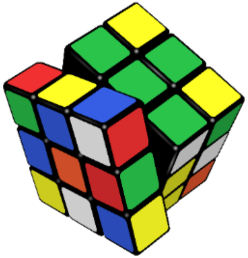
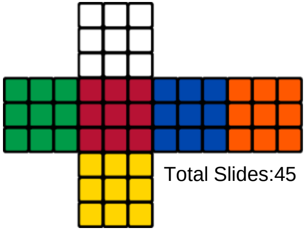


Big Data Engineering For Analytics

Data Sources and Formats

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

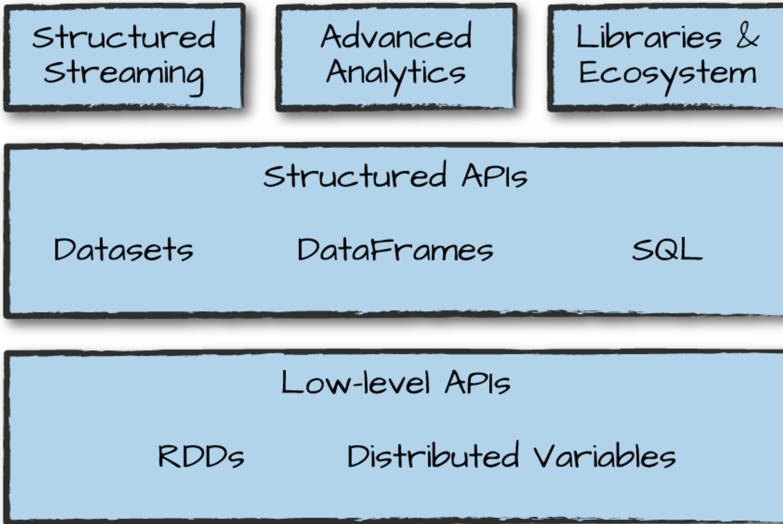


Total Slides:45

Agenda

- CSV, JSON
- Avro
- Parquet
- ORC
- Arrow

Review: Spark End User Libraries

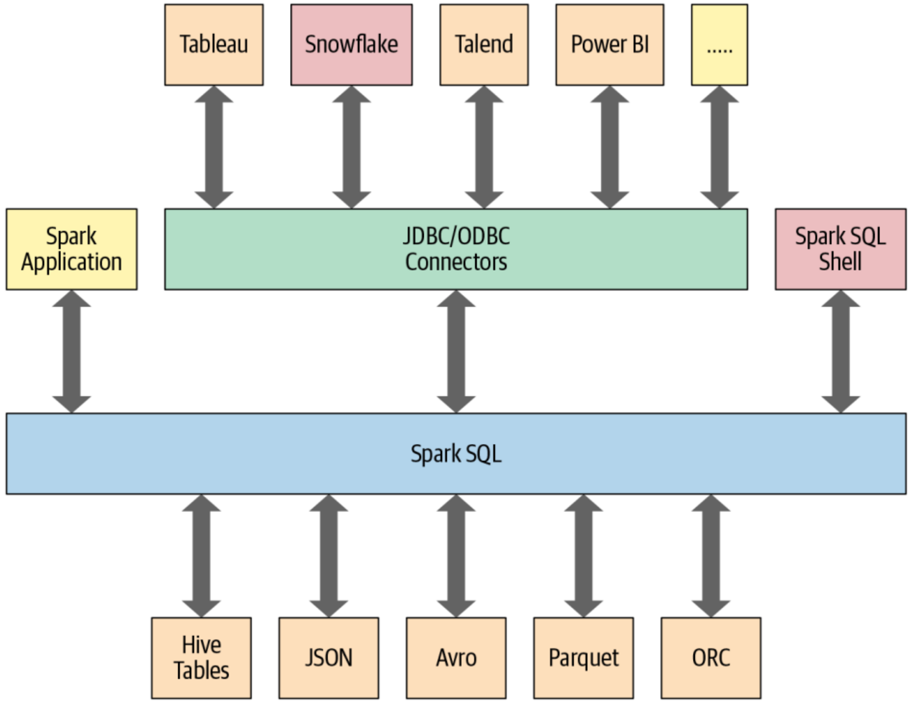


Spark SQL: work with structured and semi-structured data such as *Hive tables, RDBMS tables, Parquet files, AVRO files, JSON files, CSV files, and more*.

Guiding principles

- Row/Columnar based
- Read/Write
- Splittable (multiple tasks can run parallel on parts of file)
- Schema Evolution (Continuously evolving schema)
- Compression Support (Snappy, LZO etc.)

Spark SQL connectors and data sources



(json, parquet, jdbc, orc, libsvm, csv, avro, text)

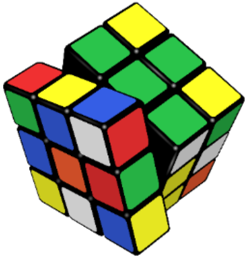
DataFrameReader methods, arguments, and options

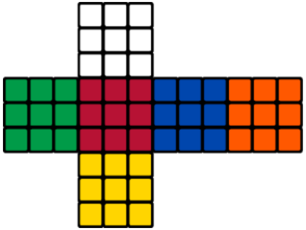
Big Data Engineering For Analytics

Method	Arguments	Description
format()	"parquet", "csv", "txt", "json", "jdbc", "orc", "avro", etc.	If you don't specify this method, then the default is Parquet or whatever is set in spark.sql.sources.default
option()	("mode", {PERMISSIVE | FAILFAST | DROPMALFORMED })("inferSchema", {true | false})("path", "path_file_data_source")	A series of key/value pairs and options. The default mode is PERMISSIVE. The "inferSchema" and "mode" options are specific to the JSON and CSV file formats.
schema()	DDL String or StructType, e.g., 'A INT, B STRING' orStructType(...)	For JSON or CSV format, you can specify to infer the schema in the option() method. Generally, providing a schema for any format makes loading faster and ensures data conforms to the expected schema.
load()	"/path/to/data/source"	The path to the data source. This can be empty if specified in option("path", "...").

DataFrameWriter methods, arguments, and options

Method	Arguments	Description
format()	"parquet", "csv", "txt", "json", "jdbc", "orc", "avro", etc.	If you don't specify this method, then the default is Parquet or whatever is set in spark.sql.sources.default
option()	("mode", {append | overwrite | ignore | error or errorifexists}) ("mode", {SaveMode.Overwrite | SaveMode.Append, SaveMode.Ignore, SaveMode.ErrorIfExists}) ("path", "path_to_write_to")	A series of key/value pairs and options. The default mode options are error or errorifexists and SaveMode.ErrorIfExists; they throw an exception at runtime if the data already exists.
bucketBy()	(numBuckets, col, col..., coln)	The number of buckets and names of columns to bucket by. Uses Hive's bucketing scheme on a filesystem.
save()	"/path/to/data/source"	The path to save to. This can be empty if specified in option("path", "...").
saveAsTable()	"table_name"	The table to save to.





CSV and JSON

Delimiter Separated Values

Pros : Human readable, all tools support it

Cons :

- IO/Storage inefficient
- No richer types - all are strings
- No support for schema evolution
- Other issues : delimiter in data, new lines within data

“Jorge”,30,”Developer”
“Bob”,32,”Developer”

Reading CSV with Spark


```
spark
     .read
     .option("header", "true")
     .option("delimiter", "\t")
     .csv(filePath)
```

```
spark
     .read
     .option("header", "true")
     .option("delimiter", "\t")
     .format("csv")
     .load(filePath)
     .show(false)
```

Python

```
df = (spark.read.format("csv")
      .option("inferSchema", "true")
      .option("header", "true")
      .load(csv_file))
df.createOrReplaceTempView("us_delay_flights_tbl")
```

Scala

```
val df = spark.read.format("csv")
 .option("inferSchema", "true")
 .option("header", "true")
 .load(csvFile)
// Create a temporary view
df.createOrReplaceTempView("us_delay_flights_tbl")
```

Writing CSV with Spark

```
frame
  .write
  .csv(filePath)
```

```
frame
  .write
  .format("csv")
  .save(filePath)
```

```
frame
  .write
  .option("header", "true")
  .option("delimiter", "\t")
  .format("csv")
  .save(filePath)
```

Python

```
frame.write.format("csv")
          .option("header", "true")
          .option("sep", ",")
          .mode("append")
          .save(csv_file)
```

Scala

```
frame.write.format("csv")
          .option("header", "true")
          .option("delimiter", ",")
          .mode("append")
          .save(csv_file)
```

SaveMode

Specifies the behavior of the save operation when data already exists

- *append*: Append contents of this DataFrame to existing data
- *overwrite*: Overwrite existing data.
- *ignore*: Silently ignore this operation if data already exists.
- *error or errorifexists (default case):* Throw an exception if data already exists.

Infer schema


```
df = spark.read.format('csv') \
                .option("inferSchema","true") \
                .option("header","true") \
                .option("sep",";") \
                .load("people.csv")
```

```
>>> df.show()
+-----+---+---------+
| name|age|      job|
+-----+---+---------+
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+---+---------+

>>> df.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = true)
 |-- job: string (nullable = true)
```

Specifying schema

Define your own schema(StructType)

```
schema = StructType([ \
    StructField("name",StringType(),True), \
    StructField("age",IntegerType(),True), \
    StructField("job",StringType(),True)])
```

Specify the schema during read

```
peopleDF = spark.read.format('csv') \
                    .option("schema","schema") \
                    .option("sep",";") \
                    .option("header","true") \
                    .load("people.csv")
```

```
>>> peopleDF.show()
+-----+---+---------+
| name|age|      job|
+-----+---+---------+
|Jorge| 30|Developer|
|  Bob| 32|Developer|
+-----+---+---------+

>>> peopleDF.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: string (nullable = true)
 |-- job: string (nullable = true)
```

JSON

Pros : Readable, some level of schema support

Cons :

- Duplicated schema
- Horrible in terms of storage
- Not splittable, linear lookups
- Aggregations require all data to be loaded into memory

```
{"name":"Michael"}
{"name":"Andy", "age":30, "job": "developer"}
{"name":"Justin", "age":19}
```

Multiline vs Record per line

Big Data Engineering For Analytics


```
],
"truncated": true,
"user": {
  "id_str": "944480690",
  "screen_name": "FloodSocial"
},
"extended_tweet": {
  "full_text": "JustanotherExtendedTweetwithmorethan140characters,generatedasadocu
    ,showingthat[\"truncated\"
    :true]andthepresenceofan\"extended_tweet\"objectwithcompletetextand\"entities\
    arsingJSON#GeoTaggedhttps://t.co/e9yhQTJSIA",
  "display_text_range": [
    0,
    249
  ],
  "entities": {
    "hashtags": [
      {
        "text": "documentation",
        "indices": [
          211,
          225
        ]
      },
      {
        "text": "parsingJSON",
        "indices": [
          226,
          238
        ]
      },
      {
        "text": "GeoTagged",
        "indices": [
          239,
          249
        ]
```

```
{"id":11348282,"id_str":"11348282","name":"NASA","screen_name":"NASA","location":"
{"id":11348282,"id_str":"11348282","name":"NASA","screen_name":"NASA","location":"
{"id":11348282,"id_str":"11348282","name":"NASA","screen_name":"NASA","location":"
```

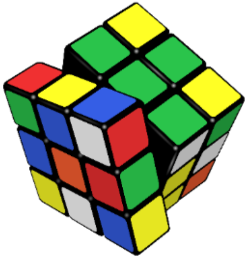
Reading and Writing JSON with Spark

```
val df = spark
      .read
      .option("multiline", "true")
      .option("mode", "PERMISSIVE")
      .json(filePath)
```

```
val df = spark
      .read
      .json(filePath)
```

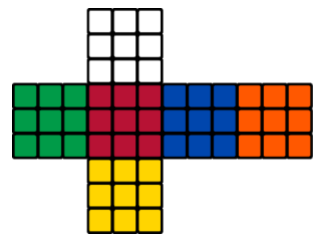
```
frame
      .write
      .format("json")
      .mode(SaveMode.Append)
      .save(filePath)
```

```
frame
      .write
      .mode(SaveMode.Overwrite)
      .json(filePath)
```



Big Data Engineering For Analytics

AVRO



Avro is row oriented

target	id	date	user	text
1	123	Saturday 8th, June	alpha1	Simple tweet1
2	234	Sunday 9th, June	alpha2	Simple tweet2

1	123	Saturday 8th, June	alpha1	Simple tweet1	2	234	Sunday 9th, June	alpha2	Simple tweet2

Avro schema definition

Avro schemas are defined using JSON:

```
{
      "namespace": "com.tweet.avro",
      "type": "record",
      "name": "TweetAvro",
      "fields":[
          {  "name": "target", "type": "int"},
          {  "name": "id", "type": "long"},
          {  "name": "date",  "type": "string"},
          {  "name": "user",  "type": "string"},
          {  "name": "text",  "type": "string"}
      ]
}
```

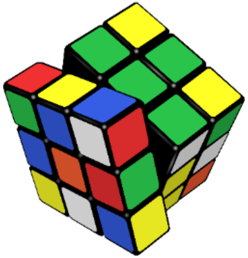
Read and write Avro from Spark

```
spark
     .read
     .format("avro")
     .load(filePath)
```

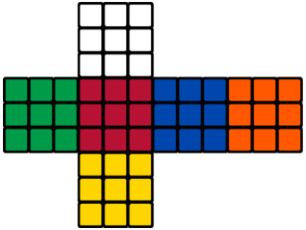
```
frame
      .write
      .format("avro")
      .save(filePath)
```

Avro

- Pros
  - ➢Splittable
  - ➢Support schema evolution
  - ➢Addition, Deletion of fields in beginning, middle, end doesn't matter
  - ➢It is a good format for data exchange. It has a data storage which is very compact, fast and efficient for analytics.
  - ➢Avro is a data serialization system.
- Cons
  - ➢ Data is not readable
  - ➢ Not integrated to every language



Parquet



Parquet is column oriented

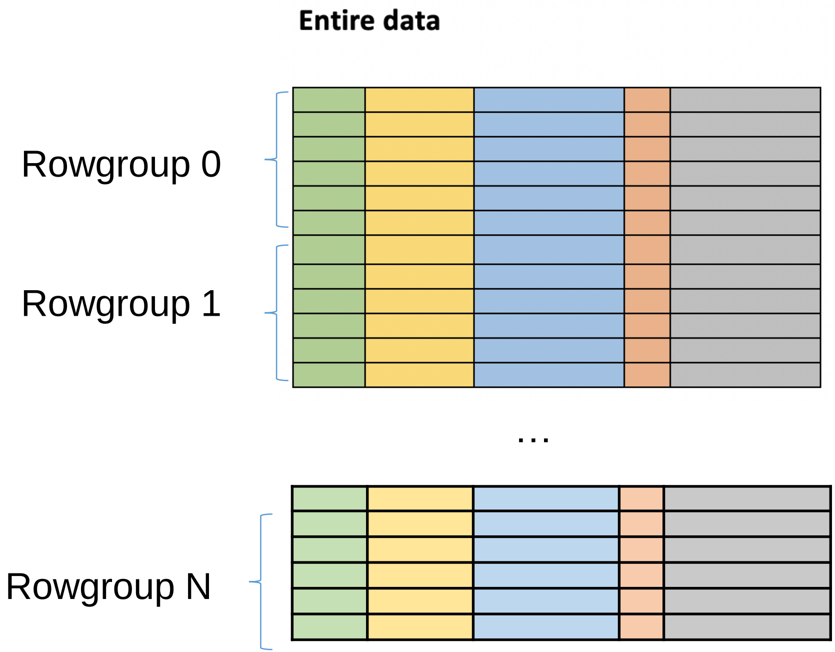
target	id	date	user	text
1	123	Saturday 8th, June	alpha1	Simple tweet1
2	234	Sunday 9th, June	alpha2	Simple tweet2

1	2	123	234	Saturday 8th, June	Sunday 9th, June	alpha1	alpha2	Parquet tweet1	Parquet tweet2

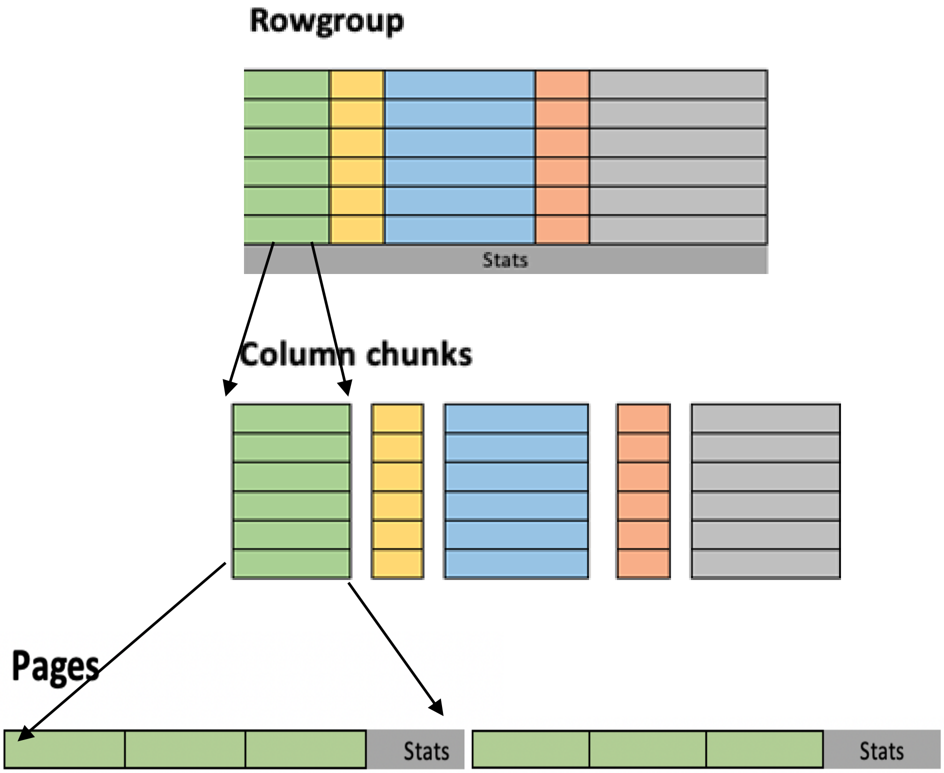
"Tables have turned"

Rowgroups

Big Data
Engineering
For
Analytics

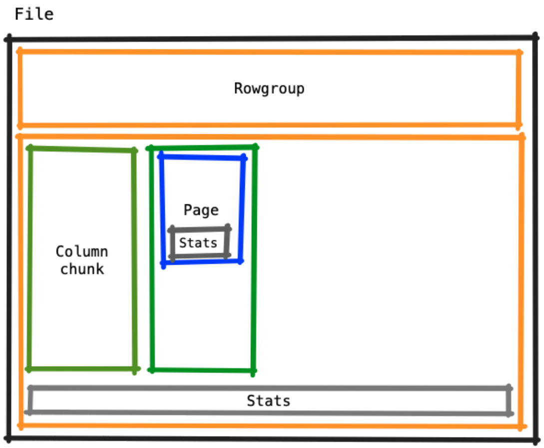

Entire data
Rowgroup 0
Rowgroup 1
...
Rowgroup N

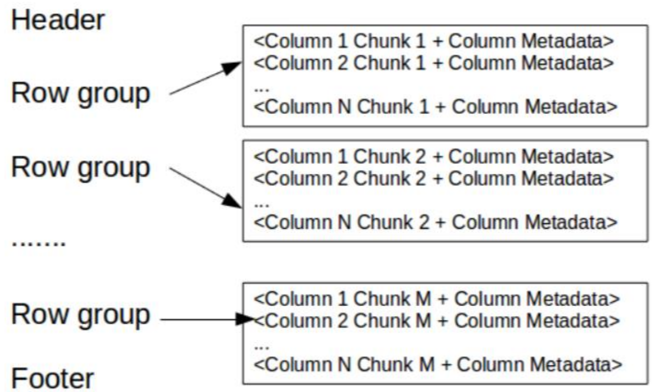
Rowgroups, Column chunks and Pages



Big Data Engineering For Analytics

Parquet format





Read and Write Parquet in Spark

Big Data Engineering For Analytics

```
spark.read
          .format("parquet")
          .load(filepath)
          .options(other options)
```

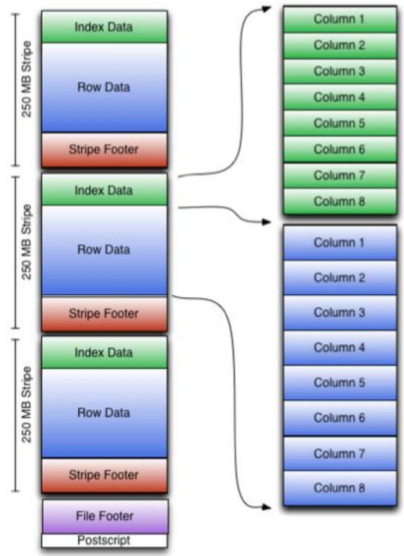
```
frame.write
          .format("parquet")
          .mode(Save mode)
          .save(filepath)
          .options(other options)
```

Parquet

- Pros
  - ➢Supports primitive, complex and logical types
  - ➢Splittable
  - ➢Supports richer schema evolution
  - ➢Highly compressed
  - ➢Great for analytical workloads
- Cons
  - ➢Writes are slower - Typical of Columnar data formats
  - ➢Not human readable

ORC

- The Optimized Row Columnar (ORC) file format provides a highly efficient way to store Hive data. It was designed to overcome limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data.
- Similar to Parquet
  - ➢Rowgroup is called a Stripe
  - ➢Supports similar encoding and compression
  - ➢Statistics stored at the same level
    - Has file level index
- Pros
  - ➢Highly compression
- Cons
  - ➢Not support schema evolution



Read and write ORC from Spark

Big Data Engineering For Analytics

Read a folder that has ORC files

```
frame = spark
            .read
            .format("orc")
            .load(filePath)
```

Write into a folder named “filePath”

```
frame
      .write
      .format("orc")
      .save(filePath)
```

ORC with snappy compressed

```
frame
      .write
      .option("compression", "snappy")
      .format("orc")
      .save(filePath)
```

Compression codecs

Name	CPU usage	Compression	Speed	Splittable
Gzip	High	Good	Slow	No
BZip2	High	Good	Slow	Yes
Snappy	Normal	Normal	Fast	Yes
LZO	Normal	Normal	Fast	Yes

- *Gzip* - A compression utility that was adopted by the GNU project. It's file have an extension of *.gz*. You can use gunzip command to decompress the files.

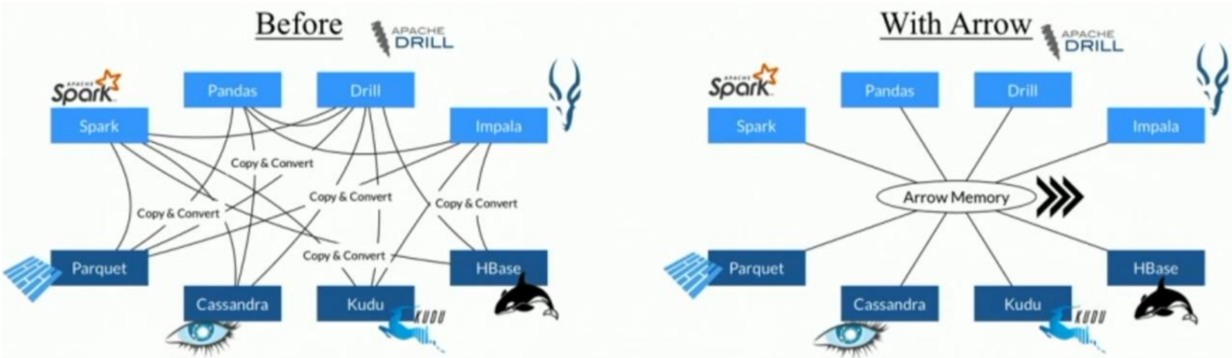
- *Bzip2* - from the usability standpoint Bzip2 and Gzip are similar. But Bzip2 has much more degree of compression then the Gzip but it is also slower . You can use Bzip2 codec space priority is higher and the data will be rarely needed to be queried.

- *Snappy* - Snbappy is the codec by Google , It provides fastest compression and decompression among all the codec but comes with a modest degree of compression.

  LZO - Similar to Snappy LZO gives fast compression and decompression with modest compression degree. LZO is licensed under GNU Public License (GPL).

Apache Arrow

Apache Arrow



Overview Apache Arrow [Julien Le Dem, Spark Summit 2017]

Before arrow:

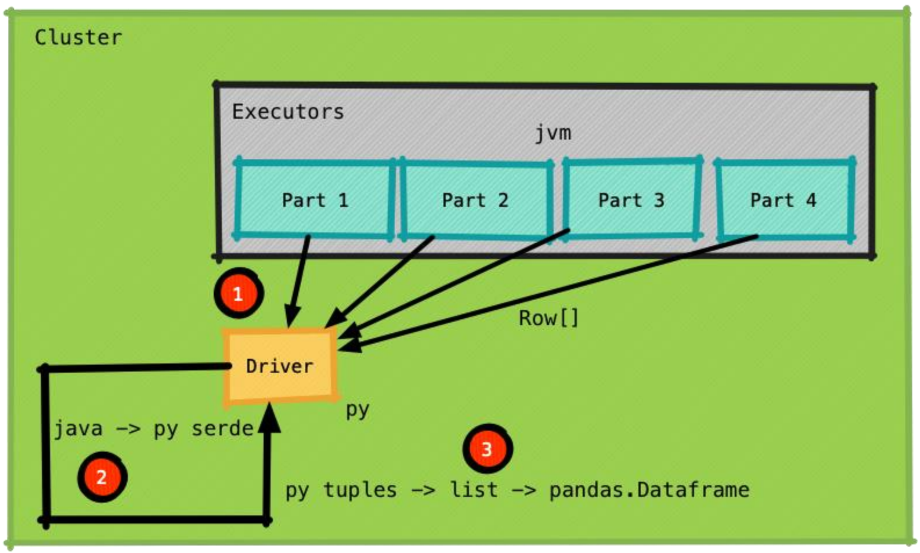
- Each system has its own internal memory format
- 70-80% CPU wasted on serialization and deserialization
- Functionality duplication and unnecessary conversions

After Arrow:

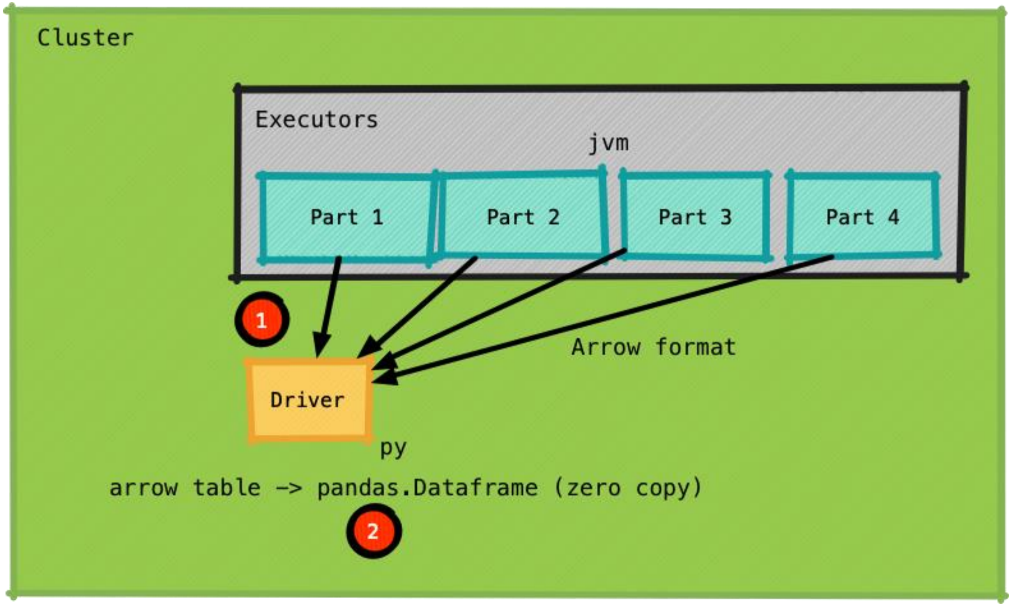
- All systems utilize the same memory format
- No overhead for cross-system communication
- Project can share functionality (e.g. parquet to arrow reader)

Spark to Pandas

```
pandasDf = sparkDf.toPandas()
```

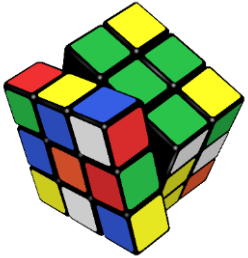


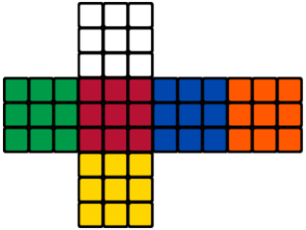
Spark to Pandas with Arrow



Arrow

- Columnar memory format for flat and hierarchical data
- No copy to any ecosystem like Java/R language
- Provide a universal data access layer to all applications
- Low loading while streaming messaging
- It supports flat and nested schemas
- Support GPU





Summary

Summary

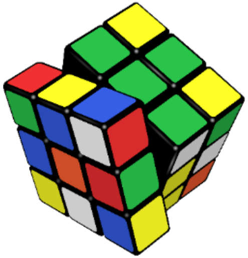
- CSV - human readable (not for large datasets)
- Semi structured JSON, XML - Not splittable
- Avro - If you are reading all or most of the columns
  - ➢Select * from the table which needs to scan the entire table
  - ➢Rich schema evolution
- Parquet/ORC
  - ➢Highly compressed
  - ➢select x, y, groupby which only needs to perform implementation on a certain columns
  - ➢Nested data
- Arrow - BE VERY EXCITED !

AVRO vs PARQUET

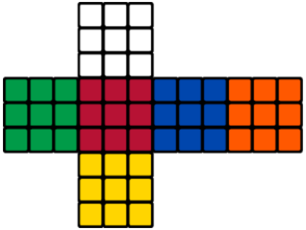
- AVRO is a row-based storage format whereas PARQUET is a columnar based storage format.
- PARQUET is much better for analytical querying i.e. reads and querying are much more efficient than writing.
- Write operations in AVRO are better than in PARQUET.
- AVRO is much matured than PARQUET when it comes to schema evolution. PARQUET only supports schema append whereas AVRO supports a much-featured schema evolution i.e. adding or modifying columns.
- PARQUET is ideal for querying a subset of columns in a multi-column table. AVRO is ideal in case of ETL operations where we need to query all the columns.

ORC vs PARQUET

- PARQUET is more capable of storing nested data.
- ORC supports ACID properties.
- ORC is more compression efficient.



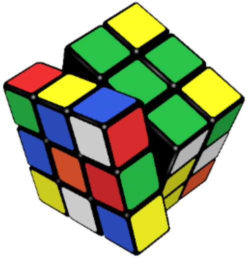
Big Data Engineering For Analytics



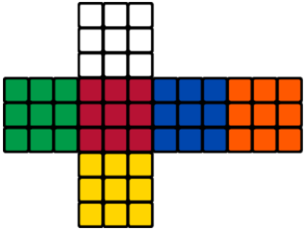
References

References

- [Hadoop: The Definitive Guide] - Tom White
- [Spark: The Definitive Guide] - Bill Chambers and Matei Zaharia
- [Dremel - Google], Sergey Melnik et al
- [Designing Data-Intensive Applications] - Martin Kleppmann
- [Arrow project]



Big Data Engineering For Analytics



Appendix

Data Format Comparisons

Types	CSV	JSON	XML	AVRO	Protocol Buffers	Parquet	ORC
text versus binary	text	text	text	metadata in JSON, data in binary	text	binary	binary
Data type	no	yes	no	yes	yes	yes	yes
Schema enforcement	no (minimal with header)	external for validation	external for validation	yes	yes	yes	yes
Schema evolution	non	yes	yes	yes	non	yes	non
Storage type	row	row	row	row	row	column	column
OLAP/OLTP	OLTP	OLTP	OLTP	OLTP	OLTP	OLAP	OLAP
Splittable	yes in its simpliest form	yes with JSON lines	non	yes	non	yes	yes
Compression	yes	yes	yes	yes	yes	yes	yes
Batch	yes	yes	yes	yes	yes	yes	yes
Stream	yes	yes	non	yes	yes	non	non
Typed data	non	non	non	non	yes	non	non
Ecosystems	popular everywhere for its simplicity	API and web	enterprise	Big Data and Streaming	RPC and Kubernetes	Big Data and BI	Big Data and BI