Why We Need Delta lake

Apache Spark & ACID :-

A → Atomicity (Either All or Nothing)
[possible at job level ~~in append mode~~ but comes at performance impact]

df.write.mode("append").csv("/tmp/test-1")

a) Means, when using spark writter API, The operation should result in all the Data being written (or) nothing written, but the save mode in spark do not utilize any locking and are not atomic.

b) Spark writtes API internally relies on a job level commit protocal to ensure some degree of atomicity. (Writes to a temp (temporary) & then to the Outfile)

c) when performing overwrite, the data will be deleted before writing out the new data

Ex :- (for No impact because of job level Atomicity in Spark)

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

No Impact

```
> scala.util.Try(spark.range(100).repartition(1).map{i =>
                  if(i > 50) {
                    Thread.sleep(5000)
                    throw new RuntimeException("Oops!")
                  }
                  i
                }.write.mode("append").csv("/tmp/test-2"))
```

Spark Job Commit Algorithm

For HDFS :-
- FileOutputCommitter version 1 → Default
  - ↳ ~~No~~ slow & Impacts Write performance
  - ↳ Provide job level Atomicity
- FileOutputCommitter version 2 → Alternative Option
  - ↳ Provides ~~not~~ job level Atomicity
  - ↳ Writes faster

For EMR :- EMRFS S3-optimized committer

For S3 :- Directory committer, Partitioned committer, Magic committer

For Databricks :- DBIO Commit Protocol

∴ Spark Commit Writter API's are not Atomic but mimics atomicity when in "append" mode. However, the Default Implementation comes with performance overhead, especially for Network intensive clusters.

For "overwrite" spark is not Atomic has spark first Deletes the Data irrespective of job results.

Apache Spark & ACID

(→ consistency (Data is always in the valid state)
[Not possible in Spark API]

```
df.write.mode("overwrite").csv("/tmp/test-2")
```

Delete + Write
↓
Time lag, when Data is not Consistent

Ex:- Impacted, fails in the middle & Data is lost.

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

>scala.util.Try(spark.range(100).repartition(10).map{i =>
                  if(i>50){
                  Thread.sleep(5000)
                  throw new RuntimeException("Oops!")
                  }
                  i
                  }.write.mode("overwrite").csv
                  ("/tmp/test-2"))
```

Due to time lag between Delete & Write operations & prior Deletion of Previous Data before Write operation Spark Overwrite job written is not Consistant & hence the Data is ~~big~~ Hadoop can only obtain eventual consistanc

Apache Spark & ACID :-

I - Isolation (An operation must be isolated from other concurrent operations)
[Spark doesn't offer isolation even at an API level.]

Spark doesn't offer isolation, which is typically done by transaction level commits in RDBMS.

Spark however offers task level & job level commits.

So, users cannot read a Data-set which a different process is overwriting

Apache Spark & ACID :—

D — Durability (once committed data is never lost)

[offers via partition tolerance.]

Durability is offered by the storage layer.

Why ACID is Critical :—

In event of runtime exception data is corrupted or lost.

faulty analytics due to inconsistent data.

Summary :— Spark is a ~~storage engine~~ Processing engine using typically HDFS/S3 for storage, YARN/Mesos/Kubernetes for cluster Resource Managment & Hive metastore for Database objects creation.

Using Hive metastore is the main reason Spark was in-able to provide ACID features required for reliable Data processing.

Other poblems :—

Missing schema enforcement.

→ Schema on Read Mechanism in Hadoop.

→ Spark doesn't support Automatic Schema evalution.

→ Data Seperation is impossible to identify the corrupted Data.

Small file problems (<128MB)

→ File Listing slows down the job performance significantly.

→ File Opening/Closing.

→ Reduced compression effectiveness due file overhead.

→ Excessive Metadata.

Mulish data skipping using partition.

→ using where clause in SQL.

→ Work only for :-

- Chronological columns (Ex: Date)
- Low cardinality columns (Ex: CountryCode)
- Combination of chronological & Cardinality columns.

→ lacks indexing