Encoders: **Encoders** are what convert your data between JVM objects and **Spark** SQL's specialized internal (tabular) representation. They're required by all Datasets! **Encoders** are highly specialized and optimized code generators that generate custom bytecode for serialization and deserialization of your data.

# How Encoders better then Kryo?

Kryo serializer leads to Spark storing every row in the dataset as a ***flat binary object.*** Instead of using Java or Kryo serializer, you can use Spark's internal encoders. You can use it automatically via spark.implicits.\_. It also uses less memory than Kryo/Java serialization.

* They contain schema information, which makes these highly optimized code generators possible, and enables optimization based on the shape of the data. Since Spark understands the structure of data in Datasets, it can create a more optimal layout in memory when caching Datasets.
* **>10x faster** than Kryo serialization (Java serialization orders of magnitude slower).

# What are case classes?

Simple immutable pojos in java which implements Serializable interface. The **case class** defines the schema of the table. These needs to be defined when using encoders for custom java objects.

Example : Employee {name,salary} a custom class

**public** **final** **class** Employee **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** String name;

**private** **long** salary;

**public** String getName() {

**return** name;

}

**public** **void** setName(**final** String name) {

**this**.name = name;

}

**public** **long** getSalary() {

**return** salary;

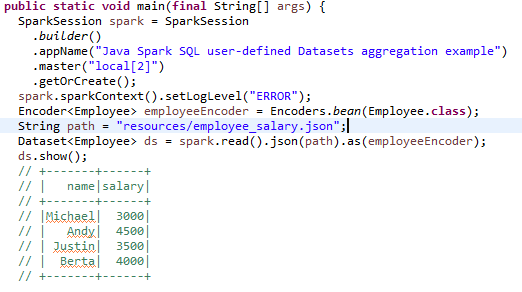
}

**public** **void** setSalary(**final** **long** salary) {

**this**.salary = salary;

}

}



Reference:

<https://spark.apache.org/docs/latest/sql-getting-started.html>

<https://towardsdatascience.com/apache-spark-dataset-encoders-demystified-4a3026900d63>

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

# What is Data Serialization?

<https://www.adaltas.com/en/2020/07/23/benchmark-study-of-different-file-format/>

<https://www.bizety.com/2019/04/02/data-serialization-protocol-buffers-vs-thrift-vs-avro/>

where to use parquet and where Avro? And which compression to use Sanppy/Gzip?

<https://medium.com/@minyodev/avro-vs-parquet-what-to-use-518ccbe8fb0c>

The fundamental difference in terms of how to use either format is this: Avro is a Row based format. If you want to retrieve the data as a whole, you can use Avro. Parquet is a Column based format. If your data consists of lot of columns but you are interested in a subset of columns, you can use Parquet.

**GZIP** compression uses more CPU resources than **Snappy** or LZO, but provides a higher compression ratio. ... **Snappy**  is a better choice for hot data, which is accessed frequently.

Google created Snappy because they needed something that offered very fast compression at the expense of the final size. For example, running a basic test with a 5.6 MB CSV file called foo.csv results in a 2.4 MB Snappy filefoo.csv.sz. Using the same file foo.csv with GZIP results in a final file size of 1.5 MB foo.csv.gz. However, Snappy used 30% CPU while GZIP used 58%.

***while writing data in sparksql you can set OPTION("compression", "snappy")***

***or sqlContext.setConf("spark.sql.parquet.compression.codec.", "snappy")***

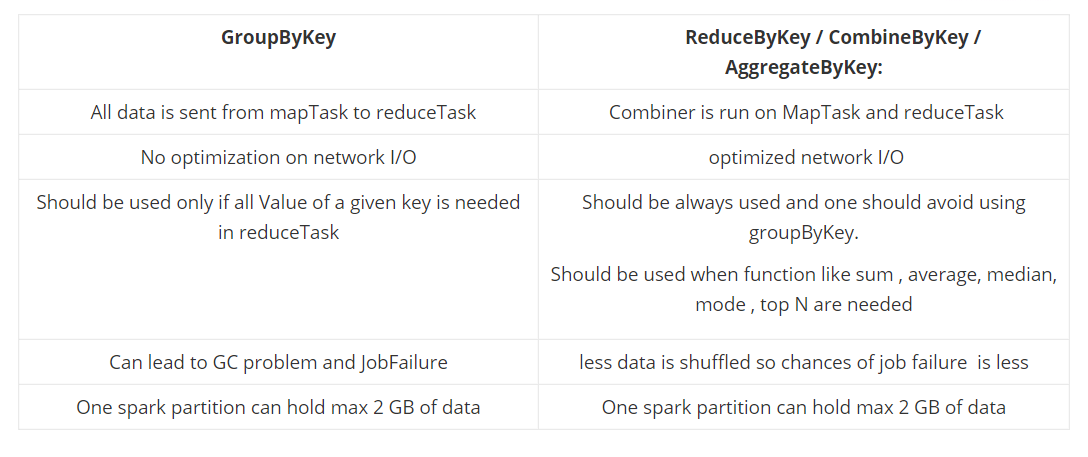
# SparkSQL vs Hive - which one is better?

1. [Hive is](https://www.educba.com/what-is-a-hive/) known to make use of HQL (Hive Query Language) whereas [Spark SQL](https://www.educba.com/what-is-apache-spark/) is known to make use of Structured Query language for processing and querying of data
2. Hive provides schema flexibility, portioning and bucketing the tables whereas Spark SQL performs SQL querying it is only possible to read data from
3. Hive provides access rights for users, roles as well as groups whereas no facility to provide access rights to a user is provided by Spark SQL
4. Hive provides the facility of selective replication factor for redundant storage of data whereas spark SQL, on the other hand, does not provide any replication factor for storing data

|  |  |
| --- | --- |
| **reduceByKey** | **CombineByKey** |
| reduceByKey internally calls ***[combineByKey](https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/PairRDDFunctions.scala" \l "L308" \t "_blank)*** | CombineByKey is the generic api  and is used by reduceByKey and aggregateByKey |
| the input type and outputType of reduceByKey are the **same** | CombineByKey is more flexible, hence one can mention the required outputType .  The output type is not necessarily required to be the same as that of the input type. |

<http://bytepadding.com/big-data/spark/reducebykey-vs-combinebykey/#:~:text=The%20only%20difference%20between%20reduceByKey,they%20function%20exactly%20the%20same%20.&text=CombineByKey%20is%20more%20flexible%2C%20hence,that%20of%20the%20input%20type>.

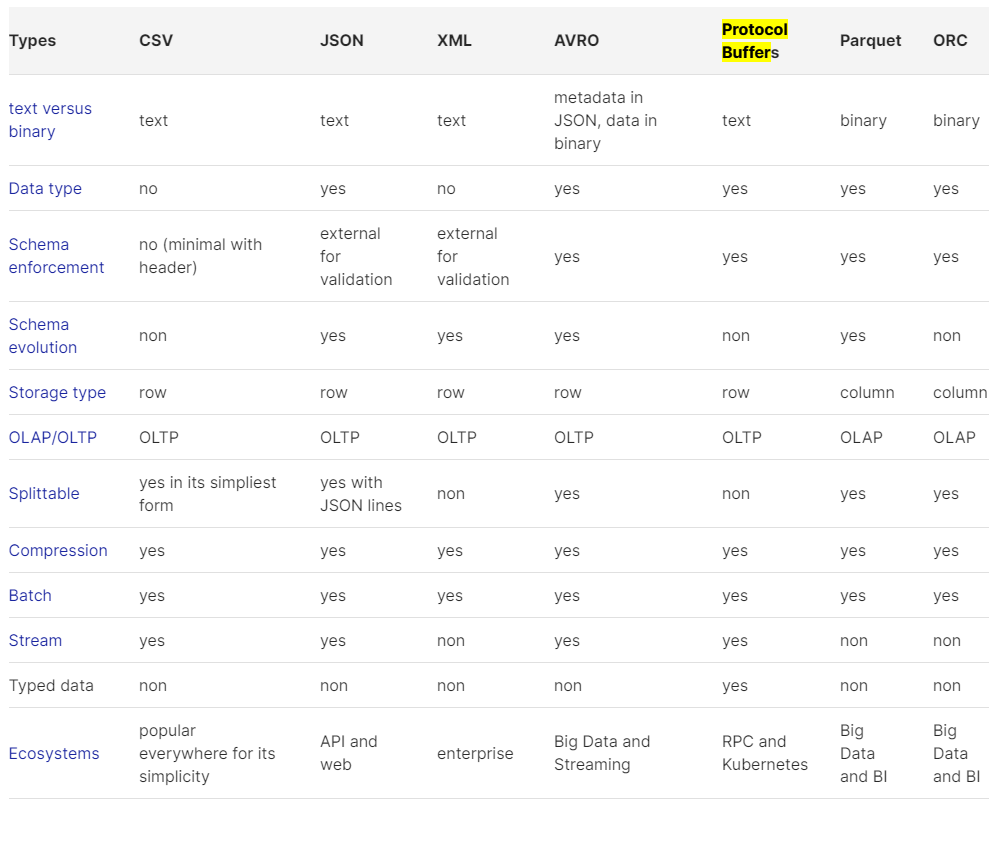
<http://bytepadding.com/big-data/spark/groupby-vs-reducebykey/>



Starting from Apache Spark 3.0, you have a new parameter “mode” that produce expected format for the plan:

* **explain(mode=”simple”)**which will display the physical plan
* **explain(mode=”extended”)** which will display physical and logical plans (like “extended” option)
* **explain(mode=”codegen”)**which will display the java code planned to be executed
* **explain(mode=”cost”)** which will display the optimized logical plan and related statistics (if they exist)
* **explain(mode=”formatted”)** which will display a splitted output composed by a nice physical plan outline, and a section with each node details

<https://medium.com/datalex/sparks-logical-and-physical-plans-when-why-how-and-beyond-8cd1947b605a#:~:text=explain(mode%3D%E2%80%9Dextended%E2%80%9D,code%20planned%20to%20be%20executed>



KAFKA :- <https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch04.html>

<https://www.youtube.com/watch?v=KOu6DVdaY24>

QUERY PLAN :- <https://www.youtube.com/watch?v=UZt_tqx4sII>