Capturing Change Data From Delta Lake

Paul Roome and Denny Lee

Who are we?

- Senior Solution Architect – Databricks
- Previously:
  - ML driven data unification @ Tamr
  - Fighting fraud & organized crime with graph analytics @ NetReveal
- Masters in Pure Mathematics - Edinburgh

databricks

Who are we?

databricks

- Developer Advocate – Databricks
- Working with Apache Spark™ since v0.6
- Former Senior Director Data Science Engineering at Concur
- Former Microsoftie: Cosmos DB, HDInsight (Isotope)
- Masters Biomedical Informatics - OHSU
- BS in Physiology - McGill

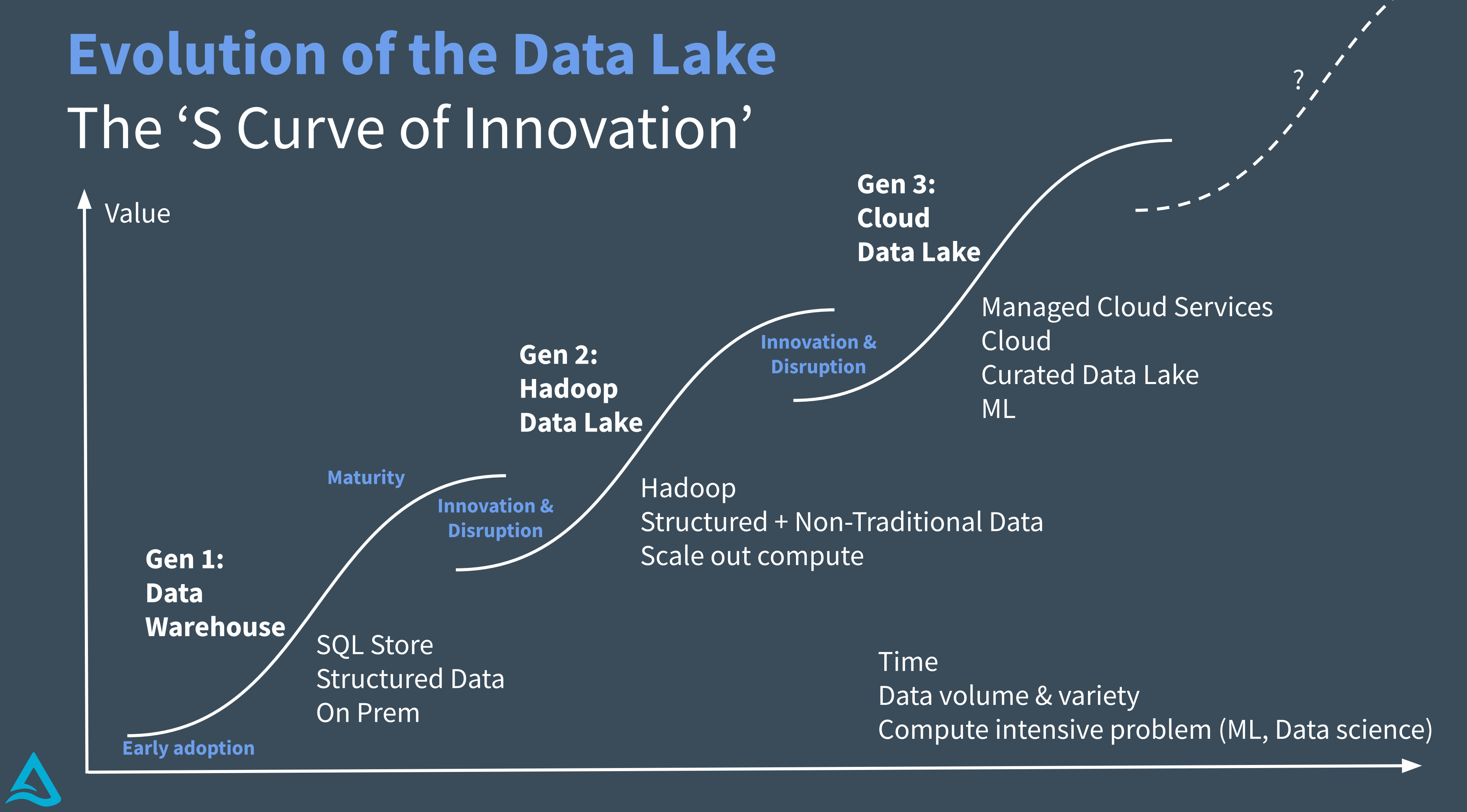
Outline

- Motivation
- Pattern 1: Bronze-silver-gold propagation
- Pattern 2: What about updates?
- Pattern 3: Can we do better?
- Summary
- Questions

Motivation

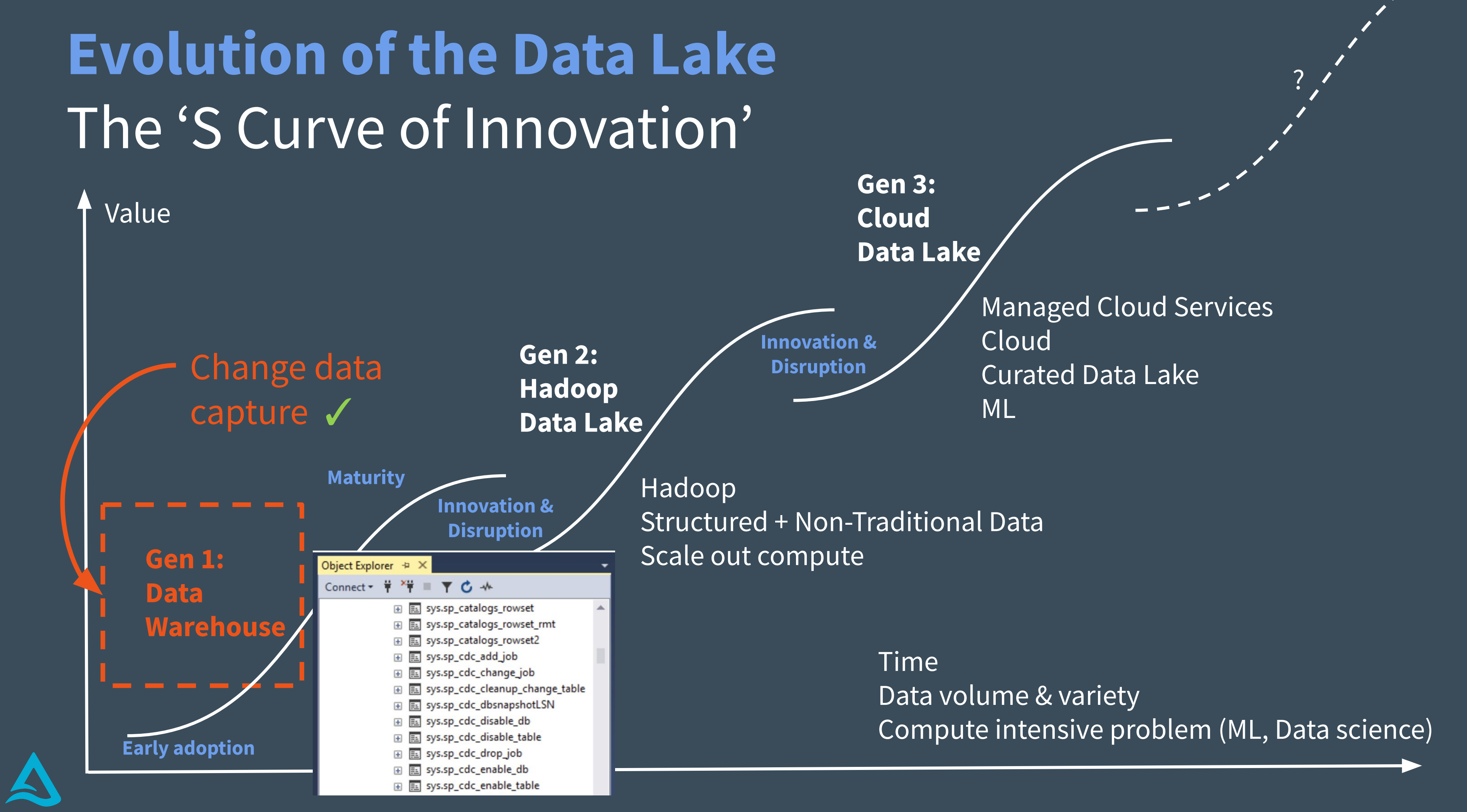
Evolution of the Data Lake

The 'S Curve of Innovation'



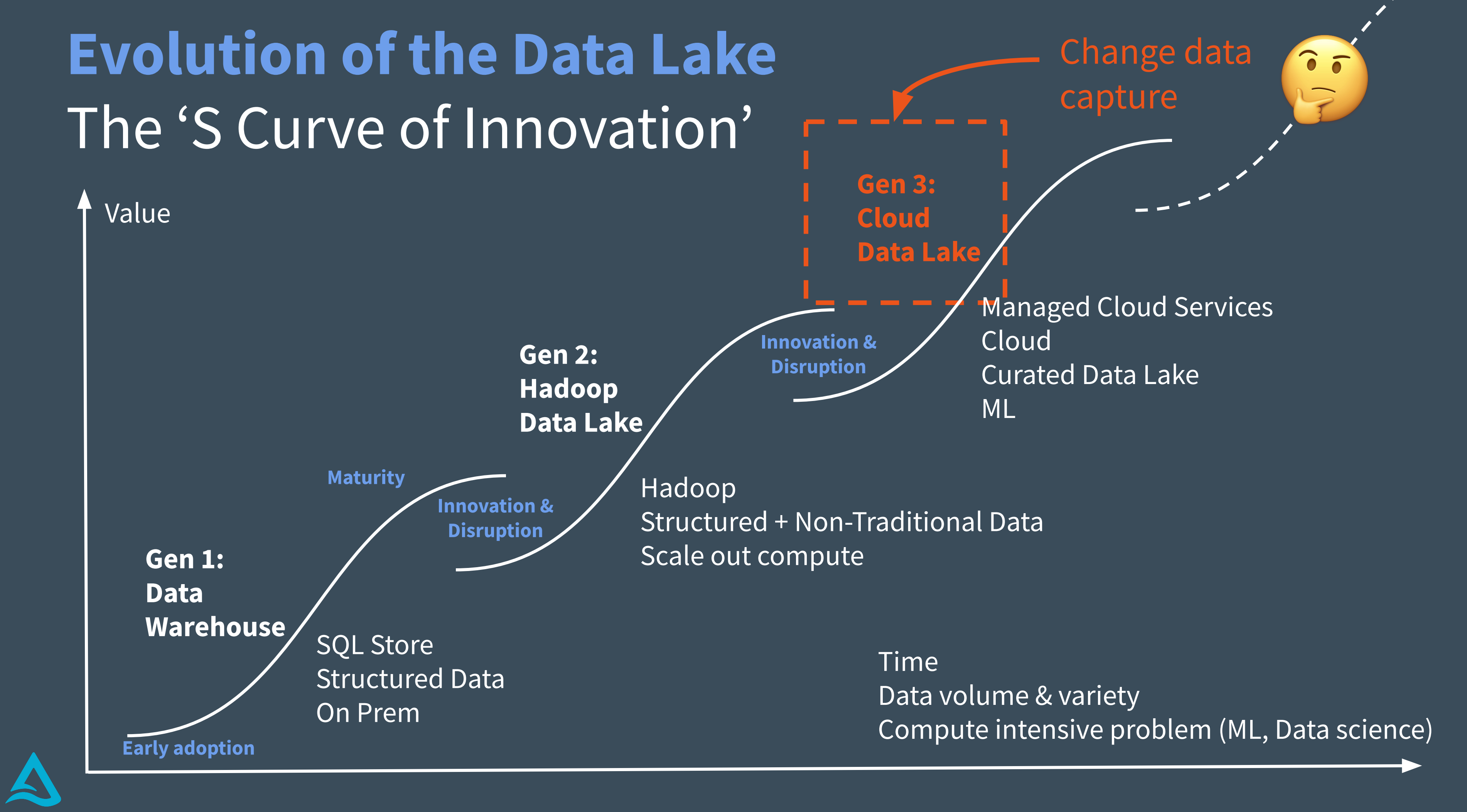
Evolution of the Data Lake

The ‘S Curve of Innovation’



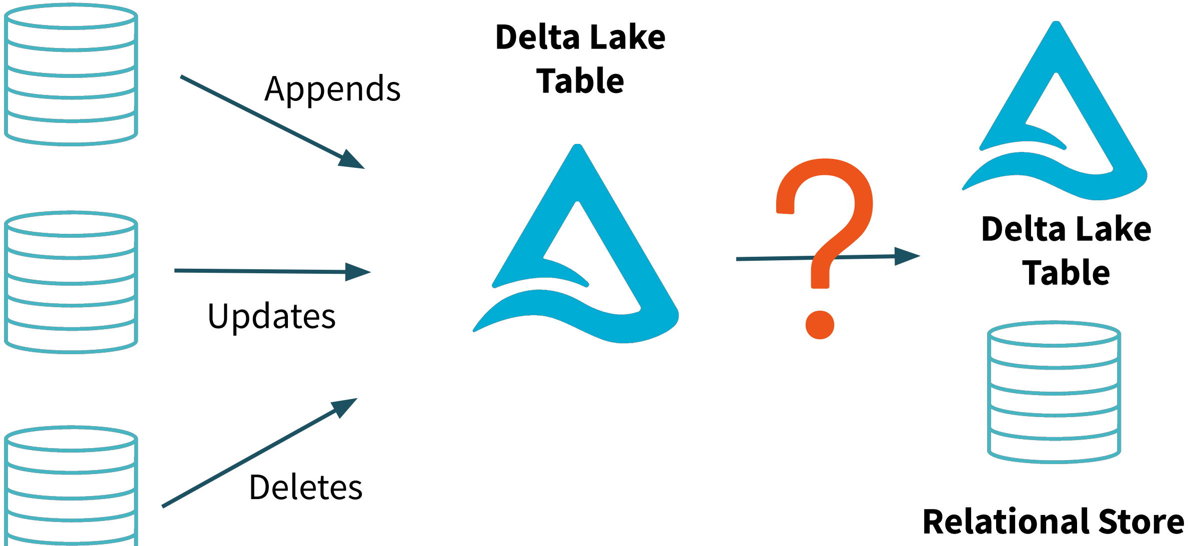
Evolution of the Data Lake

The ‘S Curve of Innovation’



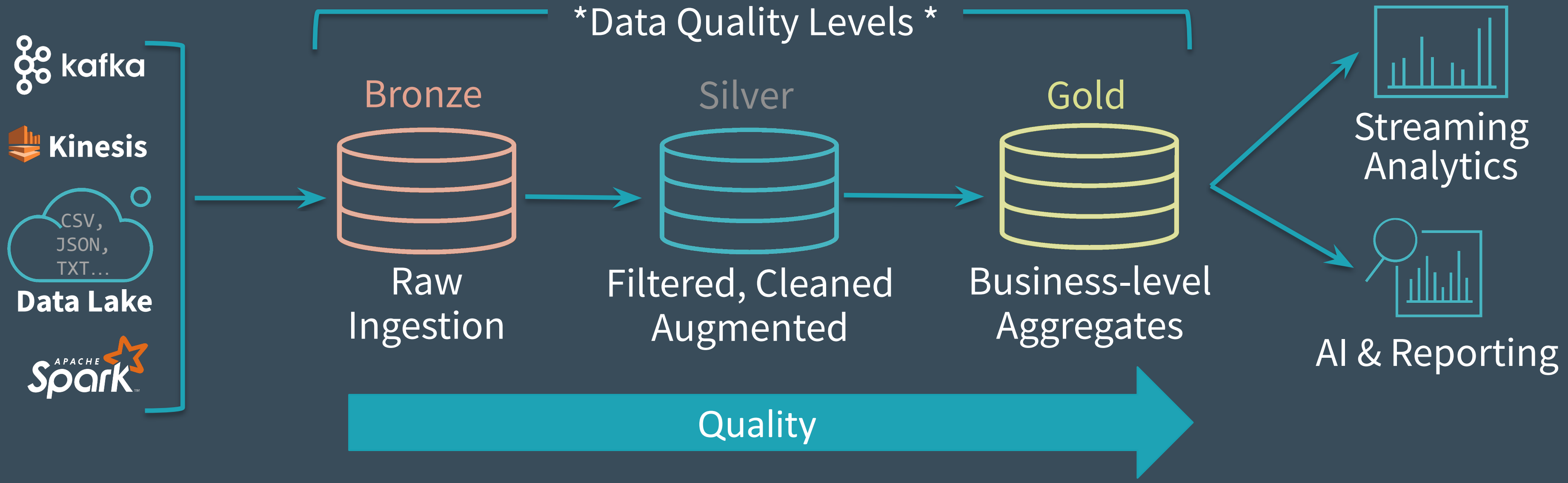
Downstream Propagation

Upstream | **Downstream**



Pattern 1:
Bronze-Silver-Gold propagation

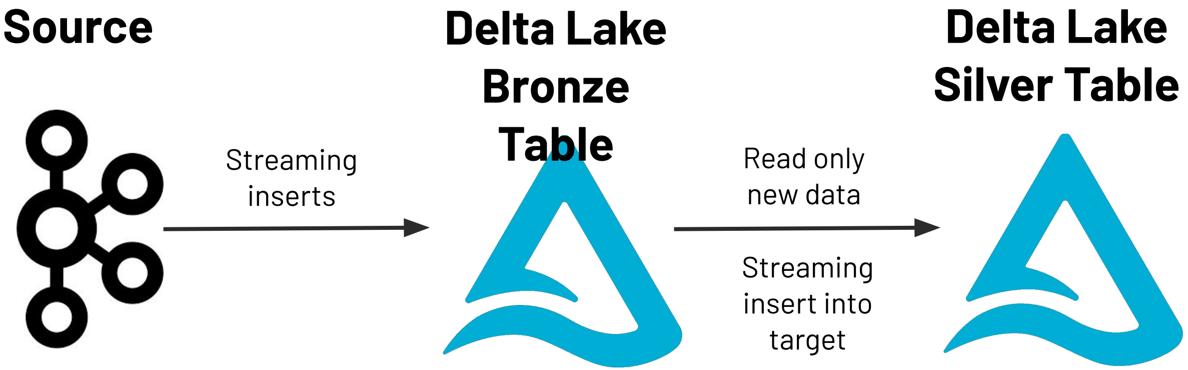
The DELTA LAKE Architecture



Delta Lake allows you to *incrementally* improve the quality of your data until it is ready for consumption.

Pattern 1a

Example: Propagation from Bronze to Silver Layer of your Delta Lake



Key Assumption: Bronze table is append only

Reading Delta table as a Stream

```
val bronzeData = spark.readStream.format("delta")
                              .load("/mnt/databricks-paul/customer-bronze")
```

>> Can **read Delta table as a stream!**

>> This is a scalable and commonly used pattern!

Writing Delta table as a Stream

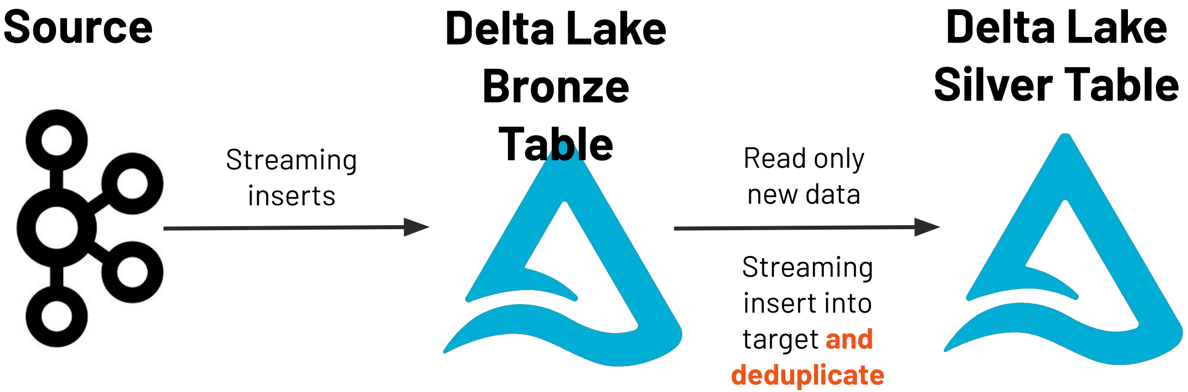
```
bronzeDataCleaned.writeStream.format("delta")
                     .option("checkpointLocation","/mnt/checkpoint/")
                     .start("/mnt/databricks-paul/customer-silver")
```

>> Use of checkpoint ensures we are reading only **new data** in each batch

Demo

Pattern 1b

Example: Propagation from Bronze to Silver Layer of your Delta Lake, **with deduplication against the sink**



Key Assumption: Bronze table is append only

Writing and Deduplicating

```
bronzeDataCleaned.writeStream.format("delta")
                .foreachBatch(deduplicateInsert _)
                .option("checkpointLocation","/mnt/checkpoint-dedupe")
                .start()
```

>> Can call a function within **foreachBatch** to perform dedupe

Deduplicating within foreachbatch

Target Delta table

New data to insert

Match condition

Insert everything that doesn't match

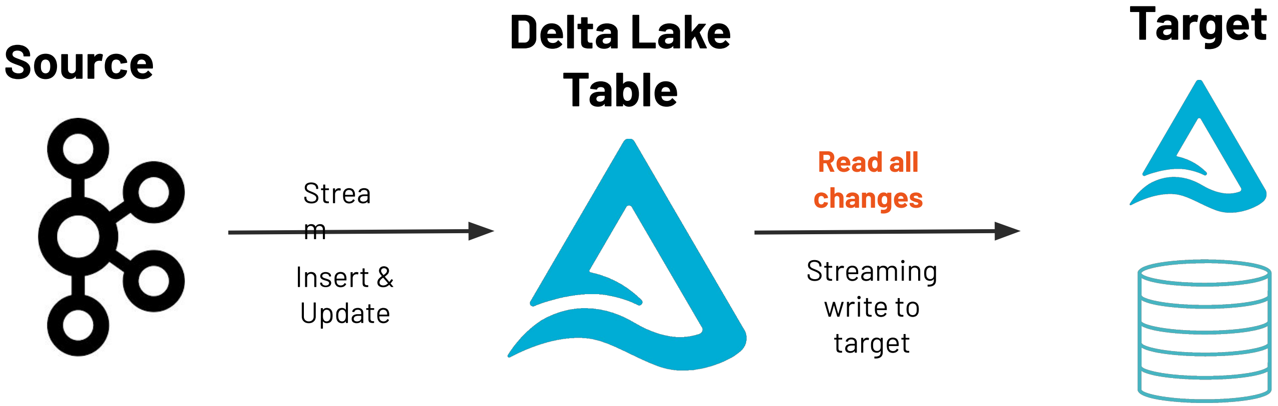
```
def deduplicateInsert(microBatch: DataFrame,
batchId: Long) {
  silverDelta
  .as("silver")
    .merge(
    microBatch.as("newBronzeData"),
    "silver.last = newBronzeData.last")
  .whenNotMatched()
  .insertAll()
  .execute()
}
```

Demo

Pattern 2: What about updates?

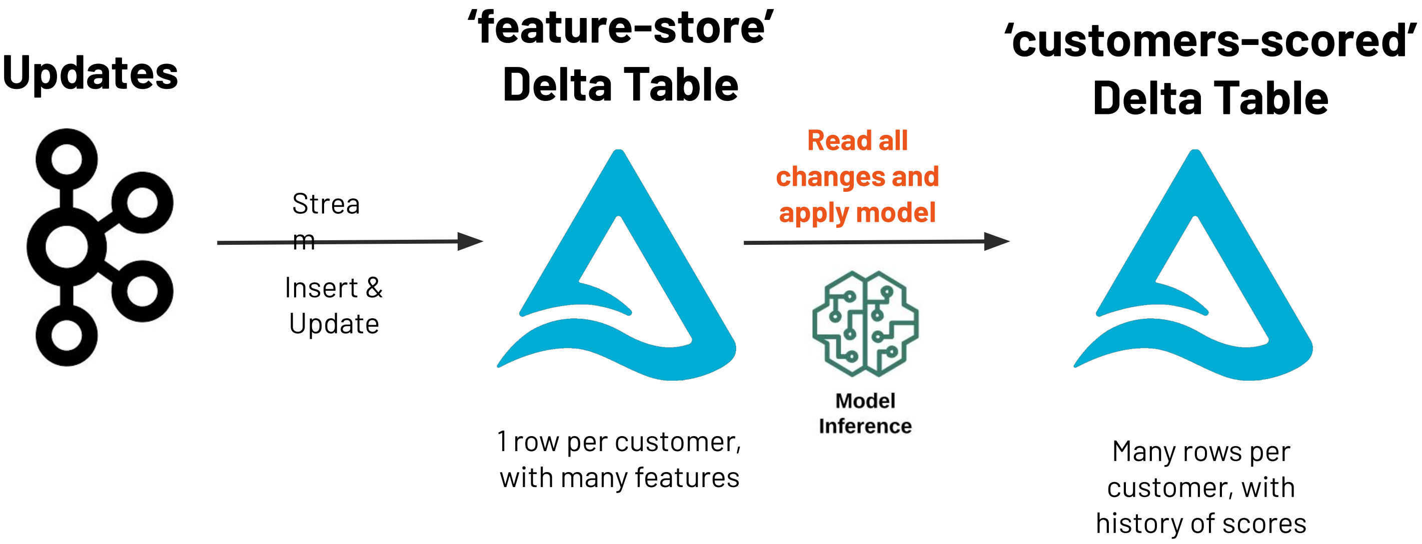
Pattern 2

Example: Propagating data from a Delta table that **has updates applied to it**



Key Assumption: Deletes don’t need to be propagated

Pattern 2 - Feature Store Example



Reading change stream from Delta table

```
val featureData = spark.readStream.format("delta")
                        .option("ignoreChanges", "true")
                        .load("/mnt/databricks-paul/feature-store")
```

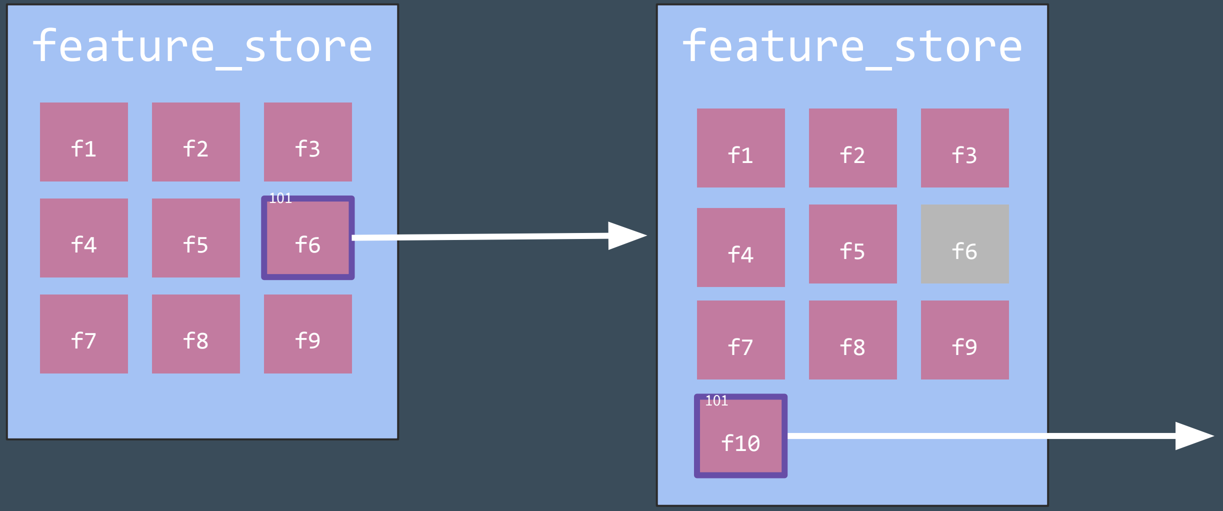
>> Setting **ignoreChanges** to **true** will emit all rewritten files in the Delta table to the stream

- >> This will be a **superset** of actually changed records
- >> **Deletes** will not be captured

Demo

Reading change stream from Delta table

```
UPDATE feature_store SET name="xxxx" WHERE customer_id=101
```



Demo

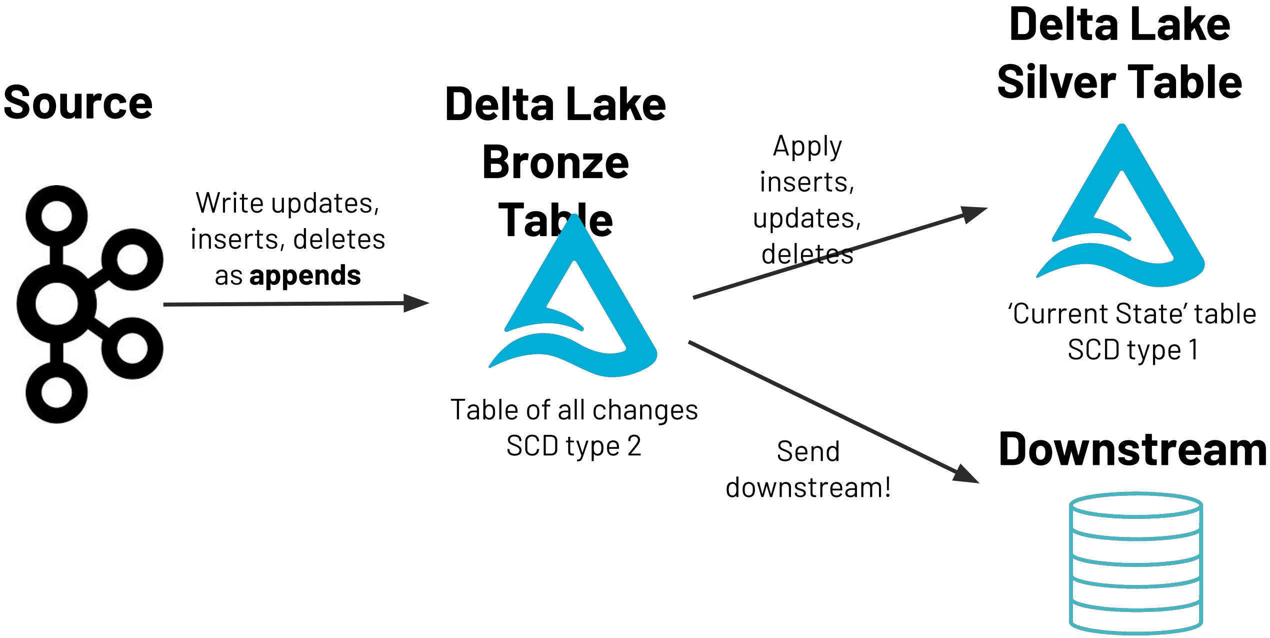
```
spark.readStream
.format("delta")
.option("ignoreChanges
"true")
```

Pattern 3: Can we do better?

Pattern 3a

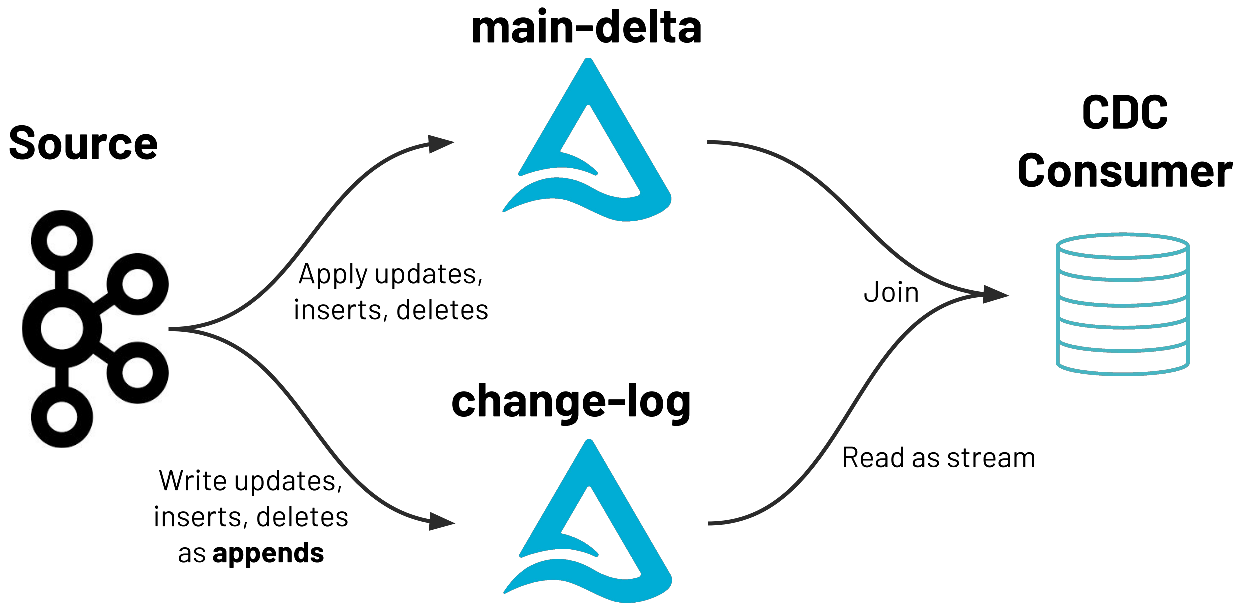
Example: Achieving a record-level change stream

>> The simple case: Re-purpose pattern 1!



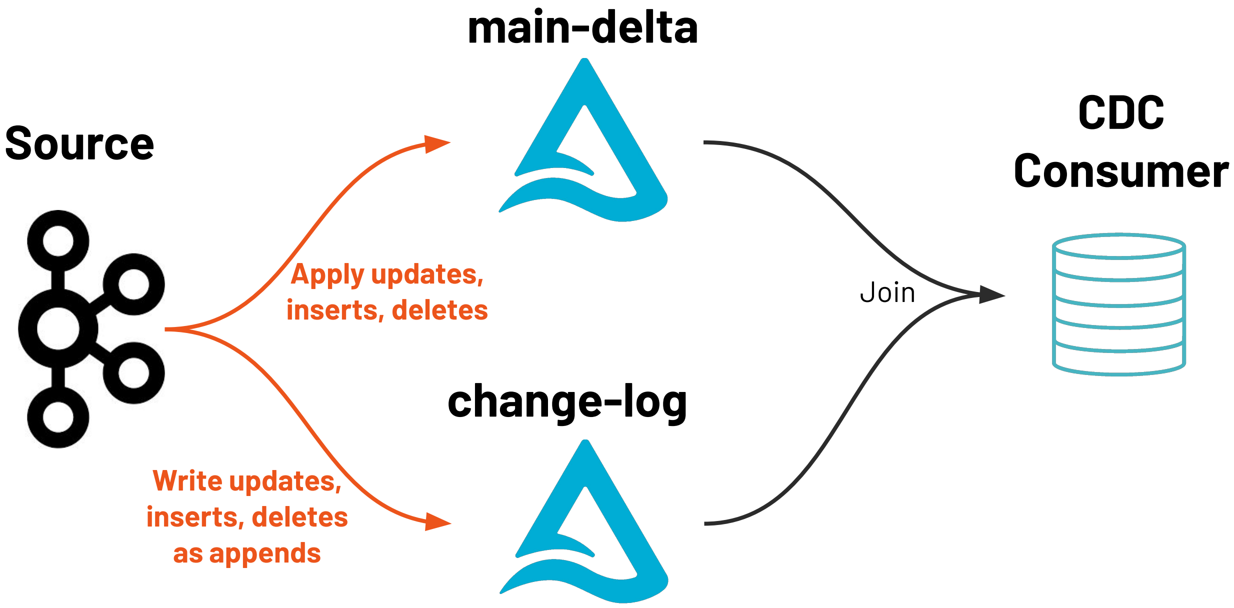
Pattern 3b

Example: Achieving a record-level change stream



Pattern 3b

Example: Achieving a record-level change stream



Writing to 2 Delta Tables

```
streamingUpserts.writeStream
                .format("delta")
                .foreachBatch(upsertToDeltaCaptureCDC _)
                .option("checkpointLocation","/update-table-checkpoint/")
                .start()
```

Magic happens here...

For simplicity in this example, we are going to assume only updates. Can easily be extended.

Pattern 3b

Starting point...

Source

id	some_string	change type
0	oranges	update
2	oranges	update

main-delta

id	some_ string	some_other_ string
0	apples	cake
1	apples	cake
2	apples	cake
3	apples	cake

change-log

id	batch_id

CDC Consumer

id	some_ string	some_other_ string

Pattern 3b

Source

id	some_string	change type

Apply updates, inserts, deletes

Write as inserts

main-delta

id	some_string	some_other_string
0	oranges	cake
1	apples	cake
2	oranges	cake
3	apples	cake

change-log

id	batch_id
0	0
2	0

CDC Consumer

id	some_string	some_other_string

Apply updates and append to change log...

Pattern 3b

main-delta

id	some_string	some_other_string
0	oranges	cake
1	apples	cake
2	oranges	cake
3	apples	cake

Source

id	some_string	change type

CDC Consumer

id	some_string	some_other_string
0	oranges	cake
2	oranges	cake

Join

Read as stream

change-log

id	batch_id
0	0
2	0

Read change-log as a stream and join to main-table

Assemble change stream

```
val changeStream = spark.readStream.
                        format("delta").
                        option("ignoreChanges","true")
                        .load("/mnt/databricks-paul/change-log/")

val changeStreamFullRecords = changeStream.join(mainTable, Seq("id"), "inner")
```

Read change log table as a stream

Grab full record from the main table

>> Delta will auto-refresh the dataframe mainTable to contain up to date data

Demo

Writing to 2 Delta Tables

```
def upsertToDeltaCaptureCDC(microBatchOutputDF: DataFrame, batchId: Long) {

    val batchDF = microBatchOutputDF.withColumn("batchId", lit(batchId))
                                     .dropDuplicates(Seq("id","batchId"))

    ...to be continued
```

- Grab latest batch of records
- Add the batchId of the micro-batch (useful to ensure idempotency later)
- Drop duplicates

Writing to 2 Delta Tables

```
def upsertToDeltaCaptureCDC(microBatchOutputDF: DataFrame, batchId: Long) {

    ...continued

    // Write 1: Merge into the main table and perform update
    mainDelta.as("m")
     .merge(
       batchDF.as("b"),
       "m.id = b.id")
     .whenMatched().updateExpr(Map(
       "some_string" -> "b.some_string"))
     .execute()

    ...tbc
```

When we find a match on ‘id’ column…

…update the column ‘some_string’

Writing to 2 Delta Tables

```
def upsertToDeltaCaptureCDC(microBatchOutputDF: DataFrame, batchId: Long) {
...continued

  // Write 2: Insert records into the change log table
  changeLog.as("c")
    .merge(
      batchDF.as("b"),
      "c.id = b.id AND c.batchId = b.batchId")
    .whenNotMatched().insertExpr(Map(
     "id" -> "b.id",
     "batchId" -> "b.batchId"))
    .execute()
}
```

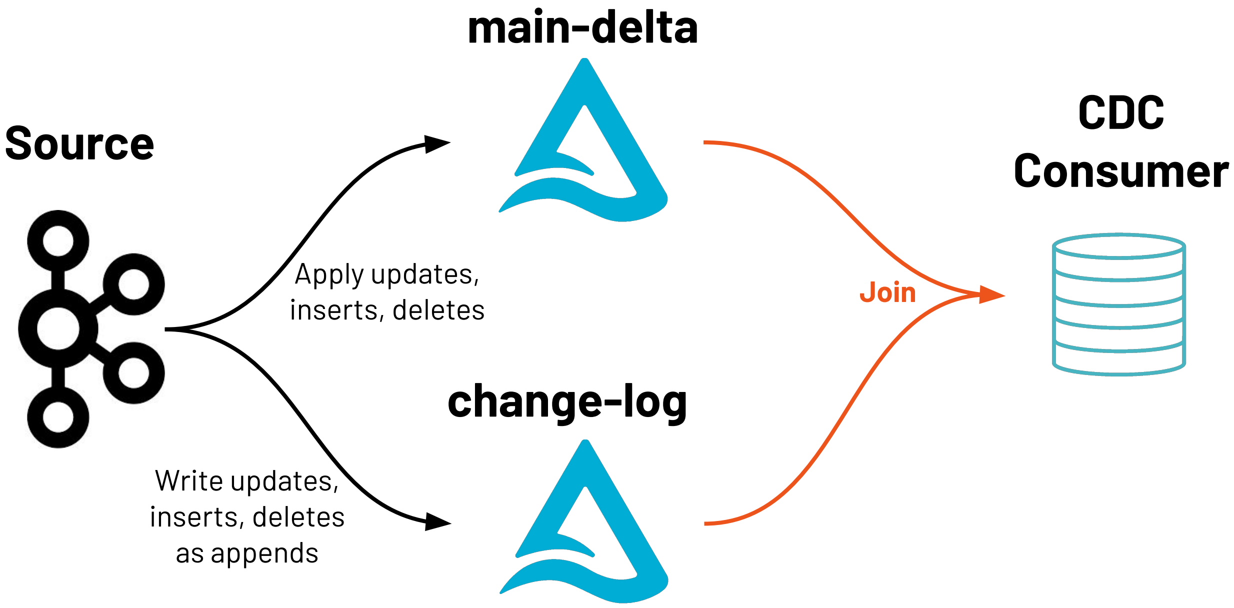
Check if this (id, batchId) pair exists in the change log (e.g. from failed run)...

...if not, insert

...if true, do nothing

Pattern 3b

Example: Achieving a record-level change stream



Assemble change stream

```
val changeStream = spark.readStream.
                        format("delta").
                        option("ignoreChanges","true")
                        .load("/mnt/databricks-paul/change-log/")

val changeStreamFullRecords = changeStream.join(mainTable, Seq("id"), "inner")
```

Read change log table as a stream

Grab full record from the main table

>> Delta will auto-refresh the dataframe mainTable to contain up to date data

Demo

Pattern 3b

Source

id	some_string	change type
0	oranges	update
2	oranges	update
10		delete

Apply updates, inserts, deletes

Write as inserts

main-delta

id	some_string	some_other_string
0	oranges	cake
1	apples	cake
2	oranges	cake
~~10~~	~~apples~~	~~cake~~

change-log

id	batch_id	change_type
0	0	update
2	0	update
10	0	delete

Join

CDC Consumer

id	some_string	some_other_string	change_type
0	oranges	cake	update
2	oranges	cake	update
10			delete

Can use this pattern to deal with **deletes**...

Summary

Summary

We’ve looked at 3 patterns of reading change streams from Delta Tables:

1. **Append only pipelines** - commonly used for bronze - silver - gold propagation
2. **Update pipelines** - ignoreChanges == ‘the easy button’ when file level changes are acceptable
3. **Change log tables** for capturing record level change streams

Thank You

"Do you have any questions for my prepared answers?"

– Henry Kissinger