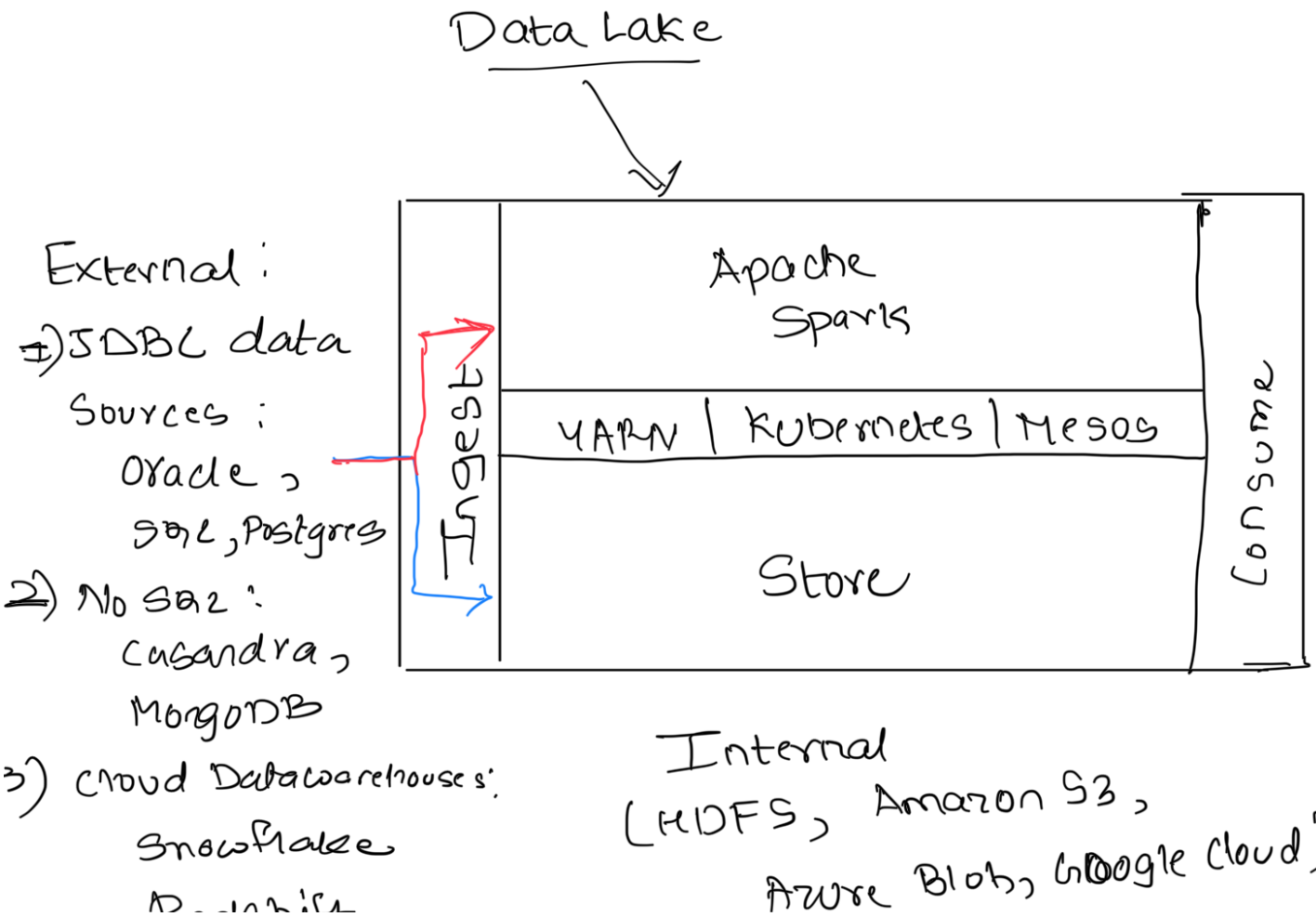
* Spark Data Sources And Sinks

- Spark is used for processing large volumes of data. However, any processing engine, including Spark must-read data from some data source.

- These data sources can be further categorized into two groups.
  1] External Data Sources
  2] Internal Data Sources



Realtime

4) Stream Integration
   Kafka, Kinesis

External Datasources:
— Your data might be stored at some application servers, Oracle, SQL Server etc.
All these systems are external to your Data Lake. You don't see them in your Data Lake conceptual diagram.
So, we categorize such data sources as external data sources.
— How can we read data from external sources?

Ans → There are two approaches:
The First approach is to bring your data to Data Lake and store them in your Lake's Distributed Storage. The most commonly used way is a suitable data integration tool. (BLUE)
The second approach is to use Spark DataFrame API to directly connect with these external systems. (Red)

Internal Data Sources:

- So your internal data sources is your distributed storage. It could be HDFS or cloud-based storage.
- Your data is stored in these systems as a data file. The mechanism of reading data from HDFS or cloud storage is same. However the difference lies in the data format.

  Example: CSV, JSON, Parquet, AVRO, Plain Text.

⇒ What is Data Sink?

- The data sinks are the final destination of the processed data.

  So you are going to load the data from an internal or external source.

  Then you will be handling it using the Spark APIs.

  Once your processing is complete, you want to save the outcome to an internal or external system.

  And these systems could be a datafile in your datalake storage, or it could ... JDBC,

be external system such as a
database or a NoSQL database

So, in general
- working with data sources is all about reading the data
- working with data sink is all about writing the data

* Spark DataFrameReader API

General Structure: DataFrameReader
.format(...)
.option("key", "value")
.schema(...)
.load()

Example :-
```
spark.read \
    .format("csv")
    .option("header", "true")
    .option("path", "/data/csv file")
    .option("mode", "FAILFAST")
    .schema(mySchema)
    .load()
```

The mode option is used for handling a corrupt a malformed record. Read modes Specify what will happen when spark comes across a malformed record.

Spark allows three read modes:

1) Permissive: This is default option. This mode sets all the fields to null when encounters a corrupt record.

2) DropMalformed: This is going to remove malformed record.

3) FailFast: This will raise an exception and terminates immediately upon encountering a malformed record.

* Spark DataFrameWriter API:

General Structure: DataFrameWriter

- format(..)
- option(...)
- partitionBy(..)
- bucketBy(..)
- sortBy(..)

• save()

Example :- DataFrame.write
• format("parquet")
• mode(SaveMode)
• option("path", "data/flights")
• save()

- Save modes specifies what will happen when Spark existing finds data at the specified location.
We have four valid modes:

1) append :- The append mode will create new files without touching the existing data at the given location.

2) overwrite :- The overwrite will remove the existing data and create new files

3) errorIfExists :- This will throw an error when you have already some data at specified location.

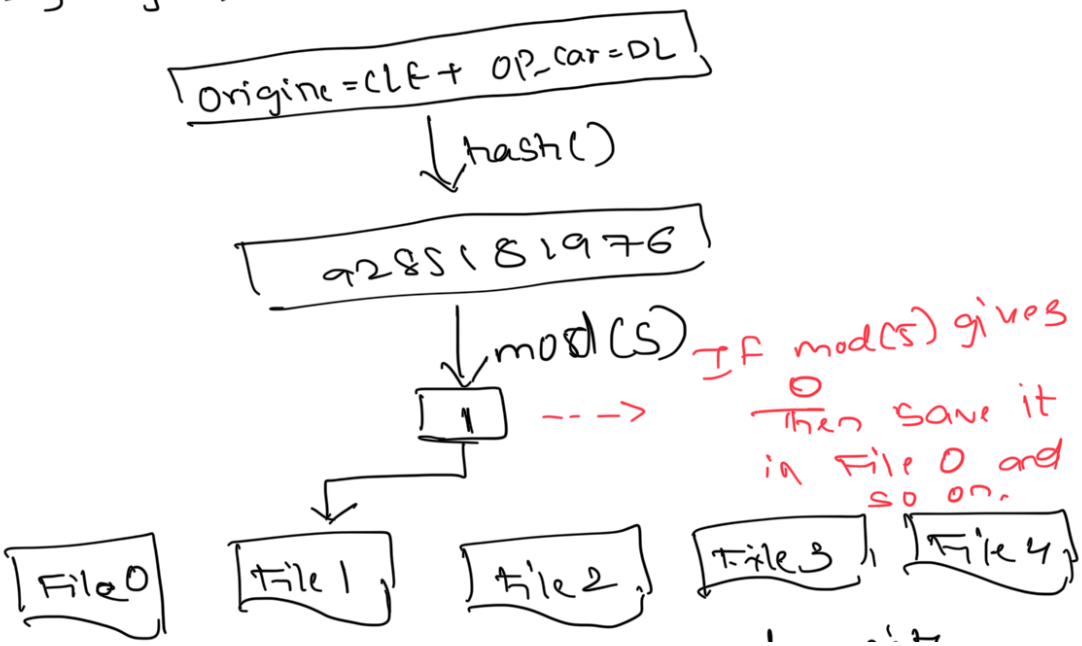
4) ignore :- The ignore will write the data if and only if the target is empty and do

directory is empty
nothing if some files already exist at the location.

- partitionBy() method will repartition your data based on a key column. You can use a single column key such as country code or you can use composite column key such as country code plus state code.
  key-based partitioning is a powerful method to break your data logically.

- bucketBy() method is used to partition your data into a fixed numbers of predefined buckets.

bucketBy(5, Origin, OP_carrier)



- sortBy() method is commonly used with bucketBy(). This option allows you to create sorted buckets.
- maxRecordsPerFile() method allows you to limit the number of records per file.

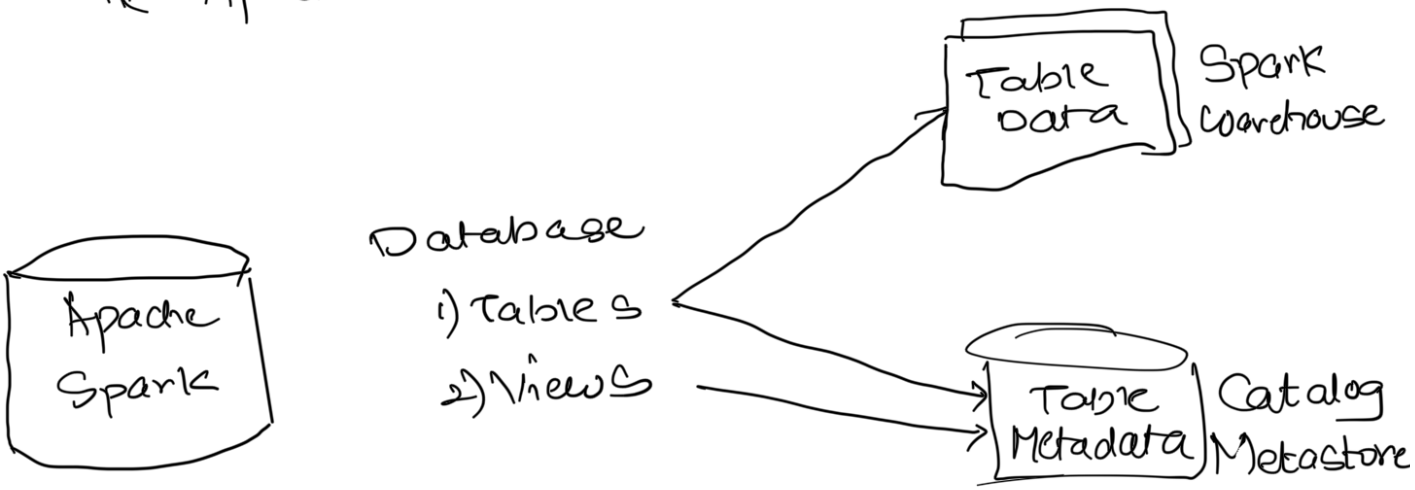
* Spark Databases and Tables

- Apache Spark is not only a set of APIs and a processing engine. It is a database itself. So you can create a database in Spark. Once you have a database, you can create database tables and views.
- These tables and views reside inside your database. The table has got two parts - Table Data - Stored in distributed storage
  - Table Metadata - The metadata is stored in a meta-store called catalog. The meta-store holds the information about the table and its data, such as

Schema, table name, database, column names, etc.

By default Spark comes with an in-memory catalog which is maintained per spark session. However, this information goes away when the session ends.

So we need a persistent and durable meta-store. Spark decided to reuse the Apache Hive Meta-store



⇒ Spark allows you to create two types of tables:

1) Managed Tables:
For these types of tables, Spark manages both: the metadata and the data.

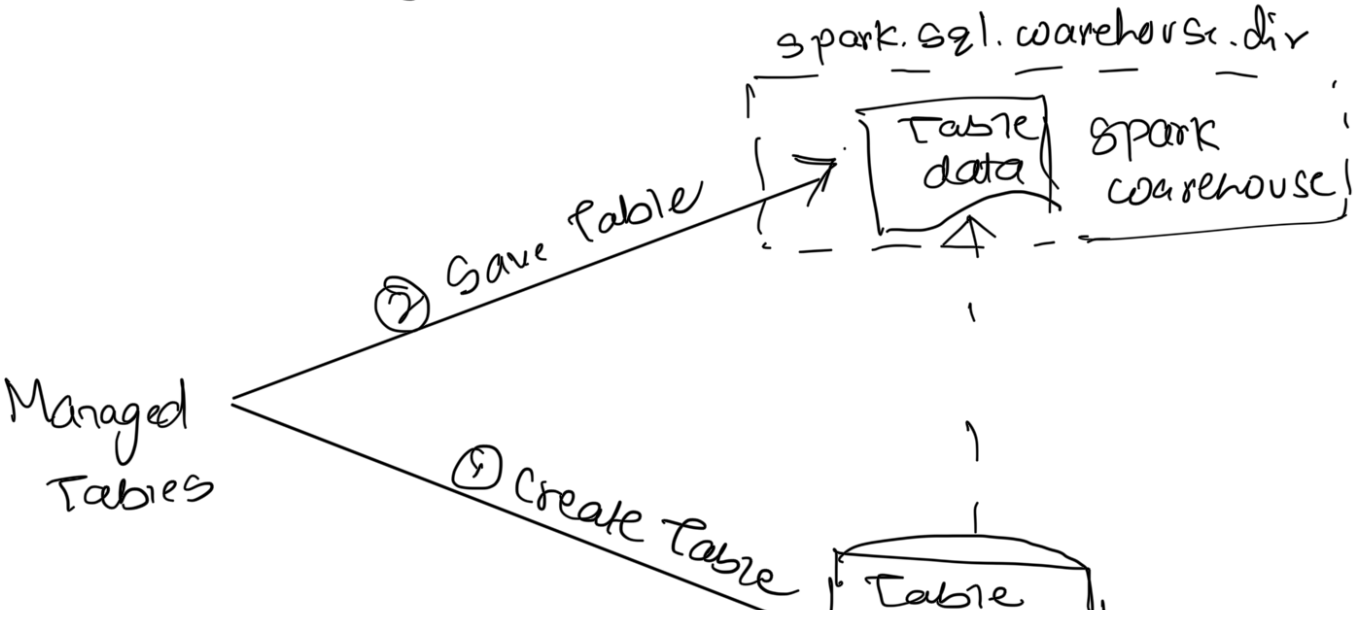
What does it mean?

It means, if you create a managed table, then the Spark is going to do two things.

- create and store meta-data about the table in meta-store
- Then spark is going to write the data inside a predefined directory location.

The directory is known as the spark.sql.warehouse.dir.

This directory is the base location where all your managed tables are stored. And your cluster admin is going to set this base directory for you.



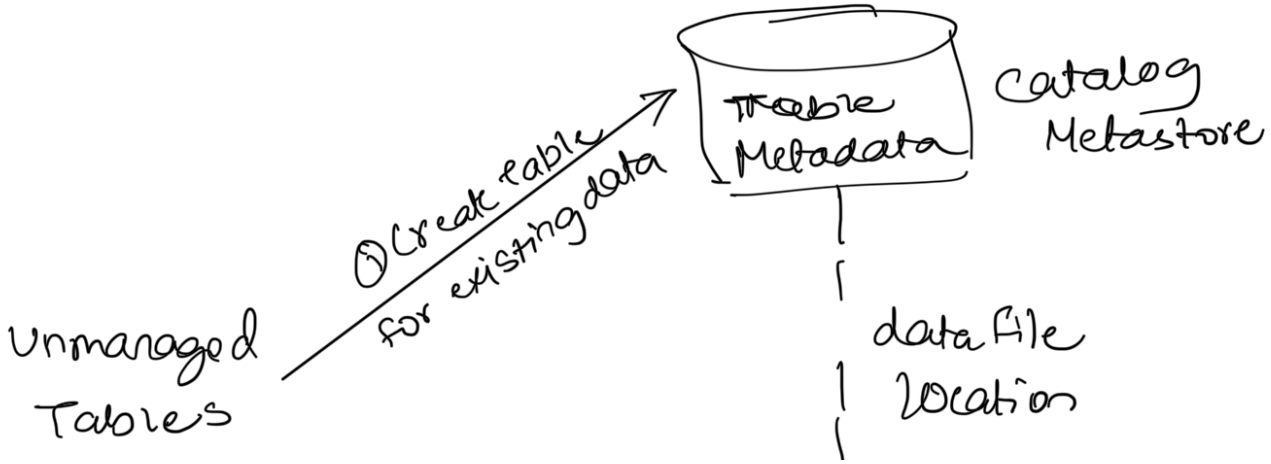
Metadata

How to create Managed Table?

```
dataframe.write
    .saveAsTable("your_table_name")
```

2] Unmanaged Tables (External Tables)

- They are also same w.r.t metadata. But they are different in terms of data storage location.
- Spark is going to do only one thing, it will create metadata for your table and store it in the meta-store.
- However, when you create an unmanaged table, then you must specify the data directory location of your table.



How to create Unmanaged Table?

Create Table your_tbl_name (col1 data type,
col2 data types
--)

Using PARQUET
Location "data_file_location"

- You do not have location facility with unmanaged table.
Because the managed table must be stored inside the warehouse directory.
However, unmanaged tables are to give you flexibility to store your data at your preferred location.

- Example:-
Suppose you already have some data and it is stored at some directory location.
Now you want to use spark SQL

And you [illegible]
on this dataset, such as Select count(*),
However, Spark SQL engine doesn't
known anything about this data.
So, you can create an unmanaged table
and map the same data to a Spark
Table.
Now, Spark will create metadata
and store it. This will allow you
to run your Spark SQL on this data.

- Drop method:
  - If you drop your managed table
    then Spark is going to delete your metadata
    and the data as well.
  - If you drop unmanaged table
    the Spark is going to remove the
    metadata and do not touch data
    files.
    Because unmanaged tables are
    designed for temporarily mapping your
    existing data and using it in Spark
    SQL.