Apache Spark L200 & Dataproc on GCE

Oct 2024
Author: yunusd@google.com

Google Cloud

- Delete me before presenting to customers

PURPOSE

This deck is for giving an Apache Spark and core Dataproc training for customer and internally. Best suited for users with prior Spark experience

HOW TO USE THIS PRESENTATION

Make a copy of this deck and tailor the presentation to your customer's specific engagement needs and agenda. The deck is not intended to be presented sequentially from end to end.

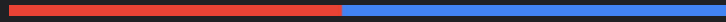
GCC can present to your customer

If you have a customer who is struggling with Spark, GCC team can do this workshop in half a day and help your customer to optimize their Spark code.

Foreword

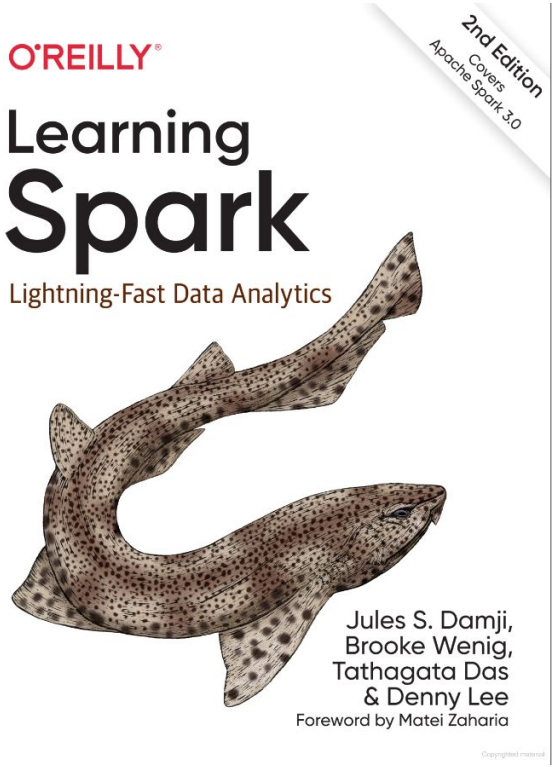
Purpose	To explain how Apache Spark works internally and how Dataproc interacts with Spark.
Intended audience	Engineers and Architects who work on Apache Spark for Data Engineering work.
Key assumptions	This deck goes into details about Spark and Dataproc. A prior experience is crucial. The audience should have deployed Spark on Dataproc and should have tried to optimize.
Delivery note	You may use this deck in Data Lake optimization engagements. Many content is taken from books and public internet. Mention this at the start of your presentation.

If you have a customer having problems with Spark, contact to yunusd@ or any GCC contact. We can help.



Disclaimer

This deck uses content from public internet and Learning Spark book (great book to advance your knowledge). They are all cited in respective slides.



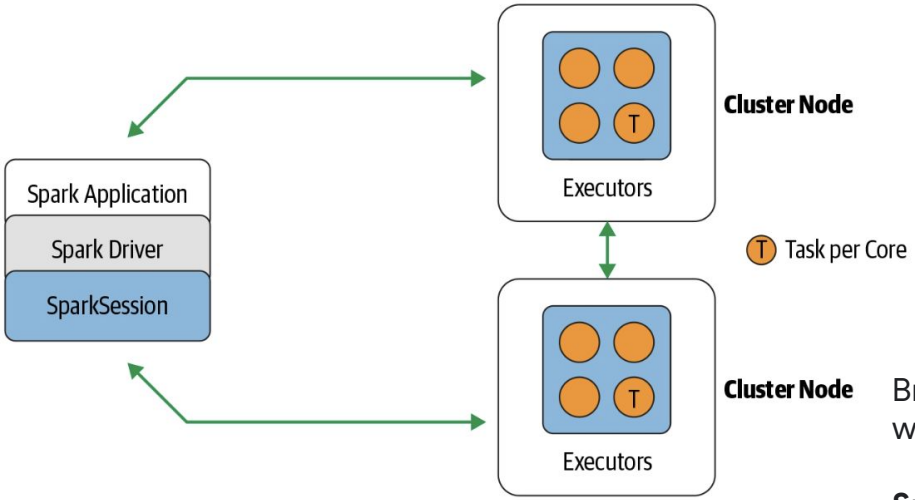
Contents

Apache Spark is

- an orchestrator
  - Single application may run several jobs (serially. Tasks inside a job run in parallel.)
- a distributed processing engine
  - Stores data in multiple machines and computes on them distributedly
- an sql engine
  - Supports both SQL syntax and Pandas style Dataframes.
- an ML framework
- a streaming processing engine
- also graph processing ...

and hence combines the challenges of Dataflow, BigQuery, Airflow.

- Partitioning is key for parallel operations
- Shuffling data is costly. Not MapReduce: small Map, huge Shuffle, and small Reduce
- Classic SQL challenges, broadcast join, filter pushdown, read only the necessary etc.

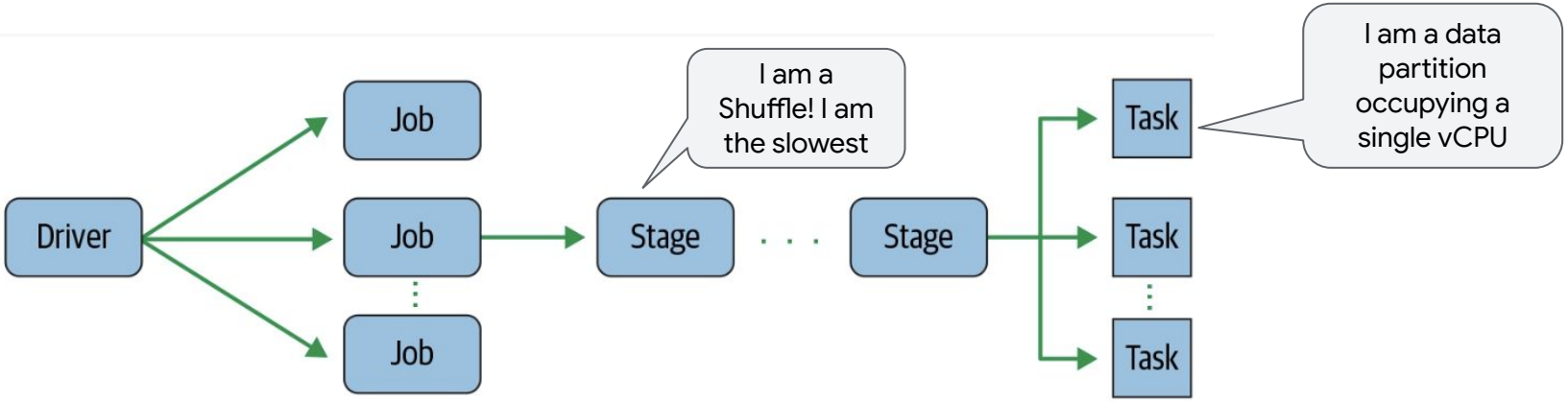


Brain/Orchestrator is the **Driver**, **Executors** are the distributed workers.

Spark session is an object that helps you interact with Spark APIs of the cluster. In an interactive shell, it is created for you. For Batch applications, you have to create the session

KEY Message: Driver is not for computation! Think it like python code for creating Composer DAGs. You don't do computation inside composer orchestrator code.

https://books.google.nl/books/about/Learning_Spark.html?id=CEb1DwAAQBAJ&redir_esc=y



Driver creates jobs, which are divided into Stages, which are again divided into Tasks.

Stages are boundaries of SHUFFLEs similar to Dataflow.

https://books.google.nl/books/about/Learning_Spark.html?id=CEb1DwAAQBAJ&redir_esc=y

Partition is atomic unit of parallelism

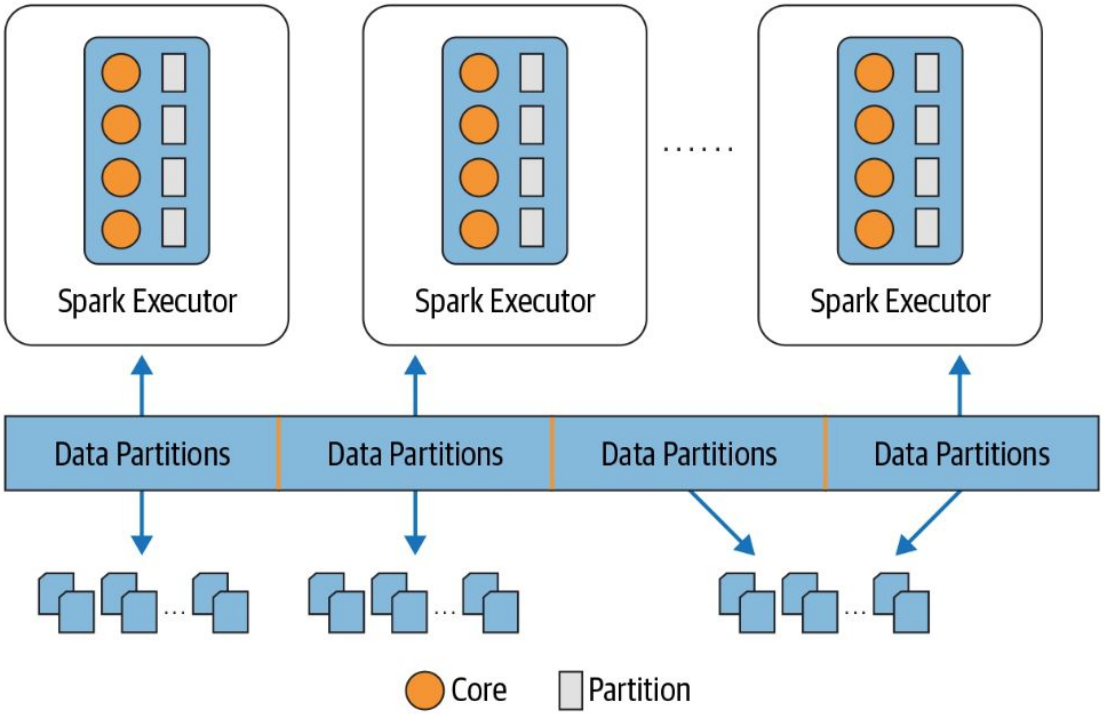


Figure 7-3. Relationship of Spark tasks, cores, partitions, and parallelism

Each partition is handled by a task which is run on a thread on a single core.

Parallelism = min(partitions, cores)

Partitions are created at source and later at stage boundaries (shuffle). e.g., for files, spark.sql.files.maxPartitionBytes determine the number of partitions.

https://books.google.nl/books/about/Learning_Spark.html?id=CEb1DwAAQBAJ&redir_esc=y

Reading From JDBC: use partitioning for parallel load

`partitionColumn, lowerBound, upperBound`	(none)	These options must all be specified if any of them is specified. In addition, `numPartitions` must be specified. They describe how to partition the table when reading in parallel from multiple workers. `partitionColumn` must be a numeric, date, or timestamp column from the table in question. Notice that `lowerBound` and `upperBound` are just used to decide the partition stride, not for filtering the rows in table. So all rows in the table will be partitioned and returned. This option applies only to reading.
		Example:
		`spark.read.format("jdbc")`
		`.option("url", jdbcUrl)`
		`.option("dbtable", "(select c1, c2 from t1) as subq")`
		`.option("partitionColumn", "c1")`
		`.option("lowerBound", "1")`
		`.option("upperBound", "100")`
		`.option("numPartitions", "3")`
		`.load()`

`numPartitions`	(none)	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections. If the number of partitions to write exceeds this limit, we decrease it to this limit by calling `coalesce(numPartitions)` before writing.

- While determining the partitions, be aware of (i) data skew where partition column is not uniformly distributed (e.g., lots of null values), (ii) overloading the source system by many parallel connections.
- Note that LOAD() is not an action, it is lazy evaluated.

Spark APIs: RDD, DataFrame, DataSets

Resilient Distributed Dataset (RDD) is the most basic abstraction. It provides:

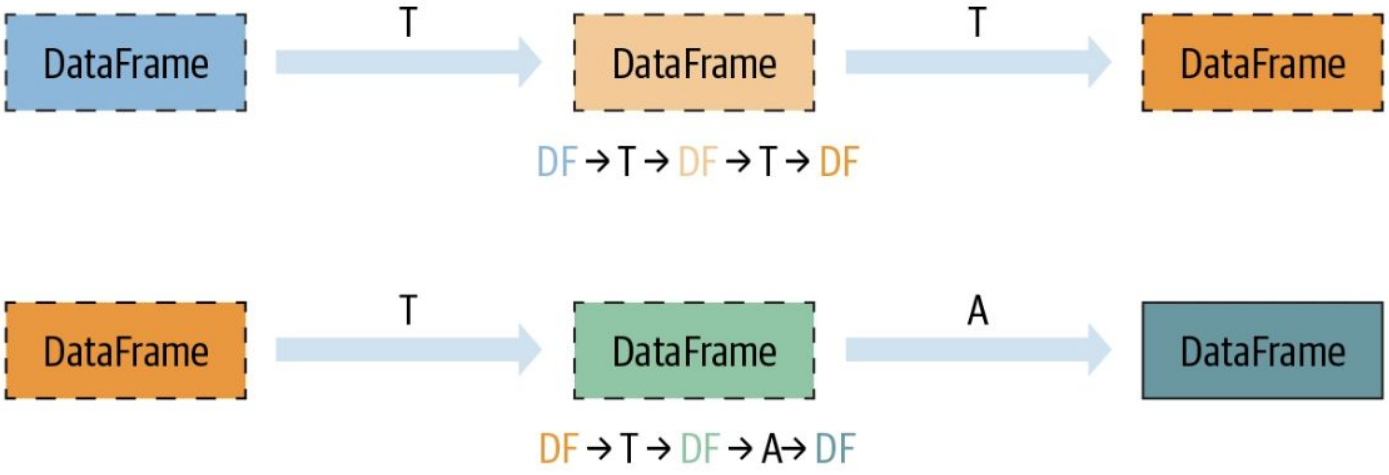
- dependencies for lineage calculation,
- Partitions with some locality information
- Compute function: partition-> Iterator[T]

RDDs are not aware of the computation, and hence doesn't allow optimizations.

DataFrames (inspired by Pandas), are in-memory tables with named schemas, columns, and data types. It provides common SQL like operations, filters, aggregations, projections..

DataSets are dataframes with strict compile time type safety. They are only available in Java and Scala, not Python and R.

Lazy Evaluation and Caching



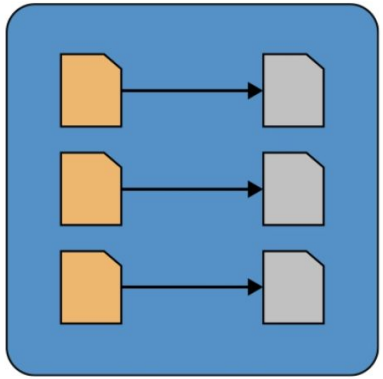
Two spark operations: Transformation and Action.

Transformations are lazily evaluated, only when an action is seen.

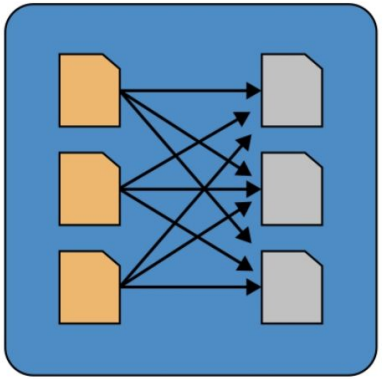
Dataframes are immutable and transformations are recorded in a **lineage** graph. When a node fails, the same state can be recovered by replaying the transformations on the lineage graph.

https://books.google.nl/books/about/Learning_Spark.html?id=CEb1DwAAQBAJ&redir_esc=y

Narrow Dependencies



Wide Dependencies



I am a shuffle. I am expensive

Which transformations require a shuffle?

Transformations and actions as Spark operations

Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

https://books.google.nl/books/about/Learning_Spark.html?id=CEb1DwAAQBAJ&redir_esc=y

```
with SparkSession.builder.appName("PythonSort").getOrCreate() as spark:

    print("reading the file")
    df = spark.read.text(sys.argv[1])# .rdd.map(lambda r: r[0])

    # split every line and word with space and then filter words that are shorter than 3 characters
    df = df.selectExpr("split(value, ' ') as words").selectExpr("explode(words) as word").filter("length(word) > 1")

    #df.cache()

    # count the occurance of each word separately and create a word and its count dataframe
    df_words = df.groupBy("word").count()
    # sort the dataframe by count in descending order
    df_words_sorted = df_words.sort("count", ascending=False)
    print("First Count")
    df_words_sorted.explain()
    df_words_sorted.count()
    print("Second count")
    df_words_sorted_filtered = df_words_sorted.filter("count > 40")
    df_words_sorted_filtered.explain()
    df_words_sorted_filtered.count()
```

```
First Count
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [count#10L DESC NULLS LAST], true, 0
   +- Exchange rangepartitioning(count#10L DESC NULLS LAST, 200), ENSURE_REQUIREMENTS, [plan_id=30]
      +- HashAggregate(keys=[word#6], functions=[count(1)])
         +- Exchange hashpartitioning(word#6, 200), ENSURE_REQUIREMENTS, [plan_id=27]
            +- HashAggregate(keys=[word#6], functions=[partial_count(1)])
               +- Filter (length(word#6) > 1)
                  +- Generate explode(words#2), false, [word#6]
                     +- Project [split(value#0,  , -1) AS words#2]
                        +- Filter ((size(split(value#0,  , -1), true) > 0) AND isnotnull(split(value#0,  , -1)))
                           +- FileScan text [value#0] Batched: false, DataFilters: [(size(split(value#0,  , -1), true) > 0), isnotnull(split(value#0,  , -1)
)], Format: Text, Location: InMemoryFileIndex(1 paths)[gs://yunus-playground-dataproc-utility-temp/example/web1_access_log_20..., PartitionFilters: [], Push
edFilters: [], ReadSchema: struct<value:string>


Second count
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [count#10L DESC NULLS LAST], true, 0
   +- Exchange rangepartitioning(count#10L DESC NULLS LAST, 200), ENSURE_REQUIREMENTS, [plan_id=184]
      +- Filter (count#10L > 40)
         +- HashAggregate(keys=[word#6], functions=[count(1)])
            +- Exchange hashpartitioning(word#6, 200), ENSURE_REQUIREMENTS, [plan_id=180]
               +- HashAggregate(keys=[word#6], functions=[partial_count(1)])
                  +- Filter (length(word#6) > 1)
                     +- Generate explode(words#2), false, [word#6]
                        +- Project [split(value#0,  , -1) AS words#2]
                           +- Filter ((size(split(value#0,  , -1), true) > 0) AND isnotnull(split(value#0,  , -1)))
                              +- FileScan text [value#0] Batched: false, DataFilters: [(size(split(value#0,  , -1), true) > 0), isnotnull(split(value#0,  ,
-1))], Format: Text, Location: InMemoryFileIndex(1 paths)[gs://yunus-playground-dataproc-utility-temp/example/web1_access_log_20..., PartitionFilters: [], P
ushedFilters: [], ReadSchema: struct<value:string>
```

In code, Sort is before Filter, but here Sort is at the end since filtering early is better

In the second DF, filter is not at the end although in the code it was after sorting. It is moved to an earlier stage with the Catalyst optimizer, showing lazy evaluation in action.

Caching Data Frames saves work

Whenever you need to use the same Data Frame multiple times, you should cache them. Otherwise, that Data Frame is re-created from scratch every time.

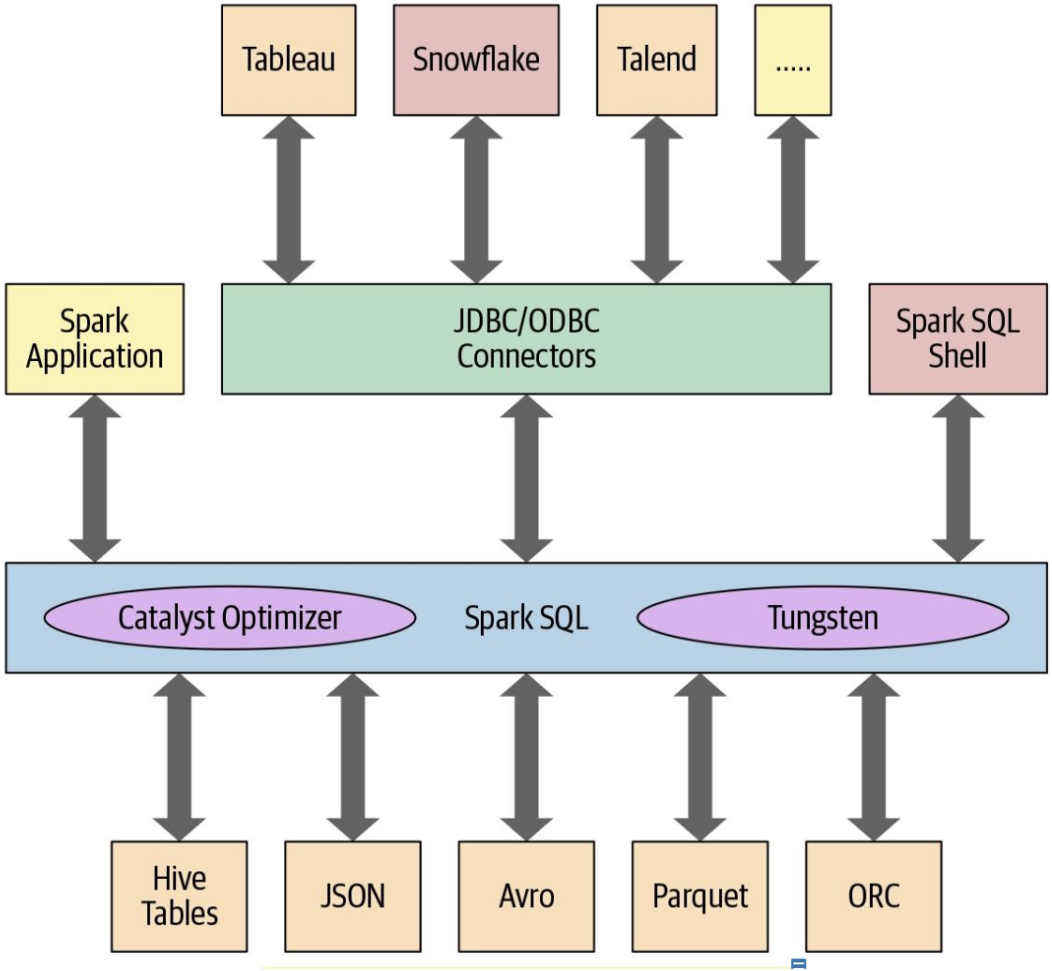
df.cache() = df.persist(MEMORY_AND_DISK)

Don't forget to df.unpersist()

Caching/persisting is a lazy operation. It doesn't happen till you invoke an action like count, collect, take. Note that if you use take(1), it caches only a single partition since only one is calculated. So use count() for explicit caching invocation.

StorageLevel	Description
MEMORY_ONLY	Data is stored directly as objects and stored only in memory.
MEMORY_ONLY_SER	Data is serialized as compact byte array representation and stored only in memory. To use it, it has to be deserialized at a cost.
MEMORY_AND_DISK	Data is stored directly as objects in memory, but if there's insufficient memory the rest is serialized and stored on disk.
DISK_ONLY	Data is serialized and stored on disk.
OFF_HEAP	Data is stored off-heap. Off-heap memory is used in Spark for storage and query execution; see "Configuring Spark executors' memory and the shuffle service".

Optimizers and Vectorized UDFs



Catalyst takes a computational query and converts it to an optimized execution plan. Rule based system.

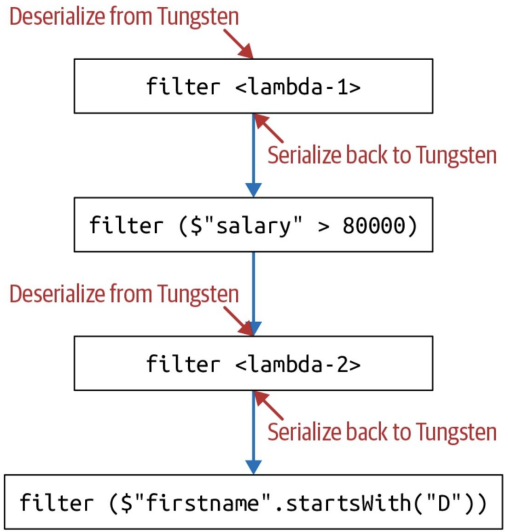
Tungsten provides low level optimizations like memory management, vectorized CPU operations, better ser/de etc. Tungsten replaces Java memory layout with its own by using pointer arithmetic, hence we observe less GC, low memory usage.

Nice read:
https://www.slideshare.net/SparkSummit/deep-dive-into-project-tungsten-josh-rosen

DSL wins against Lambda

Domain Specific Language statements can be easily optimized. They can run on serialized Tungsten format. But if you run Lambda's, the objects have to be converted into JVM memory format. Avoid Lambdas. If you have to use, run them in sequence.

```
import java.util.Calendar
val earliestYear = Calendar.getInstance.get(Calendar.YEAR) - 40

personDS

 // Everyone above 40: lambda-1
 .filter(x => x.birthDate.split("-")(0).toInt > earliestYear)

 // Everyone earning more than 80K
 .filter($"salary" > 80000)

 // Last name starts with J: lambda-2
 .filter(x => x.lastName.startsWith("J"))

 // First name starts with D
 .filter($"firstName".startsWith("D"))
 .count()
```



https://books.google.nl/books/about/Learning_Spark.html?id=CEb1DwAAQBAJ&redir_esc=y

@udf -> @pandas_udf
pandas_udf is vectorized, operates on batches

Convert row-at-a-time UDFs to Pandas UDFs for vectorized operations.

With Pandas UDFs, Data partitions in Spark are converted into Arrow record batches, which can temporarily lead to high memory usage in the JVM. To avoid possible out of memory exceptions, you can adjust the size of the Arrow record batches by setting the `spark.sql.execution.arrow.maxRecordsPerBatch` configuration to an integer that determines the maximum number of rows for each batch

https://docs.databricks.com/en/udf/pandas.html

59.6 seconds

```
from scipy import stats

@udf('double')
def cdf(v):
    return float(stats.norm.cdf(v))

%timeit df.withColumn('cumulative_probability',
cdf(df.v)).agg(count(col('cumulative_probability'))).show()
```

563 ms

```
@pandas_udf('double')
def pandas_cdf(v: pd.Series) -> pd.Series:
    return pd.Series(stats.norm.cdf(v))

%timeit df.withColumn('cumulative_probability',
pandas_cdf(df.v)).agg(count(col('cumulative_probability'))).show()
```

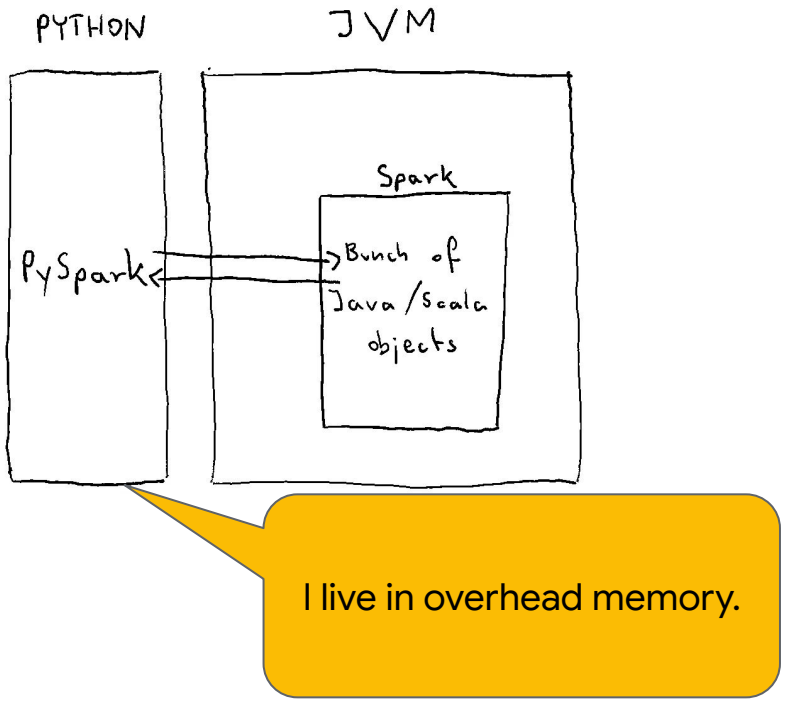
PySpark and Memory Management

PySpark - Easy to use but wastes heavily

In PySpark, Py4J library helps us connect to JVM of Spark via interprocess communication.

When we want to run a python transformation:

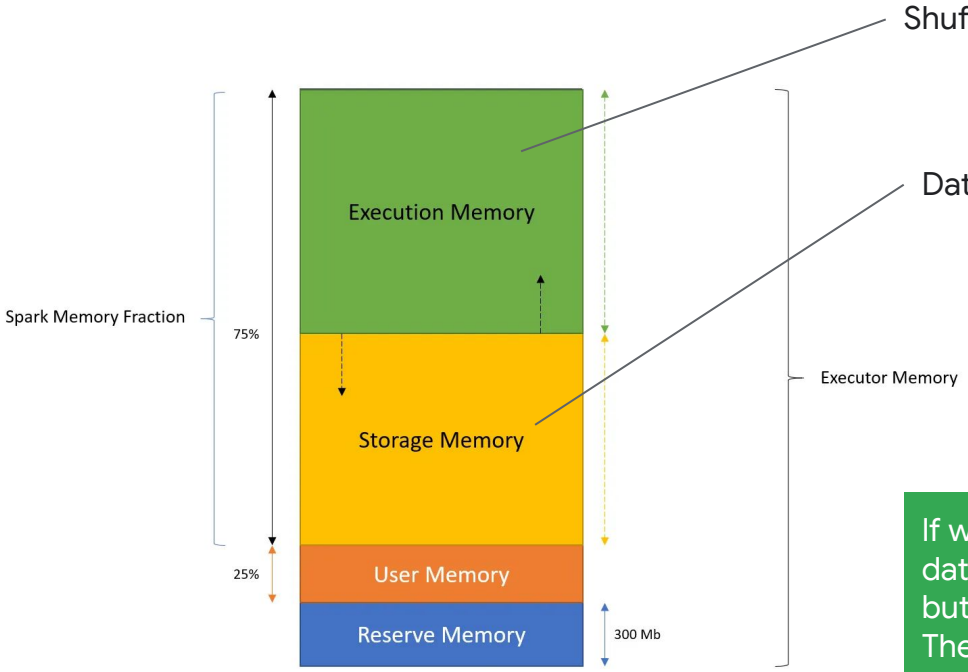
- Spark runs a separate Python program for every executor in every node
- Serializes data and the python functions, sends them to Python to run and then get back the modified dataframes



https://medium.com/analytics-vidhya/how-does-pyspark-work-step-by-step-with-pictures-c011402ccd57

Unified Memory for Executors (on-heap)

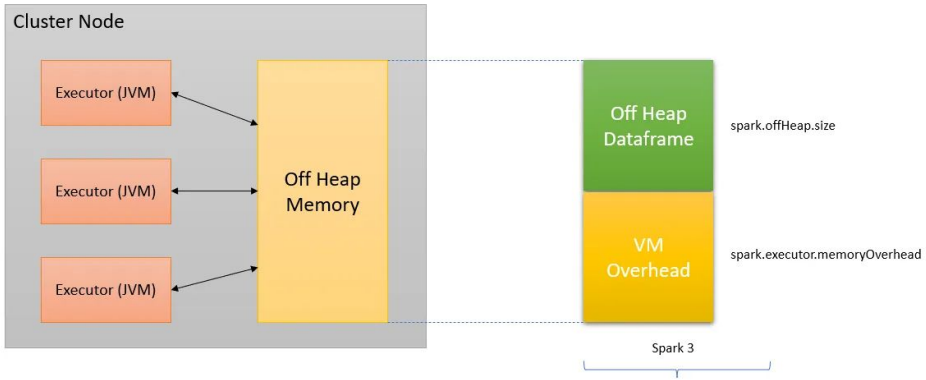
Storage and Execution share the same block



If we don't have enough space for caching data, it spills to disk making application slower but doesn't crash with OOM.
There are different caching options, memory/disk/both etc.

https://medium.com/swlh/spark-oom-error-closeup-462c7a01709d

Off-Heap memory & Memory Overhead



- Off-heap has two parts:
  - **Memory Overhead**. Default 10%, Dataproc Serverless sets it to 40% for PySpark. Since PySpark stores all the data in memory overhead. When you want to run a python transformation, data in tungsten is copied to Python , processed and then sent back. `spark.executor.memoryOverheadFactor`
  - **Off Heap Data Frame cache** -> disabled by default, use it when GC works too much due to large datasets.

`spark.memory.offHeap.enabled`

https://www.youtube.com/watch?v=dPHrykZL8Cg

Driver Memory

You can control it via:

- Spark.driver.memory
- spark.driver.memoryOverheadFactor

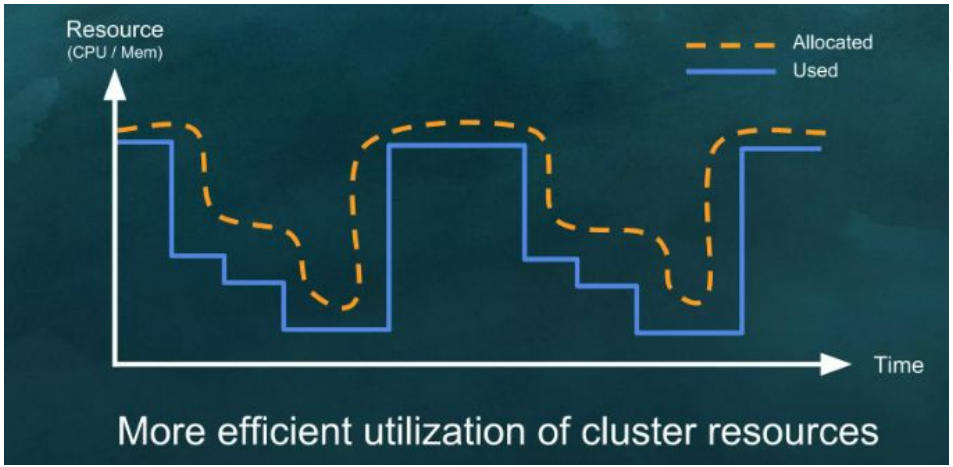
Ideally Drivers should not use much memory. They are the coordinators.

Real danger is pulling data into Driver like by using df.collect(), external connections getting data into Driver (JDBC from driver), Broadcast Join distributed data over Driver.

Adaptive Features

Dynamic Allocation

Enabled by default in Dataproc



When task queue backlog increases, more executors are requested and executors are released when they are idle.

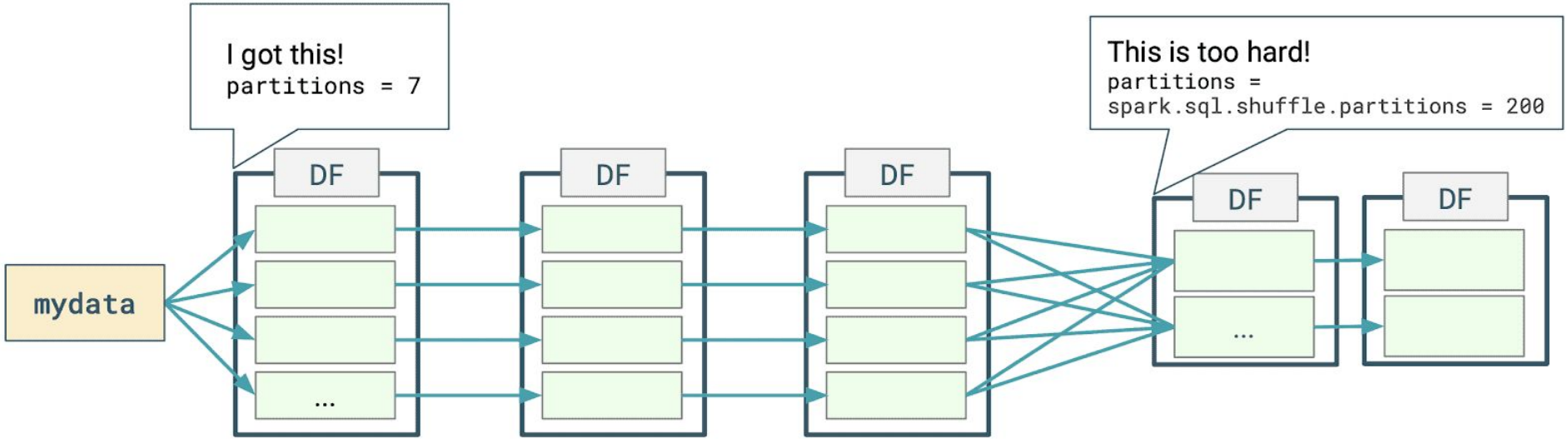
```
spark.dynamicAllocation.enabled true
spark.dynamicAllocation.minExecutors 2
spark.dynamicAllocation.schedulerBacklogTimeout 1m
spark.dynamicAllocation.maxExecutors 20
spark.dynamicAllocation.executorIdleTimeout 2min
```

Deleting executors may lead to loss of data during shuffles. Executors are responsible for distributing their partition to others. During long shuffles, executors may go down and lose data. Solution: **external shuffle service** at each node which is responsible for sharing data.

Executors with cached data are never removed. Always unpersist cached data if unused. Additionally, if you still want them to be removed, set:
`spark.dynamicAllocation.cachedExecutorIdleTimeout`

https://spark.apache.org/docs/latest/job-scheduling.html
https://www.slideshare.net/slideshow/dynamic-allocation-in-spark/49434386#2
https://community.cloudera.com/t5/Community-Articles/Dynamic-Allocation-in-Apache-Spark/ta-p/368095

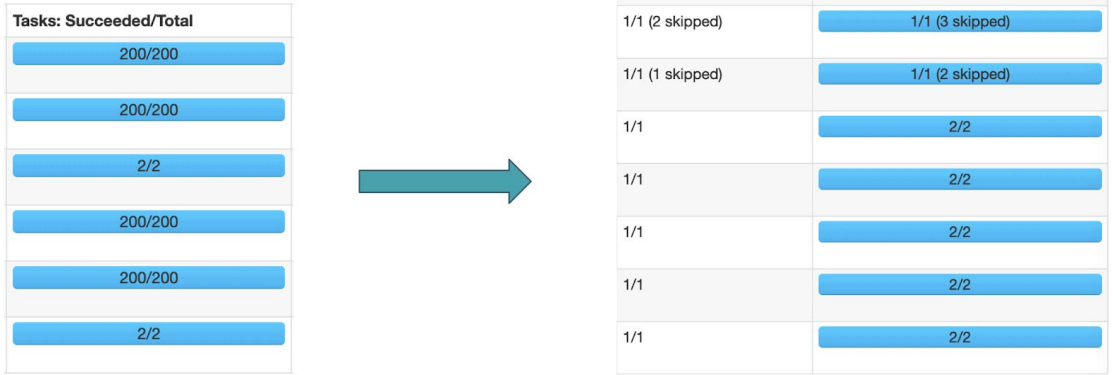
Spark 3 - Adaptive Query Engine (1)



Then in a shuffle operation, partitions are determined by spark.sql.shuffle.partitions value.
You don't exactly control size of the partitions. You may change the value for each stage but it requires lots of work. E.g., a customer ingestion job into BigQuery had around 150 separate jobs, each with multiple stages.

https://blog.cloudera.com/how-does-apache-spark-3-0-increase-the-performance-of-your-sql-workloads/

Spark 3 - Adaptive Query Engine (2)
Adapt the number of shuffle partitions



AQE partition fix happens in shuffle boundaries! So you may still start with default partitions at the beginning!

On the left 200 default partitions. On the right, with AQE, partitions are merged.
You are able to determine the expected size of each partition like 64MB, AQE then merges the partitions to find the balance. It also detects large skewed partitions and divides them.
Parameters: https://spark.apache.org/docs/latest/configuration.html#spark-sql

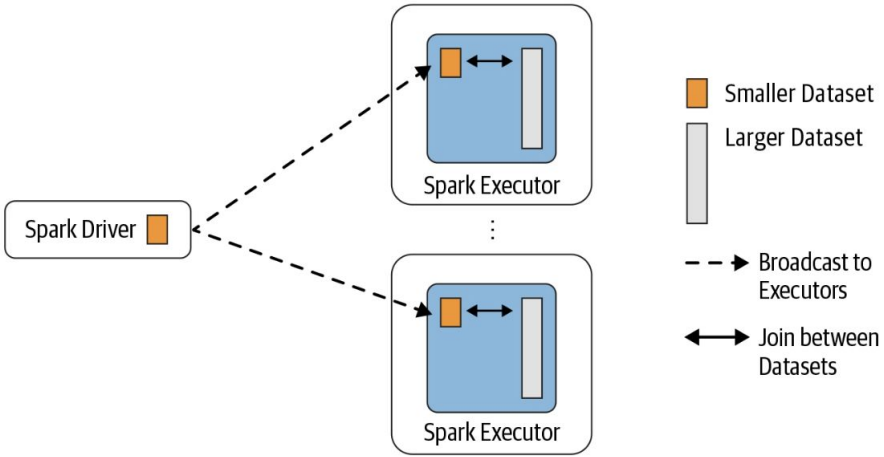
Spark 3 - Adaptive Query Engine (3)

Adaptive Broadcast join

Broadcast join is usually the fastest.
Small table is collected first at the driver and then distributed to each executor. Therefore, it is memory heavy operation.
`spark.sql.adaptive.autoBroadcastJoinThreshold` determines the size of table to allow broadcasting. Default is -1 (not effective), set it to around 100-200MB.

You may also use hints:

```
-- Join Hints for broadcast join
SELECT /*+ BROADCAST(t1) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ BROADCASTJOIN (t1) */ * FROM t1 left JOIN t2 ON t1.key = t2.key;
SELECT /*+ MAPJOIN(t2) */ * FROM t1 right JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle sort merge join
SELECT /*+ SHUFFLE_MERGE(t1) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGEJOIN(t2) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGE(t1) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```



https://books.google.nl/books/about/Learning_Spark.html?id=CEb1DwAAQBAJ&redir_esc=y

Don’t read unnecessary data

Filter pushdown, dynamic partition pruning

If the underlying source supports, Spark skips reading unnecessary data. Parquet, BigQuery, Databases etc.

Filter pushdown:

```
dfwithCasting=spark.table('spark3db.withCasting')
dfwithCasting.filter(col('age')>40).select("*").explain(extended=True)
```

```
== Physical Plan ==
*(1) Project [id#54, name#55, age#56, position#57]
+- *(1) Filter (isnotnull(age#56) && (age#56 > 40))
   +- *(1) FileScan parquet spark3db.withcasting[id#54,name#55,age#56,position#57] Batched: true, Format: Parquet, Location: InMemoryFileIndex[gs://spark3-hive-bucket/spark3db/withCasting], PartitionFilters: [], PushedFilters: [IsNotNull(age), GreaterThan(age,40)], ReadSchema: struct<id:string,name:string,age:int,position:string>
```

If source was string, the filtering wouldn’t work

Message: Use proper data formats that allow early filtering, partition selection. No CSV, JSON!

Partition pruning:

```
dfwithPartition=spark.table('spark3db.withPartition')
dfwithPartition.filter(col('age')>40).select("*").explain(True)
```

```
== Physical Plan ==
*(1) FileScan parquet spark3db.withpartition[id#99,name#100,position#101,age#102] Batched: true, Format: Parquet, Location: PrunedInMemoryFileIndex[gs://spark3-hive-bucket/spark3db/withPartition/age=43, gs://spark3-hive-b...], PartitionCount: 2, PartitionFilters: [isnotnull(age#102), (age#102 > 40)], PushedFilters: [], ReadSchema: struct<id:string,name:string,position:string>
```

Dynamic Partition pruning. Filtered dimension table determines the Age and City and then prunes partitions in fact table for that Age range

```
df_fact=spark.table('spark3db.withPartition_fact')
df_dim=spark.table('spark3db.withoutPartition_dim').filter(col('city')=='Bangalore')
df_join=df_fact.join(df_dim,'age')
```

```
== Physical Plan ==
*(2) Project [age#3, id#0, name#1, position#2, id#29, name#30, city#32]
+- *(2) BroadcastHashJoin [age#3], [age#31], Inner, BuildRight, false
   :- *(2) Filter dynamicpruningexpression(age#3 IN dynamicpruning#112)
   :  :  +- SubqueryBroadcast dynamicpruning#112, 0, [age#31], [id#214]
   :  :     +- ReusedExchange [id#29, name#30, age#31, city#32], BroadcastExchange HashedRelationBroadcastMode(List(cast(input[2, int, false] as bigint)),false), [id#196]
   :  +- *(2) ColumnarToRow
   :     +- FileScan parquet spark3db.withpartition_fact[id#0,name#1,position#2,age#3] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex[gs://spark3-hive-bucket/spark3db/withPartition_fact/age=20, gs://spark3-hive-bu...], PartitionFilters: [isnotnull(age#3), dynamicpruningexpression(age#3 IN dynamicpruning#112)], PushedFilters: [], ReadSchema: struct<id:string,name:string,position:string>
   :           +- ReusedSubquery SubqueryBroadcast dynamicpruning#112, 0, [age#31], [id#214]
   +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[2, int, false] as bigint)),false), [id#196]
      +- *(1) Filter ((isnotnull(city#32) AND (city#32 = Bangalore)) AND isnotnull(age#31))
         +- *(1) ColumnarToRow
            +- FileScan parquet spark3db.withoutpartition_dim[id#29,name#30,age#31,city#32] Batched: true, DataFilters: [isnotnull(city#32), (city#32 = Bangalore), isnotnull(age#31)], Format: Parquet, Location: InMemoryFileIndex[gs://spark3-hive-bucket/spark3db/withoutPartition_dim], PartitionFilters: [], PushedFilters: [IsNotNull(city), EqualTo(city,Bangalore), IsNotNull(age)], ReadSchema: struct<id:string,name:string,age:int,city:string>
```

https://connecttoaparup.medium.com/apache-spark-3-0-dpp-bd05c8d58524

Store data wisely

File Format vs Table Format

Data Lake Engines	Spark, Hive, Presto, BigQuery
Table Formats	Hive, Iceberg, Delta Lake, Hudi, BigQuery
File Formats	CSV, JSON, ORC, AVRO, PARQUET, CAPACITOR
Data Lake Storage	GCS, S3, ADLS, HDFS

Hive Table Format (Partitioning):

```
|-date=20210517/
   |- hour=17/
        |- part_000.parquet
             .
             .
        |- part_10.parquet
   |- hour=18/
```

Splittable vs unsplittable file types

Compressed CSV files cannot be partitioned. The whole file acts as a single partition. Then you have to use uncompressed CSV for proper partitioning.

In contrast, Parquet, ORC are columnar file formats that support compression, and certain codecs, like Snappy, are splittable.

A Parquet compression reduce size around 75%. Less data in storage and in network!

Benchmark of CSV and Parquet

How are file partitions calculated

Partitioning logic is provided here. Summary:

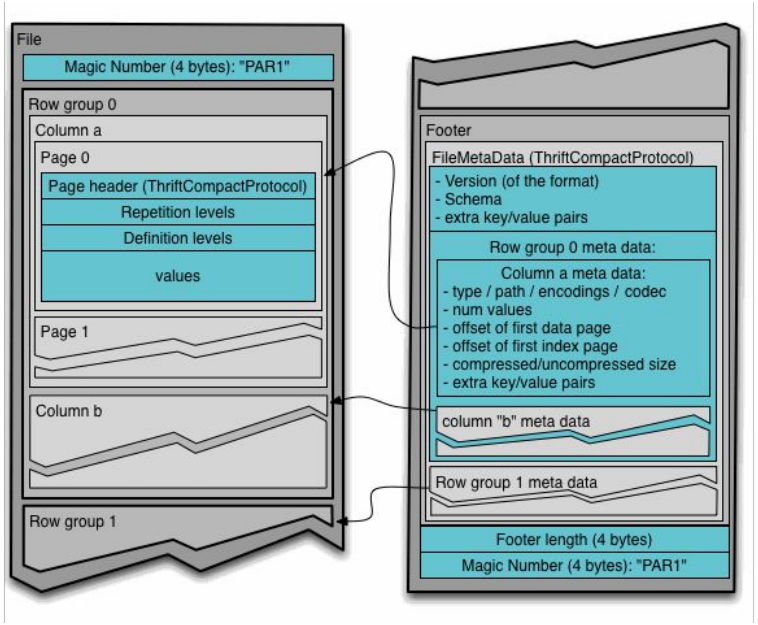
1. The code uses a "Next Fit Decreasing" strategy to assign files to partitions. This means:
    a. Files are processed in the order they are presented.
    b. A new partition is created if adding the current file would exceed the maxSplitBytes limit.
    c. Each file also incurs an openCostInBytes overhead, representing the cost of opening and processing it.
2. maxSplitBytes is calculated based on maxPartitionBytes, openCostInBytes and minPartitionNum (defaults to number of cores) config parameters
3. If the number of partitions turn out to be more than the maximum number of partitions (filesMaxPartitionNum - defaults to None), the code recalculates the desired split size (desiredSplitBytes) based on the total data size and the maximum partition limit. It then re-partitions the files using this new split size.

```
// Assign files to partitions using "Next Fit Decreasing"
partitionedFiles.foreach { file =>
  if (currentSize + file.length > maxSplitBytes) {
    closePartition()
  }
  // Add the given file to the current partition.
  currentSize += file.length + openCostInBytes
  currentFiles += file
}
closePartition()
partitions.toSeq
```

```
def maxSplitBytes(sparkSession: SparkSession, calculateTotalBytes: => Long): Long = {
  val defaultMaxSplitBytes = sparkSession.sessionState.conf.filesMaxPartitionBytes
  val openCostInBytes = sparkSession.sessionState.conf.filesOpenCostInBytes
  val minPartitionNum = sparkSession.sessionState.conf.filesMinPartitionNum
    .getOrElse(sparkSession.leafNodeDefaultParallelism)
  val totalBytes = calculateTotalBytes
  val bytesPerCore = totalBytes / minPartitionNum

  Math.min(defaultMaxSplitBytes, Math.max(openCostInBytes, bytesPerCore))
}
```

spark.sql.files.maxPartitionBytes	Default: 128MB	The maximum number of bytes to pack into a single partition when reading files. This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.
spark.sql.files.openCostInBytes	Default: 4MB	The estimated cost to open a file, measured by the number of bytes that could be scanned in the same time. This is used when putting multiple files into a partition. It is better to over-estimate, then the partitions with small files will be faster than partitions with bigger files (which is scheduled first). This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.
spark.sql.files.minPartitionNum	default value is `spark.sql.leafNodeDefaultParallelism` whose default value is the `spark.default.parallelism` and whose default is `the number of cores`	The suggested (not guaranteed) minimum number of split file partitions. If not set, the default value is `spark.sql.leafNodeDefaultParallelism`. This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.
spark.sql.files.maxPartitionNum	none	The suggested (not guaranteed) maximum number of split file partitions. If it is set, Spark will rescale each partition to make the number of partitions is close to this value if the initial number of partitions exceeds this value. This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.

Parquet file format helps with filtering



Page Index stores statistics about column values like max, min. To help us skip reading when there is a filter.

With partitions, files are stored in different directories. We don't read files at all with partition filter.

With bucketing, parquet files store sorted values with defined max/min so easier to skip chunks. We only read headers and related parts.

Skip shuffle with partitioning

```
// Do the join
val usersOrdersDF = ordersDF.join(usersDF, $"users_id" === $"uid")

== Physical Plan ==
InMemoryTableScan [transaction_id#40, quantity#41, users_id#42, amount#43,
state#44, items#45, uid#13, login#14, email#15, user_state#16]
  +- InMemoryRelation [transaction_id#40, quantity#41, users_id#42, amount#43,
state#44, items#45, uid#13, login#14, email#15, user_state#16],
StorageLevel(disk, memory, deserialized, 1 replicas)
        +- *(3) SortMergeJoin [users_id#42], [uid#13], Inner
           :- *(1) Sort [users_id#42 ASC NULLS FIRST], false, 0
           :  +- Exchange hashpartitioning(users_id#42, 16), true, [id=#56]
           :     +- LocalTableScan [transaction_id#40, quantity#41, users_id#42,
amount#43, state#44, items#45]
           +- *(2) Sort [uid#13 ASC NULLS FIRST], false, 0
              +- Exchange hashpartitioning(uid#13, 16), true, [id=#57]
                 +- LocalTableScan [uid#13, login#14, email#15, user_state#16]
```

Users table is joined with Orders on user id. Since tables are large, no broadcast join and instead SortMergeJoin with a shuffle.

Userid and UID's are shuffled among executors to make them appear at the same executors for local sort and merge.

```
// Save as managed tables by bucketing them in Parquet format
usersDF.orderBy(asc("uid"))
  .write.format("parquet")
  .bucketBy(8, "uid")
  .mode(SaveMode.OverWrite)
  .saveAsTable("UsersTbl")

ordersDF.orderBy(asc("users_id"))
  .write.format("parquet")
  .bucketBy(8, "users_id")
  .mode(SaveMode.OverWrite)
  .saveAsTable("OrdersTbl")

// Cache the tables
spark.sql("CACHE TABLE UsersTbl")
spark.sql("CACHE TABLE OrdersTbl")

// Read them back in
val usersBucketDF = spark.table("UsersTbl")
val ordersBucketDF = spark.table("OrdersTbl")

// Do the join and show the results
val joinUsersOrdersBucketDF = ordersBucketDF
  .join(usersBucketDF, $"users_id" === $"uid")
```

```
== Physical Plan ==
*(3) SortMergeJoin [users_id#165], [uid#62], Inner
:- *(1) Sort [users_id#165 ASC NULLS FIRST], false, 0
:  +- *(1) Filter isnotnull(users_id#165)
:     +- Scan In-memory table `OrdersTbl` [transaction_id#163, quantity#164,
users_id#165, amount#166, state#167, items#168], [isnotnull(users_id#165)]
:           +- InMemoryRelation [transaction_id#163, quantity#164, users_id#165,
amount#166, state#167, items#168], StorageLevel(disk, memory, deserialized, 1
replicas)
:                 +- *(1) ColumnarToRow
:                    +- FileScan parquet
```

If we bucket (or partition) them in advance, when they are loaded same userid/uid data are already in the same executor and hence no need to shuffle again!

If you use a dimension table like users in several joins, bucket/partition all the tables first and then perform the joins. Benefit from data locality.

https://books.google.nl/books/about/Learning_Spark.html?id=CEb1DwAAQBAJ&redir_esc=y

What about other systems?

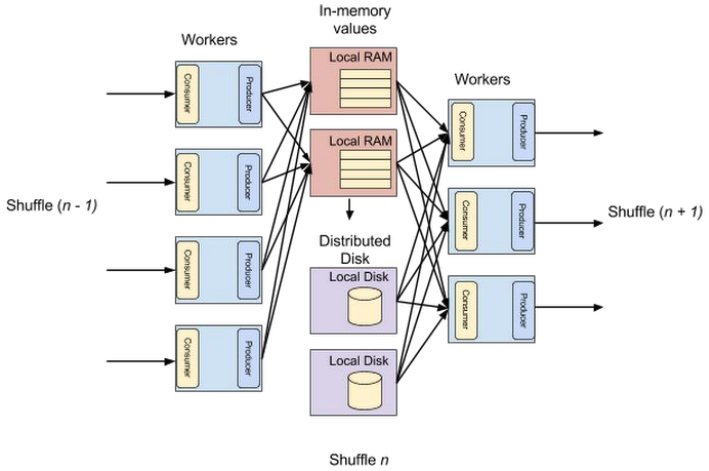
Inspirational: What about other systems?

External Shuffle Service

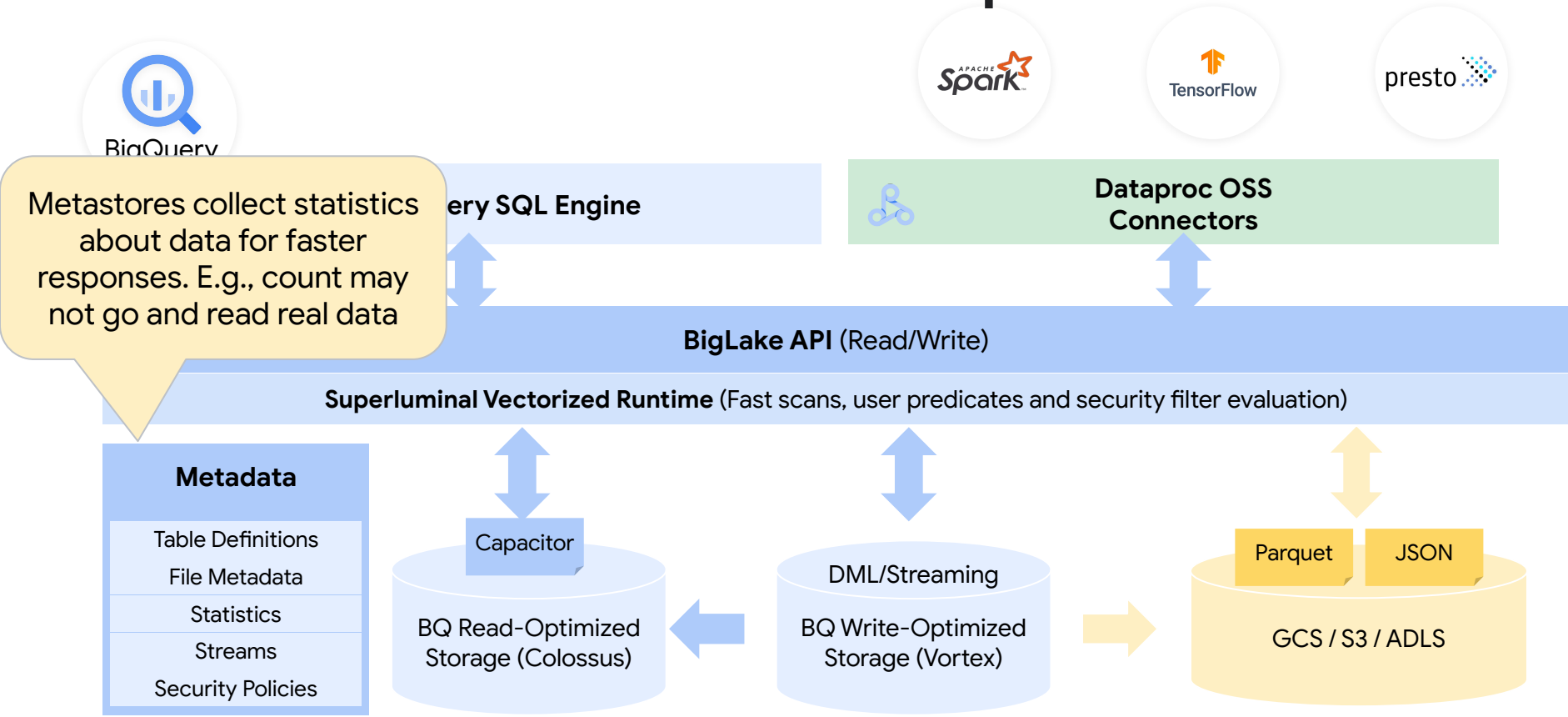
The Dataflow Shuffle feature moves the shuffle operation & data out of the worker VMs and into the Dataflow service backend.

- Smaller VMs, only for processing
- Easy horizontal autoscaling
- Dynamic work rebalancing (divide data more if a worker is too busy)
- Dynamic thread scaling (more threads if VM CPU utilization is low)

BigQuery also have external [in-memory shuffle service] similar to Dataflow. Workers start consuming data before not all data is available.



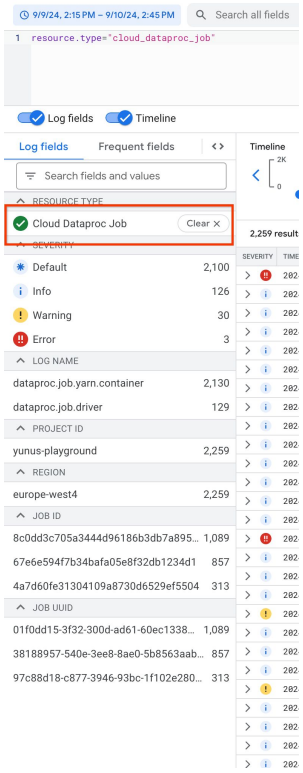
Use Metastores for faster queries

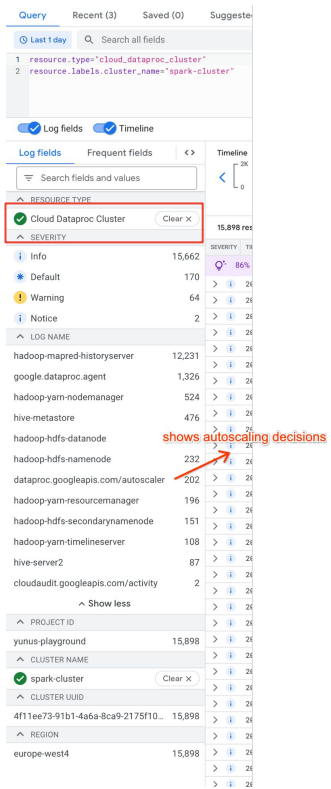


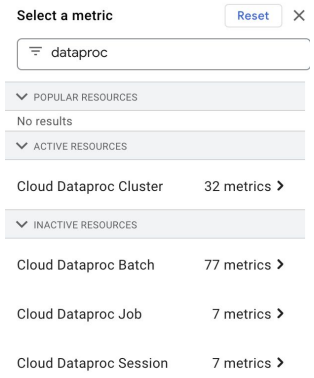
Debugging with Logs and Spark UI

Logs and Metrics

!!! Based on your deployment mode (client or cluster), configure the log collection correctly!!!







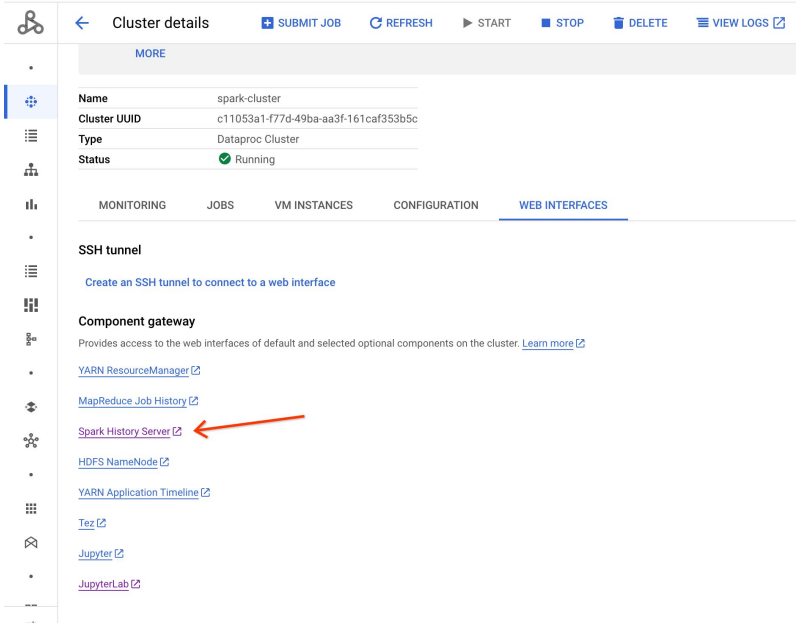
- `--metric-sources` : Required to enable custom metric collection. Specify one or more of the following metric sources: `spark`, `flink`, `hdfs`, `yarn`, `spark-history-server`, `hiveserver2`, `hivemetastore`, and `monitoring-agent-defaults`. The metric source name is case insensitive, for example, either "yarn" or "YARN" is acceptable.

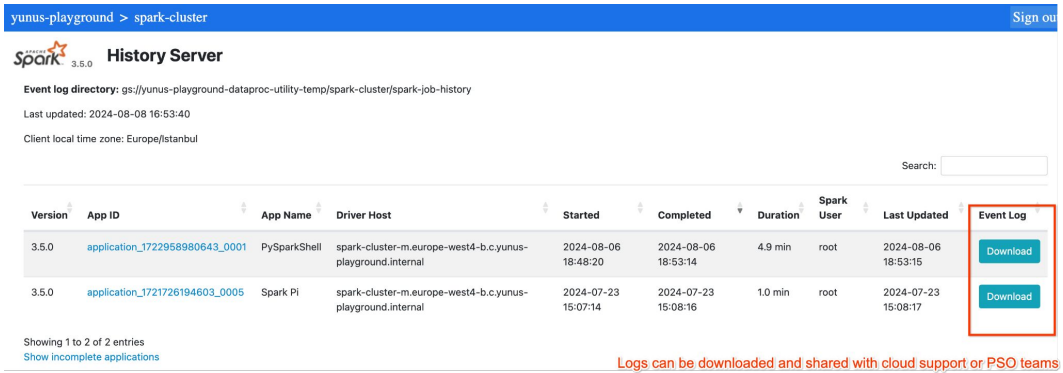
Not available for versions 2.2+. Install via init-action. It provides VM memory, CPU and disk use statistics

Spark UI - Best place for debugging

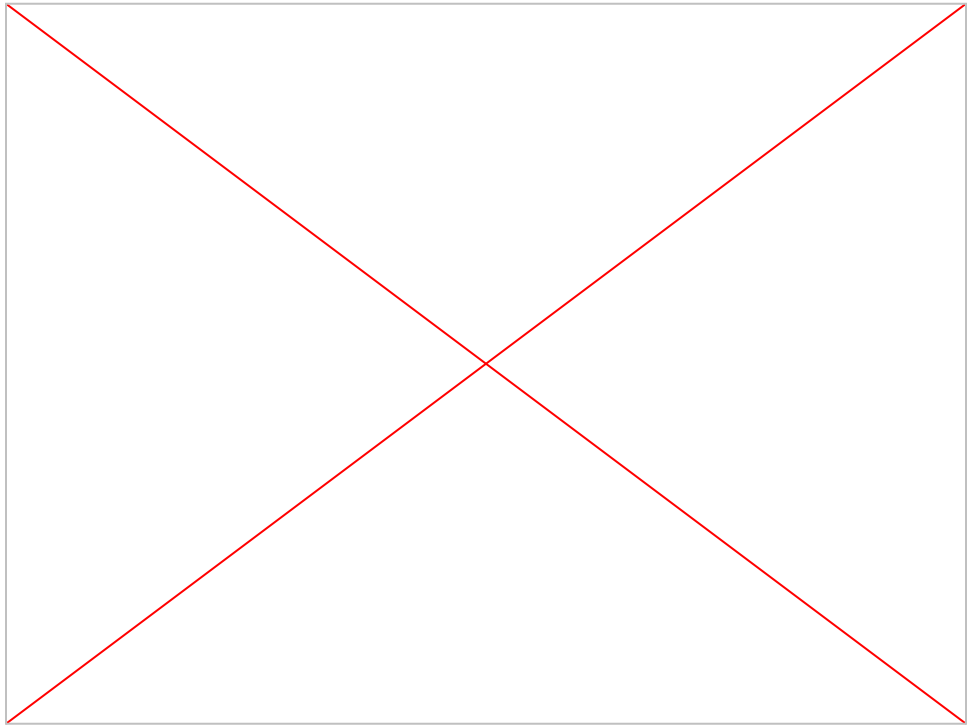
Support teams can't access to Spark UI. You may easily share logs by downloading from UI or getting directly from the GCS bucket.

`gs://<DATAPROC-TEMP-BUCKET>/<CLUSTER_UUID>/spark-job-history/application_…`



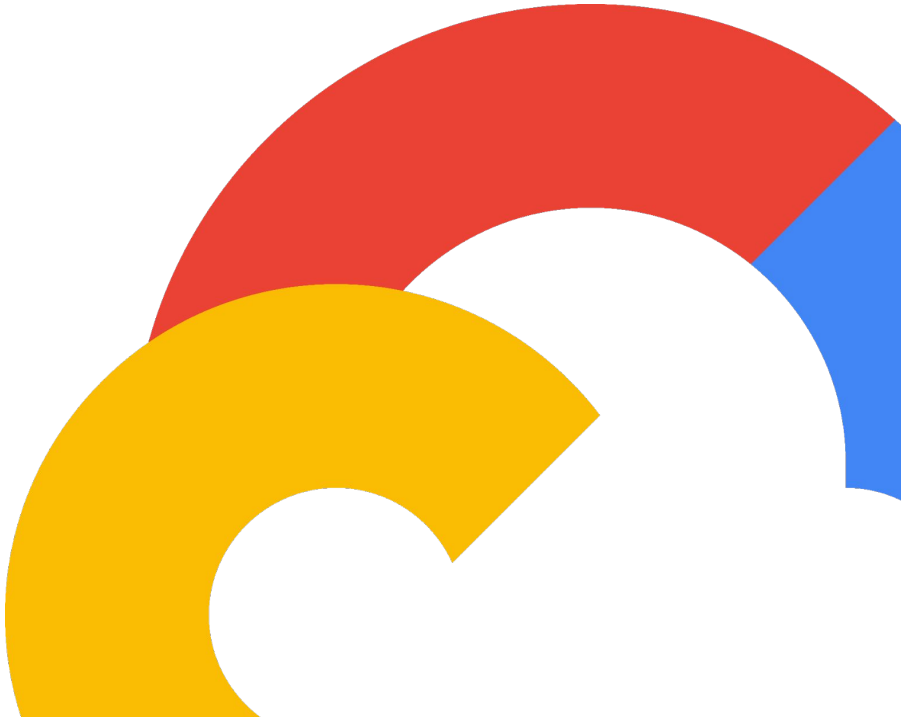


Spark UI



http://screencast/cast/NDYyNzY0OTEzMjQ5NDg0OHxkMjk4NzY4Ni0wNw
https://www.youtube.com/watch?v=rNpzrkB5KQQ

Dataproc



Google Cloud

Dataproc

Fully **managed** & highly **scalable** service for running Apache Spark, Apache Hadoop, & 30+ open source tools & services.

Flexibility of consumption
Use Serverless, or manage clusters on Google Compute Engine (GCE)

Open-Source
No lock-in

Enterprise grade security
Fine grained access control and personal auth

Integrated Google Cloud services
Enable data users through integrations with Vertex AI, BigQuery, & Dataplex

Seamless AI/ML

Dataproc types

Dataproc on GCE:

- Dataproc runs on VMs in customer's project. 2-5 minutes cluster startup.
- Based on Apache BigTop project.
- Suitable for both ephemeral and long running clusters.
- Not possible to update, only via SSH. Re-create instead.
- Not just Spark. Almost all OSS Big Data tech. Can be extended with initialization actions & components

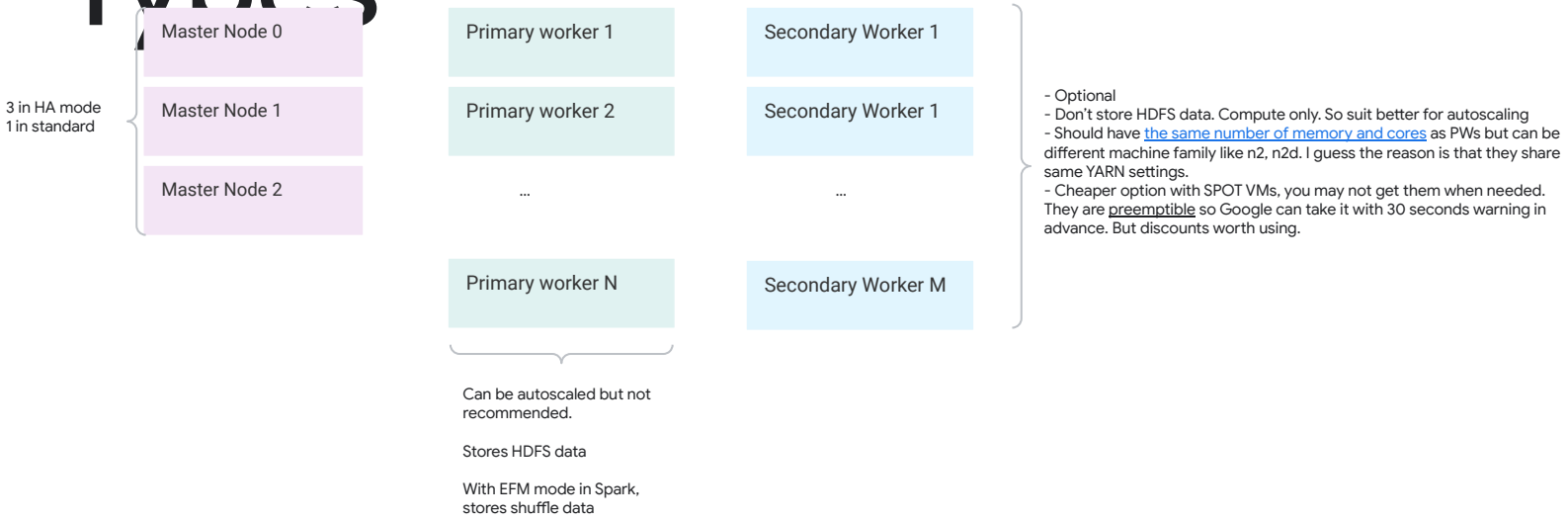
Dataproc Serverless (only Spark)

- Future of Spark in Google Cloud
- New features like autotune, performance enhancements
- Can be called from BigQuery as a stored procedure

Dataproc on GKE (only Spark)

- Replaces YARN with Kubernetes scheduler
- Based on open source Spark K8S operator

Dataproc Machine Types



Dataproc disks

Verdict: If possible (no commits), use Local SSDs. Otherwise, observe disk IOPS from monitoring and adjust sizing.

Dataproc VMs can attach any type of disks.

There are two main types:
Persistent Disks (networked attached) - durable and large but eats from network throughput
Local SSD (attached to the machine) - super fast, lost when error or machines are deleted, not part of commits, limited availability

IOPS limits for zonal Persistent Disk

	Zonal standard PD	Zonal balanced PD	Zonal SSD PD	Zonal extreme PD	Zonal SSD PD multi-writer mode
Read IOPS per GiB	0.75	6	30	–	30
Write IOPS per GiB	1.5	6	30	–	30
Read IOPS per VM*	7,500	80,000	100,000	120,000	100,000
Write IOPS per VM*	15,000	80,000	100,000	120,000	100,000

IOPS is per-GB. Larger disks mean more IO and hence more throughput.

IOPS limits for Local SSDs

# of attached Local SSD disks	Total storage space (GiB)	Capacity per disk (GiB)	IOPS		Throughput (MiBps)	
			Read	Write	Read	Write
1	375	375	170,000	90,000	660	350
2	750	375	340,000	180,000	1,320	700
3	1,125	375	510,000	270,000	1,980	1,050
4	1,500	375	680,000	360,000	2,650	1,400
5	1,875	375	680,000	360,000	2,650	1,400

This is huge! 1TB PD-balanced has 6K read IOPS while Local SSD has 510K!!

1TB PD-Balanced costs around $110 per month while equivalent three Local SSDs cost $88. Moreover Spot VMs make it even more cheaper.

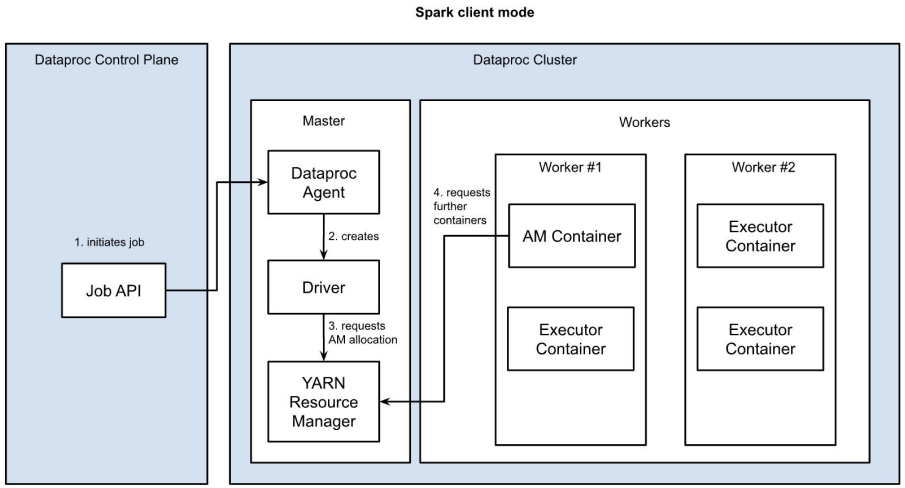
DPMS as the Metastore

- Leverage Google's Dataproc Metastore Service as an enterprise-wide common metastore when working with structured data assets
- Define your external tables once and reference many times
- Enable automated integration with Cloud Catalog
- Common metastore for Dataplex, Dataproc clusters and serverless Spark

```
gcloud metastore services create $DATAPROC_METASTORE_SERVICE_NM \
    --location=$LOCATION \
    --labels=used-by=all-$BASE_PREFIX-clusters \
    --network=$VPC_NM \
    --port=9083 \
    --tier=Enterprise \
    --hive-metastore-version=3.1.2
```

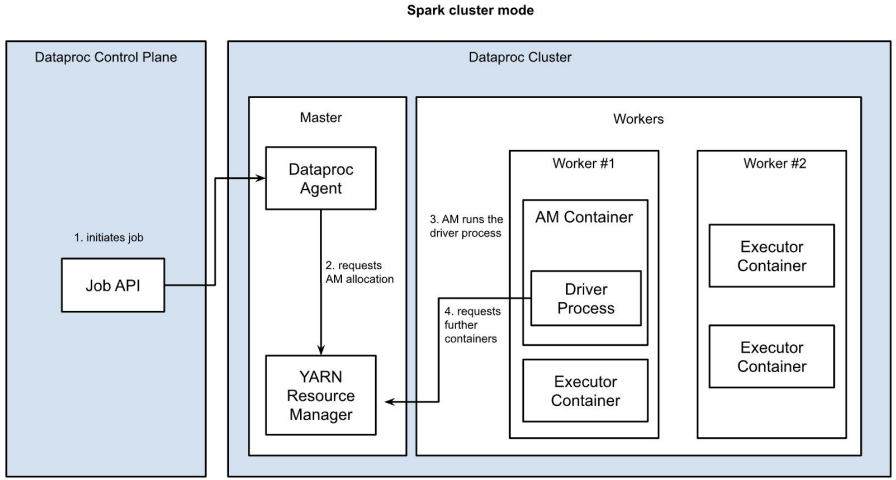
Job submission process

Client vs Cluster mode



Driver is in Master nodes. Max 3 master nodes are available. So the number of drivers is limited.

xroc	dataproc.scheduler.driver-size-mb	number	The average driver memory footprint, which determines the maximum number of concurrent jobs a cluster will run. The default value is 1GB. A smaller value, such as 256, may be appropriate for Spark jobs.
xroc	dataproc.scheduler.job-submission-rate	number	Jobs are throttled if this rate is exceeded. The default rate is 1.0 QPS.
xroc	dataproc.scheduler.max-concurrent-jobs	number	The maximum number of concurrent jobs. If this value is not set when the cluster is created, the upper limit on concurrent jobs is calculated as max((masterMemoryMb - 3584) / masterMemoryMbPerJob, 5). masterMemoryMb is determined by the master VM's machine type. masterMemoryMbPerJob is 1024 by default, but is configurable at cluster creation with thedataproc:dataproc.scheduler.driver-size-mb cluster property.
xroc	dataproc.scheduler.max-memory-used	number	The maximum amount of the RAM that can be used. If current usage is above this threshold, new jobs cannot be scheduled. The default is 0.9 (90%). If set to 1.0, master-memory-utilization job throttling is disabled.
xroc	dataproc.scheduler.min-free-memory.mb	number	The minimum amount of free memory in megabytes needed by the Dataproc job driver to schedule another job on the cluster. The default is 256 MB.

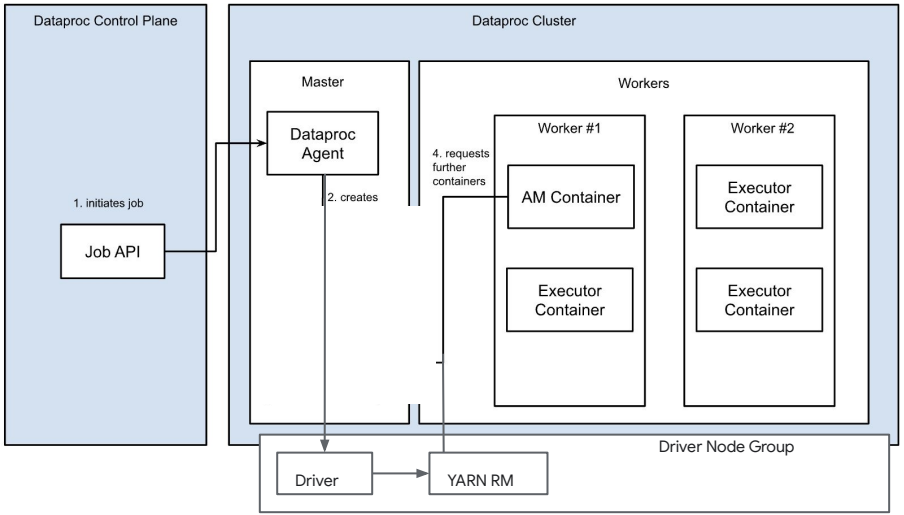


Driver is in worker nodes. So you are not limited to masters only, hence more applications in parallel. Set the dataproc.scheduler.max-concurrent-jobs to high number, still in effect but you don't have to protect master VMs.

But: during Autoscaling or VM Preemption you may lose the whole application if driver is gone. YARN restarts it though.

There is dataproc:am.primary_only property at cluster startup to ensure that drivers run in primary workers at least. But due to a YARN node labeling issue, autoscaling is not possible.

Driver node groups



Master nodes are limited to 3. So we are limited in running drivers.

Driver Node Groups are a separate set of VMs, only reserved for Spark Drivers. You may create as many as you want.

They don’t autoscale. You can change size manually.

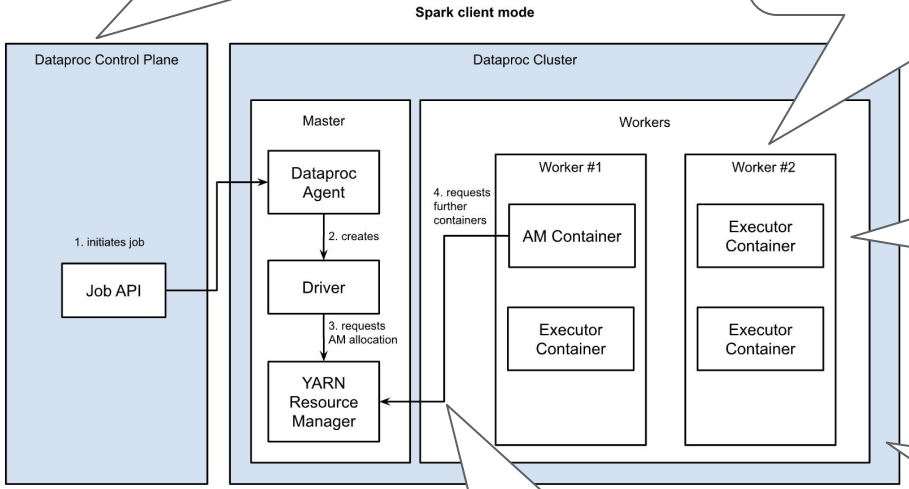
Recommended for long running large shared clusters.

Life of a Job

There is Dataproc Job Queue (only client mode) and then YARN job queue. Dataproc Job queue size is a function of total master memory and per-driver memory. It is configurable. Troubleshooting.

Yarn containers run executors as a JVM process with memory limits (-Xmx). Apart from JVM limit, YARN is permissive in memory and also no CPU limitation by default.

If you need to limit CPUs and memory strictly, you should enable cgroups. No one does as it is a complicated setup.



When there is not enough memory left in a VM (default less than 64MB), earlyoom tool kills the process with highest memory usage. Kills are logged under `google.dataproc.oom-killer`. Added to metric: `dataproc.googleapis.com/node/problem_count` More info

Since JVM is controlled via Xmx and Spark can store excess cache data in disk, it is probably overhead memory that blows up.

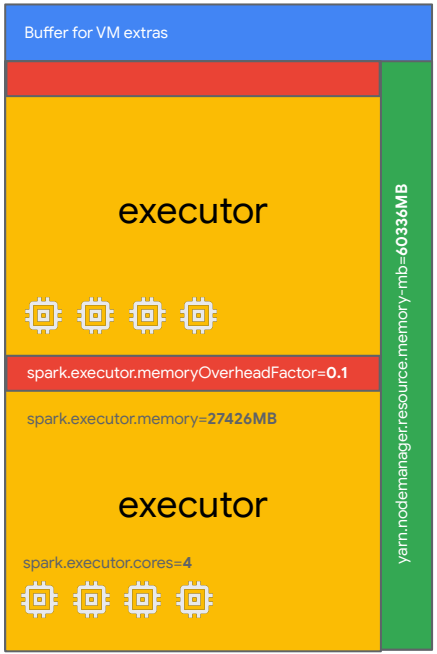
Single Job can dominate the whole cluster. To prevent, you should introduce YARN queues or limit a job's percentage of resource use.
It is hard to figure out the best setup. So autoscaling and job scoped clusters are better for SLA dependent applications.

YARN uses capacity-scheduler to request containers for the executors.

Need for more executors is determined by the lack of memory. To consider CPU too, you should change resource calculator to DominantResourceCalculator

```
<property>
    <name>yarn.scheduler.capacity.resource-calculator</name>
    <value>org.apache.hadoop.yarn.util.resource.DominantResourceCalculator</value>
</property>
```

Defaults for Memory and CPU assigned per VM



Example at left: **n2-highmem8 -> 8 cores, 64GB RAM**.

By default every VM gets two executors.
First some buffer is set (a combination of ratios and hardcoded minimums). YARN is not allowed to use the whole VM.

Then 2 executors share the rest of memory and cores. Each executor has overhead memory fraction of 0.1 by default.

In Dataproc Serverless, with PySpark, memory Overhead is set to 40% of executor memory since python objects live under overhead.

You may change these defaults for all the jobs during cluster creation or per-job during job submission.

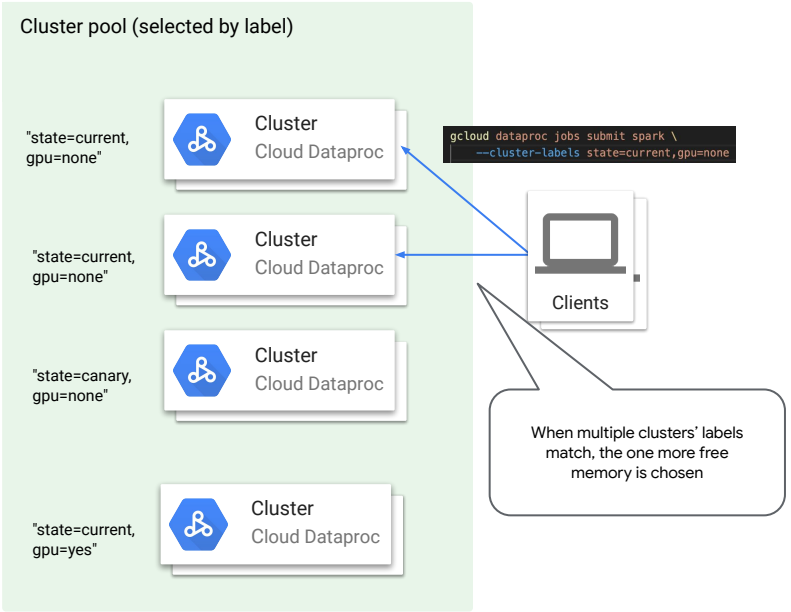
Mostly you keep settings as it is till you hit an OOM error or you decide that executors are too big and wasting resources.

You may also change machine type instead of touching these values - this is mostly what Cloud Support recommends.

Cluster Pooling for canary deployments, dynamic cluster selection

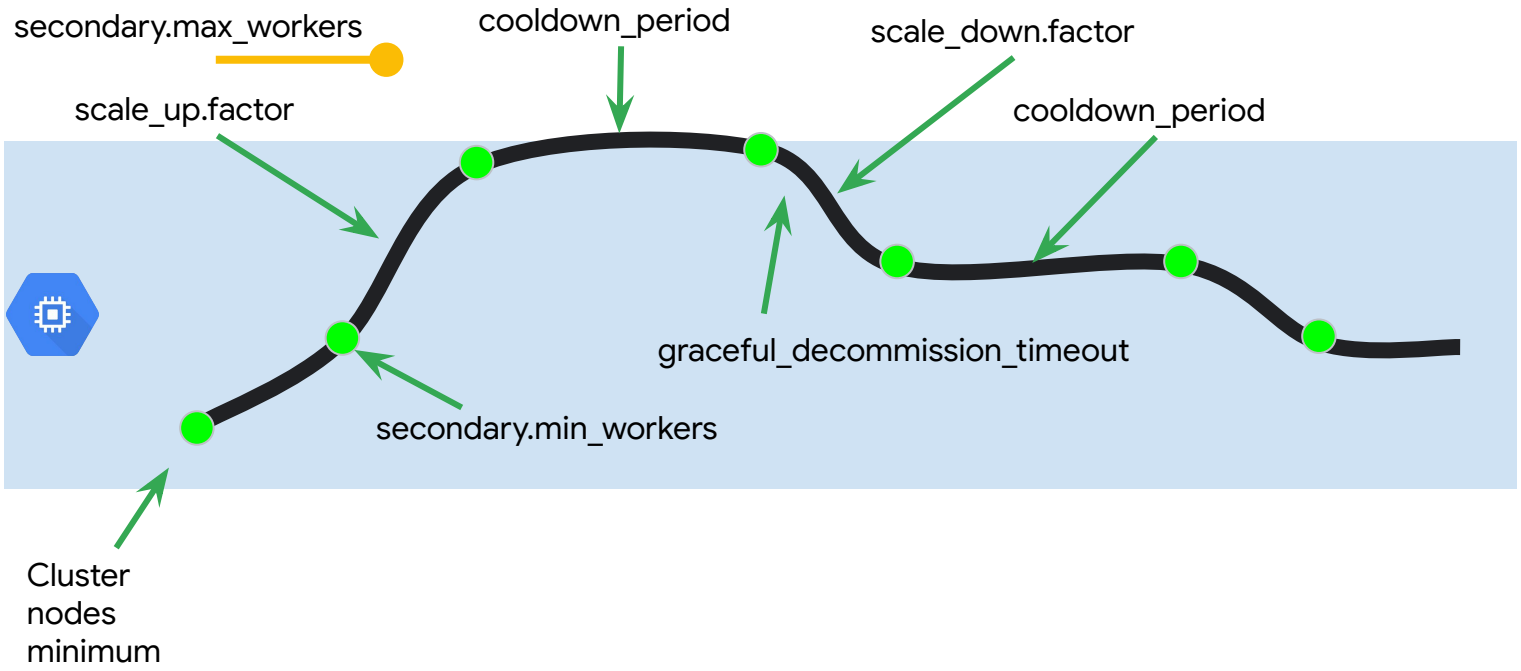
- Cluster labels can be used to select clusters at job submission instead of cluster names
- You may use the same label in multiple clusters, in that case those clusters are “pooled”. The one that has more free memory and in healthy is selected by the Dataproc Submit API
- You may also throttle the number of jobs submitted to a cluster by using

  dataproc.scheduler.max-concurrent-jobs setting



Dataproc Autoscaling

How Cloud Dataproc Auto Scaling works



Autoscaling Policy API

- Define a policy once and **reuse** it for multiple clusters
  - Updating an autoscaling policy **updates all clusters using that policy**
  - An Autoscaling Policy **cannot be deleted while in use**

- Autoscaling can be enabled / disabled at any time
  - **Enable** autoscaling by attaching an autoscaling policy at cluster creation time, or later!
  - **Update** the cluster to use a different autoscaling policy
  - **Disable** autoscaling by detaching the autoscaling policy

Autoscaling Policy Example

```
workerConfig:
  minInstances: 10
  maxInstances: 10
  weight: 1
secondaryWorkerConfig:
  minInstances: 0
  maxInstances: 100
  weight: 1
basicAlgorithm:
  cooldownPeriod: 2m
  yarnConfig:
    scaleUpFactor: 0.05
    scaleDownFactor: 1.0
    scaleUpMinWorkerFraction: 0.0
    scaleDownMinWorkerFraction: 0.0
    gracefulDecommissionTimeout: 1h
```

Graceful Decommission Timeout

Optionally, a **graceful decommission timeout** can be specified in order to give the YARN Resource Manager time to gracefully drain yarn nodes before Dataproc deletes them.

It is recommended to specify a graceful decommission timeout in order to **prevent loss of job progress**.

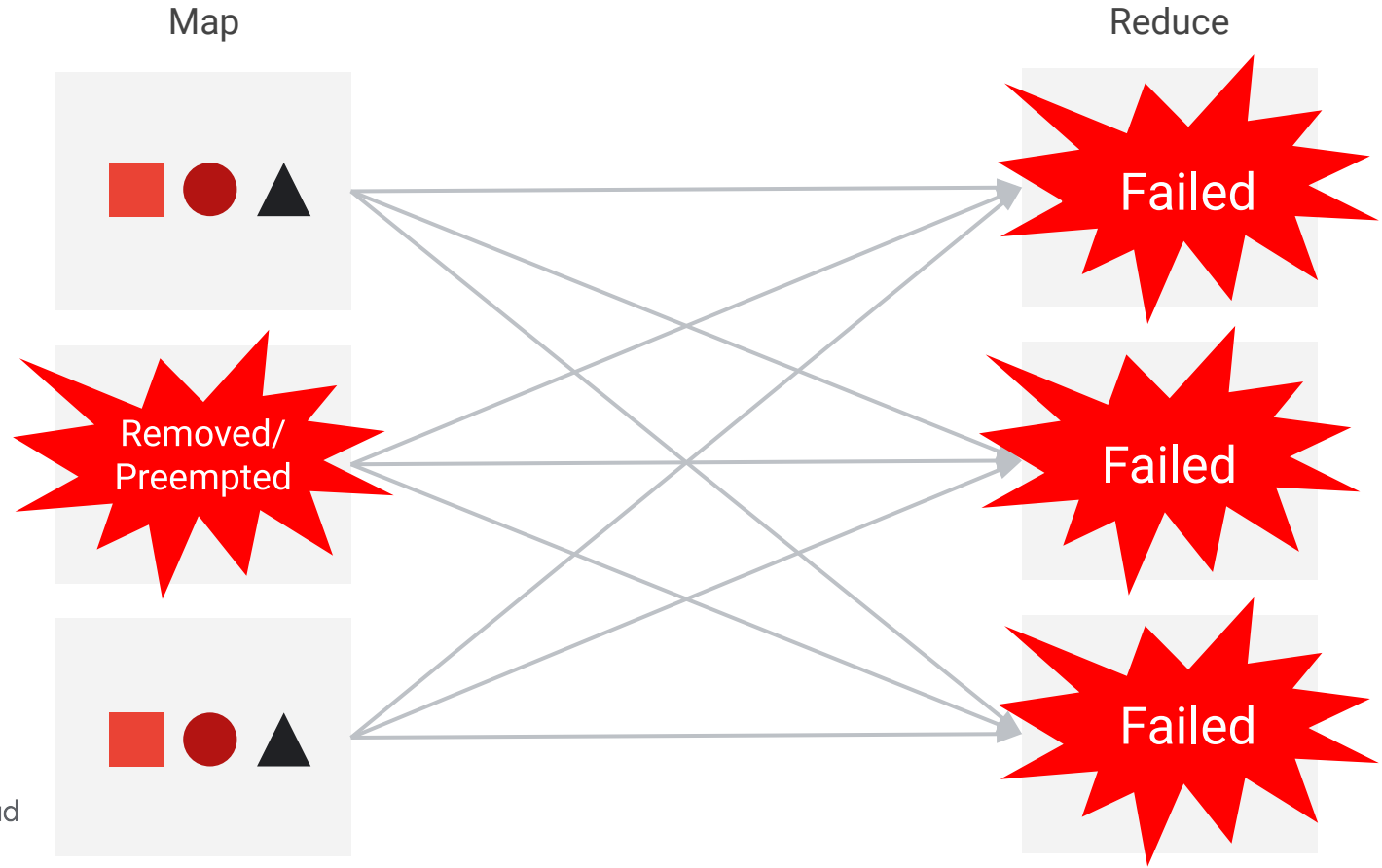
The YARN Resource Manager only marks a node DECOMMISSIONED once **both** these conditions are met:

- All **containers** running on the node have finished
- All **applications** that at some point previously ran a container on the node have finished
  - This is intended to prevent shuffle data loss, which can alternatively be mitigated by using EFM

This means that the graceful decommission timeout should be set to **longer than the duration of the longest application** that will run on the cluster.

Unfortunately, downscaling and machine preemptions impose a problem: loss of intermediate data.

Shuffle with preemption



Solution: Write intermediate shuffle data to the Spark shuffle service on primary workers (EFM)

EFM Challenges

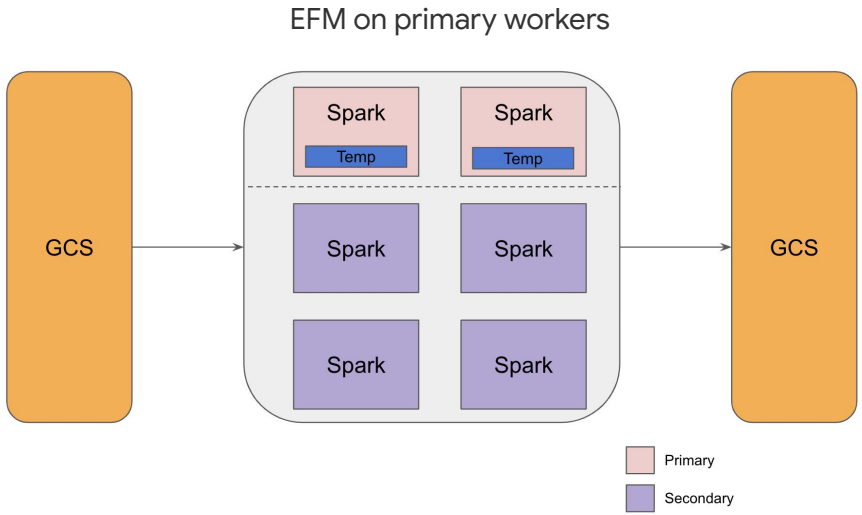
EFM saves shuffle data on primary workers.

EFM Benefits

- All shuffle data are stored on primary workers which are not preemptable nodes
- Improve the Spark job reliability
- Secondary nodes can scale down faster

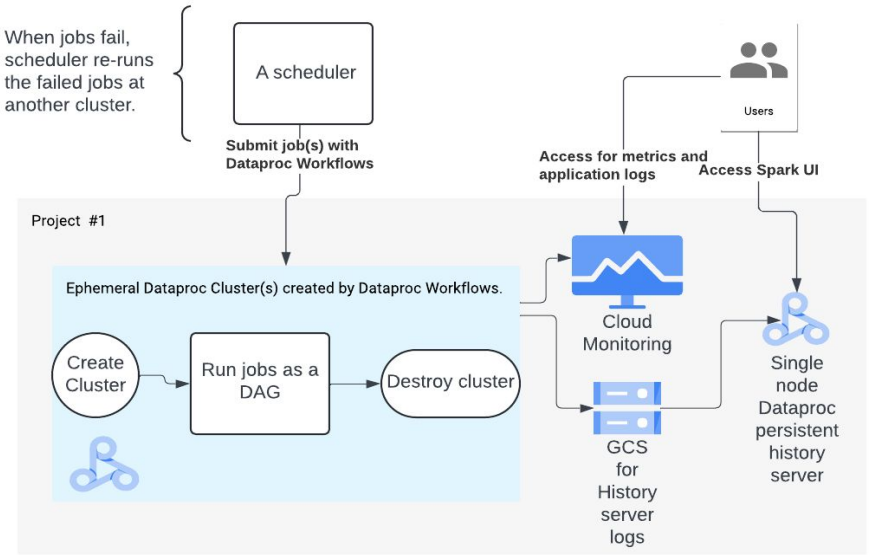
EFM Challenges

- Tuning primary worker nodes (Amount of disks required)
- Hotspotting on primary workers. (All the traffic goes to primary)
- Difficult to scale down primary worker nodes



Ephemeral vs Long-lived clusters

Job Scoped Cluster Architecture



- Cluster is created in minutes
- Several jobs are run serially or in parallel
- Cluster is shutdown
- Dataproc workflows or an external orchestrator can handle the process
- Logs/Metrics are stored durably beyond the lifetime of the cluster

Don’t pay for empty clusters

Long Lived clusters work! Many large customers use in production.

Nevertheless, listen to Google SRE principles:

- Global systems fail globally

- Every single point of failure fails eventually

	Long running	Job scoped
Cost	Even with autoscaling, minimum number of Master and Primary Worker VMs are kept all the time. 3 masters are required for protection against hadoop service failures. **Cost Attribution** is not easy. You may charge per job, create a dashboard for job durations etc. Not optimal.	You only pay for the cluster when you run a job. Cluster start and destroy periods are not charged. Single Master would be enough since we don’t have to keep cluster on all the time. We re-ran jobs in case of a failure. **Cost attribution** is not an issue at all.
Maintenance	When services are down like Hadoop/Yarn services, we have to restart them by logging into the VMs.	After we right size the cluster and jobs, we don’t have to maintain the clusters as we will be creating a new one every time.
Version Conflicts	All jobs should have the same version. When a new version of Spark is available, all jobs have to be migrated at the same time. Upgrades will be painful and mostly teams try to avoid.	We may run multiple clusters in parallel with different versions and configurations.
Head of line blocking	Unrelated jobs may block others from running by consuming all the resources in the cluster. YARN queues is the solution but it simply complicates the maintenance. You have to guess the job demands in advance.	Large jobs may be scheduled in their own clusters so that they don’t interfere with others.
Monitoring	Since everything runs in a single cluster, there is a single place to look.	By moving logs and metrics out of the cluster, we don’t have an issue of having multiple clusters.
Disaster Recovery	We have to move the clusters to another region. Probably a manual operation.	We will be running multiple clusters at the same time. At any point in time we can run them in different regions so that we are never bound to a single region. Note that, data source should also be taken into consideration when running in multiple regions.
IP address requirements	Shared clusters with limits have more predictable IP limits.	By setting auto scaling limits for jobs, we can constrain the number of IPs. Where there are not enough IPs, we may use separate VPCs or use large machines.

Scheduled Cluster Deletion

Even if you don’t use job scoped clusters, you may introduce scheduled deletion to avoid paying for idle cluster costs. Remember to recreate when you need them.

Flag	Description	Finest Granularity	Min Value	Max Value
`--max-idle`[1]	The duration from the moment when the cluster enters the idle state to the moment when the cluster starts to delete. Provide the duration in `IntegerUnit` format, where the unit can be "s, m, h, d" (seconds, minutes, hours, days, respectively). Examples: "30m" or "1d" (30 minutes or 1 day from when the cluster becomes idle).	1 second	5 minutes	14 days
`--expiration-time`[2]	The time to start deleting the cluster in ISO 8601 datetime format. An easy way to generate the datetime in correct format is through the Timestamp Generator. For example, "2017-08-22T13:31:48-08:00" specifies an expiration time of 13:21:48 in the UTC -8:00 time zone.	1 second	10 minutes from the current time	14 days from the current time
`--max-age`[2]	The duration from the moment of submitting the cluster create request to the moment when the cluster starts to delete. Provide the duration in `IntegerUnit` format, where the unit can be "s, m, h, d" (seconds, minutes, hours, days, respectively). Examples: "30m" (30 minutes from now); "1d" (1 day from now).	1 second	10 minutes	14 days

Connectors

Handy OSS connectors

Many Open Source connectors are provided to make interactions with storage systems easier.

Spark - BigQuery

Read Write BQ with Dataframes.

Filter pushdown, Query pushdown

Hadoop - GCS

GCS acts like a Hadoop Compatible File System (HCFS)

Hive - BigQuery

Use Hive as an interface for BigQuery

Spark - BigTable

Read/Write BigTable with Dataframes

Spark - PubSubLite

Read/Write from PubSubLite in streaming microbatches

Common Debugging Questions for Infra team

My job doesn’t start! (1)

In Dataproc GCE there are two main queues:

- Dataproc Job queue (outside of cluster)
- Yarn pending queue (inside of the cluster)

Dataproc Job Queue is only for Client mode. To protect Master VMs, Dataproc API puts the jobs in queue. The capacity is a function of Master memory and parameters. Checkout slide . Metrics that checks state:

`job/state` GA *(project)*
Job state

GAUGE, BOOL,
cloud_dataproc_job

Indicates whether job is currently in a particular state or not. True indicates in that state and False indicates exited that state. Sampled every 60 seconds. After sampling, data is not visible for up to 120 seconds.
`state`: The state of the job such as PENDING or RUNNING.

`cluster/job/duration` GA *(project)*
Job state duration

DELTA, DISTRIBUTION, s
cloud_dataproc_cluster

The time jobs have spent in a given state. Sampled every 60 seconds. After sampling, data is not visible for up to 120 seconds.
`job_type`: The type of job such as HADOOP_JOB or SPARK_JOB.
`state`: The state of the job such as PENDING or RUNNING.

Yarn Pending applications from cloud monitoring or YARN resource manager UI:

`cluster/yarn/apps` GA *(project)*
YARN active applications

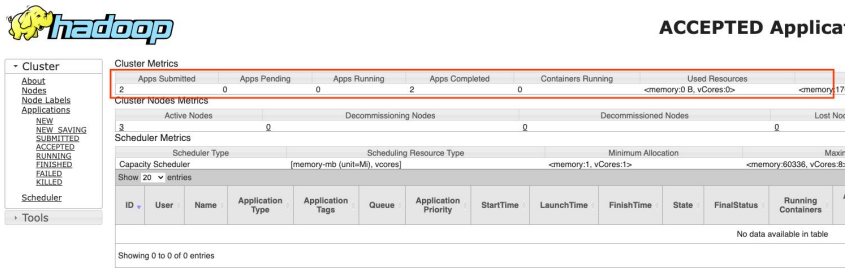
GAUGE, INT64, 1
cloud_dataproc_cluster

Indicates the number of active YARN applications. Sampled every 60 seconds. After sampling, data is not visible for up to 120 seconds.
`status`: The status of YARN application such as running, pending, completed, failed, killed.

`cluster/yarn/containers` GA *(project)*
YARN containers

GAUGE, INT64, 1
cloud_dataproc_cluster

Indicates the number of YARN containers. Sampled every 60 seconds. After sampling, data is not visible for up to 120 seconds.
`status`: The status of YARN container such as allocated, pending, reserved.



https://cloud.google.com/dataproc/docs/concepts/jobs/troubleshoot-job-delays
https://cloud.google.com/dataproc/docs/concepts/jobs/troubleshoot-jobs#console

My job doesn’t start! (2)

Job doesn’t start since we don’t have enough capacity:

- Client mode: play with the job concurrency parameters or move to Driver Node Groups
- Cluster mode: jobs don’t wait in Dataproc Job queue

For both modes, jobs may wait in YARN queues. So we need

- larger clusters (increase auto scaling limit)
- or someone consuming too much resource, limit them with `spark.dynamicAllocation.maxExecutors`, `spark.dynamicAllocation.executorAllocationRatio`

But job owners can override them! Spark UI environment tab shows all the parameters of jobs.

Why am I getting OOM?

https://cloud.google.com/dataproc/docs/support/troubleshoot-oom-errors -> how to discover processes that are killed by earlyoom-killer

The Job owner can change the driver and executor memory to fit to their needs: spark.executor.memory[OverheadRatio], spark.driver.memory[OverheadRatio]

These should fit into your nodes though!

But mostly, OOM errors are due to large partitions, doing operations inside Driver etc. This has nothing to do with the Infrastructure.

Infra is only involved if the machine type offered does not have enough memory for that specific job.

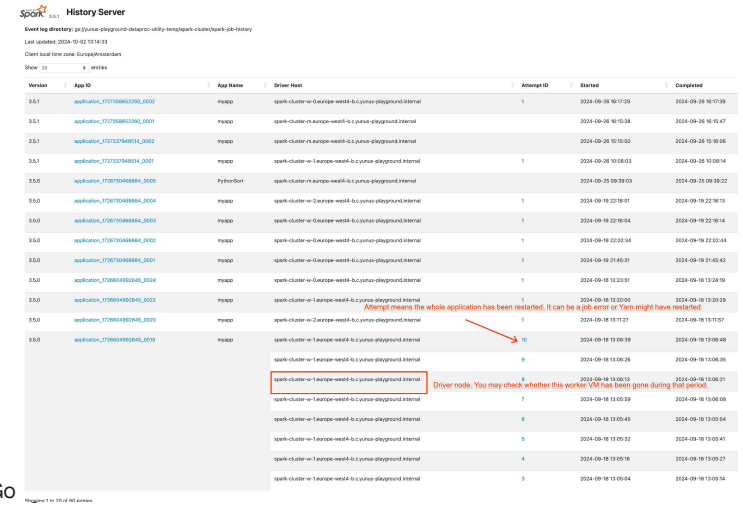
My application is slow! (1)

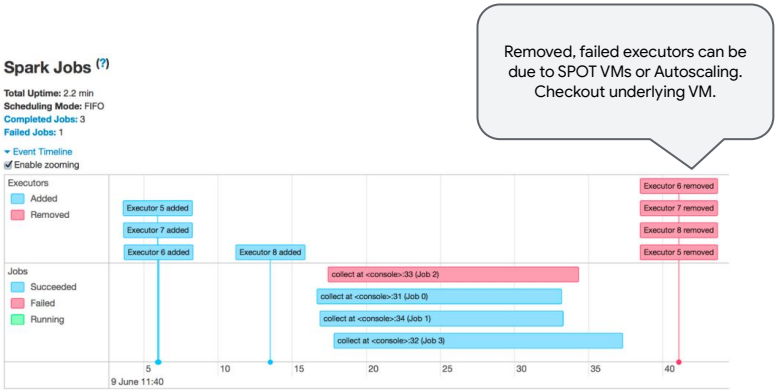
- Is the application using
  - proper caching,
  - less shuffle,
  - data spilling to disk
  - enough partitioning?
- Use Spark UI to debug. Find the longest Job and then check its SparkSQL plan.

My application is slow! (2)

Preemption or Scaling-in may kill processes

- Autoscaling can kill executors and even kill the drivers. Killing driver starts the whole application from scratch, killing executor has partial impact.
- Killed driver can be seen in SparkUI or Yarn.
- Preempted VMs exist under [audit logs]

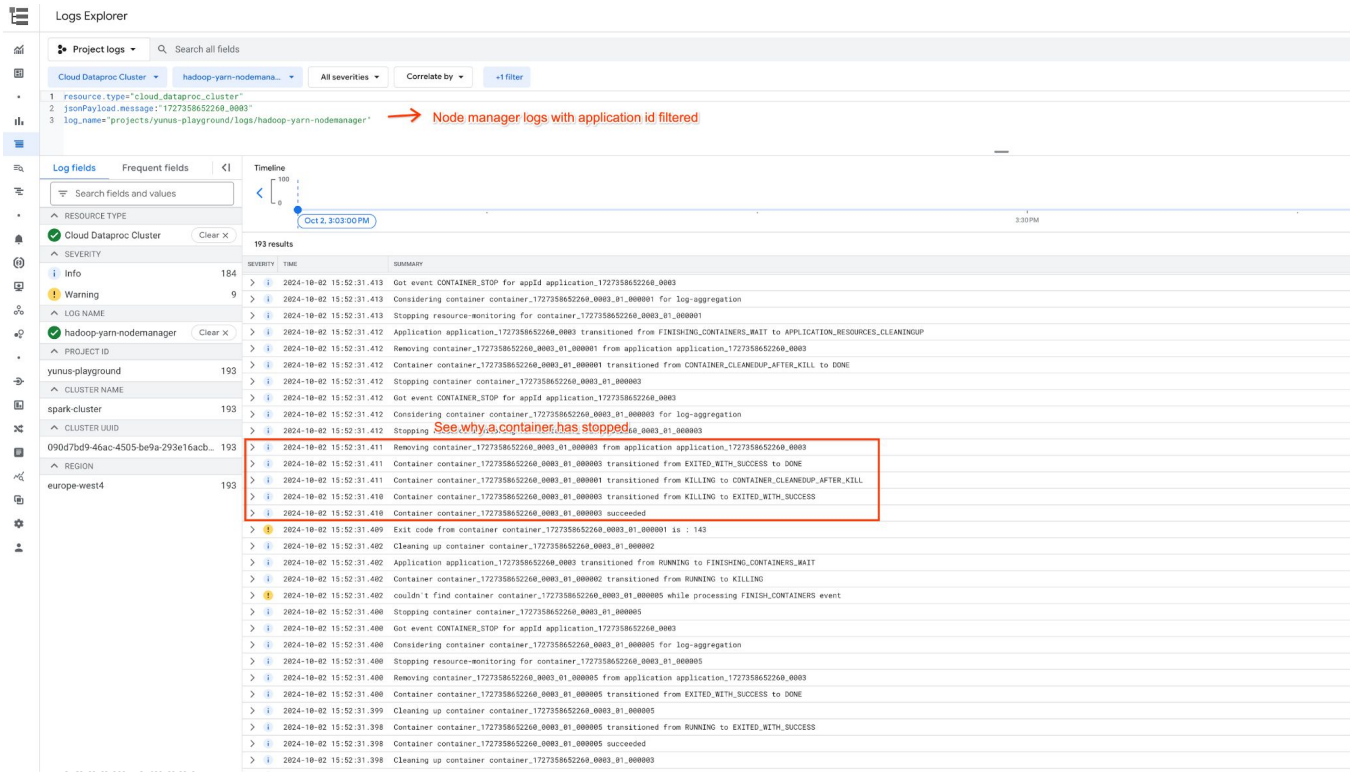




Removed, failed executors can be due to SPOT VMs or Autoscaling. Checkout underlying VM.

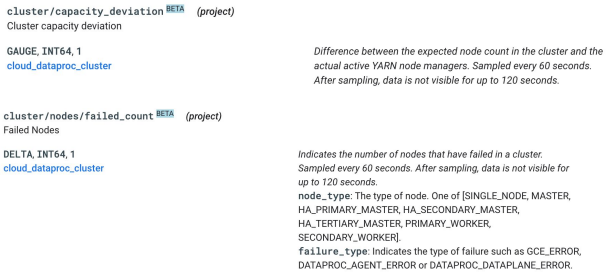
https://docs.databricks.com/en/optimizations/spark-ui-guide/failing-spark-jobs.html

Yarn Node Manager logs show state changes



My application is slow! (3)
Have enough VMs?

- Does the cluster have enough resources to run executors?
  - Max parallelism = min(partitions, executors x cores per executor)
- If the problem is executors, then you may increase cluster size.
- Checkout auto scaling behavior from
  - Logs: dataproc.googleapis.com/autoscaler
  -



`cluster/capacity_deviation` BETA *(project)*
Cluster capacity deviation

GAUGE, INT64, 1
cloud_dataproc_cluster

Difference between the expected node count in the cluster and the actual active YARN node managers. Sampled every 60 seconds. After sampling, data is not visible for up to 120 seconds.

`cluster/nodes/failed_count` BETA *(project)*
Failed Nodes

DELTA, INT64, 1
cloud_dataproc_cluster

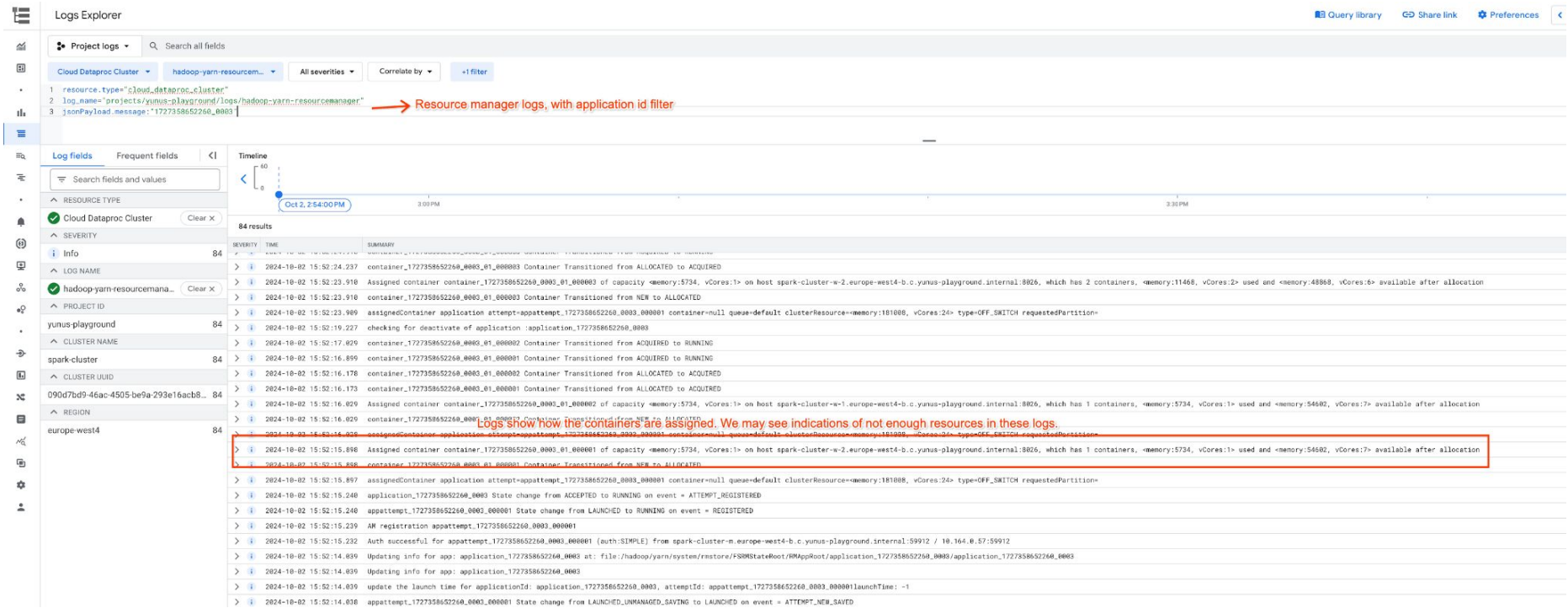
Indicates the number of nodes that have failed in a cluster. Sampled every 60 seconds. After sampling, data is not visible for up to 120 seconds.
`node_type`: The type of node. One of [SINGLE_NODE, MASTER, HA_PRIMARY_MASTER, HA_SECONDARY_MASTER, HA_TERTIARY_MASTER, PRIMARY_WORKER, SECONDARY_WORKER].
`failure_type`: Indicates the type of failure such as GCE_ERROR, DATAPROC_AGENT_ERROR or DATAPROC_DATAPLANE_ERROR.

- What is constrained? Memory or CPU?

- Default YARN capacity scheduler only checks Memory.

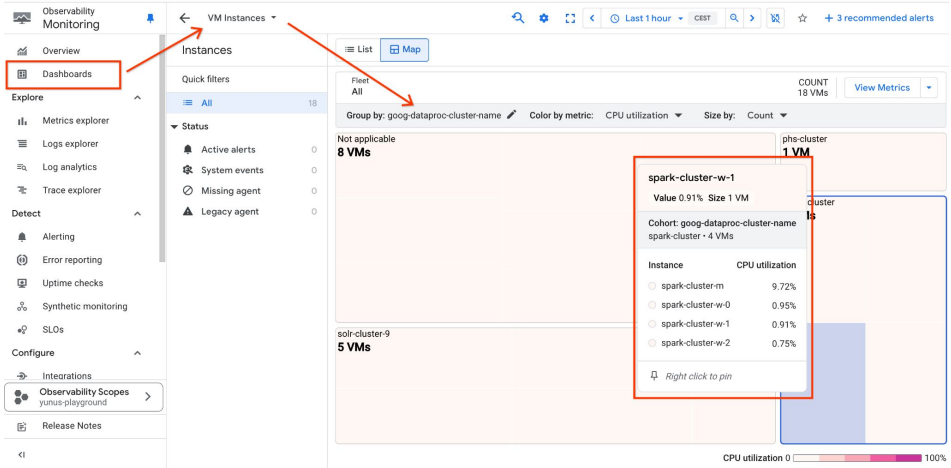
- if you set it to DominantResourceCalculator, then both CPU and Memory are considered for scaling.

How Yarn Resource manager assign resources, See what is constrained?



Have we overprovisioned the cluster?

- Install Ops Agent to get VM level utilization.
- Low CPU utilization and Low Memory utilization at VM level shows waste.

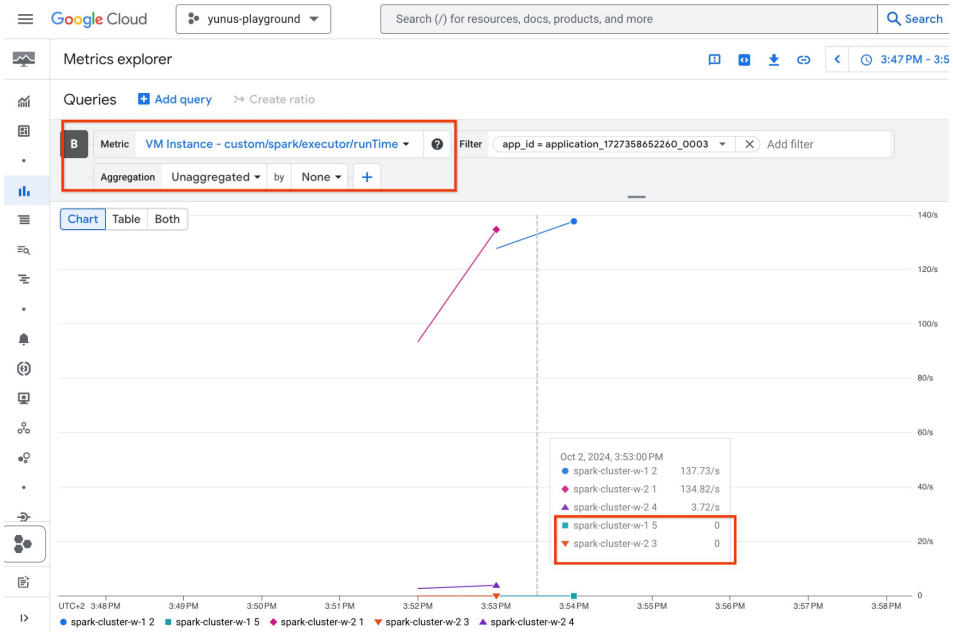


Ideally Application owners should adjust their Executor and Driver sizes.

Dynamic Allocation gets resources back for long running jobs, but for short jobs, manual intervention is needed.

Set `spark.executor.instances=1` to start with only a single executor. Default is 2.

Have we overprovisioned the cluster?

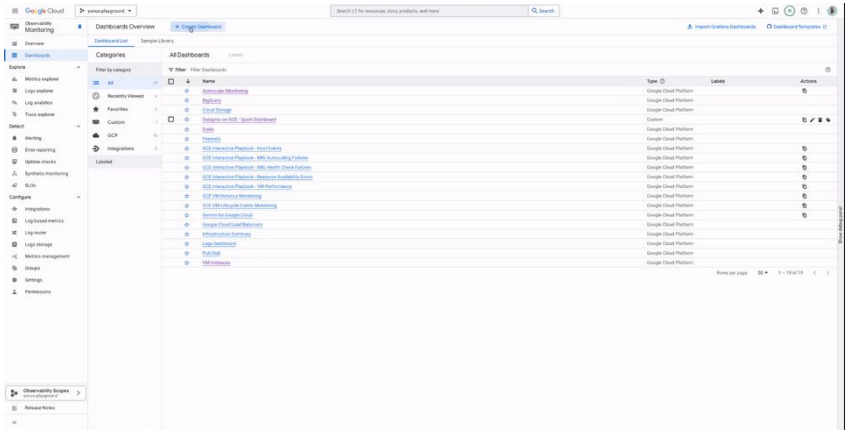


Enable Spark metrics and then for an application, we can see whether it used its executors or not.

Custom dashboard for Dataproc on GCE monitoring

We prepared a custom dashboard for Dataproc on GCE. You may create it from the JSON definition that we provide.

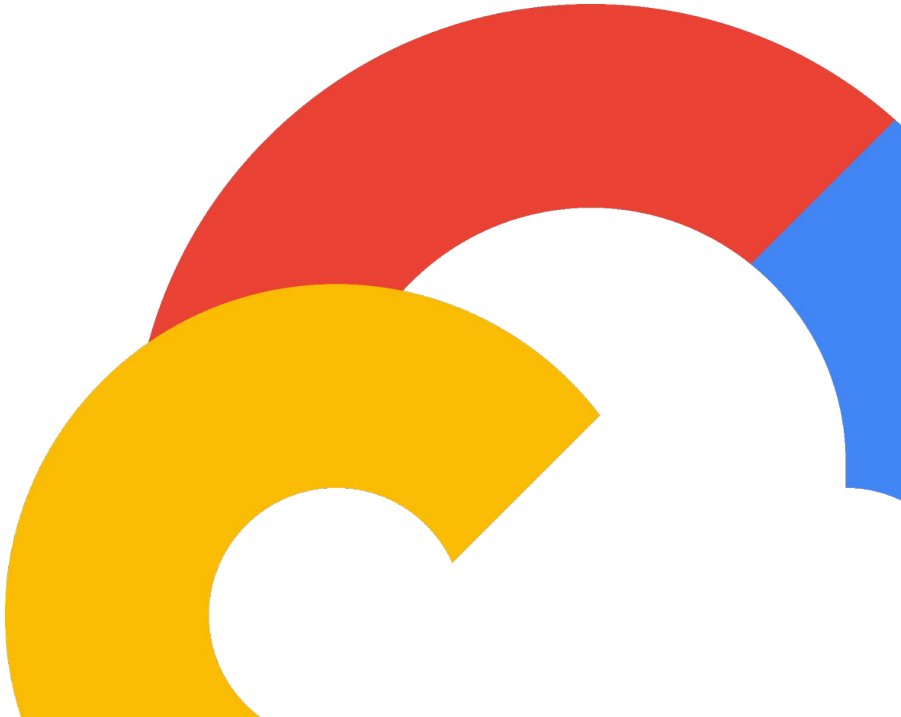
You may set this Dashboard per project, or to a central project where all the metrics and logs are forwarded.



Live Debugging

Let’s go over an example code and see what we can improve

Thank you



Feedback form:
https://forms.gle/bTPYRiJe5FAvX3hPA

Google Cloud