Data Lakes on ACID

BouvetGeekOne 2022-10-20

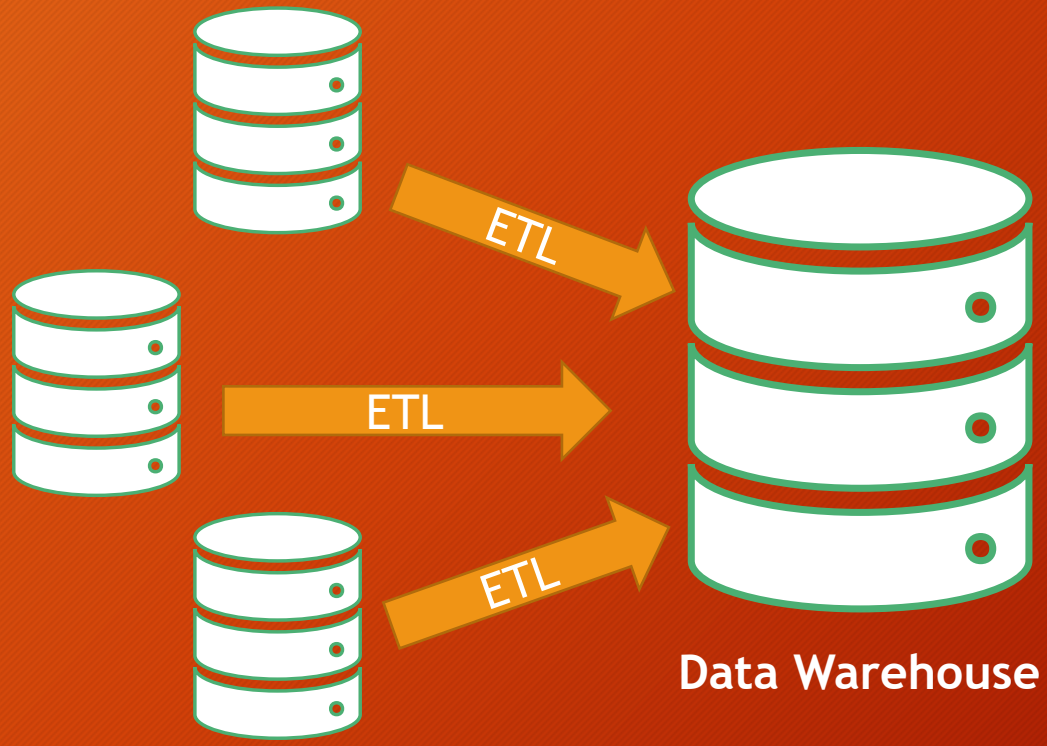
Gareth Western

@gareth
gareth.western@bouvet.no

Data Lake vs Data Warehouse

Data warehouse handles analytical queries

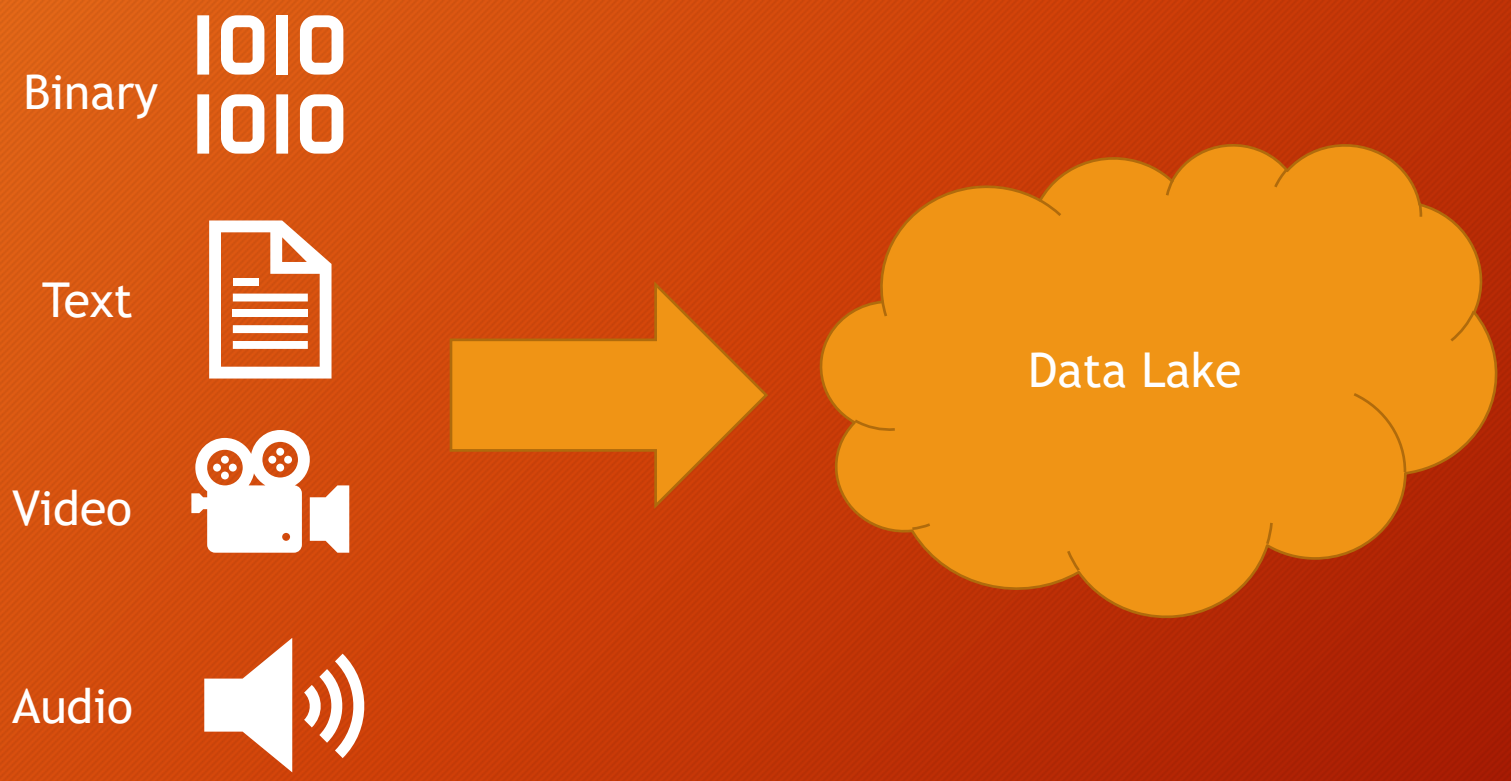
Online Transaction **Processing (OLTP)**



Data Warehouse

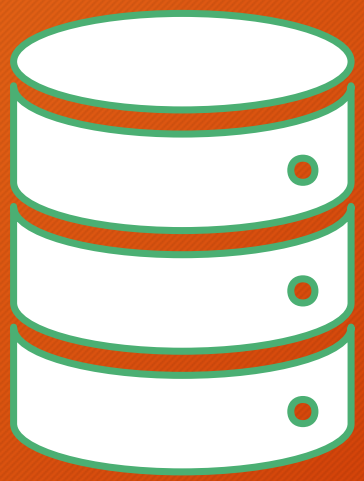
Online Analytical Processing (OLAP)

Data lake stores semi/un-structured data



(This data is usually text-based)

Data lakehouse is a combination of the two



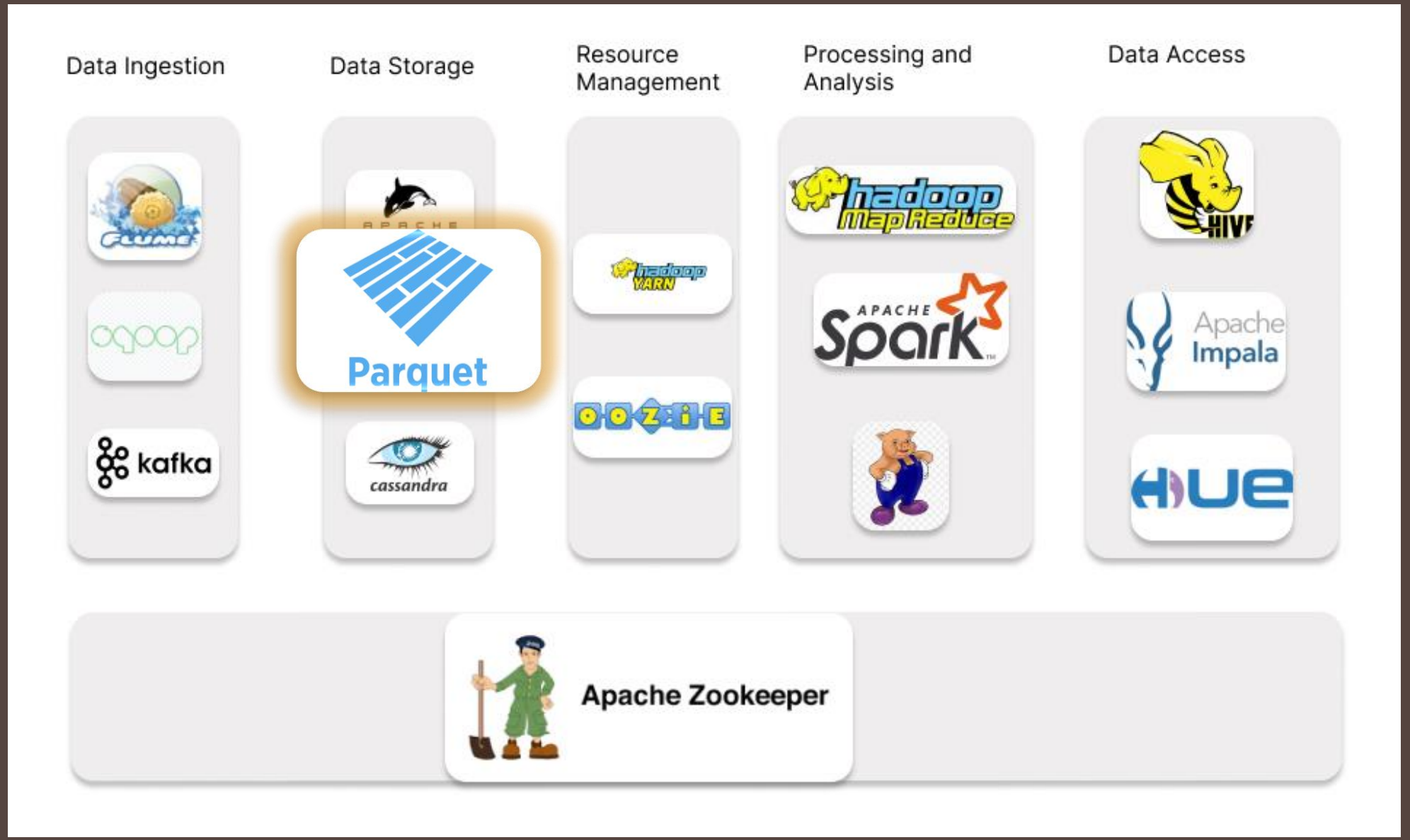
Data Warehouse

+

The “lake” poses some challenges

1. Inconsistent formats
2. Difficult to query
3. Inefficient storage
4. Updating existing data
5. Recovering from bad writes
6. Keeping track of changes

The rise of Big Data led to new technologies



Source: [analyticsvidhya]

A bit of Parquet

- Inspired by the Dremel paper
- Binary, column-oriented format
- Well-suited for analytical queries
- Includes metadata
- Supports multiple encoding and compression schemes

Dremel: Interactive Analysis of Web-Scale Datasets

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer,
Shiva Shivakumar, Matt Tolton, Theo Vassilakis
Google, Inc.
{melnik,andrey,jlong,gromer,shiva,mtolton,theov}@google.com

ABSTRACT

Dremel is a scalable, interactive ad-hoc query system for analysis of read-only nested data. By combining multi-level execution trees and columnar data layout, it is capable of running aggregation queries over trillion-row tables in seconds. The system scales to thousands of CPUs and petabytes of data, and has thousands of users at Google. In this paper, we describe the architecture and implementation of Dremel, and explain how it complements MapReduce-based computing. We present a novel columnar storage representation for nested records and discuss experiments on few-thousand node instances of the system.

1. INTRODUCTION

Large-scale analytical data processing has become widespread in web companies and across industries, not least due to low-cost storage that enabled collecting vast amounts of business-critical data. Putting this data at the fingertips of analysts and engineers has grown increasingly important; interactive response times often make a qualitative difference in data exploration, monitoring, online customer support, rapid prototyping, debugging of data pipelines, and other tasks.

Performing interactive data analysis at scale demands a high degree of parallelism. For example, reading one terabyte of compressed data in one second using today's commodity disks would require tens of thousands of disks. Similarly, CPU-intensive exchanged by distributed systems, structured documents, etc. lend themselves naturally to a *nested* representation. Normalizing and recombining such data at web scale is usually prohibitive. A nested data model underlies most of structured data processing at Google [21] and reportedly at other major web companies.

This paper describes a system called Dremel[1] that supports interactive analysis of very large datasets over shared clusters of commodity machines. Unlike traditional databases, it is capable of operating on *in situ* nested data. *In situ* refers to the ability to access data 'in place', e.g., in a distributed file system (like GFS [14]) or another storage layer (e.g., Bigtable [8]). Dremel can execute many queries over such data that would ordinarily require a sequence of MapReduce (MR [12]) jobs, but at a fraction of the execution time. Dremel is not intended as a replacement for MR and is often used in conjunction with it to analyze outputs of MR pipelines or rapidly prototype larger computations.

Dremel has been in production since 2006 and has thousands of users within Google. Multiple instances of Dremel are deployed in the company, ranging from tens to thousands of nodes. Examples of using the system include:

- Analysis of crawled web documents.
- Tracking install data for applications on Android Market.
- Crash reporting for Google products.
- OCR results from Google Books.

Row-based works well for reading entire rows

Name	Age	Favourite Colour
Anders	40	Blue
Bjørn	40	Red
Oda	31	Red
Ada	29	Green

- Horizontal partitioning
- OLTP ☑
- OLAP ✗

Anders	40	Blue	Bjørn	40	Red	Oda	31	Red	…

Column-based is better for analytics

Name	Age	Favourite Colour
Anders	40	Blue
Bjørn	40	Red
Oda	31	Red
Ada	29	Green

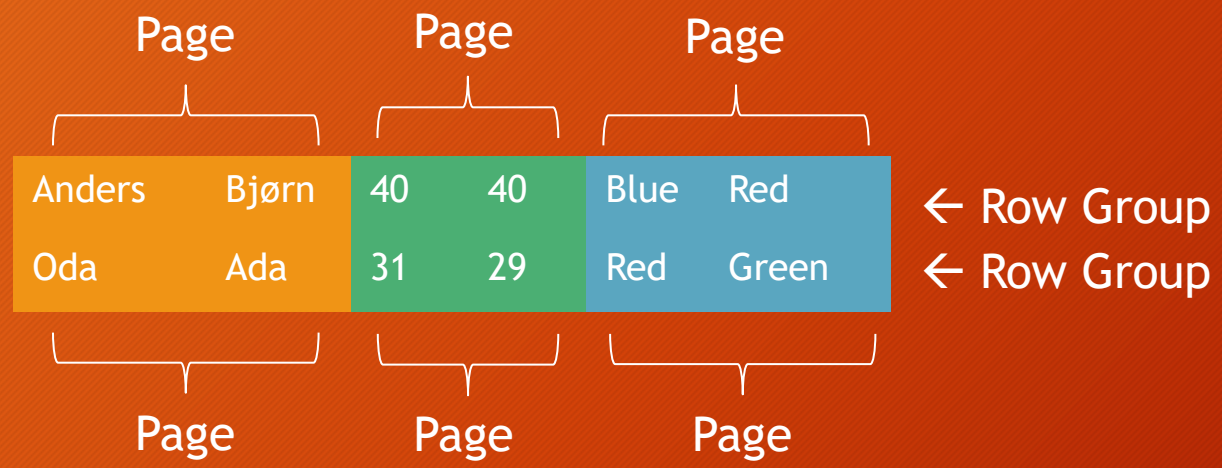
- Vertical partitioning
- OLTP ✗
- OLAP ☑

Anders	Bjørn	Oda	Ada	40	40	31	29	Blue	…

Hybrid approach brings the best of both

Name	Age	Favourite Colour
Anders	40	Blue
Bjørn	40	Red
Oda	31	Red
Ada	29	Green

- Vertical & Horizontal partitioning
- Divide the data into *Row Groups* and *Pages*



Metadata and statistics optimizes queries

Name	Age	Favourite Colour
Anders	40	Blue
Bjørn	40	Red
Oda	31	Red
Ada	29	Green

Schema
Name: String
Age: Integer
Fav. Colour: String

Stats
Min age: 29
Max age: 40

Dictionary encoding helps to reduce filesize

- Build a dictionary of column values
  - Anders → 0
  - Bjørn → 1
  - Oda → 2
  - Ada → 3

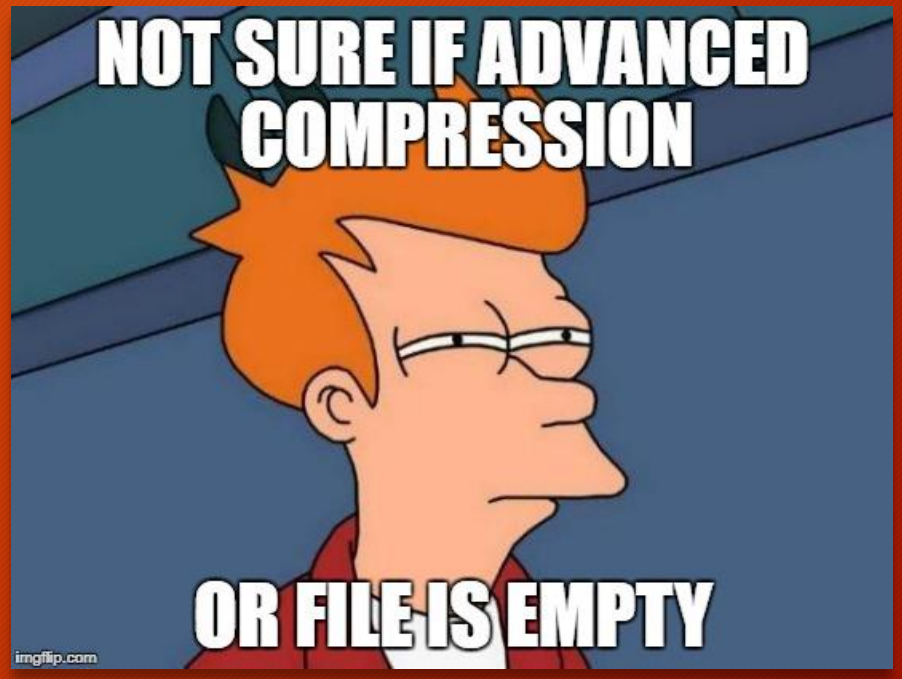
  - Blue → 0
  - Red → 1
  - Green → 2

Name	Age	Favourite Colour
Anders	40	Blue
Bjørn	40	Red
Oda	31	Red
Ada	29	Green

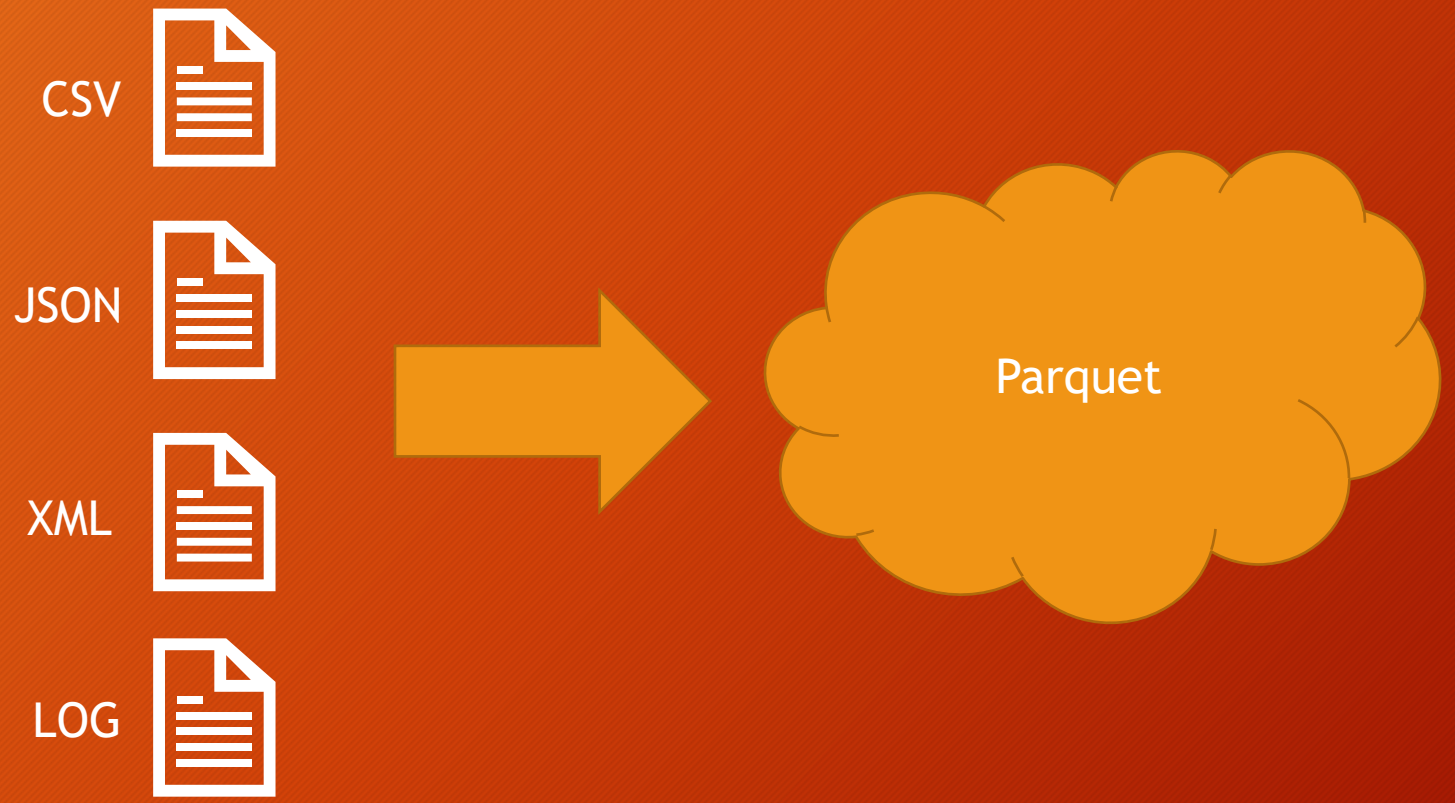
Name	Age	Favourite Colour
0	40	0
1	40	1
2	31	1
3	29	2

Compression reduces filesize even more

- Snappy – well-balanced compression / speed
- LZO – similar to snappy
- GZIP – better compression, but slower



Standardising on parquet improves the lake



Using parquet has solved *some* problems

1. ~~Inconsistent formats~~
2. ~~Difficult to query~~
3. ~~Inefficient storage~~
4. Updating existing data
5. Recovering from bad writes
6. Keeping track of changes

Adding ACID to the data lake

- **Atomicity**
  - Transactions must be treated as a single “unit” which either fail or succeed completely.
- **Consistency**
  - Transactions must ensure that the database goes from one valid state to another.
- **Isolation**
  - Concurrent transactions must be handled as if the transactions were executed sequentially.
- **Durability**
  - One committed, a transaction must stay committed.

Introducing the *Delta* Lakehouse

Three Delta Features to Win Friends and Influence People

- Transaction Log
- Optimize / Z-Order
- Time Travel

Source: pintrest

1. The Transaction Log (AKA Delta Log)

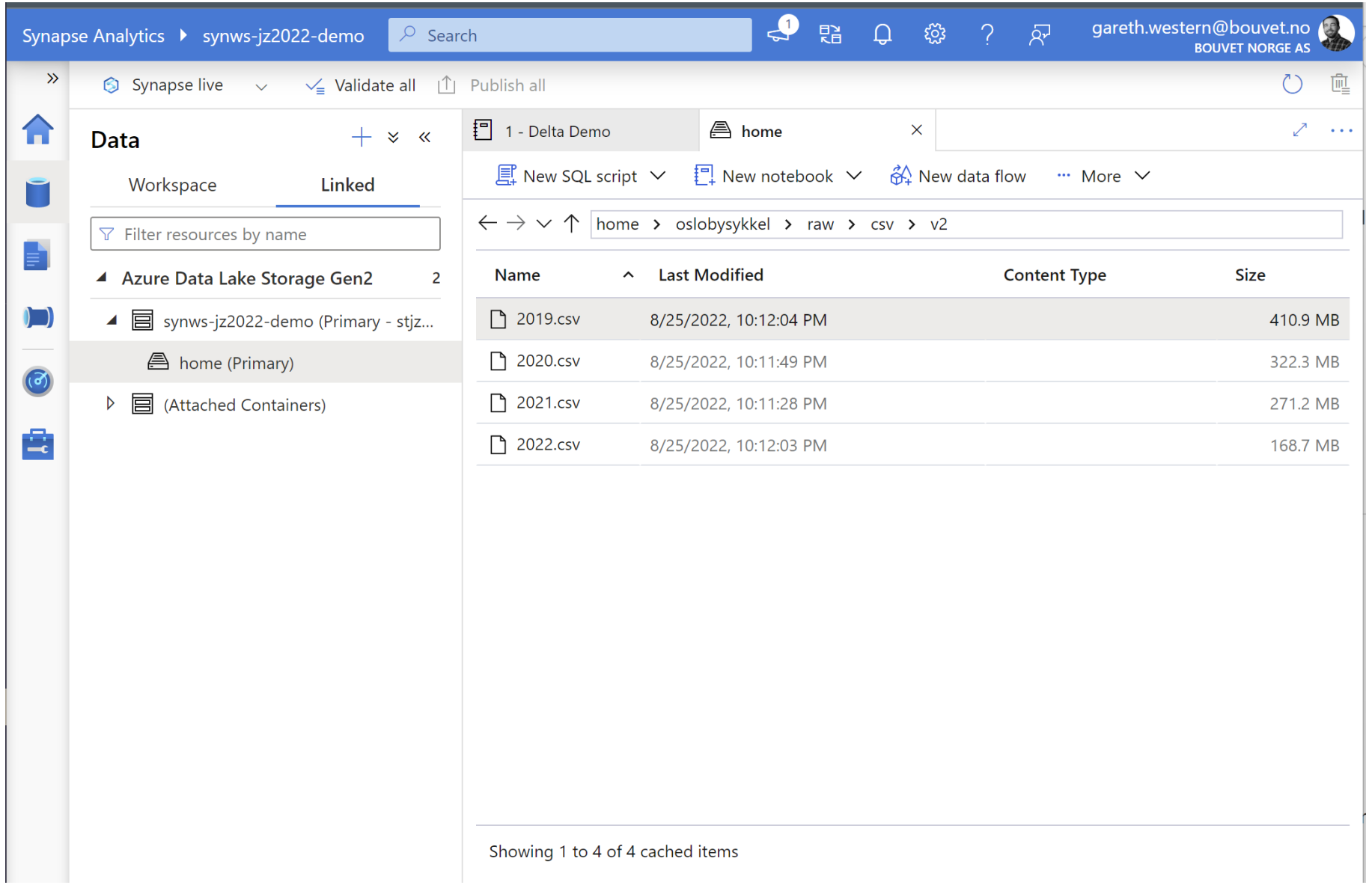
Azure Synapse Analytics

About Oslo City Bike

The most efficient way to get around Oslo is on a bike. City bikes are primarily used for short rides and as a supplement to public transport. About 100,000 people in Oslo use city bikes, and in 2018 over 2.7 million bike trips were made.

The city bike scheme in Oslo is a collaboration between the city of Oslo and Clear Channel Norway AS. The municipality makes public advertising space available and gets a city bike space in return. The scheme is financed by subscriptions, advertisements on the stations, and sponsorship.

Demo – Oslo City Bike Trips


Synapse Analytics
synws-jz2022-demo
Search
gareth.western@bouvet.no
BOUVET NORGE AS
Synapse live
Validate all
Publish all
Data
Workspace
Linked
Filter resources by name
Azure Data Lake Storage Gen2
2
synws-jz2022-demo (Primary - stjz...
home (Primary)
(Attached Containers)
1 - Delta Demo
home
New SQL script
New notebook
New data flow
More
home
oslobysykkel
raw
csv
v2
Name
Last Modified
Content Type
Size
2019.csv
8/25/2022, 10:12:04 PM
410.9 MB
2020.csv
8/25/2022, 10:11:49 PM
322.3 MB
2021.csv
8/25/2022, 10:11:28 PM
271.2 MB
2022.csv
8/25/2022, 10:12:03 PM
168.7 MB
Showing 1 to 4 of 4 cached items

API V1 - 2016 - 2018

Here we read trip data from the "v1" API, covering the years from 2016 - 2018.

The schema here is very simple, with only 4 fields.

+ Code + Markdown

```
1   v1Data = (spark.read
2           .format("csv")
3           .option("header", "true")
4           .option("inferSchema", "true")
5           .load(f"{data_path}/raw/csv/v1/*.csv")
6   )
7
8   print(f"Total number of records: {v1Data.count()}")
9   print(v1Data.printSchema())
10  display(v1Data.limit(5))
```

✓ - Command executed in 47 sec 528 ms on 9:47:46 AM, 9/08/22

```
Total number of records: 7747817
root
 |-- Start station: integer (nullable = true)
 |-- Start time: string (nullable = true)
 |-- End station: integer (nullable = true)
 |-- End time: string (nullable = true)
```

None

View Table Chart ⟼ Export results

Start station	Start time	End station	End time
159	2016-06-19 14:02:57 +0200	229	2016-06-19 14:23:29 +0200
253	2016-06-19 14:03:06 +0200	222	2016-06-19 16:15:48 +0200
210	2016-06-19 14:03:06 +0200	246	2016-06-19 14:48:24 +0200

4. Join the v1 data with the dimension table for mapping both start_station_id and end_station_id

```
import pyspark.sql.functions as F

v1Data = (v1Data
    .withColumnRenamed("Start station", "start_station_id")
    .withColumnRenamed("End station", "end_station_id")
    .withColumnRenamed("Start time", "started_at")
    .withColumnRenamed("End time", "ended_at")
    .withColumn("start_year", F.year("started_at"))
    .withColumn("start_month", F.month("started_at"))
    .withColumn("api_version", F.lit(1))
)
display(v1Data.limit(5))
```

[4] ✓ - Command executed in 18 sec 763 ms on 9:48:07 AM, 9/08/22

View: Table | Chart Export results

started_at	ended_at	start_year	start_month	api_version	start_station_id	end_station_id
2016-10-01 06:00:08 +0200	2016-10-01 06:09:47 +0200	2016	10	1	783	782
2016-10-01 06:00:41 +0200	2016-10-01 06:11:07 +0200	2016	10	1	625	825
2016-10-01 06:01:02 +0200	2016-10-01 06:15:52 +0200	2016	10	1	510	442
2016-10-01 06:01:20 +0200	2016-10-01 06:15:04 +0200	2016	10	1	514	464
2016-10-01 06:02:16 +0200	2016-10-01 06:04:29 +0200	2016	10	1	506	464

Write to Delta Table

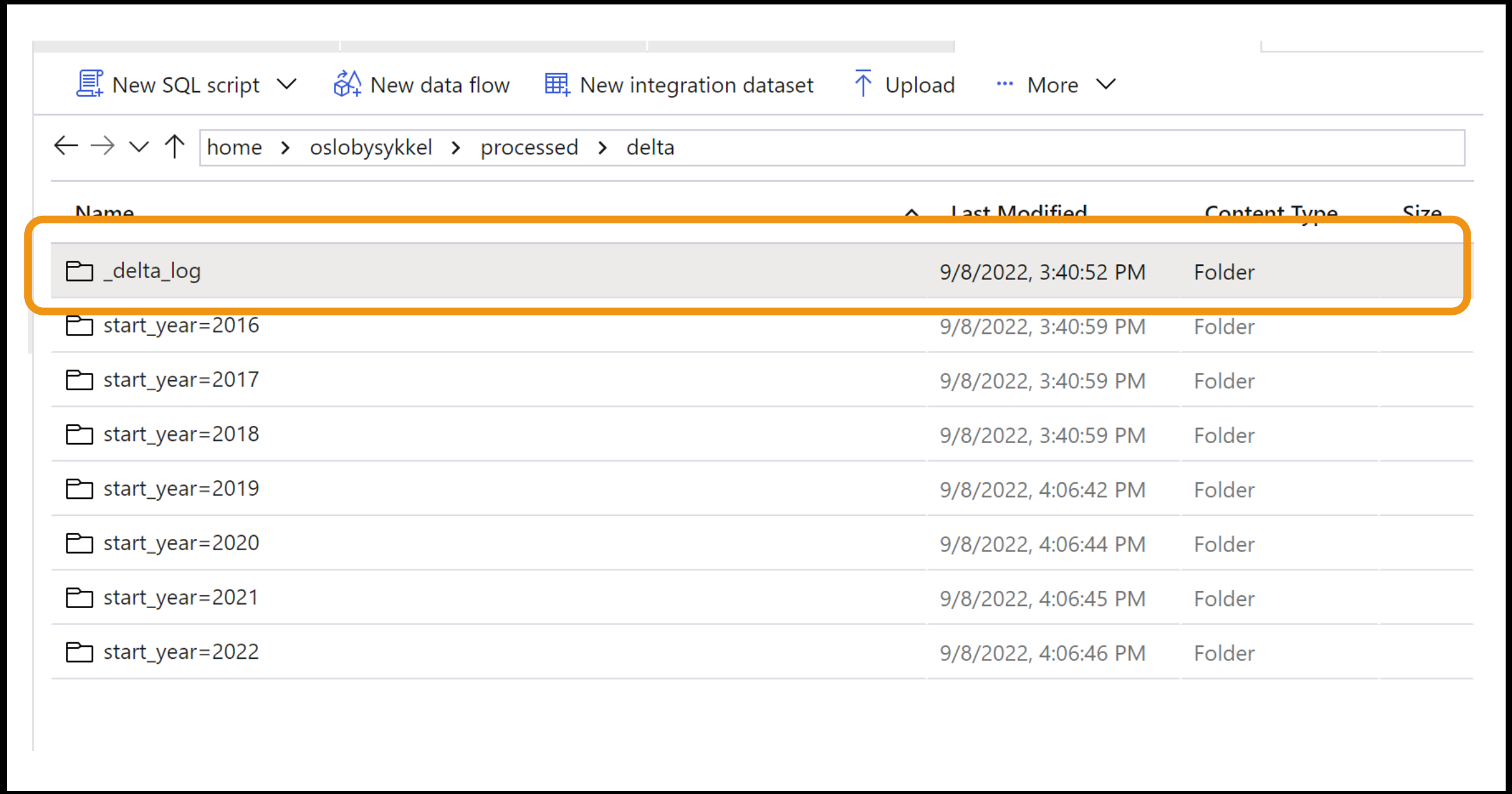
Now we'll write the data to a Delta table!

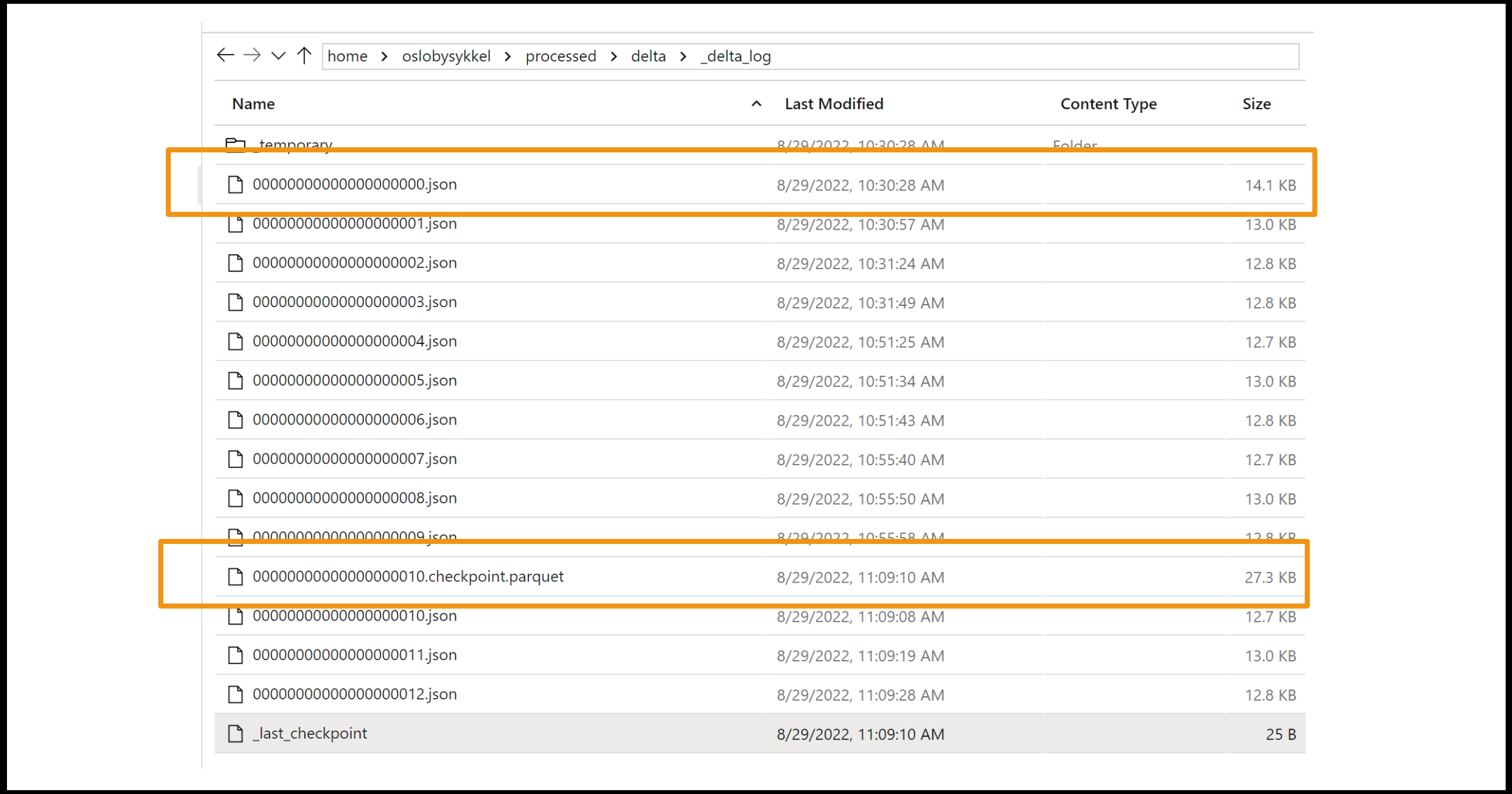
Note that we can tell Spark to automatically partition the data by `start_year` and `start_month`.

[5]

```
(v1Data.write
    .mode("append")
    .partitionBy("start_year", "start_month")
    .format("delta")
    .save(f"{data_path}/processed/delta")
)
```

✓ - Command executed in 22 sec 920 ms on 9:48:30 AM, 9/08/22


New SQL script
New data flow
New integration dataset
Upload
More
home
oslobysykkel
processed
delta
Name
Last Modified
Content Type
Size
_delta_log
9/8/2022, 3:40:52 PM
Folder
start_year=2016
9/8/2022, 3:40:59 PM
Folder
start_year=2017
9/8/2022, 3:40:59 PM
Folder
start_year=2018
9/8/2022, 3:40:59 PM
Folder
start_year=2019
9/8/2022, 4:06:42 PM
Folder
start_year=2020
9/8/2022, 4:06:44 PM
Folder
start_year=2021
9/8/2022, 4:06:45 PM
Folder
start_year=2022
9/8/2022, 4:06:46 PM
Folder

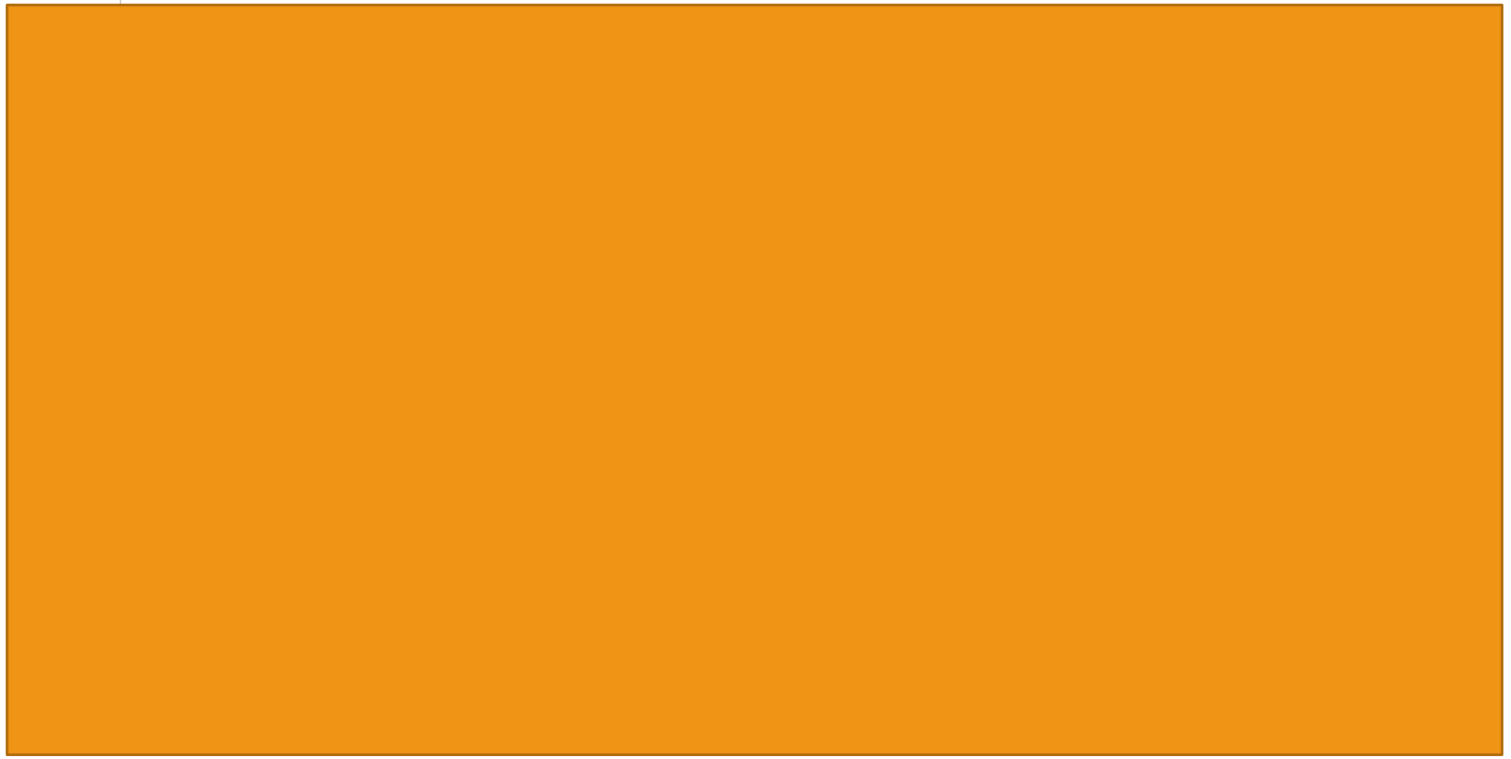

home › oslobysykkel › processed › delta › _delta_log
Name
Last Modified
Content Type
Size
_temporary
8/29/2022, 10:30:28 AM
Folder
00000000000000000000.json
8/29/2022, 10:30:28 AM
14.1 KB
00000000000000000001.json
8/29/2022, 10:30:57 AM
13.0 KB
00000000000000000002.json
8/29/2022, 10:31:24 AM
12.8 KB
00000000000000000003.json
8/29/2022, 10:31:49 AM
12.8 KB
00000000000000000004.json
8/29/2022, 10:51:25 AM
12.7 KB
00000000000000000005.json
8/29/2022, 10:51:34 AM
13.0 KB
00000000000000000006.json
8/29/2022, 10:51:43 AM
12.8 KB
00000000000000000007.json
8/29/2022, 10:55:40 AM
12.7 KB
00000000000000000008.json
8/29/2022, 10:55:50 AM
13.0 KB
00000000000000000009.json
00000000000000000010.checkpoint.parquet
8/29/2022, 11:09:10 AM
27.3 KB
00000000000000000010.json
8/29/2022, 11:09:08 AM
12.7 KB
00000000000000000011.json
8/29/2022, 11:09:19 AM
13.0 KB
00000000000000000012.json
8/29/2022, 11:09:28 AM
12.8 KB
_last_checkpoint
8/29/2022, 11:09:10 AM
25 B

Single transaction


```
{
    "protocol": {…
    }
} {
    "metaData": {
        "id": "c2c87577-83a0-4db5-8281-906301a4a5b4",
        "format": {…
        },
        "schemaString": "{\"type\":\"struct\",\"fields\":[{\"name\":\"started_at\",\"type\":\"string\",\"nullab
        "partitionColumns": [],
        "configuration": {},
        "createdTime": 1661761823592
    }
} {
    "add": {
        "path": "part-00000-4c88d73b-452f-4c01-b50f-7f506a7f90a3-c000.snappy.parquet",
        "partitionValues": {},
        "size": 8461535,
        "modificationTime": 1661761828700,
        "dataChange": true,
        "stats": "{\"numRecords\":285149,\"minValues\":{\"started_at\":\"2019-04-02 22:18:47.926000+00:00\",\"e
    }
} {
    "remove": {…
    }
} {
    "commitInfo": {
        "timestamp": 1661761828758,
        "operation": "WRITE",
        "operationParameters": {…
        },
        "isolationLevel": "Serializable",
        "isBlindAppend": true,
        "operationMetrics": {
            "numFiles": "8",
            "numOutputRows": "2245247",
            "numOutputBytes": "66540451"
        },
    }
}
```

Parquet metadata


```
{
    "protocol": {…
    }
} {
    "metaData": {
        "id": "c2c87577-83a0-4db5-8281-906301a4a5b4",
        "format": {…
        },
        "schemaString": "{\"type\":\"struct\",\"fields\":[{\"name\":\"started_at\",\"type\":\"string\",\"nullab
        "partitionColumns": [],
        "configuration": {},
        "createdTime": 1661761823592
    }
} {
```

List of actions

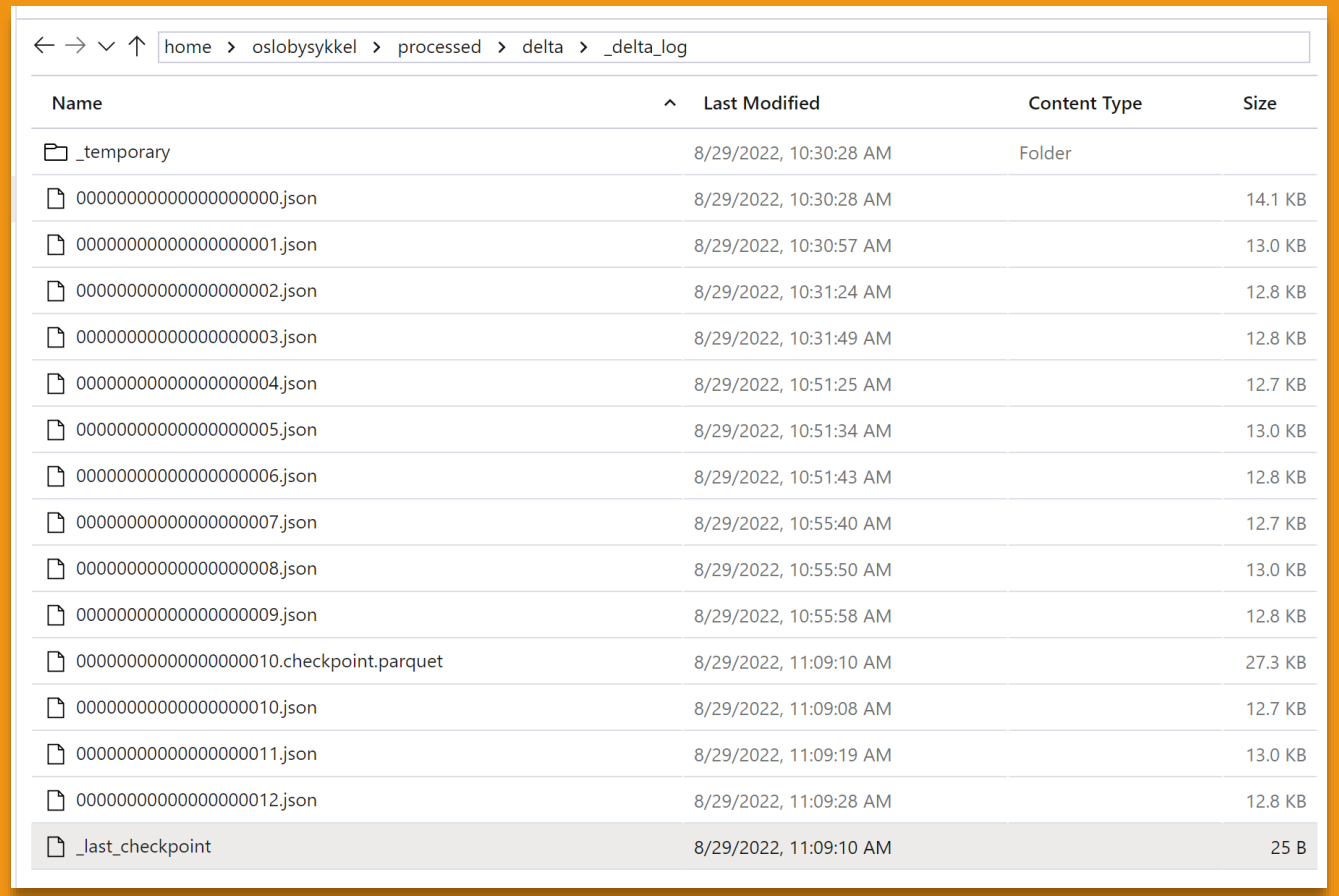
```
18  } {
19      "add": {
20          "path": "part-00000-4c88d73b-452f-4c01-b50f-7f506a7f90a3-c000.snappy.parquet",
21          "partitionValues": {},
22          "size": 8461535,
23          "modificationTime": 1661761828700,
24          "dataChange": true,
25          "stats": "{\"numRecords\":285149,\"minValues\":{\"started_at\":\"2019-04-02 22:18:47.926000+00:00\",\"e
26      }
27  } {
28      "remove": {…
30      }
31  } {
```

Commit metadata


```
} {
    "commitInfo": {
        "timestamp": 1661761828758,
        "operation": "WRITE",
        "operationParameters": {…
        },
        "isolationLevel": "Serializable",
        "isBlindAppend": true,
        "operationMetrics": {
            "numFiles": "8",
            "numOutputRows": "2245247",
            "numOutputBytes": "66540451"
        },
    }
}
```

Mutual Exclusion and *Optimistic Concurrency Control* ensures “Isolation”

Recording all transactions to disk ensures “Durability”



2. Optimize / Z-Order to optimize data files

Name	Age
Anders	..
Bjørn	...

Name	Age
Oda	..
Ada	..

Name	Age
Gareth	..
Hanne	..

Name	Age
Dag	..
Claudio	..

OPTIMIZE
Z-ORDER BY NAME

Name	Age
Ada	..
Anders	..
Bjørn	..
Claudio	..

Name	Age
Dag	..
Gareth	..
Hanne	..
Oda	..

SELECT COUNT(*)
WHERE name = "Gareth"

```python
from delta.tables import DeltaTable

deltaTable = DeltaTable.forPath(spark, f"{data_path}/processed/delta")

deltaTable.optimize().executeZOrderBy("start_station_id")

# If you have a large amount of data and only want to optimize a subset of it, you can specify an optional partition predicate using `where`
deltaTable.optimize().where("start_year='2022'").executeZOrderBy("start_station_id")
```

✓ - Command executed in 155 ms on 10:02:58 AM, 9/08/22

```
1   display(deltaTable.history())
```

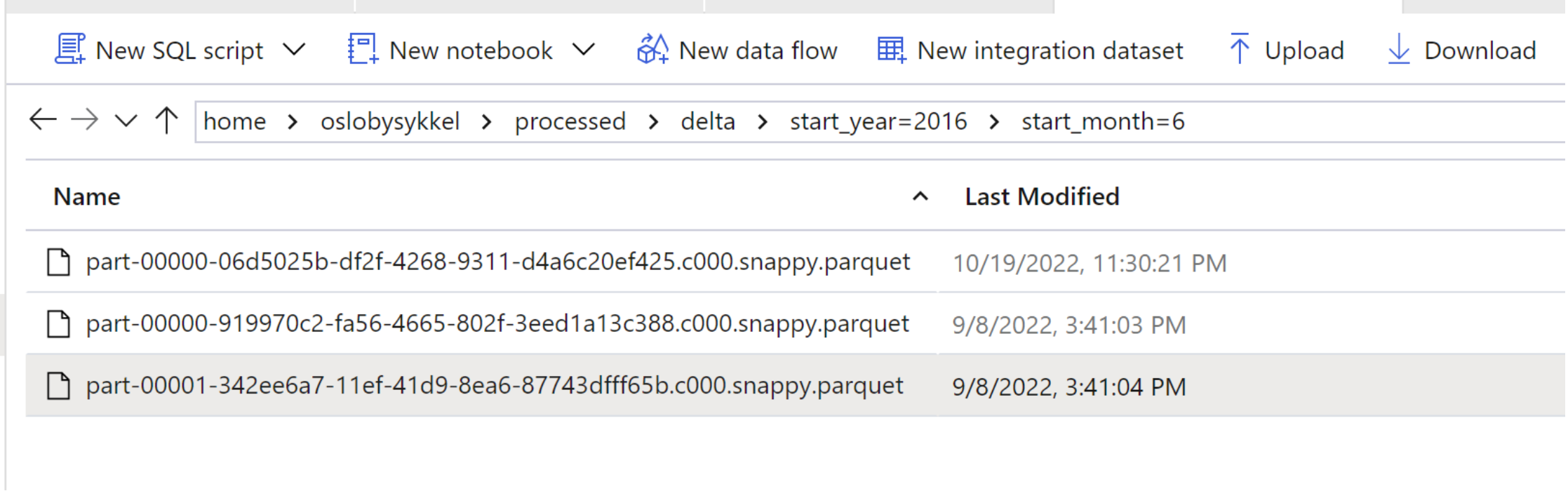
✓ 3 sec - Command executed in 2 sec 940 ms by gareth.western on 11:38:06 PM, 10/19/22

ecution Succeeded **Spark** 2 executors 8 cores

View in monitoring Open Spark UI

View Table Chart Export results

version	timestamp	userId	userName	operation	operationParameters	job
313	2022-10-19 21:34:38.275	undefined	undefined	OPTIMIZE	▶ "{"predicate":"[\"true\"]","zOrderBy	undefined
312	2022-09-08 14:45:02.402	undefined	undefined	STREAMING UPDATE	▶ "{"outputMode":"Append","queryI	undefined
311	2022-09-08 14:44:52.52	undefined	undefined	STREAMING UPDATE	▶ "{"outputMode":"Append","queryI	undefined
310	2022-09-08 14:44:41.941	undefined	undefined	STREAMING UPDATE	▶ "{"outputMode":"Append","queryI	undefined


New SQL script
New notebook
New data flow
New integration dataset
Upload
Download
home
oslobysykkel
processed
delta
start_year=2016
start_month=6
Name
Last Modified
part-00000-06d5025b-df2f-4268-9311-d4a6c20ef425.c000.snappy.parquet
10/19/2022, 11:30:21 PM
part-00000-919970c2-fa56-4665-802f-3eed1a13c388.c000.snappy.parquet
9/8/2022, 3:41:03 PM
part-00001-342ee6a7-11ef-41d9-8ea6-87743dfff65b.c000.snappy.parquet
9/8/2022, 3:41:04 PM

3. Time Travel

Source: Slate.com

```
1  display(deltaTable.history())
```

✓ 3 sec - Command executed in 2 sec 940 ms by gareth.western on 11:38:06 PM, 10/19/22

ecution Succeeded **Spark** 2 executors 8 cores

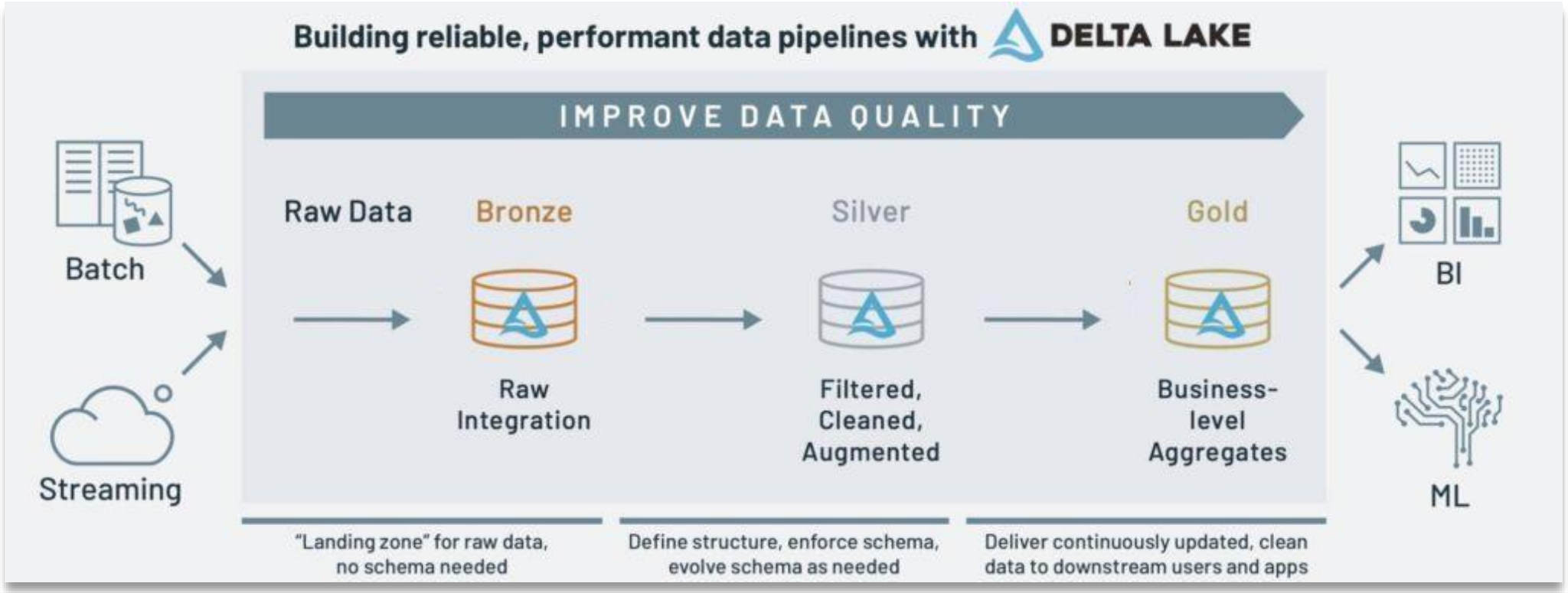
View in monitoring Open Spark UI

View Table Chart Export results

version	timestamp	userId	userName	operation	operationParameters	job
313	2022-10-19 21:34:38.275	undefined	undefined	OPTIMIZE	▶ "{"predicate":"[\"true\"]","zOrderBy	undefined
312	2022-09-08 14:45:02.402	undefined	undefined	STREAMING UPDATE	▶ "{"outputMode":"Append","queryI	undefined
311	2022-09-08 14:44:52.52	undefined	undefined	STREAMING UPDATE	▶ "{"outputMode":"Append","queryI	undefined
310	2022-09-08 14:44:41.941	undefined	undefined	STREAMING UPDATE	▶ "{"outputMode":"Append","queryI	undefined

```
1   deltaTable.restoreToVersion(0) # restore table to oldest version
2   # or
3   deltaTable.restoreToTimestamp('2019-02-14') # restore to a specific timestamp
```

✓ - Command executed in 14 sec 692 ms on 10:03:33 AM, 9/08/22



Medallion Architecture

Apache Spark™

docs | source code

Spark

This connector allows Apache Spark™ to read from and write to Delta Lake.

Apache Flink

docs | source code

Flink standalone

This connector allows Apache Flink to write to Delta Lake.

PrestoDB

docs | source code

PrestoDB standalone

This connector allows PrestoDB to read from Delta Lake.

Trino

docs | source code

Trino

This connector allows Trino to read from and write to Delta Lake.

Delta Standalone

docs | source code

Scala Java standalone

This library allows Scala and Java-based projects (including Apache Flink, Apache Hive, Apache Beam, and PrestoDB) to read from and write to Delta Lake.

Apache Hive

docs | source code

Hive standalone

This connector allows Apache Hive to read from Delta Lake.

Delta Rust API

docs | source code

Rust Python Ruby

This library allows Rust (with Python and Ruby bindings) low level access to Delta tables and is intended to be used with data processing frameworks like datafusion, ballista, rust-dataframe, vega, etc.

SQL Delta Import

docs | source code

SQL JDBC

This utility is for importing data from a JDBC source into a Delta Lake table.

Kafka Delta Ingest

docs | source code

Kafka Rust

This project builds a highly efficient daemon for streaming data through Apache Kafka into Delta Lake.

Azure Synapse

docs

Azure Synapse

The current version of Delta Lake included with Azure Synapse has language support for Scala, PySpark, and .NET.

MinIO

docs | source code

MinIO community

This connector allows you to read and write from Delta tables on MinIO storage.

Athena

docs | source code

Athena AWS manifest

This utility allows Athena to read from Delta Lake using a manifest file.

Snowflake (Beta)

docs

Snowflake

This preview allows Snowflake to read from Delta Lake via an external table.

DataHub

source code

DataHub community

This connector allows DataHub to extract Delta Lake metadata.

Redshift

docs | source code

Redshift AWS manifest

This utility allows AWS Redshift to read from Delta Lake using a manifest file.

Ceph

source code

Ceph community

This connector allows you to read and write from Delta tables on Ceph storage.

Datastream Connector

source code

GCS Datastream badal.io community

As Datastream streams changes to files to Google Cloud Storage, this connector streams these files and writes the changes to Delta Lake.

dlt | SparkR

docs | source code

SparkR community

This package allows SparkR to read from and write to Delta Lake.

Power BI

docs | source code

PowerBI community

This connector allows Power BI to read from Delta Lake.

Java | Delta Sharing (Beta)

source code

Java Delta Sharing community

This connector allows as Java client to read from Delta Sharing endpoint.

Apache Beam (Beta)

docs | source code

Beam standalone community

This connector allows Apache Beam to read from Delta Lake.

Power BI | Delta Sharing (Beta)

docs

PowerBI Delta Sharing

This connector allows Power BI to read from Delta Sharing endpoint.

Databricks

docs

Databricks Azure GCP AWS

Delta Lake is included within Databricks allowing it to read from and write to Delta Lake.

Athena Query Federation (Beta)

docs | source code

AWS Athena standalone community

This connector allows AWS Athena to read from Delta Lake.

node.js | Delta Sharing (Beta)

source code

node.js Delta Sharing community

This connector allows node.js to read from Delta Sharing endpoint.

Starburst

docs

Starburst Azure GCP AWS

The Starburst Delta Lake connector is an extended version of the Trino/Delta Lake connector with configuration and usage identical.

Delta Integrations

Delta Sharing – an open protocol for secure data sharing

DuckDB – SQLite for OLAP

read_delta.py > ...

```python
import duckdb
from deltalake import DeltaTable

trips = DeltaTable("./delta").to_pyarrow_dataset()

conn = duckdb.connect()

df = conn.execute('''
SELECT start_station_id, end_station_id, start_year, start_month, COUNT(*) AS cnt
FROM trips
GROUP BY start_station_id, end_station_id, start_year, start_month
ORDER BY cnt DESC
''').df()

print(df)
```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL AZURE JUPYTER

	start_station_id	end_station_id	start_year	start_month	cnt
0	606	504	2017	6	1235
1	504	606	2018	5	1234
2	606	504	2017	8	1124
3	504	606	2017	8	1115
4	606	504	2017	7	1107
...	...	...	...	...	...
1481070	381	575	2016	4	1
1481071	782	532	2016	4	1
1481072	553	506	2016	4	1
1481073	545	560	2016	4	1
1481074	534	545	2016	4	1

[1481075 rows x 5 columns]

Mission accomplished!

1. ~~Inconsistent formats~~
2. ~~Difficult to query~~
3. ~~Inefficient storage~~
4. ~~Updating existing data~~
5. ~~Recovering from bad writes~~
6. ~~Keeping track of changes~~

- Delta Lake = Parquet++

Learn More

- Notebook:
  - https://github.com/gdubya/data-lakes-on-acid
- ACID for your data lake
  - Delta: https://delta.io
  - Iceberg: https://iceberg.apache.org
  - Hudi: https://hudi.apache.org
- Hadoop
  - Home: https://hadoop.apache.org
  - Parquet: https://parquet.apache.org
  - Spark: https://spark.apache.org
- DuckDB: https://duckdb.org