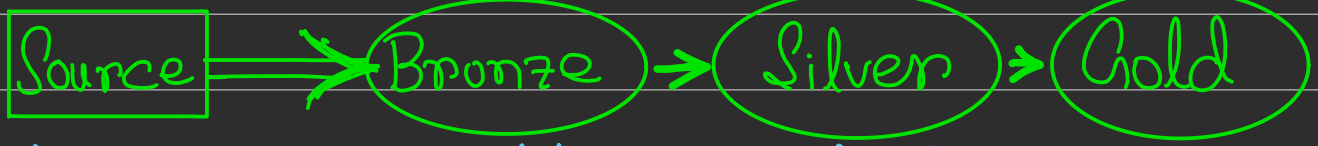
- Python variable usage :- {variable}
  Hive SQL variable usage :- ${variable}

- Data Lakehouse enables design patterns and use cases associated with data warehouses, stored in an open format (feature of delta lake) in economical cloud object storage aka Data Lake

Lakehouse Medallion Architecture :-



Files or integrated systems

Raw data + Metadata

Validated data with atomic grain

Refined, aggregated data.

Not all pipelines need to follow this above architecture.

This above architecture takes advantage of the Delta Lakes' Table Scoped Transactional guarantees.

Enhanced incremental data processing by integration of Delta Lake & Spark Structured Streaming

⇒ BRONZE LAYER :-

⇒ Replaces traditional data lake.

⇒ Maintains the raw state of the data source. Captures exactly without change.

⇒ Main objective is to get data into the Lakehouse

⇒ Should be majorly <u>append-only</u>.

⇒ SILVER LAYER :-

⇒ Collection of tables that stores a validated copy of all data events.

⇒ Uses <u>Delta Lake Tables with SQL table names.</u>

⇒ No aggregation - preserves data grain.

⇒ Eliminates duplicates. Enforces Production Schema.

⇒ GOLD LAYER :-

⇒ Powers ML apps, reporting, dashboards.
⇒ Allows fine-grained permissions.

⇒ Contains data that has been transformed to knowledge.

⇒ STREAMING DESIGN PATTERNS

⇒ Spark Structured Streaming can be used to process data in near-realtime, as well as in batches of incremental data, overculling the complexities to track data changes over time.

Ex:- We have a "bronze" table and using it as source, we want to update "silver" table, using structured streaming:-

```
from pyspark.sql import functions as F
def update_silver():
    query = spark.readStream.table("Bronze")
                .withColumn("processed_time", F.current_timestamp())
                .writeStream.options("checkpointLocation", "...")
                .trigger(availableNow=True)
                .table("Silver")
    query.awaitTermination()
```

⇒ When the above update_silver() function will be executed, all the new rows added to "Bronze" table, will be added to the "silver" table and can be understood as Incremental Load.

Though, Structured Streaming is used, .trigger(availableNow = True) and query.awaitTermination() will cause stream to run once, for a very short duration, till the updation is done. Thus, it is making it as a Batch process.

⇒ Using this, we can easily track the propagating data through a series of tables.
This architecture is called Medallion / Multi-Hop / bronze-silver-gold architecture.

⇒ Applying Custom write logic on Streaming Data using foreachBatch()

This always does writing in batches, even for streaming data. Thus, inside it, we only use write (not writeStream)

this parameter gets the new set of records, that the streaming query reads currently

```
def write_twice( microBatchDF, batchId ):
    appId = "write_twice"

    microBatchDF.select( "id", "name", F.current_timestamp().alias("processed_time") )  \
    .write.option("txnVersion", batchId).option("txnAppId", appId).mode("append")   \
    .saveAsTable("silver_name")

    microBatchDF.select( "id", "value", F.current_timestamp().alias("processed_time") ) \
    .write.option("txnVersion", batchId).option("txnAppId", appId).mode("append")   \
    .saveAsTable("silver_value")
    #write_twice() function body ends here

def split_stream():
    query = (
        spark.readStream.table("Bronze")
        .writeStream.foreachBatch( write_twice )
        .option( "checkpointLocation", f"{DA.paths.checkpoints}/split_stream" )
        .trigger(availableNow = True)
        .start()
    )
    query.awaitTermination()
```

to this function, each new set of records will be passed as a dataframe in the 1st parameter, and batch Id of that batch of data, as the next parameter.

Here in the split_stream() function (that actually handles the streaming query), we just read the streaming data from Bronze table, and write it using writeStream

In writeStream, we need not set output mode or location, as everything is handled in the custom writing logic in write_twice() function.
We just pass each incremental set of records to the custom logic function, using foreachBatch(), and the rest part is handled there.

split_stream() is only responsible for starting & stopping of stream reading & writing, and passing incremental records to custom logic function.
The set of new records, is called micro batch.

⇒ When we write data from single table into multiple down stream tables, setting txnVersion & txnAppId options make each write Idempotent.

⇒ Using foreachBatch() to write to multiple tables keeps all the tables in sync.
On the other hand,
Using multiple streams for multi-table writes makes each write independent.

⇒ Significance of .trigger() :-

⇒ In streaming read or write, if we do not set .trigger(), then, by default, stream operation runs every 500 ms.

⇒ If we set trigger, and an interval in

that, then the trigger runs, after that interval, repeated by.

=> Setting trigger (availableNow = True), does not set any interval, and runs the stream check only once.

=> Checking whether a streaming query is still active :-

```
query = spark.readStream.table("bronze")
print(query.isActive)  => True
```

isActive member variable contains a boolean value saying whether a stream is currently active.

⇒ Checking progress of a query :-

[query.recentProgress contains the no.

⇒ of queries currently processed by a stream.

⇒ Can only be used inside a function, where the query is passed as a parameter.

⇒ Checking the number & name of active streams

⇒ spark.streams.active contains a list of Stream objects, that are currently active.

⇒ We can see the no. of active streams, and their names with the following prog:-

```
print(len(spark.streams.active))

for stream in spark.streams.active:
    print(stream.name)
```

the stream object contains a name member variable, containing the stream / streaming query name.

⇒ A stream can be be given a custom name, by passing the name as a string parameter to the .queryName() attribute, while defining the streaming query.

```
type(query) == type(spark.streams.active[0])
```

⇒ Active streams can be stopped as follows:

```
for stream in spark.streams.active:
    stream.stop()  ≈  query.stop()
    stream.awaitTermination()
```

Joining a static table with a streaming table :-

```
staticDF = spark.read.table("1")
streamDF = spark.readStream.table("2")

query = ( staticDF.join(streamDF, staticD.id == streamDF.id, "inner")
          .select("staticDF.*", "streamDF.*")
          .writeStream
          .option("checkpointLocation", "...")
          .table("join_table")
        )
```

as no trigger is mentioned, this stream will continuously run every 500ms

joined results will be saved to this table.

When a stream table & static table is joined, using writeStream as above, then :-

Even if the stream continuously runs, if we make any changes or updation in the static table, no changes will be updated in the resulting join_table.

b> If we make any updation in the streaming table (optionally, also updation is done on static table, then the resulting join-table will be updated with changes, to reflect the most recent version of both static table & streaming table.

Each microbatch of the streaming data, joins with the current values of static table, in case of stream-static join.