databricks 01 - Data Engineering with Delta

(https://databricks.com)

Building a Spark Data pipeline with Delta Lake

With this notebook we are buidling an end-to-end pipeline consuming our customers information.

We are implementing a *medaillon / multi-hop* architecture, but we could also build a star schema, a data vault or follow any other modeling approach.

With traditional systems this can be challenging due to:

- data quality issues
- running concurrent operations
- running DELETE/UPDATE/MERGE operations on files
- governance & schema evolution
- poor performance from ingesting millions of small files on cloud blob storage
- processing & analysing unstructured data (image, video...)
- switching between batch or streaming depending of your requirements...

Overcoming these challenges with Delta Lake

What's Delta Lake? It's a OSS standard that brings SQL Transactional database capabilities on top of parquet files!

Used as a Spark format, built on top of Spark API / SQL

- **ACID transactions** (Multiple writers can simultaneously modify a data set)
- **Full DML support** (UPDATE/DELETE/MERGE)
- **BATCH and STREAMING** support
- **Data quality** (Expectations, Schema Enforcement, Inference and Evolution)
- **TIME TRAVEL** (Look back on how data looked like in the past)
- **Performance boost** with Z-Order, data skipping and Caching, which solve the small files problem

DELTA LAKE

Exploring the dataset

Let's review first the raw data landed on our blob storage

```
%run ./includes/SetupLab
```

```
userRawDataDirectory = rawDataDirectory + '/users'
print('User raw data under folder: ' + userRawDataDirectory)

# Listing the files under the directory
for fileInfo in dbutils.fs.ls(userRawDataDirectory): print(fileInfo.name)
```

Achieve the same with a "unix-like" command

```
%fs ls /cloud_lakehouse_labs/retail/raw/users
```

Review the raw user data received as JSON

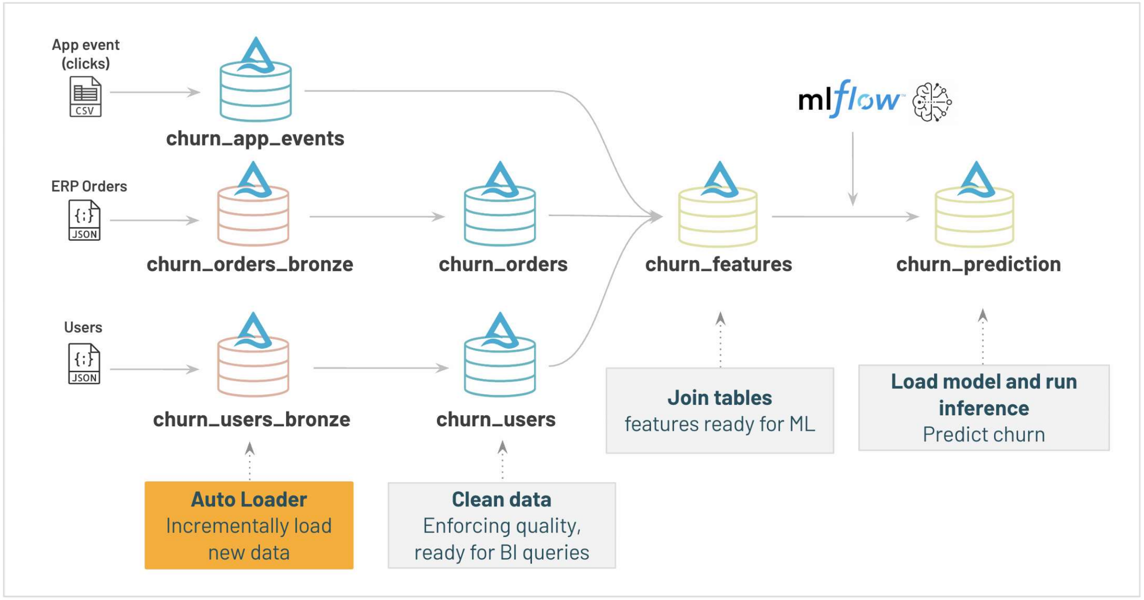
```
%sql
SELECT * FROM json.`/cloud_lakehouse_labs/retail/raw/users`
```

Exercise: Try to explore the orders and events data under the /orders and /events subfolders respectively

1/ Loading our data using Databricks Autoloader (cloud_files)

The Autoloader allows us to efficiently ingest millions of files from a cloud storage, and support efficient schema inference and evolution at scale.

Let's use it to ingest the raw JSON & CSV data being delivered in our blob storage into the *bronze* tables



Storing the raw data in "bronze" Delta tables, supporting schema evolution and incorre

```
def ingest_folder(folder, data_format, table):
  bronze_products = (spark.readStream
                      .format("cloudFiles")
                      .option("cloudFiles.format", data_format)
                      .option("cloudFiles.inferColumnTypes", "true")
                      .option("cloudFiles.schemaLocation",
                              f"{deltaTablesDirectory}/schema/{table}") #Autoloader will automatically infer all the schema & evolution
                      .load(folder))
  return (bronze_products.writeStream
            .option("checkpointLocation",
                    f"{deltaTablesDirectory}/checkpoint/{table}") #exactly once delivery on Delta tables over restart/kill
            .option("mergeSchema", "true") #merge any new column dynamically
            .trigger(once = True) #Remove for real time streaming
            .table(table)) #Table will be created if we haven't specified the schema first

ingest_folder(rawDataDirectory + '/orders', 'json', 'churn_orders_bronze')
ingest_folder(rawDataDirectory + '/events', 'csv', 'churn_app_events')
ingest_folder(rawDataDirectory + '/users', 'json',  'churn_users_bronze').awaitTermination()
```

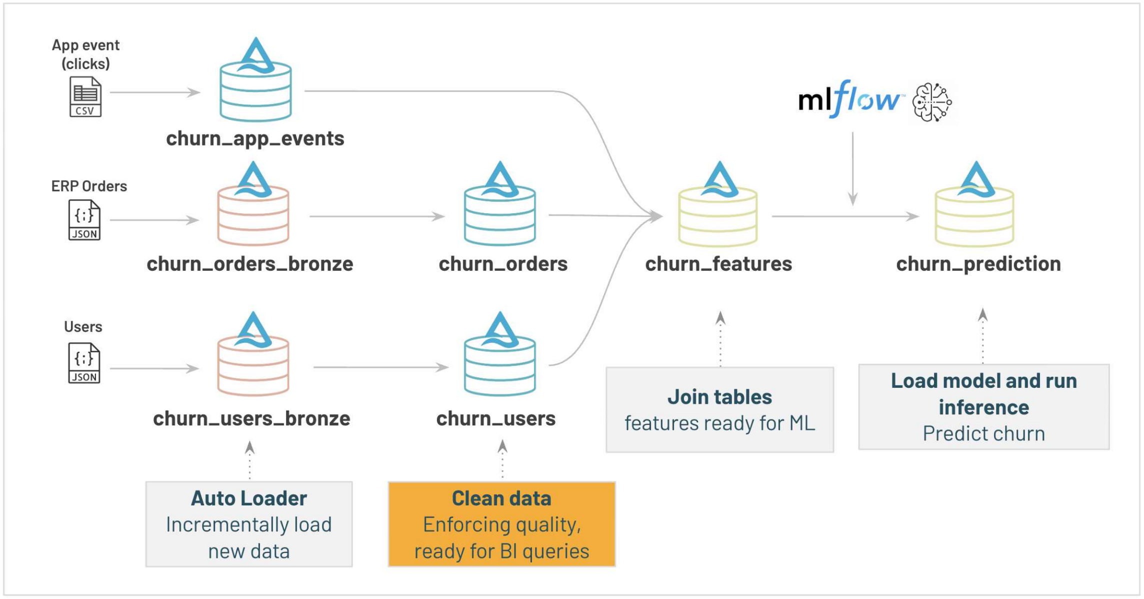
Our user_bronze Delta table is now ready for efficient querying

```
%sql
-- Note the "_rescued_data" column. If we receive wrong data not matching existing schema, it will be stored here
select * from churn_users_bronze;
```

2/ Silver data: anonimized table, date cleaned

We can chain these incremental transformation between tables, consuming only new data.

This can be triggered in near realtime, or in batch fashion, for example as a job running every night to consume daily data.



Silver table for the users data

```
from pyspark.sql.functions import sha1, col, initcap, to_timestamp

(spark.readStream
        .table("churn_users_bronze")
        .withColumnRenamed("id", "user_id")
        .withColumn("email", sha1(col("email")))
        .withColumn("creation_date", to_timestamp(col("creation_date"), "MM-dd-yyyy H:mm:ss"))
        .withColumn("last_activity_date", to_timestamp(col("last_activity_date"), "MM-dd-yyyy HH:mm:ss"))
        .withColumn("firstname", initcap(col("firstname")))
        .withColumn("lastname", initcap(col("lastname")))
        .withColumn("age_group", col("age_group").cast('int'))
        .withColumn("gender", col("gender").cast('int'))
        .drop(col("churn"))
        .drop(col("_rescued_data"))
      .writeStream
        .option("checkpointLocation", f"{deltaTablesDirectory}/checkpoint/users")
        .trigger(once=True)
        .table("churn_users").awaitTermination())
```

```
%sql select * from churn_users;
```

Silver table for the orders data

```
(spark.readStream
        .table("churn_orders_bronze")
        .withColumnRenamed("id", "order_id")
        .withColumn("amount", col("amount").cast('int'))
        .withColumn("item_count", col("item_count").cast('int'))
        .withColumn("creation_date", to_timestamp(col("transaction_date"), "MM-dd-yyyy H:mm:ss"))
        .drop(col("_rescued_data"))
      .writeStream
        .option("checkpointLocation", f"{deltaTablesDirectory}/checkpoint/orders")
        .trigger(once=True)
        .table("churn_orders").awaitTermination())
```

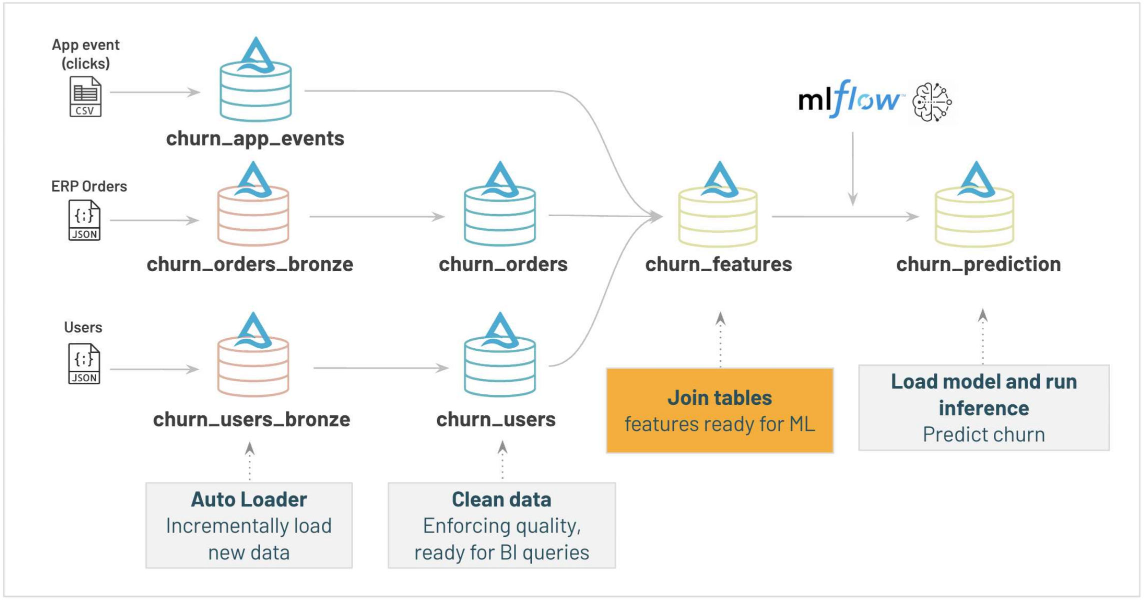
```
%sql select * from churn_orders;
```

3/ Aggregate and join data to create our ML features

We are now ready to create the features required for our churn prediction.

We need to enrich our user dataset with extra information which our model will use to help predicting churn, sucj as:

- last command date
- number of item bought
- number of actions in our website
- device used (ios/iphone)
- ...



Creating a "gold table" to be used by the Machine Learning practitioner

```
spark.sql(
  """
    CREATE OR REPLACE TABLE churn_features AS
      WITH
        churn_orders_stats AS (
          SELECT
            user_id,
            count(*) as order_count,
            sum(amount) as total_amount,
            sum(item_count) as total_item,
            max(creation_date) as last_transaction
          FROM churn_orders
          GROUP BY user_id
        ),
        churn_app_events_stats AS (
          SELECT
            first(platform) as platform,
            user_id,
            count(*) as event_count,
            count(distinct session_id) as session_count,
            max(to_timestamp(date, "MM-dd-yyyy HH:mm:ss")) as last_event
          FROM churn_app_events GROUP BY user_id
        )
        SELECT
          *,
          datediff(now(), creation_date) as days_since_creation,
          datediff(now(), last_activity_date) as days_since_last_activity,
          datediff(now(), last_event) as days_last_event
        FROM churn_users
        INNER JOIN churn_orders_stats using (user_id)
        INNER JOIN churn_app_events_stats using (user_id)
  """
)

display(spark.table("churn_features"))
```

Exploiting the benefits of Delta

(a) Simplifing operations with transactional DELETE/UPDATE/MERGE operations

Traditional Data Lakes struggle to run even simple DML operations. Using Databricks and Delta Lake, your data is stored on your blob storage with transactional capabilities. You can issue DML operation on Petabyte of data without having to worry about concurrent operations.

We just realised we have to delete users created before 2016-01-01 for compliance; let's

```
%sql DELETE FROM churn_users where creation_date < '2016-01-01T03:38:55.000+0000';
```

Delta Lake keeps the history of the table operations

```
%sql describe history churn_users;
```

We can leverage the history to travel back in time, restore or clone a table, enable CDC

```
%sql
 -- the following also works with AS OF TIMESTAMP "yyyy-MM-dd HH:mm:ss"
select * from churn_users version as of 1 ;
```

```
%sql
-- You made the DELETE by mistake ? You can easily restore the table at a given version / date:
RESTORE TABLE churn_users TO VERSION AS OF 1

-- Or clone it (SHALLOW provides zero copy clone):
-- CREATE TABLE user_gold_clone SHALLOW|DEEP CLONE user_gold VERSION AS OF 1

-- Turn on CDC to capture insert/update/delete operation:
-- ALTER TABLE myDeltaTable SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
```

(b) Optimizing for performance

Ensuring that all our tables are storage-optimized

```
%sql
ALTER TABLE churn_users    SET TBLPROPERTIES (delta.autooptimize.optimizewrite = TRUE, delta.autooptimize.autocompact = TRUE );
ALTER TABLE churn_orders   SET TBLPROPERTIES (delta.autooptimize.optimizewrite = TRUE, delta.autooptimize.autocompact = TRUE );
ALTER TABLE churn_features SET TBLPROPERTIES (delta.autooptimize.optimizewrite = TRUE, delta.autooptimize.autocompact = TRUE );
```

Our user table will be queried mostly by 3 fields; let's optimize the table for that!

```
%sql
OPTIMIZE churn_users ZORDER BY user_id, firstname, lastname
```

Next up

- Exploring, discovering, and governing data access with Unity Catalog ($./01.1%20-%20Unity%20Catalog)
- Simplifying Data Pipelines with Delta Live Tables ($./01.2%20-%20Delta%20Live%20Tables)