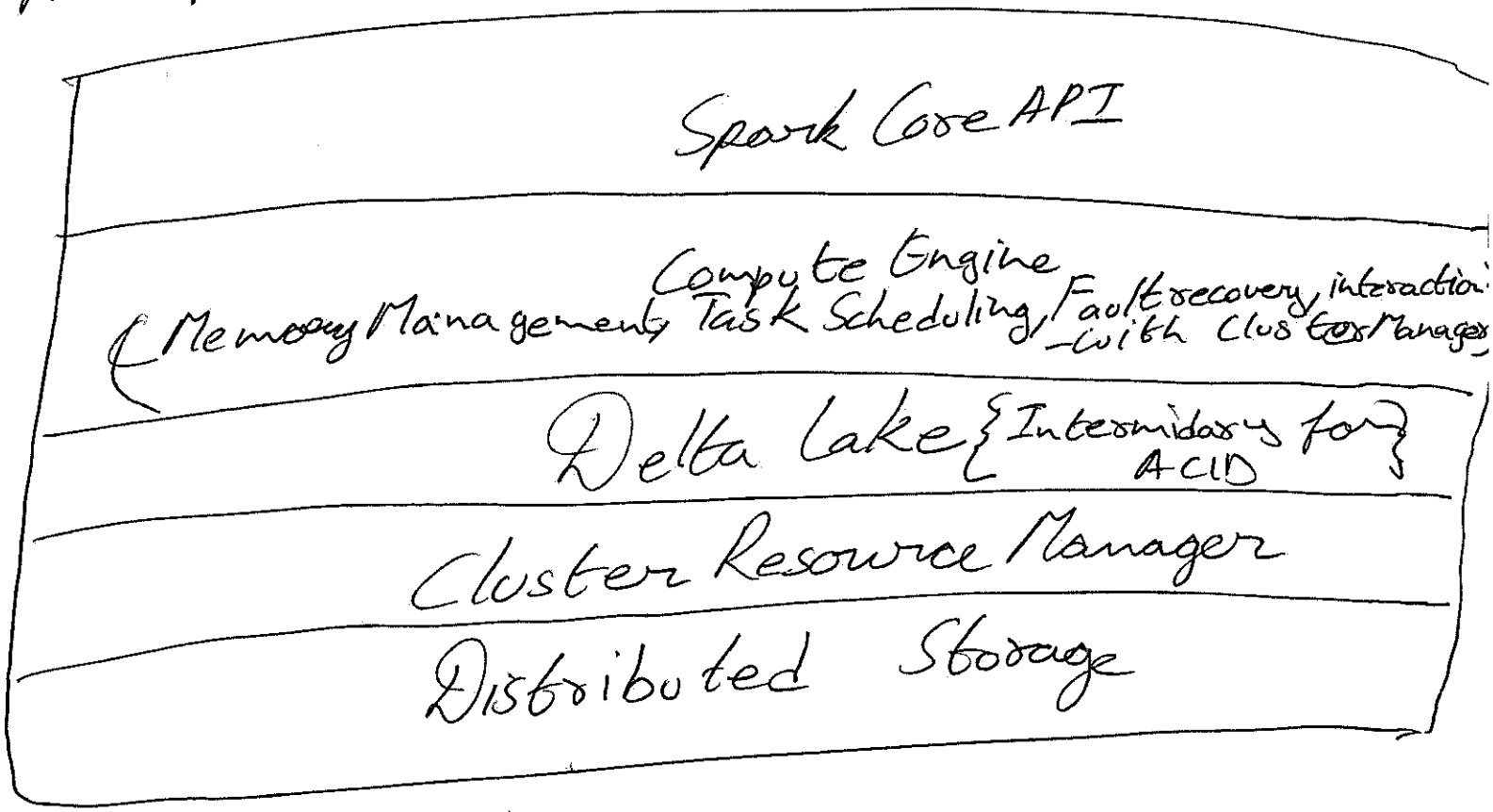
Delta lake for Apache Spark

(How does it work?)

What is Delta Lake :-

Delta Lake provides an intermidiate layer ~~for~~ in between Spark & Storage Layer to provide ACID compliance. As, storage layer in Hadoop is typically ~~the~~ the non ACID part.



Start Delta lake :-

```
$spark2-shell --master yarn \
--jars="delta/delta-core_2.11-0.3.0.jar"
```

(or)

```
$ spark2-shell --master yarn \
--packages io.delta:delta-core_2.11:0.3.0
```

Code changes for Delta:-

Initial:-

```
> spark.range(100).repartition(2).write.
    .mode("overwrite").csv("/tmp/test-1/")
```

```
> scala.util.Try(spark.range(100).
    repartition(2).map{ i =>
      if(i>50){
        Thread.sleep(50000)
        throw new RuntimeException("Oops!")
      }
      i
    }.write.mode("overwrite").csv("/tmp/test-2/")
```

changed:-

```
> spark.range(100).repartition(2).write.mode("
    overwrite").format("delta")
    .save("/tmp/test-1/")
```

```
> scala.util.Try(spark.range(100).repartition(2)
    .map{i=>
      if(i>50){
```

```
        Thread.sleep(5000)
        throw new RuntimeException("Oops!")
      }
      i
    }.write.mode("overwrite").format
     ("delta").save("/tmp/test-1")
```

but fails because of the mismatch in the column names, as shown below:-

```
> spark.range(100).pointSchema
--id: long (nullable = false)

> spark.range(100).repartition(1).map{ i =>
        if (i > 50) {
            Thread.sleep(5000)
            throw new RuntimeException("Oops!")
        }
        i
    }.pointSchema
--value: long (nullable = true)
```

Rechanged ~~to~~ fit the Columnname —

```
>spark.range(100).repartition(1).map{i =>
        if (i>50){
                Thread.sleep(5000)
                throw new RuntimeException("Oops!")
        }
        i
}.select(C$"value".as("id"))
 .write.mode("Overwrite")
 .format("delta")
 .save("/tmp/delta-test-2/")
```

Descipt the Runtime error data is not lost.

testing for Data Atomicity :-

```
> spark.read.format("delta").load("/tmp/test-2/")
                            .count
```

~~test~~

Hence, Delta lake uses log files ~~to~~ in "_deltalog" directory to provide Atomicity for Runtime errors & to ~~read the~~ maintain a list of Data additions & deletions.

Sample ETL :-

```
$ hadoop fs -cat /tmp/data-batch2/file
FName, LName, Phone, Age
Salman, Khan, 9876543210, 52
Akshay, Kumar, 9823456789, 48
```

Back End Data loading :-

```
> val df = spark.read.format("csv").option
        ("header", "true").option
        ("inferSchema", "true").load
        ("/tmp/data-batch2/")

> val df1 = df.select($"FName", $"LName",
        $"Phone", $"Age",
        (when($"Age" > 50, "old").otherwise("Young"))
        .alias("AgeGroup"))

> df1.write.format("delta").mode("overwrite")
        .save("/tmp/stars")

> df1.show()
```

FName	LName	Phone	Age	AgeGroup
Salman	Khan	9876543210	52	Old
Akshay	Kumar	9823456789	48	Young

~~Data Manipulation~~

Reading Data from Delta lake :—

> val dt = spark.read.format("delta").load("/tmp/Stocks")

↓

Reads as a dataframe.

> import io.delta.tables.—

> val dt = DeltaTable.forPath(spark, "/tmp/Stocks")

↓

Reads as a "io.delta.tables.DeltaTable"

With Delta Table => additions, Updations & conversion to standard Dataframe is possible.

> dt.delete("FName == 'Salmaan'")

> dt.toDF.show

FName	LName	Phone	Age	AgeGroup
Akshay	Kumar	9823456789	48	Young

> dt.updateExpr("FName == 'Akshay'", Map("Age" → "Age+5"))

Upsert using Delta Lake

```
> import io.delta.tables._
> val dt = DeltaTable.forPath(spark, "/tmp/Staas")
> dt.toDF.show
```

FName	LName	Phone	Age	AgeGroup
Salman	Khan	9876543210	52	Old
AKshay	Kumar	9823456789	48	Young

db batch

```
> $ hadoop fs -cat /tmp/data-batch2/*
FName, LName, Phone, Age
AKshay, Kumar, 6832256789, 51 => Update
Saif Ali, Khan, 9123456789, 55 => New
Sanjay, Dutt, 9833712345, 58 => New
```

```
> val df = spark.read.format("csv").option
("header", "true").schema("FName
STRING, LName STRING, Phone STRING
, Age DOUBLE").load("/tmp/data-batch2")
```

```
> val df1 = df.select($"FName", $"LName", $"Phone"
, $"Age", (when($"Age" > 50, "Old")
.otherwise("Young")).alias("AgeGroup"))
```

```
> df1.show
```

FName	LName	Phone	Age	AgeGroup
Akshay	Kumarr	65325678	57.0	old
Saif Ali	Khan	912345678	55	old
Sanjay	Dutt	9843324345	58	old

New batch Data

```
> dt.as("stores")
    .merge(df1.as("inputs"), "stores.FName
                              = inputs.FName")
    .whenMatched()
    .updateExpr(
        Map(
            "LName" -> "inputs.LName",
            "Phone" -> "inputs.Phone",
            "Age" -> "inputs.Age",
            "AgeGroup" -> "inputs.AgeGroup"
            )
        )
    .whenNotMatched
    .insertAll
    .execute();
```

Upsert operation

> dt.toDF.show // New data inserted & Updated

FName	LName	Phone	Age	AgeGroup
Saif Ali	Khna	912345789	55	old
Akshay	Kumar	6832567789	51	old
Sanjay	Dutt	9843321345	58	old
Salman	Khan	9876543210	52	old

Time travel using Delta Lake

> dt.history.show(false) // to get the // commit history

// Using commit versions

>spark.read.format("delta").option("versionAsOf", 0).load("/tmp/Stars").show

>spark.read.format("delta").option("versionAsOf", 5).load("/tmp/Stars").show

// time travel using commit time

// There are few specifics of reading Data via commit time.

// User can't read before the first commit time & after the last commit time. Trying to read beyond the first & last commit time will result in an exception due to lack of data snapshot in these time periods.

// commit_time_of_version0 ≤ commit_time_of_version1 will result in Data@version0

// to get the most recent timestamp Data load delta lake with-out timestamp.

->spark.read.format("delta").option("timestampAsOf", "2019-09-25 15:50:00.000").load("/tmp/stars").show

↓

for Data before 2019-09-25 15:50:00.000 time

->spark.read.format("delta").load("/tmp/stars").show

↓

for most recent Data.

Summary :-

Atomocity :- [Either all or nothing]
failed job Data in Delta Doesn't get updated in the logs.
Due to loging mechanism inbetween old Data is not lost.

Consistency :- [Data is always in the valid state]
till Data writting is completed by the current ETL, the previous version is available consistantly to the users. Only once the transactional log is updated the data view is changed.

Isolation :- [Can operation must be isolated]
Delta tables are auto refresh & Data is always aviable. Hence Isolation is achieved.

Durability :- [Once committed data is never lost]
Delta table are stored are distributed storage system. Hence, data is always ~~cons~~ available due to partition Tolerence.

Schema enforcement :- ~~Schema is checked~~ Delta Lake automically validates that the schema of the Dataframe being written is compatible with the schema of the table.

Time Travel :- Delta lake also provides built-in data versioning for easy roll backs & reproducing reports.

Small file problems:—

Small file problems in Delta Lake can be handeled by "compaction" using "Auto OPTIMIZE" (or) "repartition"

Ex:- val df = spark.read.format("delta").load("/some/data")
↓
1000 files

df.repartition(10).write.format("delta").mode("overwrite")
.save("/some/data")
↓
10 10 files
(1000 ~~orig~~ previous data
10 current compacted

User can use "vaccum()" command to delete the old data files. like: VACCUM Table name
(or)
VACCUM delta.'/data/events'

Updation & Deletion:—

uses "merge", "whenMatched", "UpdateExps" & "whenNotMatched", "insertAll" & "execute()" commands to facilitate data addition & Updation

Using Optimize :-

OPTIMIZE delta.'/data/events'
(or)
OPTIMIZE events
(or)
OPTIMIZE events where data >= '2017-01-01'

Mulisth Data Skipping:—

"Z-ordering" is a technique to colocate related information in same set of files. This co-locality is automatically used for Data Skipping in-addition to "partitionBy" like:—

```
df.write.format("delta").mode("overwrite").
    partitionBy("Origin").save("/tmp/flights_delta")
spark.sql("DROP TABLE IF EXISTS flights")
spark.sql("create table flights Using DELTA
          LOCATION '/tmp/flights_delta'")
spark.sql("OPTIMIZE flights ZORDER BY (column_name)")
```