Build an Open Data Lakehouse with Spark, Delta and Trino on S3

Data lakes represent the first step towards gaining insights from ever-growing data. In many cases, it is the first place collected data lands on to the data system. Data lake creates two challenges:

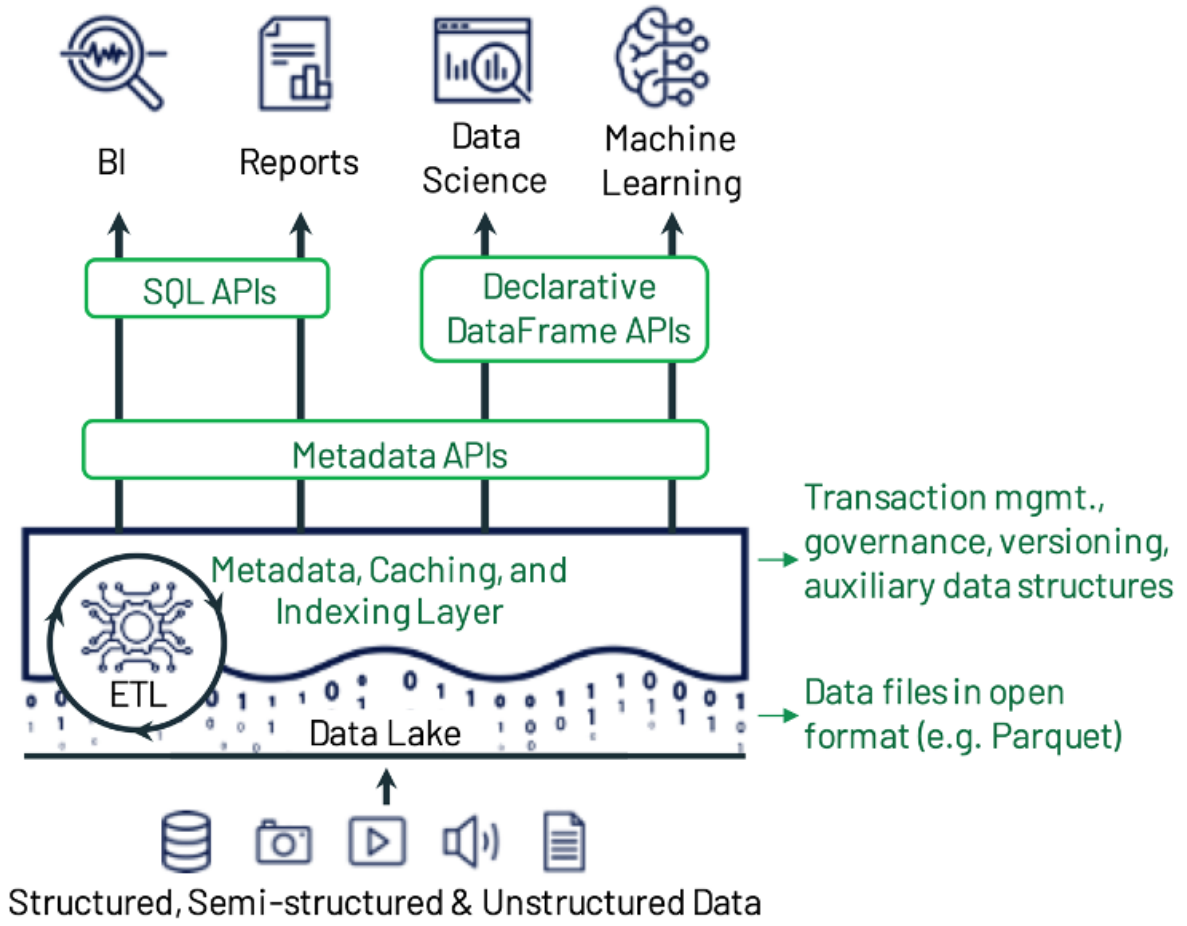
- Data quality and governance. Everything is just a file/object in a data lake.
- Performance. Limited query optimisation, such as metadata, indexing, etc.

Data warehouses, on the other hand, often time is the final destination of analytical data. Data in a data warehouse may come from the data lake, or directly from the sources. Data warehouse challenge includes:

- Limited support for unstructured data.
- Performance for machine learning. SQL over ODBC/JDBC is not efficient for ML. ML needs direct access to data in an open format.

Enter data lakehouse. A data lakehouse system tries to solve these challenges by combing the strengths of data lakes and warehouses. Key features of a data lakehouse include:

- Designed for both SQL and machine learning workloads.
- ACID transactions.
- Partition evolution.
- Schema evolution.
- Time-travel query.
- (near) Data warehouse performance.



Example data lakehouse system design from the paper by Michael Armbrust1, Ali Ghodsi1,2, Reynold Xin1, Matei Zaharia

Implementing a data lakehouse

Key components in a data lakehouse implementation include:

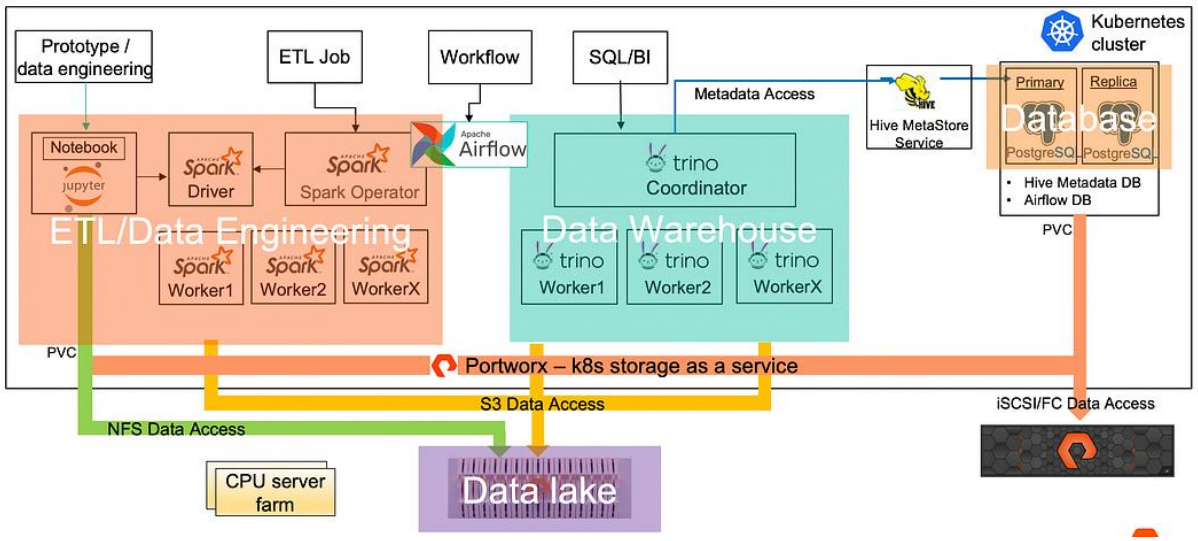
- Leverage existing data lake and open data format. Table data is typically stored as Parquet or ORC files in HDFS or S3 data lake.
- Add metadata layers for data management. Popular open-source choices include Delta Lake, Apache Iceberg, and Apache Hudi. They typically store metadata in the same data lake as JSON or Avro format. And have a catalog pointer to the current metadata.
- Analytics engine that supports the data lakehouse spec. Apache Spark, Trino, and Dremio are among the most popular ones.

The below, I will explain my process of implementing a simple data lakehouse system using open-source software. This implementation can run with cloud data lakes like Amazon S3, or on-premise ones such as Pure Storage FlashBlade S3.

Getting ready for data lakehouse

To implement a data lakehouse system, we need to be familiar and ready for data lake and warehouse first. In my case, I have already set up my FlashBlade S3 data lake, Spark, and Trino data warehouse.

Press enter or click to view image in full size



Adding data lakehouse metadata management

One thing that is missing in my previous setup is the metadata management layer for data lakehouse. I choose Delta Lake for this, because it is easy to get started, has less dependency on Hadoop and Hive, and its documentation is good.

Delta Lake is implemented as Java libraries. Only four jars are required to add Delta Lake to an existing Spark environment: delta-core, delta-storage, antlr4-runtime, and jackson-core-asl. Download these jars from Maven repo, and add them under the $SPARK_HOME/jars directory. Because I run Spark on Kubernetes, I add these jars into my Spark container image.

Next, I add the following configurations to my Spark session, so that Spark will use the Delta catalog and its SQL extension.

spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog

That's all we need for Delta Lake and Spark integration.

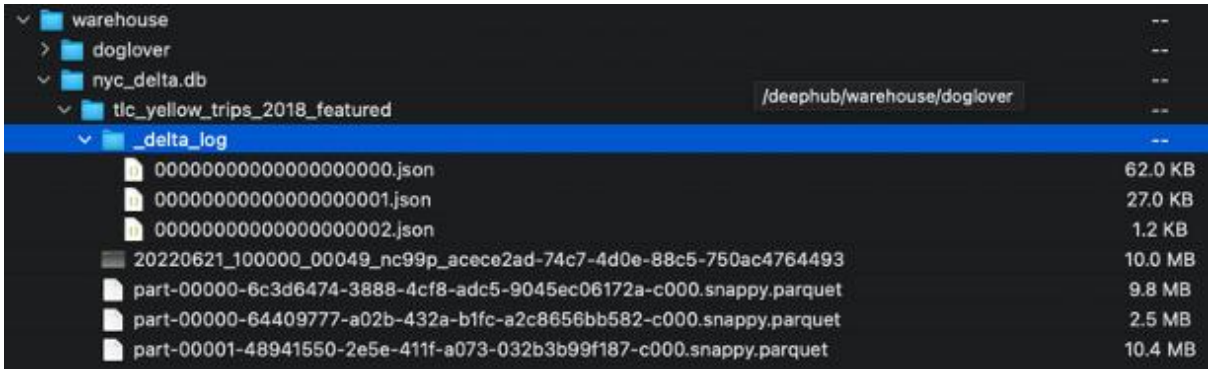
Data lakehouse with Delta Lake and Spark

Now let's demonstrate some data lakehouse features with Delta Lake and Spark.

Save a Spark dataframe in Delta format:

df.write.format('delta').mode('overwrite').save('s3a://warehouse/nyc_delta.db/tlc_yellow_trips_2018_featured')

This writes both the Parquet data files and Delta Lake metadata (JSON) in the same FlashBlade S3 bucket.



Delta Lake data and metadata in FlashBlade S3

To read back Delta Lake data into Spark dataframes:

```
df_delta =
spark.read.format('delta').load('s3a://warehouse/nyc_delta.db/tlc_yellow_trips_2018_featured')
```

Delta Lake provides programmatic APIs for conditional update, delete, and merge (upsert) data into tables.

```
from delta.tables import *
from pyspark.sql.functions import *delta_table = DeltaTable.forPath(spark,
's3a://warehouse/nyc_delta.db/tlc_yellow_trips_2018_featured')# Update fare_amount by adding 1
to it
delta_table.update(
   condition = expr('trip_duration == 15 and trip_distance == 2.7'),
   set = {'fare_amount': expr('fare_amount + 1')})
```

Transaction is not easy, if possible, in a data lake, but is built-in with a data lakehouse. Transactions create snapshots. I can query previous snapshots of my Delta table by using time travel queries. If I want to access the data that has been overwritten, I can query a snapshot of the table before I overwrote the first set of data using the versionAsOf option.

```
# Read older versions of data using time travel queries
df_features = spark.read.format('delta').option('versionAsOf',
0).load('s3a://warehouse/nyc_delta.db/tlc_yellow_trips_2018_featured')
```

I can also retrieve a Delta table history like this:

```
from delta.tables import *features_table = DeltaTable.forPath(spark,
's3a://warehouse/nyc_delta.db/tlc_yellow_trips_2018_featured')
full_history = features_table.history()
full_history.show()
```

```
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+--------------------+------------+--------------------+
|version|          timestamp|userId|userName|operation| operationParameters| job|notebook|clusterId|readVersion|isolationLevel|isBlindAppend|    operationMetrics|userMetadata|         engineInfo|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+--------------------+------------+--------------------+
|      3|2022-06-21 09:53:22|  null|    null|   DELETE|{predicate -> ["(...|null|    null|     null|          2|  Serializable|        false|{numRemovedFiles ...|        null|Apache-Spark/3.2....|
|      2|2022-06-21 09:50:02|  null|    null|    MERGE|{predicate -> ((f...|null|    null|     null|          1|  Serializable|        false|{numTargetRowsCop...|        null|Apache-Spark/3.2....|
|      1|2022-06-21 09:48:23|  null|    null|   UPDATE|{predicate -> ((t...|null|    null|     null|          0|  Serializable|        false|{numRemovedFiles ...|        null|Apache-Spark/3.2....|
|      0|2022-06-21 09:43:52|  null|    null|    WRITE|{mode -> Overwrit...|null|    null|     null|       null|  Serializable|        false|{numFiles -> 78, ...|        null|Apache-Spark/3.2....|
+-------+-------------------+------+--------+---------+--------------------+----+--------+---------+-----------+--------------+-------------+--------------------+------------+--------------------+
```

Delta table versions

SQL on data lakehouse

While Spark is great for general ETL with its Dataframe APIs, for advanced analytics and business intelligence, SQL is preferred. In the below, I add data lakehouse support to my existing Trino data warehouse using the Trino Delta Lake Connector.

To configure the Delta Lake Connector, add the following to the catalog/delta.properties file, and restart Trino.

```
hive.metastore.uri=thrift://metastore:9083
hive.s3.endpoint=http://192.168.170.22
hive.s3.ssl.enabled=false
hive.s3.path-style-access=true
delta.enable-non-concurrent-writes=true
```

With this, Trino can understand the Delta spec, query and update the above Spark Delta format output.

Connect to Trino Delta catalog:

```
trino-cli --server trino:8080 --catalog delta
```

Create a Delta table in Trino, and query the data.

```
USE nyc_delta;

CREATE TABLE IF NOT EXISTS delta.nyc_delta.tlc_yellow_trips_2018_featured (
   fare_amount DOUBLE,
   trip_distance DOUBLE,
   trip_duration INTEGER,
   passenger_count INTEGER,
   pickup_year INTEGER,
   pickup_month INTEGER,
   pickup_day_of_week INTEGER,
   pickup_hour_of_day INTEGER
)
WITH (LOCATION = 's3a://warehouse/nyc_delta.db/tlc_yellow_trips_2018_featured')
;

SELECT * FROM tlc_yellow_trips_2018_featured LIMIT 10;
```

Update Trino Delta table(require delta.enable-non-concurrent-writes set to true):

update tlc_yellow_trips_2018_featured set fare_amount=20.0 where trip_duration = 15 and trip_distance=3.74;

Note that updates, or transactions in general, is not supported in classic Trino tables on S3. To update even just one row in a table, we need to re-populate the entire partition or table. With transaction support in a Trino Delta table this becomes much easy as shown above. Transactions are first stored in the _delta_log directory, and later merged back to the base Parquet files in the backend.

An example Delta log including a Delete operation

I also want to stress that, although ACID transaction is normally a built-in feature for data lakehouse system, this is not meant to be used for general OLTP purpose. Transactions in a data lakehouse system should be infrequent.

As of the time of this blog, Trino's Delta Lake connector supports common Delta/Trino SQL type mapping, and common queries including Select, Update and so on. Advanced Delta features, such as time travel queries, are not supported yet. For those features, use the APIs from the Delta Lake library.