

# Spark Performance and Profiling

# Spark scheduler

- Multiple jobs can be submitted to a Spark context from different threads
- By default Spark uses FIFO → Jobs will wait for each other
- Spark can be set to “fair” mode → uses round robin to assign tasks between jobs
  - ***spark.scheduler.mode = “FAIR”***  

```
spark.conf().set("spark.scheduler.mode", "FAIR");
```
- Scheduling pools – a way to share resources between users
  - A pool can be configured per thread:  

```
spark.sparkContext().setLocalProperty("spark.scheduler.pool", "user_pool1");
```

# Partitioning of a text file

Case 1: Reading an uncompressed files

→ Every partition is a Hadoop file block (usually 64/128/256 mb)

Question: What happens if a line is split between two blocks?

→ Answer: The mapper reads two blocks 😊

Benefits:

- good parallelism
- less GC pressure

Drawbacks:

- IO

# Partitioning of a text file

## Case 1: Reading a compressed files

- Every partition is a Hadoop file – compressed file can't be split!
- The mapper will read all the split blocks of a single file`

## Benefits:

- Less IO

## Drawbacks:

- May result in big partitions
- CPU intensive

“But not all compressions are created equal!!!“

# Common compressions

- The tradeoff is usually between Speed vs Storage
- By default use Snappy/lz4

Compression	Splittable (on Hadoop)	Speed	Compression RATIO
Snappy	No	Extremely fast	Low
LZO	No (Yes if indexed)	Fast	Medium
Gzip	No	Medium	High
Bzip2	Yes	Slow	High
lz4	Yes	Fastest	Low

# When to compress what?

- Splittable or not?
  - ◆ Up to ~1GB you can use non splittable method
  - ◆ You can take advantage of Spark partitions:  
e.g. saving 10GB dataset with 100 partitions yields 100 files of about 100MB each
- If splittable isn't needed - *Snappy* is the best option
- Prefer *GZip* in case you REALLY need to save space
- If the files are greater than 1GB - use splittable method  
*lz4* will be the best option
- Spark uses *lz4* for intermediate files

# What about wholeTextFile

Remember that:

```
RDD<Tuple2<String, String>> textRDD = spark.sparkContext().wholeTextFiles("path", 10);
```

???

- The executor reads the entire file
- We don't like it! why??
  - ◆ Lack of parallelism
  - ◆ GC
- Might be necessary with problematic file formats (e.g. XML)

***You should definitely try to avoid that!***

# Spark Files Rule of thumb

- When saving files aim these files not to be too big and not too small
- Aim to the file system block size
- Aim to have number of partition correlate to your resources (workers)
-



# Data locality in Spark

- The driver knows where every partition is located
- Locality levels (by priority):
  1. **PROCESS\_LOCAL** – data is in the same JVM as the running code
  2. **NODE\_LOCAL** – data is on the same node
  3. **NO\_PREF** – data is accessible equally quickly from anywhere
  4. **RACK\_LOCAL** – data is on the network, in a different server in the same rack
  5. **ANY** – data is on the network and not in the rack

# Data locality in Spark

Tasks (144)

Page: [1](#) [2](#) [>](#)

2 Pages. Jump to  . Show  items in a page.

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	Task Deserialization Time	GC Time	Result Serialization Time	Getting Result Time	Peak Execution Memory	Input Size / Records	Write Time	Shuffle Write Size / Records	Er
0	340886	0	SUCCESS	PROCESS_LOCAL	1 / 10.103.252.130 <a href="#">stdout</a> <a href="#">stderr</a>	2018/07/16 08:47:33	3 s	18 ms	16 ms		0 ms	0 ms	32.3 MB	36.1 MB / 942361	31 ms	1080.2 KB / 23471	
1	340887	0	SUCCESS	PROCESS_LOCAL	4 / 10.103.226.135 <a href="#">stdout</a> <a href="#">stderr</a>	2018/07/16 08:47:33	4 s	18 ms	18 ms	84 ms	0 ms	0 ms	32.3 MB	33.9 MB / 885604	23 ms	674.7 KB / 12930	
2	340888	0	SUCCESS	PROCESS_LOCAL	10 / 10.103.251.177 <a href="#">stdout</a> <a href="#">stderr</a>	2018/07/16 08:47:33	4 s	18 ms	9 ms	0.1 s	0 ms	0 ms	32.3 MB	33.8 MB / 884479	40 ms	664.2 KB / 12635	
3	340889	0	SUCCESS	PROCESS_LOCAL	11 / 10.103.236.113 <a href="#">stdout</a> <a href="#">stderr</a>	2018/07/16 08:47:33	3 s	20 ms	18 ms	43 ms	0 ms	0 ms	32.3 MB	33.7 MB / 879479	0.1 s	1118.6 KB / 23326	
4	340890	0	SUCCESS	PROCESS_LOCAL	0 / 10.103.232.48 <a href="#">stdout</a> <a href="#">stderr</a>	2018/07/16 08:47:33	4 s	19 ms	18 ms	0.8 s	0 ms	0 ms	32.3 MB	33.9 MB / 885207	93 ms	1092.6 KB / 22581	
5	340891	0	SUCCESS	PROCESS_LOCAL	3 / 10.103.239.124 <a href="#">stdout</a> <a href="#">stderr</a>	2018/07/16 08:47:33	3 s	22 ms	15 ms	94 ms	0 ms	0 ms	32.3 MB	33.8 MB / 879479	20 ms	1360.0 KB / 29419	
6	340892	0	SUCCESS	PROCESS_LOCAL	5 / 10.103.252.21 <a href="#">stdout</a> <a href="#">stderr</a>	2018/07/16 08:47:33	3 s	21 ms	11 ms	40 ms	0 ms	0 ms	32.3 MB	33.9 MB / 882707	19 ms	1046.3 KB / 21506	
7	340893	0	SUCCESS	PROCESS_LOCAL	6 / 10.103.224.181 <a href="#">stdout</a> <a href="#">stderr</a>	2018/07/16 08:47:33	3 s	23 ms	16 ms	0.5 s	0 ms	0 ms	32.3 MB	34.0 MB / 889479	21 ms	614.9 KB / 11623	

# Data locality in Spark


How does it work?

→ Spark attempts to schedule a task as close to the process it can

→ Scenario:

- ◆ Node A has Executor A and its busy
- ◆ Node B has Executor B and its idle – it also processed all of his local partitions...

→ What will Spark do ?

- ◆ Wait.. 
- ◆ Till a threshold is reached – and then the task is rescheduled
- ◆ Like everything - this behavior can be changed

# Caching

- One of the strongest features of Spark is the ability to cache an RDD.
- Spark can cache the items of the RDD in memory or on disk.
- You can avoid expensive re-calculations this way.
- The `cache()` and `persist()` store the content in memory.
- You can provide a different storage level by supplying it to the `persist` method.

# Caching

- You can store the RDD in memory, disk or try in memory and if it fails fallback to disk.
- In the memory it can be stored either deserialized or serialized.
- Note that serialized is more space-efficient but CPU works harder.
- You can also specify that the cache will be replicated, resulting in a faster cache in case of partition loss.

# DataFrame caching

- Spark caches DataFrames using their columnar format
- By default every column is compressed
- Configurable:
  - ◆ `spark.sql.inMemoryColumnarStorage.compressed = true`
  - ◆ `spark.sql.inMemoryColumnarStorage.batchSize = 10000`

# In-Memory Caching

- When using memory as a cache storage, you have the option of either using serialized or de-serialized state.
- Serialized state occupies less memory at the expense of more CPU.
- However, even in a de-serialized mode, Spark still needs to figure the size of the RDD partition.
  - Spark can't know beforehand the size of the partition
- Spark uses the SizeEstimator utility for this purpose.
- Sometimes can be quite expensive (for complicated data structures)

# In-Memory Caching

What is the real size in memory??

Jobs	Stages	Storage	Environment	Executors	SQL	JDBC/ODBC Server
Storage						
RDDs						
RDD Name		Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
*Filter (isnotnull(r#38767) && (r#38767 = 1)) +- *RunningWindowFunction [user_id#37626, dt#37682, platform_id#37629, vip_level#38298, vip_points#38297, LTV#37631, MaxLTV#38299, first_vip_date#38301, price#37627, row_number() window specification (user_id#37626, dt#37682 DESC NULLS LAST, specified window frame (RowFrame, unbounded preceding \$(), current row \$())) AS r#38767], [user_id#37626], [dt#37682 DESC NULLS LAST] +- *Sort [user_id#37626 ASC NULLS FIRST, dt#37682 DESC NULLS LAST], false, 0 +- Exchange hashpartitioning (user_id#37626, 200) +- InMemoryTableScan [user_id#37626, dt#37682, platform_id#37629, vip_level#38298, vip_points#38297, LTV#37631, MaxLTV#38299, first_vip_date#38301, price#37627] +- InMemoryRelation [user_id#37626, dt#37682, platform_id#37629, vip_level#38298, vip_points#38297, LTV#37631, MaxLTV#38299, first_vip_date#38301, price#37627], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas) +- InMemoryTableScan [user_id#37626, dt...		Disk Serialized 1x Replicated	200	100%	612.6 KB	420.6 KB
*HashAggregate(keys=[user_id#35512, platform_user_id#35513, device_type#35555, model#35533, RegDate#40170, First_Deposit_Date#39585, RegPlatform#36191, age#40173, gender#40174, country_code#40175, Traffic_Type#36192, ReferralType#40177, Language#37859, Currency#38588, vip_level#40180, Approve_Emails#40181, Allow_Push#36622, IsTest#40183, balance#40184L, FB_Date#39813, Email#37858, FB_ID#36090, IsCrown#40188], functions=[], output=[user_id#35512, platform_user_id#35513, device_type#35555, model#35533, RegDate#40170, First_Deposit_Date#39585, RegPlatform#36191, age#40173, gender#40174, country_code#40175, Traffic_Type#36192, ReferralType#40177, Language#37859, Currency#38588, vip_level#40180, Approve_Emails#40181, Allow_Push#36622, IsTest#40183, balance#40184L, FB_Date#39813, Email#37858, FB_ID#36090, IsCrown#40188]) +- *HashAggregate(keys=[user_id#35512, platform_user_id#35513, device_type#35555, model#35533, RegDate#40170, First_Deposit_Date#39585, RegPlatform#36191, age#40173, gender#40174, country_code#4...		Disk Serialized 1x Replicated	200	100%	441.6 MB	350.9 MB



# Unpersist

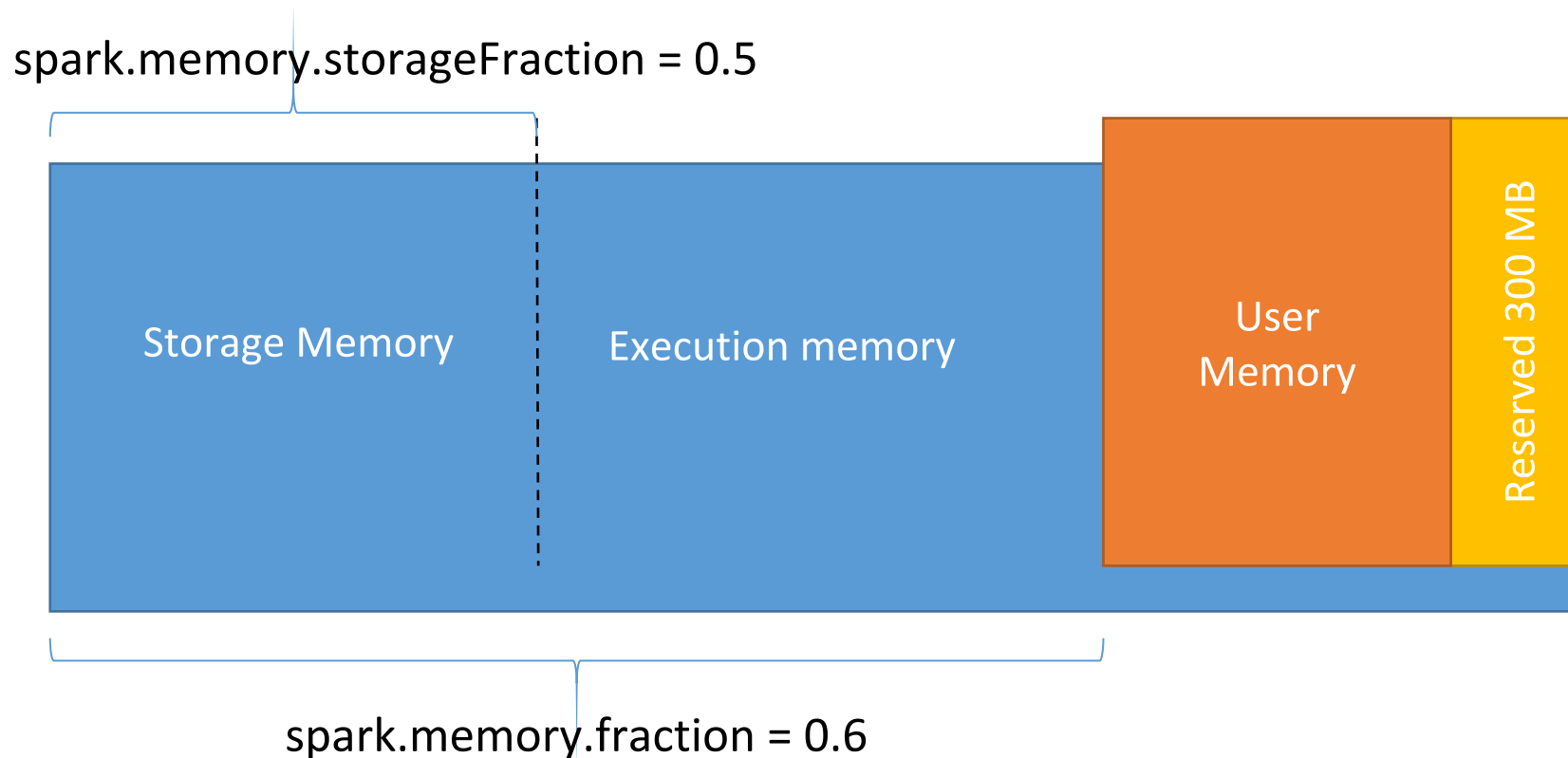
- Caching is based on LRU.
- If you don't need a cache, remove it with the `unpersist()` method.

# Memory Management in Spark

- Memory usage in spark falls under one of the following:
  - ◆ Execution memory - refers to the memory who used for computation (shuffles, joins, sorts, aggregations and etc...)
  - ◆ Storage memory - refers to the memory who used for caching data and propagating data across the cluster
  - ◆ User memory - user data structure, internal Spark metadata, guarding from oom and more...
- First 2 categories share the same memory region (AKA 'M')
- If needed, execution may evict memory from storage.  
It can't, however, take storage region below some threshold 'R'
- 'R' will always left for the storage

# Memory Management in Spark

- The Unified Model - storage and execution memory are dynamic!!
- Configuration is percent from the heap size
- You probably better don't change it...



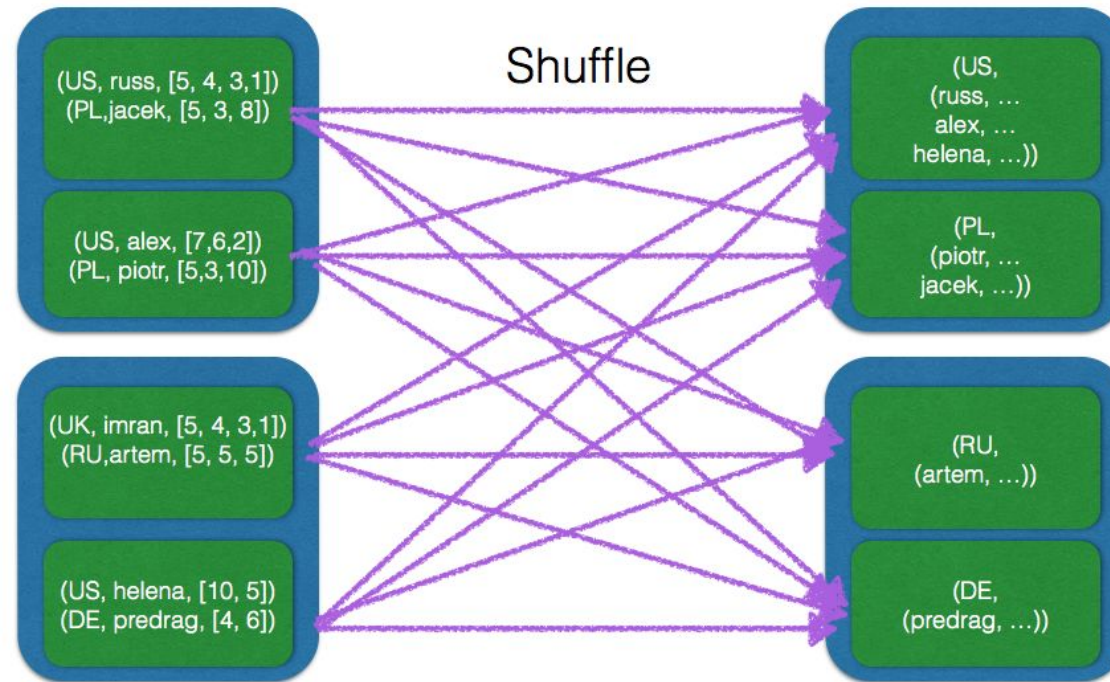
# Shuffle

- Shuffle operations repartition the data across the network.
- Can be very expensive operations in Spark.
- You must be aware where and why shuffle happens.
- Order is not guaranteed inside a partition.
- Popular operations that cause shuffle are: *groupBy\**, *reduceBy\**, *sort\**, *aggregateBy\** and *join/intersect* operations on multiple RDDs.

# Everyday I'm shuffling

GroupBy Requires ALL of the  
Previous RDD's Partitions to Compute Any Single Partition

```
val rdd = sc.cassandraTable("sc2","points")  
.groupBy(_.getString("country"))
```



# Shuffle

- Shuffle occurs through two steps:
  - ◆ Emitting the data – called “mapping” and the task is a “mapper” task
  - ◆ Consuming the data – called “reducing” and the task is a “mapper” task
- The mapper writes the shuffled data to disk
- The reducer reads the shuffled data from different mappers (using sockets – no direct access to files)

# Hash Shuffle

- *spark.shuffle.manager = hash*
- The default shuffle before Spark 1.2
- Each mapper creates one file per reducer that is using it later on
- M mappers, R reducers, C cores will create  $M * R$  files:
  - ◆ If we have 50k partitions will end up with  $50k * 50k = 2500k$  (that is 2.5 Billion files!)
  - ◆ REAL network issue
  - ◆ Amount of open file per machine
  - ◆ Output buffer size
  - ◆ And much more...
- Can reduce the number of files to  $E * C * R$  instead of  $M * R$ :
  - ***spark.shuffle consolidateFiles = true***

# Sort Shuffle

- *spark.shuffle.manager = sort*
- The default shuffle after Spark 1.2
- Data is sorted, written into one file and indexed
- Assume: M mappers, R reducers: 2M files are created
- 50K partitions will yield 100k files
- Every mapper creates two files:
  - ◆ data output file sorted by reducer id
  - ◆ Index file for the data output file



# Sort Shuffle

- But what if we have a few partitions? We need a fallback
- A version of HashShuffle is used
  - ◆ ByPassMergeSortShuffle
  - ◆ When the numbers of “reducers” is less than the threshold
- Configurable:
  - ***spark.shuffle.sort.bypassMergeThreshold = 200***

# Sort Shuffle

The mapper :

- Locally combine results – for example reduceByKey
- Outputs the records – for groupBy or join

Uses a special HashMap implementations:

- AppendOnlyMap – only in memory
- ExternalAppendOnlyMap – in memory and backed by disk

Reducer will try to combine the records in memory and spill to disk if needed

# Spilling to disk

- Spilling to disk is slow
- You can see in the Spark UI whether if you are spilling
  - ◆ Shuffle Spill (Memory) – is the data size in the memory before the spill
  - ◆ Shuffle Spill (Disk) – is the data size on the disk (usually compressed)
- Try to use more reducers (more partitions)
- Try to reduce the amount of data in shuffle

Shuffle Write Size / Records	Shuffle Spill (Memory)	Shuffle Spill (Disk)
23.3 MB / 96	184.9 GB	4.4 GB
23.3 MB / 96	185.0 GB	4.4 GB

# Physical plan optimizations

- Broadcast join – broadcasts one of the DataFrames
  - ◆ Collects the DataFrame
  - ◆ Passes it as a broadcast parameter
- Occurs when one of the DataFrames is smaller than the threshold
  - ◆ ***spark.sql.autoBroadcastJoinThreshold = 10485760 (10MB)***

# File Formats and Spark

Choosing the right file format is crucial!

→ It affects performance and stability

Formats without schema:

- JSON
- CSV
- Text

Formats with Schema:

- Avro
- Parquet
- ORC

# To Schema or not to Schema?

## Pros:

- We want a clear API between sub-systems
- Schemas provide Spark more space for optimizations
- We are used to schemas (DB, API contracts)

## Cons:

- Schema management – migrations, versioning ..
- Less flexible

# Avro

- Binary row based container data format
- The file format:
  - Metadata (schema in JSON format, # blocks, # records .. )
  - data blocks that contain rows (can be compressed)
- Fast serialization and deserialization
- Supported compressions: Snappy and Deflate (GZip)
- Not built-in in Spark
  - Use DataBricks spark-avro package

# Parquet

- Columnar binary file format
- The file format:
  - Blocks of column chunks + metadata
  - File metadata - index of columns and chunks
- Supports compression per column: Snappy, Gzip, lz4 and others..
- Optimized for nested data structures, nested columns are "flattened":
  - {"social": {"likes": 1}} => "social.likes"
- Good Spark support

```
4-byte magic number "PAR1"
<Column 1 Chunk 1 + Column Metadata>
<Column 2 Chunk 1 + Column Metadata>
...
<Column N Chunk 1 + Column Metadata>
<Column 1 Chunk 2 + Column Metadata>
<Column 2 Chunk 2 + Column Metadata>
...
<Column N Chunk 2 + Column Metadata>
...
<Column 1 Chunk M + Column Metadata>
<Column 2 Chunk M + Column Metadata>
...
<Column N Chunk M + Column Metadata>
File Metadata
4-byte length in bytes of file metadata
4-byte magic number "PAR1"
```

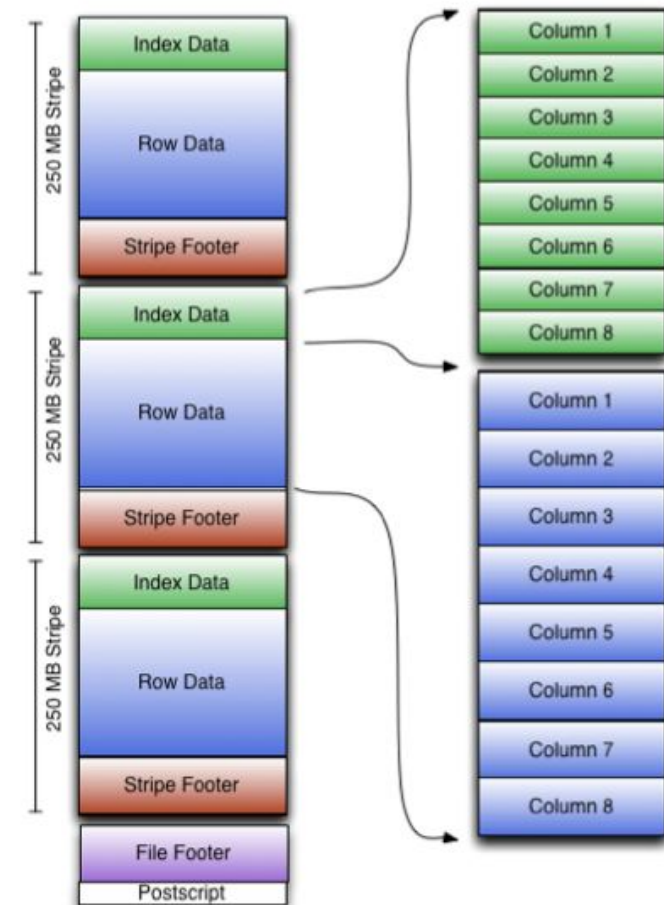


# Parquet types

- Primitive types:
  - Boolean – 1 bit
  - INT32, INT64, INT96: X bit signed ints
  - FLOAT, DOUBLE
  - BYTE\_ARRAY
- Strings are stored as byte arrays with UTF8 encoding
- Complex types are supported through logical type definition
- **No null type!** Nulls are not encoded in the data

# ORC – Optimized Row Columnar

- Binary columnar format
- Created for Hive to improve the performance of read/write/process
- The file format:
  - Stripes of rows composed of :
    - Index data – min/max and row positions
    - Column chunks
    - Stripe footer
  - File footer:
    - column level aggregates: min,max,count,sum
    - column data types
    - stripes metadata and locations
- Supported compressions: Snappy, Zlib
- Schema is kept in Hive

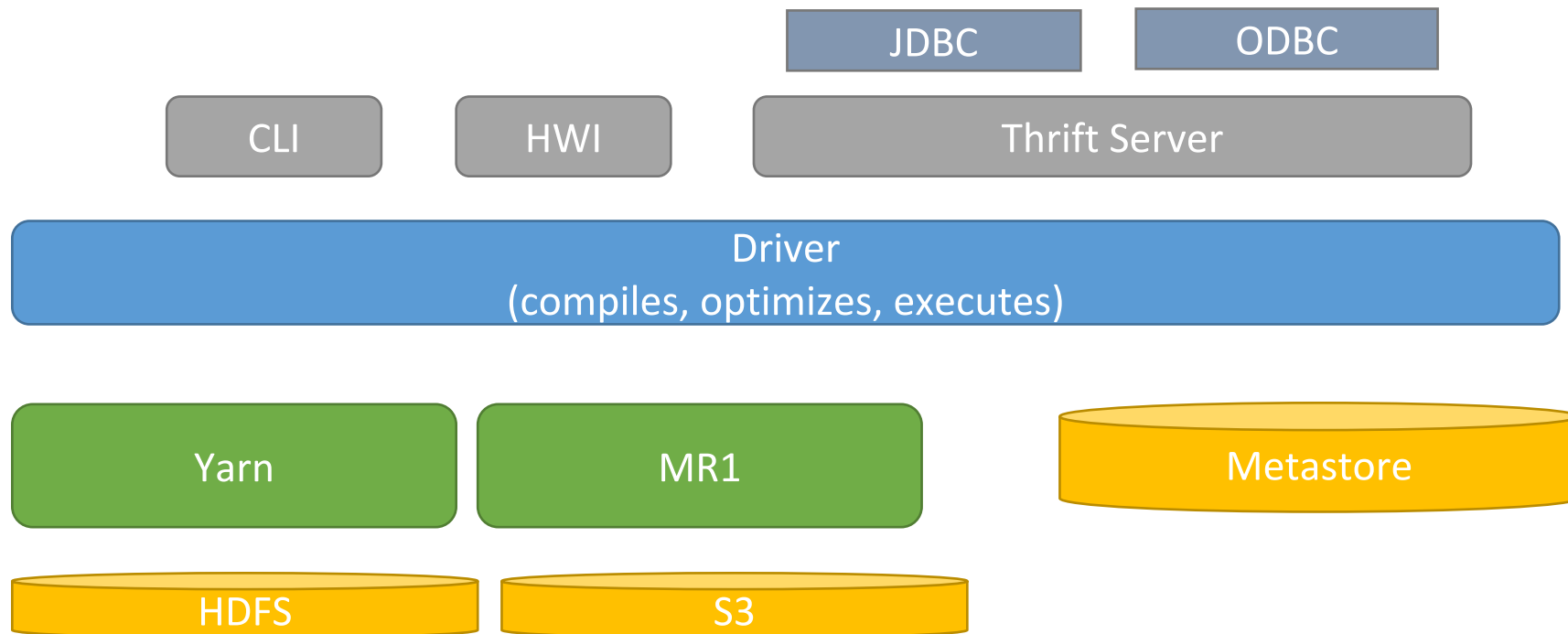


# File formats summary

- No single format to rule them all
- If you have to work with rows – go with Avro
- If you are doing data analytics choose Parquet
  - It has better Spark support (vectorized read)
- Use ORC if you are already in the Hive eco-system and want ACID transactions
- Use compression, snappy for performance, GZip for stale data

# Hive

- An SQL-like (HiveQL) query interface on top of Hadoop
- Started at Facebook and became an Apache project



# Processing a query in hive

1. The query is received through the **UI/Thrift/CLI** and passed to the **Driver**
2. The **Driver** passes the query to the **Compiler**
3. The **Compiler** asks the **Metastore** about the relations in the query and creates an execution plan
4. The plan is passed to the **Optimizer** – it does optimizations such as column pruning and predicate pushdown
5. The **Driver** passes the plan to the **Execution Engine**
6. The **Execution Engine** lunches a MR/Yarn/Spark job that works on data from S3 or HDFS
7. The **Execution Engine** returns the results to the driver which returns them to the caller

# The metastore

- Backed by RDBMS with ORM (DataNucleus)
- Has information about the tables
  - The schema of the tables
  - Physical information about partitions, file locations and serialization
- Abstraction for data discovery

# File formats in Hive

- Have the same characteristics as in Spark
- For row based processing use Avro
- For columnar – use ORC – has a vectorized read implemented in Hive
- Hive with ORC files also supports ACID transactions

# Spark SQL and Hive

- Spark SQL has a seamless integration with Hive
- It can access the Hive metastore to execute queries
- From Spark 1.4.0 single Spark build can communicate with different Hive metastore versions
- Requires *hive-site.xml*, *core-site.xml* and *hdfs-site.xml* placed in the conf/ directory



# Objects Serialization

- By default Spark uses Java Serialization (*ObjectOutputStream*) and those can work with any object that implements Java's *Serializable* object
- However, in order to provide better performance you can use Kyro:

```
spark.sparkContext().getConf().set("spark.serializer", KryoSerializer.class.getName());  
spark.sparkContext().getConf().set("spark.kryo.registrator", Revenue.class.getName());
```
- Kyro will work without registration as well but with storing the full class name with each object (and you don't want that)

# Don't forget!

- Tune memory and GC algorithms
- Tune the amount of executors and cores per executor