > Cache > Explode > PartialGroupBy (term). Unfortunately, this solution is actually slower than the original one at 21 minutes on the first stage, while the original has combined stage 1 & 2 at 17 minutes. The later stages are the exact same at 6.5 minutes, which results in 27.5 minutes total runtime. Note that this experiment is done on VirtualBox VMs within the same laptop host, which means the network latency is about non-existent. The speed of shuffling in comparison to Python UDF depends on the network latency, shuffle size, disk write speed, CPU speed, and the algorithm performed on both methods. High CPU speed favours Python UDF while low latency, low shuffle size, and high disk write speed favours shuffling.

Why Python UDF is Slow

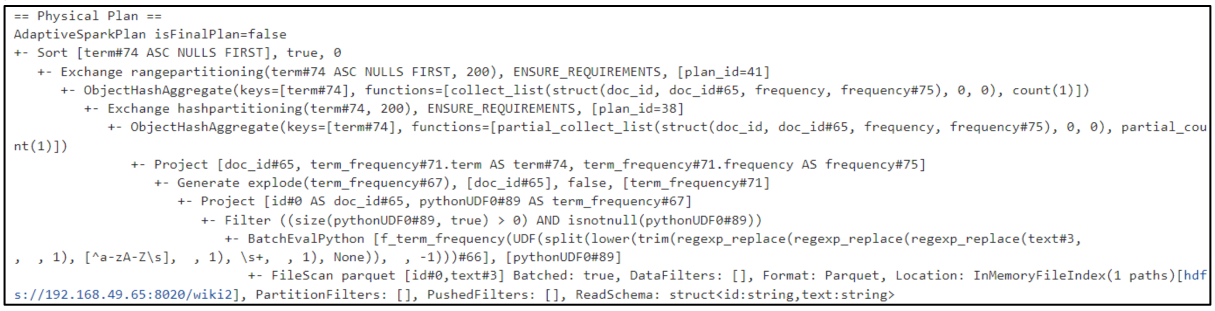
Even when it is still being executed in memory, Python UDF itself is slow because it is run on Python, which is a very slow interpreted language to begin with. More to that, to perform the UDF calculation, Dataframe needs to be transferred from JVM to Python then back to JVM, which means there will be serialization & deserialization happening.

Python UDF – Nondeterministic

One other Python UDF approach which I tried is using Nondeterministic instead of the default caching that needs to be performed to avoid same computation:

```
udf_term_frequency = udf(f_term_frequency, ArrayType(StructType([
    StructField("term", StringType()),
    StructField("frequency", DoubleType())
]))).asNondeterministic()

df_term_frequency = df_rm_stopword.select(col("id").alias("doc_id"),
udf_term_frequency("words").alias("term_frequency"))
```

Above is the only change needed, by adding *asNondeterministic()* at the end of the UDF definition & removing the caching on the *df_term_frequency*. Below is the created physical plan:



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [term#74 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(term#74 ASC NULLS FIRST, 200), ENSURE_REQUIREMENTS, [plan_id=41]
      +- ObjectHashAggregate(keys=[term#74], functions=[collect_list(struct(doc_id, doc_id#65, frequency, frequency#75), 0, 0), count(1)])
         +- Exchange hashpartitioning(term#74, 200), ENSURE_REQUIREMENTS, [plan_id=38]
            +- ObjectHashAggregate(keys=[term#74], functions=[partial_collect_list(struct(doc_id, doc_id#65, frequency, frequency#75), 0, 0), partial_cou
nt(1)])
               +- Project [doc_id#65, term_frequency#71.term AS term#74, term_frequency#71.frequency AS frequency#75]
                  +- Generate explode(term_frequency#67), [doc_id#65], false, [term_frequency#71]
                     +- Project [id#0 AS doc_id#65, pythonUDF0#89 AS term_frequency#67]
                        +- Filter ((size(pythonUDF0#89, true) > 0) AND isnotnull(pythonUDF0#89))
                           +- BatchEvalPython [f_term_frequency(UDF(split(lower(trim(regexp_replace(regexp_replace(regexp_replace(text#3,
,  , 1), [^a-zA-Z\s],  , 1), \s+,  , 1), None)),  , -1)))#66], [pythonUDF0#89]
                              +- FileScan parquet [id#0,text#3] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex(1 paths)[hdf
s://192.168.49.65:8020/wiki2], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:string,text:string>
```

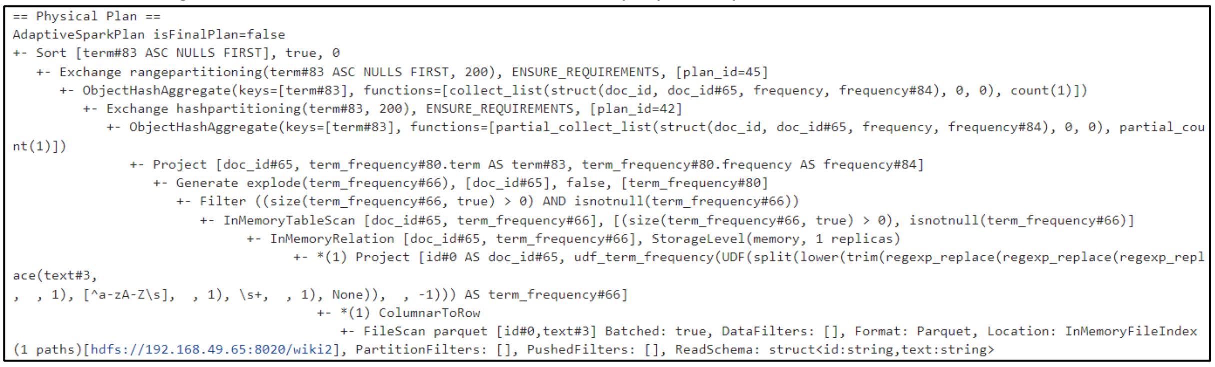
The reason why there isn't any recalculation of the UDF is because nondeterministic means the function isn't guaranteed to produce the same output for the same input that it is given every different time it is called. An easy example to a nondeterministic function is the one that involves a random number. However, this isn't actually an ideal approach because nondeterministic function prevents Spark's optimizer from doing any optimization around it such as predicate pushdown. The result is better than the deterministic one at 19 minutes at first stage, which is still slower compared to the original approach. This approach is probably faster than deterministic Python UDF because of the serialization that occurred at the caching and other operations that happens during caching. The final total execution time of this solution is 25.5 minutes.

Scala UDF

Next approach I tried is using Scala UDF instead of Python, which should be by default faster due to being executed directly on the JVM:

```
class TermFrequencyUDF extends UDF1[Seq[String], Seq[(String, Double)]] {
  override def call(wordsList: Seq[String]): Seq[(String, Double)] = {
    val wordsCount = wordsList.size.toDouble
    val tfMap = wordsList.groupBy(identity).mapValues(_.size / wordsCount)
    tfMap.toSeq
  }
}
```

Above is the scala code, it does exactly what the Python UDF does, just in scala. The first stage of this execution results in 13 minutes of runtime, which is faster than the original's combined stage 1 & 2 at 17 minutes. This is the physical plan:



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [term#83 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(term#83 ASC NULLS FIRST, 200), ENSURE_REQUIREMENTS, [plan_id=45]
      +- ObjectHashAggregate(keys=[term#83], functions=[collect_list(struct(doc_id, doc_id#65, frequency, frequency#84), 0, 0), count(1)])
         +- Exchange hashpartitioning(term#83, 200), ENSURE_REQUIREMENTS, [plan_id=42]
            +- ObjectHashAggregate(keys=[term#83], functions=[partial_collect_list(struct(doc_id, doc_id#65, frequency, frequency#84), 0, 0), partial_cou
nt(1)])
               +- Project [doc_id#65, term_frequency#80.term AS term#83, term_frequency#80.frequency AS frequency#84]
                  +- Generate explode(term_frequency#66), [doc_id#65], false, [term_frequency#80]
                     +- Filter ((size(term_frequency#66, true) > 0) AND isnotnull(term_frequency#66))
                        +- InMemoryTableScan [doc_id#65, term_frequency#66], [(size(term_frequency#66, true) > 0), isnotnull(term_frequency#66)]
                              +- InMemoryRelation [doc_id#65, term_frequency#66], StorageLevel(memory, 1 replicas)
                                    +- *(1) Project [id#0 AS doc_id#65, udf_term_frequency(UDF(split(lower(trim(regexp_replace(regexp_replace(regexp_repl
ace(text#3,
,  , 1), [^a-zA-Z\s],  , 1), \s+,  , 1), None)),  , -1))) AS term_frequency#66]
                                       +- *(1) ColumnarToRow
                                          +- FileScan parquet [id#0,text#3] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex
(1 paths)[hdfs://192.168.49.65:8020/wiki2], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:string,text:string>
```

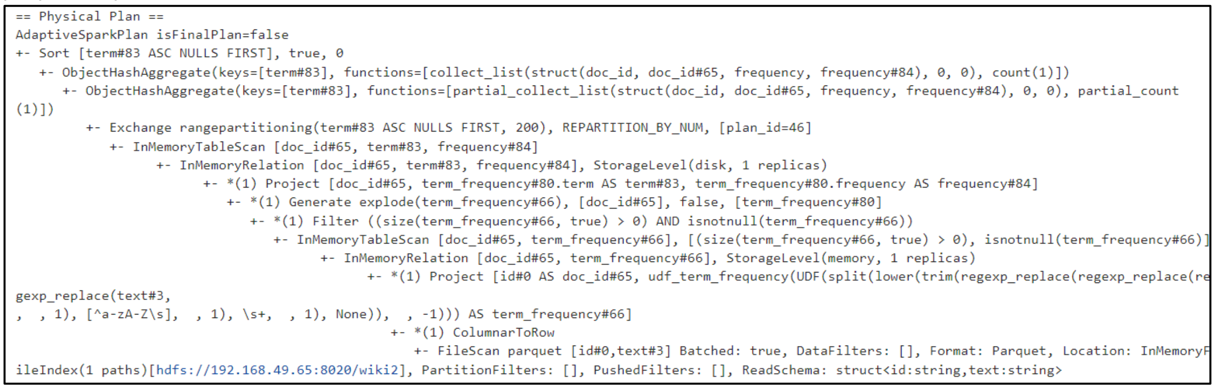
Notice that it has no "BatchEvalPython" compared to the Python UDF approaches. Total execution time for this DAG is 19.5 minutes.

Scala UDF – Repartition By Range

Since using Scala UDF is proven to be the fastest so far, the next 2 optimization is just editing the Scala UDF approach. This one tries to reduce the number of shuffle even further by doing *repartitionByRange*. This is the edited part from the group by term & order by stage:

```
df_unstruct_persisted = df_unstruct.persist(StorageLevel.DISK_ONLY)

df_repart_range = df_unstruct_persisted.repartitionByRange(200, "term")

df_termdoc = df_repart_range.groupBy("term").agg(collect_list(struct("doc_id",
"frequency")).alias("term_frequency_array"), (log(df_count / count("*")) /
log(lit(10))).alias("idf"))

df_termdoc_ordered = df_termdoc.orderBy("term")
```

This combines original's stage 4 & 5 into: Shuffle Read > GroupBy (term) > Sort. Here is the physical plan:



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [term#83 ASC NULLS FIRST], true, 0
   +- ObjectHashAggregate(keys=[term#83], functions=[collect_list(struct(doc_id, doc_id#65, frequency, frequency#84), 0, 0), count(1)])
      +- ObjectHashAggregate(keys=[term#83], functions=[partial_collect_list(struct(doc_id, doc_id#65, frequency, frequency#84), 0, 0), partial_count
(1)])
         +- Exchange rangepartitioning(term#83 ASC NULLS FIRST, 200), REPARTITION_BY_NUM, [plan_id=46]
            +- InMemoryTableScan [doc_id#65, term#83, frequency#84]
                  +- InMemoryRelation [doc_id#65, term#83, frequency#84], StorageLevel(disk, 1 replicas)
                        +- *(1) Project [doc_id#65, term_frequency#80.term AS term#83, term_frequency#80.frequency AS frequency#84]
                           +- *(1) Generate explode(term_frequency#66), [doc_id#65], false, [term_frequency#80]
                              +- *(1) Filter ((size(term_frequency#66, true) > 0) AND isnotnull(term_frequency#66))
                                 +- InMemoryTableScan [doc_id#65, term_frequency#66], [(size(term_frequency#66, true) > 0), isnotnull(term_frequency#66)]
                                       +- InMemoryRelation [doc_id#65, term_frequency#66], StorageLevel(memory, 1 replicas)
                                             +- *(1) Project [id#0 AS doc_id#65, udf_term_frequency(UDF(split(lower(trim(regexp_replace(regexp_replace(re
gexp_replace(text#3,
,  , 1), [^a-zA-Z\s],  , 1), \s+,  , 1), None)),  , -1))) AS term_frequency#66]
                                                +- *(1) ColumnarToRow
                                                   +- FileScan parquet [id#0,text#3] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryF
ileIndex(1 paths)[hdfs://192.168.49.65:8020/wiki2], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:string,text:string>
```

Disk cache is crucial because if not, the whole HDFS scan to Scala UDF operation will be done twice because we are dealing with range partitioning method just like sort that needs to have 2 stages. This way, the second stage will read from the disk cache instead. The first stage does everything from Scan HDFS to UDF, followed by writing to disk and using it to find the range partitioning sample. The second stage would be just reading data from disk cache then performing the actual shuffle. One downside to this approach is that there is no partial aggregation that happens before the shuffle, which means if you have low cardinality aggregation, you might miss filtering a lot of data from getting send over the network. The final result is that it is slightly slower by 30 seconds compared to the original Scala UDF approach at total of 20 minutes. The first stage is the same at 13 minutes, the second stage is only 1.5 minutes, while the final stage is 5.5 minutes.

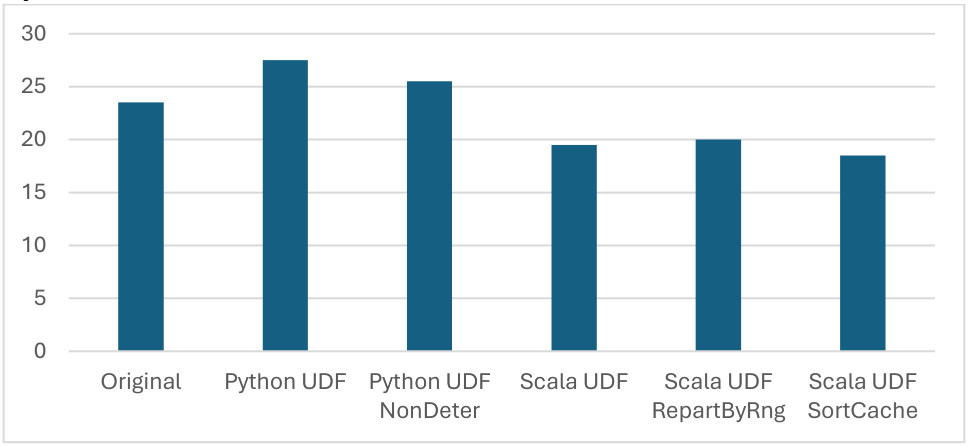
Scala UDF – Sort Cache

From the previous repartition trial, I learned that disk cache is needed so that the range partitioning won't perform whole stage operations twice. So I apply this same method, but to the original Scala UDF, at between these:

```
df_termdoc = spark.sql(f"""
with cte as (
SELECT term, COLLECT_LIST(STRUCT(doc_id, frequency)) AS term_frequency_array,
log({df_count} / count(*)) / log(10) as idf
FROM v_word_count
GROUP BY term
)
SELECT term, TRANSFORM(term_frequency_array, x -> STRUCT(x.doc_id as doc_id,
x.frequency * idf as tfidf)) as term_tfidf_array
FROM cte
""")

df_termdoc_cache = df_termdoc.persist(StorageLevel.DISK_ONLY)

df_termdoc_ordered = df_termdoc_cache.sort("term")
```

The physical plan isn't looking any different except that it has a step for disk cache. This approach would be beneficial if the aggregation computation is very expensive and if the aggregation output itself is much smaller than the input since during the second iteration it will be scanning way less data. However, if the aggregation computation isn't expensive and there isn't much data getting aggregated, this caching approach might do worse especially if the disk write is slow. The result is the first group by term stage computing for 3 minutes which is slower than either group by term stages on original Scala UDF that each performs at 2+ minutes. This makes sense since there is an extra operation that serialized & write data to disk. However, the second group by term stage only runs for 1 minute, making this version faster than original Scala UDF by 1 minute, which total runtime at 18.5 minutes.

Speed Chart



Common Failure: Java heap space

The problem I encountered the most is "Java heap space", and my most common fix to this is to tune the spark.memory.fraction lower, which adds user memory & decrease spark memory. User memory is especially important for the used Scala UDF since it does create a lot of user-defined data structure. Increasing spark.memory.fraction instead would be great at reducing memory spill. Using Python UDF is less prone to Java heap space error since the temporary dictionary that is getting created is stored in python process instead. However, we need to be careful since this means Python process use extra memory from the OS that is independent from JVM's memory. There are definitely some memory spills in this whole experiment, but from many experiments I found that most of the memory spill that happened didn't affect the time significant enough, it usually just add at max 1 minute.