Cloud Data Lakes

Amir H. Payberah
payberah@kth.se
2022-10-05

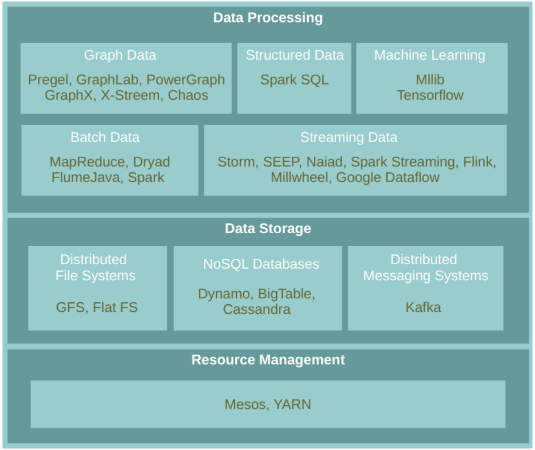
The Course Web Page

https://id2221kth.github.io

The Questions-Answers Page

https://tinyurl.com/bdenpwc5

Where Are We?



Data Processing
Graph Data
Pregel, GraphLab, PowerGraph
GraphX, X-Streem, Chaos
Structured Data
Spark SQL
Machine Learning
Mllib
Tensorflow
Batch Data
MapReduce, Dryad
FlumeJava, Spark
Streaming Data
Storm, SEEP, Naiad, Spark Streaming, Flink,
Millwheel, Google Dataflow
Data Storage
Distributed
File Systems
GFS, Flat FS
NoSQL Databases
Dynamo, BigTable,
Cassandra
Distributed
Messaging Systems
Kafka
Resource Management
Mesos, YARN

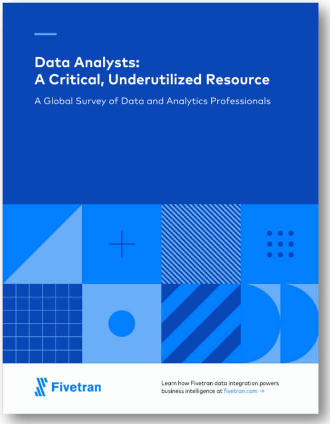
What Are The Challenges?

The Biggest Challenges With Data Today

- Data quality
- Staleness
- Data volume
- Scale

Fivetran Data Analyst Survey

- 60% reported data quality as top challenge.
- 86% of analysts had to use stale data, with 41% using data that is > 2 months old.
- 90% regularly had unreliable data sources over the last 12 months

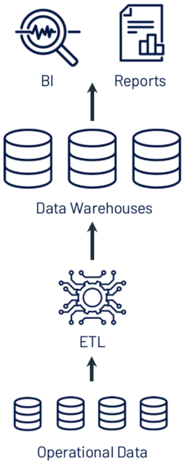


Getting high-quality, timely data is hard!

The Evolution of Data Management

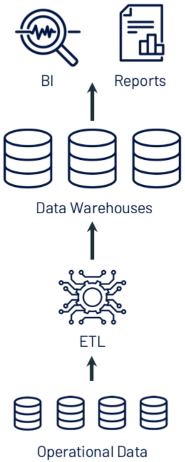
Data Warehouses (1980s)

- ETL (Extract, Transform, Load) data directly from operational database systems.



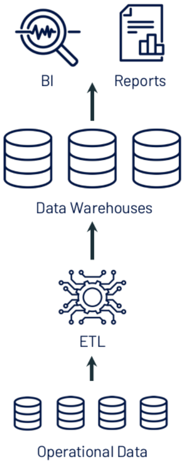
Data Warehouses (1980s)

- ETL (Extract, Transform, Load) data directly from operational database systems.
- Purpose-built for SQL analytics and BI: schemas, indexes, caching, etc.



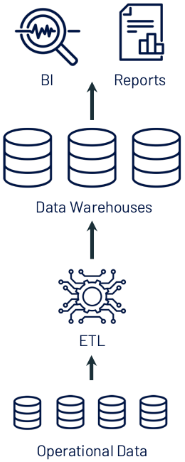
Data Warehouses (1980s)

- ETL (Extract, Transform, Load) data directly from operational database systems.
- Purpose-built for SQL analytics and BI: schemas, indexes, caching, etc.
- Powerful management features such as ACID transactions and time travel



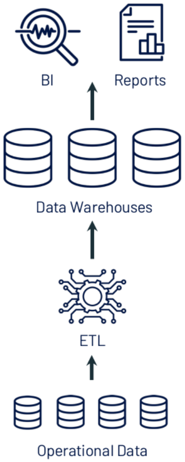
Data Warehouses - Problems (2010s)

- Could not support rapidly growing unstructured and semi-structured data: time series, logs, images, documents, etc.



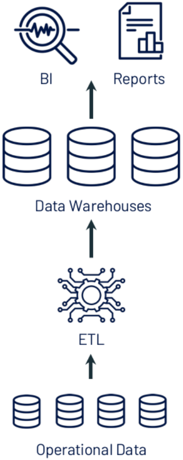
Data Warehouses - Problems (2010s)

- Could not support rapidly growing unstructured and semi-structured data: time series, logs, images, documents, etc.
- High cost to store large datasets.



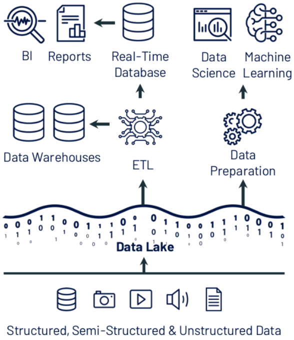
Data Warehouses - Problems (2010s)

- Could not support rapidly growing unstructured and semi-structured data: time series, logs, images, documents, etc.
- High cost to store large datasets.
- No support for data science and ML.



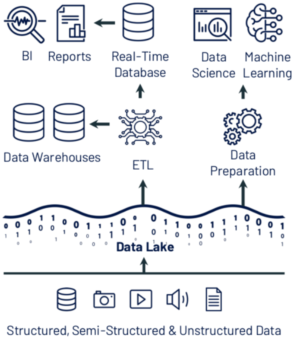
Data Lakes (2010s)

- Low-cost storage to hold all raw data, e.g., Amazon S3, and HDFS.



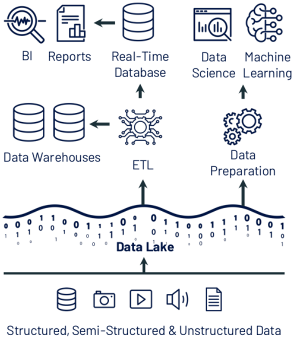
Data Lakes (2010s)

- Low-cost storage to hold all raw data, e.g., Amazon S3, and HDFS.
- ETL jobs then load specific data into warehouses, possibly for further ELT.



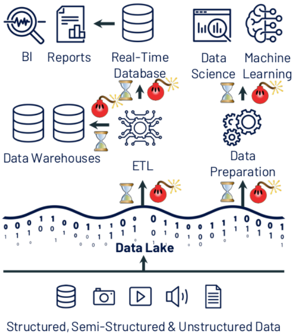
Data Lakes (2010s)

- Low-cost storage to hold all raw data, e.g., Amazon S3, and HDFS.
- ETL jobs then load specific data into warehouses, possibly for further ELT.
- Directly readable in ML libraries (e.g., TensorFlow and PyTorch) due to open file format.



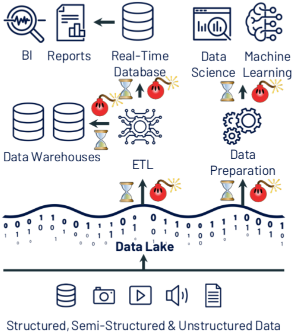
Data Lakes - Problems (Todays)

- Cheap to store all the data, but system architecture is much more complex!



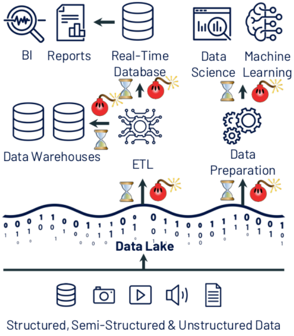
Data Lakes - Problems (Todays)

- Cheap to store all the data, but system architecture is much more complex!
- Data reliability suffers:
  - Multiple storage systems with different semantics, SQL dialects, etc.
  - Extra ETL steps that can go wrong.



Data Lakes - Problems (Todays)

- Cheap to store all the data, but system architecture is much more complex!
- Data reliability suffers:
  - Multiple storage systems with different semantics, SQL dialects, etc.
  - Extra ETL steps that can go wrong.
- Timeliness suffers and high cost:
  - Extra ETL steps before data is available in data warehouses.
  - Continuous ETL, duplicated storage



Data Lake vs. Data Warehouse

- Data Lake stores all data irrespective of the source and its structure whereas Data Warehouse stores data in quantitative metrics with their attributes.

Data Lake vs. Data Warehouse

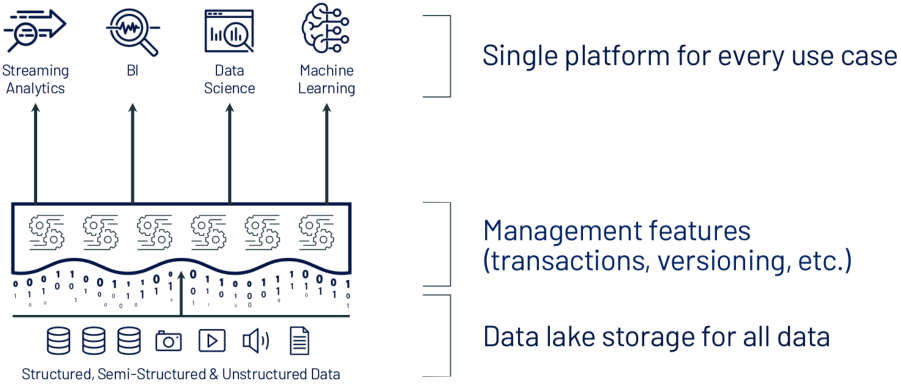
- Data Lake stores all data irrespective of the source and its structure whereas Data Warehouse stores data in quantitative metrics with their attributes.
- Data Lake defines the schema after data is stored whereas Data Warehouse defines the schema before data is stored.

Data Lake vs. Data Warehouse

- Data Lake stores all data irrespective of the source and its structure whereas Data Warehouse stores data in quantitative metrics with their attributes.
- Data Lake defines the schema after data is stored whereas Data Warehouse defines the schema before data is stored.
- Data Lake uses the ELT process while the Data Warehouse uses ETL process.

Lakehouse

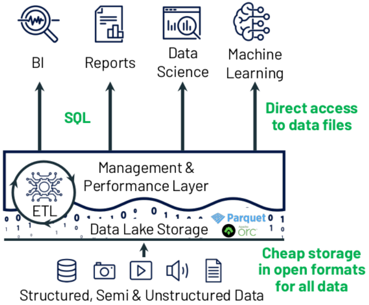
Lakehouse Vision



- Lakehouse systems combine the benefits of Data Warehouses and Data Lakes while simplifying enterprise data architectures.

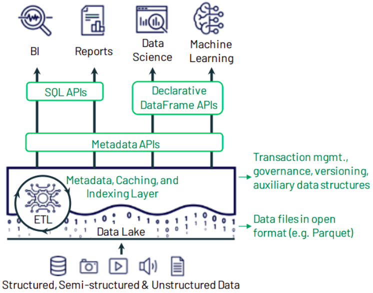
Lakehouse Systems

- Implement Data Warehouse management and performance features on top of directly-accessible data in open formats.



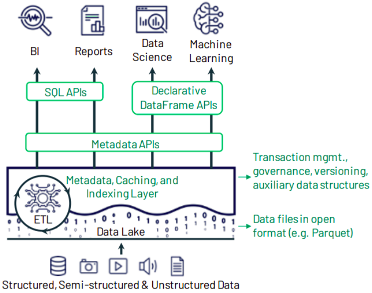
Key Technologies Enabling Lakehouse

- Metadata layers for Data Lakes



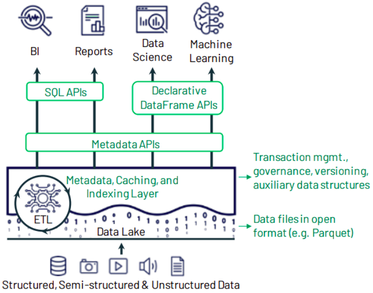
Key Technologies Enabling Lakehouse

- Metadata layers for Data Lakes
- New query engine designs



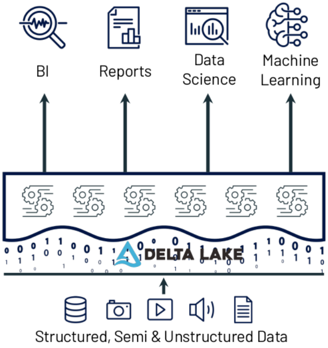
Key Technologies Enabling Lakehouse

- Metadata layers for Data Lakes
- New query engine designs
- Declarative access for data science and ML



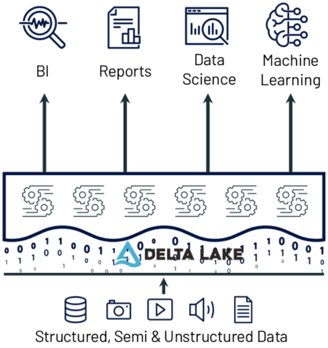
Metadata Layers for Data Lakes

- Add transactions, versioning, and more ...



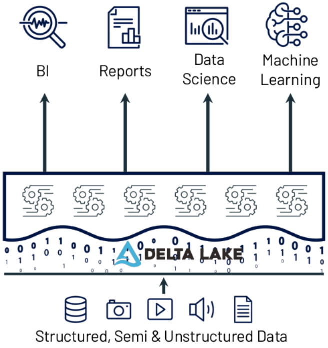
Metadata Layers for Data Lakes

- Add transactions, versioning, and more ...
- Track which files are part of a table version to offer rich management features like transactions.



Metadata Layers for Data Lakes

- Add transactions, versioning, and more ...
- Track which files are part of a table version to offer rich management features like transactions.
- Implemented in multiple systems, such as Delta Lake.



New Query Engine Designs

- Great SQL performance on Data Lake storage systems and file formats.

New Query Engine Designs

- Great SQL performance on Data Lake storage systems and file formats.
- Directly-accessible file storage optimizations can enable high SQL performance:

New Query Engine Designs

- Great SQL performance on Data Lake storage systems and file formats.
- Directly-accessible file storage optimizations can enable high SQL performance:
  - Caching hot data in RAM/SSD

New Query Engine Designs

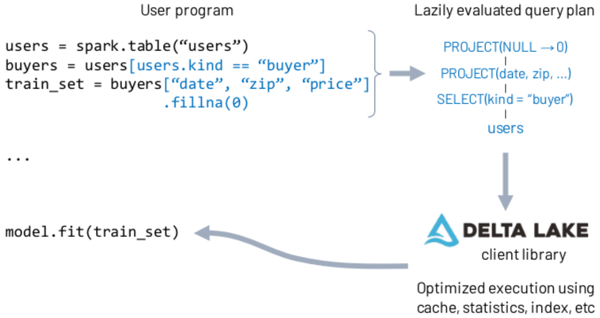
- Great SQL performance on Data Lake storage systems and file formats.
- Directly-accessible file storage optimizations can enable high SQL performance:
  - Caching hot data in RAM/SSD
  - Data layout within files to cluster co-accessed data

New Query Engine Designs

- Great SQL performance on Data Lake storage systems and file formats.
- Directly-accessible file storage optimizations can enable high SQL performance:
  - Caching hot data in RAM/SSD
  - Data layout within files to cluster co-accessed data
  - Auxiliary data structures like statistics and indexes

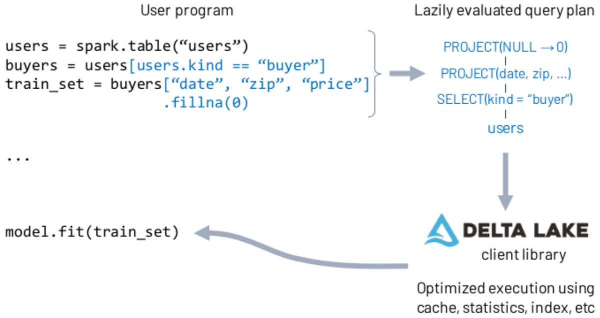
Declarative Access for Data Science and ML

- New declarative interfaces for I/O enable further optimization.



Declarative Access for Data Science and ML

- New declarative interfaces for I/O enable further optimization.
- Example: Spark DataFrame API compiles to relational algebra.



DELTA LAKE

Delta Lake

- Delta Lake is an open source storage layer that brings reliability to Data Lakes.

Delta Lake

- Delta Lake is an open source storage layer that brings reliability to Data Lakes.
- Provides ACID transactions.

Delta Lake

- Delta Lake is an open source storage layer that brings reliability to Data Lakes.
- Provides ACID transactions.
- Provides scalable metadata handling.

Delta Lake

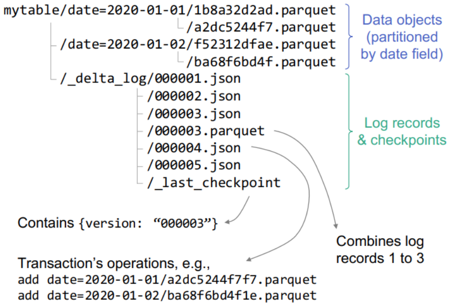
- Delta Lake is an open source storage layer that brings reliability to Data Lakes.
- Provides ACID transactions.
- Provides scalable metadata handling.
- Provides time travel and versioning.

Delta Lake

- Delta Lake is an open source storage layer that brings reliability to Data Lakes.
- Provides ACID transactions.
- Provides scalable metadata handling.
- Provides time travel and versioning.
- Unifies streaming and batch data processing.

Delta Lake Table

- Delta Lake Table is a directory (e.g., `mytable`) that holds data objects and a log of transaction operations.



DeltaLog

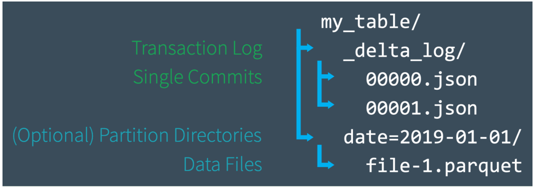
- DeltaLog is a transaction log that tracks all changes that users make to the table.

DeltaLog

- DeltaLog is a transaction log that tracks all changes that users make to the table.
- Delta Lake uses the DeltaLog for many features including ACID transactions, scalable metadata handling, time travel, etc.

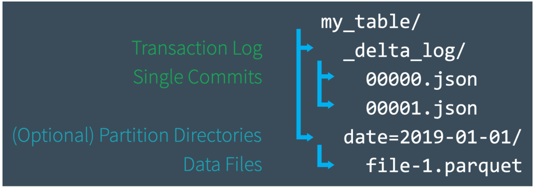
DeltaLog Structure (1/2)

- When a user creates a Delta Lake Table, its DeltaLog is automatically created in the `_delta_log` subdirectory.



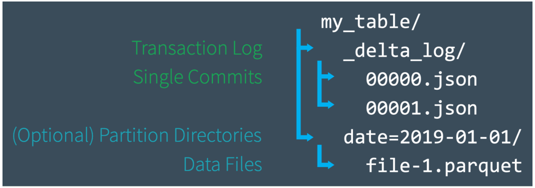
DeltaLog Structure (1/2)

- When a user creates a Delta Lake Table, its DeltaLog is automatically created in the `_delta_log` subdirectory.
- Any changes to that table are then recorded as ordered, atomic commits in the DeltaLog.



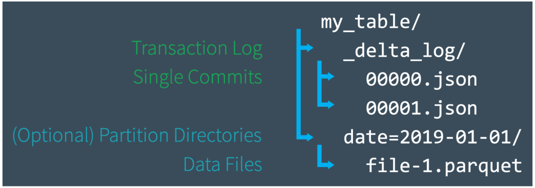
DeltaLog Structure (1/2)

- When a user creates a Delta Lake Table, its DeltaLog is automatically created in the `_delta_log` subdirectory.
- Any changes to that table are then recorded as ordered, atomic commits in the DeltaLog.
- Each commit is written out as a JSON file, starting with `000000.json`.



DeltaLog Structure (1/2)

- When a user creates a Delta Lake Table, its DeltaLog is automatically created in the `_delta_log` subdirectory.
- Any changes to that table are then recorded as ordered, atomic commits in the DeltaLog.
- Each commit is written out as a JSON file, starting with `000000.json`.
- Additional changes to the table generate subsequent JSON files in ascending numerical order, e.g., `000001.json`, 000002.json, and so on.



Deltalog Structure (2/2)

- Assume you add some records to a table from data files `1.parquet` and `2.parquet`.

Deltalog Structure (2/2)

- Assume you add some records to a table from data files `1.parquet` and `2.parquet`.
- That transaction would automatically be added to the DeltaLog, saved to disk as commit `000000.json`.

Deltalog Structure (2/2)

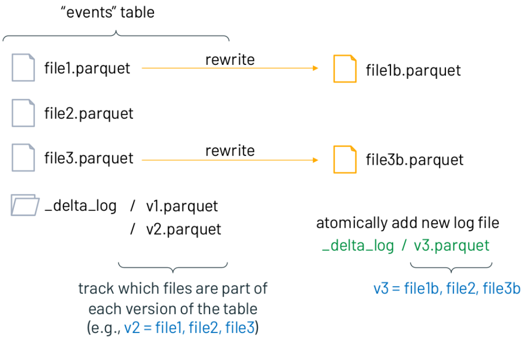
- Assume you add some records to a table from data files `1.parquet` and `2.parquet`.
- That transaction would automatically be added to the DeltaLog, saved to disk as commit `000000.json`.
- Then, assume remove those files and add `3.parquet` instead.

Deltalog Structure (2/2)

- Assume you add some records to a table from data files `1.parquet` and `2.parquet`.
- That transaction would automatically be added to the DeltaLog, saved to disk as commit `000000.json`.
- Then, assume remove those files and add `3.parquet` instead.
- Those actions would be recorded as the next commit in the DeltaLog, as `000001.json`.

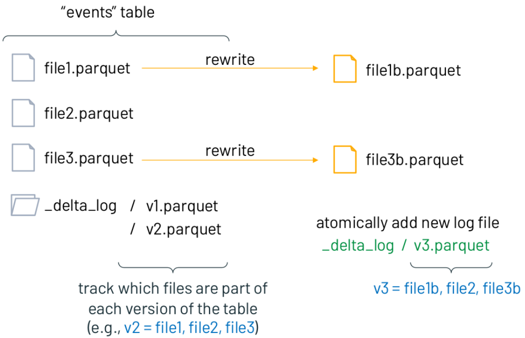
Delta Lake Transaction Example

- Query: delete all events data about customer no. 17



Delta Lake Transaction Example

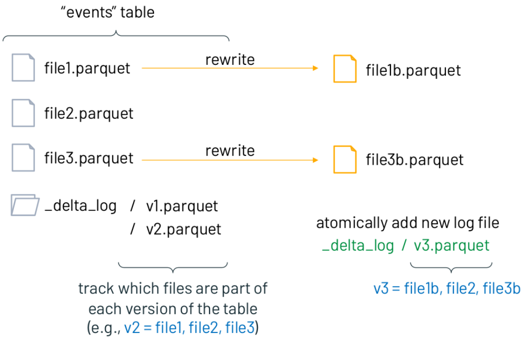
- Query: delete all events data about customer no. 17



- Clients now always read a consistent table version!

Delta Lake Transaction Example

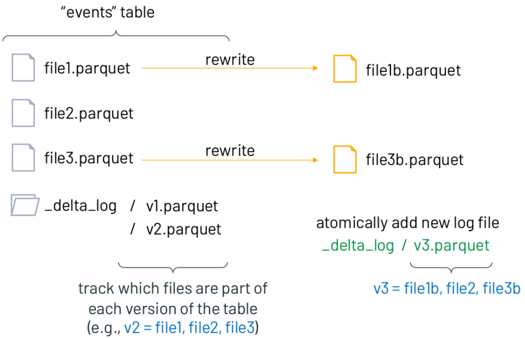
- Query: delete all events data about customer no. 17



- Clients now always read a consistent table version!
  - If a client reads v2 of log, it sees file1, file2, file3 (no delete)

Delta Lake Transaction Example

- Query: delete all events data about customer no. 17



- Clients now always read a consistent table version!
  - If a client reads v2 of log, it sees file1, file2, file3 (no delete)
  - If a client reads v3 of log, it sees file1b, file2, file3b (all deleted)

Actions and Commits

- Each log record object (e.g., `000003.json`) contains a commit, i.e., an array of actions recoreded as atomic, ordered units.

Actions and Commits

- Each log record object (e.g., `000003.json`) contains a commit, i.e., an array of actions recoreded as atomic, ordered units.
- Change metadata: name, schema, partitioning, etc.

Actions and Commits

- Each log record object (e.g., `000003.json`) contains a commit, i.e., an array of actions recoreded as atomic, ordered units.
- Change metadata: name, schema, partitioning, etc.
- Add/remove file: adds/removes a file

Actions and Commits

- Each log record object (e.g., `000003.json`) contains a commit, i.e., an array of actions recoreded as atomic, ordered units.
- Change metadata: name, schema, partitioning, etc.
- Add/remove file: adds/removes a file
- Protocol evolution: upgrades the version of the transaction protocol

Actions and Commits

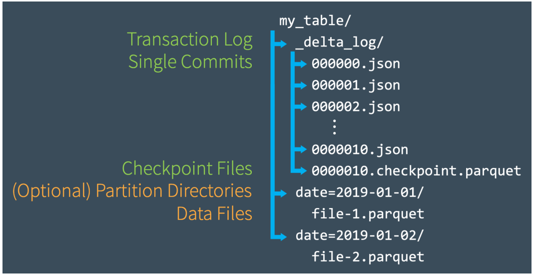
- Each log record object (e.g., `000003.json`) contains a commit, i.e., an array of actions recoreded as atomic, ordered units.
- Change metadata: name, schema, partitioning, etc.
- Add/remove file: adds/removes a file
- Protocol evolution: upgrades the version of the transaction protocol
- Set transaction: records an idempotent transaction id

Actions and Commits

- Each log record object (e.g., `000003.json`) contains a commit, i.e., an array of actions recoreded as atomic, ordered units.
- Change metadata: name, schema, partitioning, etc.
- Add/remove file: adds/removes a file
- Protocol evolution: upgrades the version of the transaction protocol
- Set transaction: records an idempotent transaction id
- Commit info: information around commit for auditing

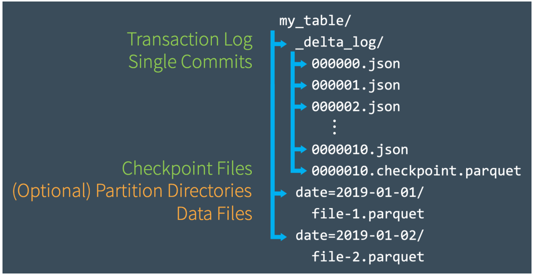
Quickly Recomputing State With Checkpoint Files

- Delta Lake automatically generates checkpoint files every 10 commits in the same `_delta_log` subdirectory.



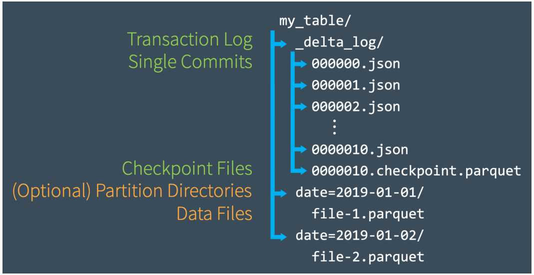
Quickly Recomputing State With Checkpoint Files

- Delta Lake automatically generates checkpoint files every 10 commits in the same `_delta_log` subdirectory.
- The checkpoint files save the entire state of the table at a point in time.



Quickly Recomputing State With Checkpoint Files

- Delta Lake automatically generates checkpoint files every 10 commits in the same `_delta_log` subdirectory.
- The checkpoint files save the entire state of the table at a point in time.
- They are in native Parquet format that is quick and easy for Spark to read.



Dealing With Multiple Concurrent Reads and Writes

- With petabytes of data, there is a high likelihood that users will be working on different parts of the data altogether.

Dealing With Multiple Concurrent Reads and Writes

- With petabytes of data, there is a high likelihood that users will be working on different parts of the data altogether.
- It allows them to complete non-conflicting transactions simultaneously.

Dealing With Multiple Concurrent Reads and Writes

- With petabytes of data, there is a high likelihood that users will be working on different parts of the data altogether.
- It allows them to complete non-conflicting transactions simultaneously.
- But, what if there is a conflict?

Dealing With Multiple Concurrent Reads and Writes

- With petabytes of data, there is a high likelihood that users will be working on different parts of the data altogether.
- It allows them to complete non-conflicting transactions simultaneously.
- But, what if there is a conflict?
- Optimistic concurrency control

Optimistic Concurrency Control (1/2)

- It ensures that the resulting state of the table after multiple concurrent writes is the same as if those writes had occurred serially, in isolation from one another.

Optimistic Concurrency Control (1/2)

- It ensures that the resulting state of the table after multiple concurrent writes is the same as if those writes had occurred serially, in isolation from one another.
- The process proceeds like this:

Optimistic Concurrency Control (1/2)

- It ensures that the resulting state of the table after multiple concurrent writes is the same as if those writes had occurred serially, in isolation from one another.
- The process proceeds like this:
  1. Record the starting table version.

Optimistic Concurrency Control (1/2)

- It ensures that the resulting state of the table after multiple concurrent writes is the same as if those writes had occurred serially, in isolation from one another.
- The process proceeds like this:
    1. Record the starting table version.
    2. Record reads/writes.

Optimistic Concurrency Control (1/2)

- It ensures that the resulting state of the table after multiple concurrent writes is the same as if those writes had occurred serially, in isolation from one another.
- The process proceeds like this:
  1. Record the starting table version.
  2. Record reads/writes.
  3. Attempt a commit.

Optimistic Concurrency Control (1/2)

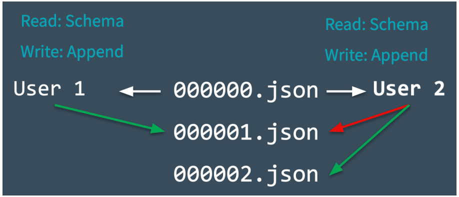
- It ensures that the resulting state of the table after multiple concurrent writes is the same as if those writes had occurred serially, in isolation from one another.
- The process proceeds like this:
  1. Record the starting table version.
  2. Record reads/writes.
  3. Attempt a commit.
  4. If someone else wins, check whether anything you read has changed.

Optimistic Concurrency Control (1/2)

- It ensures that the resulting state of the table after multiple concurrent writes is the same as if those writes had occurred serially, in isolation from one another.
- The process proceeds like this:
  1. Record the starting table version.
  2. Record reads/writes.
  3. Attempt a commit.
  4. If someone else wins, check whether anything you read has changed.
  5. Repeat.

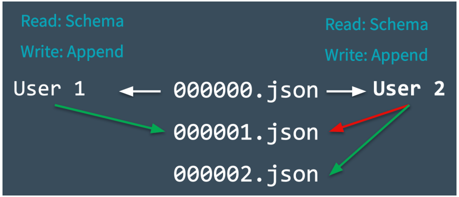
Optimistic Concurrency Control (2/2)

- Two users read from the same table, then each attempts to add some data to it.



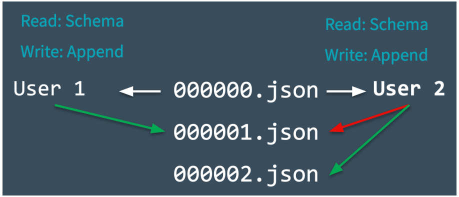
Optimistic Concurrency Control (2/2)

- Two users read from the same table, then each attempts to add some data to it.
- Here, we run into a conflict because only one commit can come next and be recorded as `000001.json`.



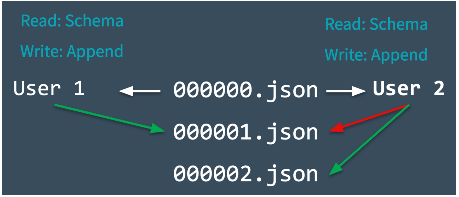
Optimistic Concurrency Control (2/2)

- Two users read from the same table, then each attempts to add some data to it.
- Here, we run into a conflict because only one commit can come next and be recorded as `000001.json`.
- Mutual exclusion: only one user can successfully make commit `000001.json` (user 1's commit is accepted, while user 2's is rejected).



Optimistic Concurrency Control (2/2)

- Two users read from the same table, then each attempts to add some data to it.
- Here, we run into a conflict because only one commit can come next and be recorded as `000001.json`.
- Mutual exclusion: only one user can successfully make commit `000001.json` (user 1's commit is accepted, while user 2's is rejected).
- However, Delta Lake does not throw an error for user 2 and handles this conflict optimistically.



Use Cases - Time Travel

- Every table is the result of the sum of all of the commits recorded in the Delta Lake DeltaLog.

Use Cases - Time Travel

- Every table is the result of the sum of all of the commits recorded in the Delta Lake DeltaLog.
- The DeltaLog provides a step-by-step instruction guide, detailing exactly how to get from the table's original state to its current state.

Use Cases - Time Travel

- Every table is the result of the sum of all of the commits recorded in the Delta Lake DeltaLog.
- The DeltaLog provides a step-by-step instruction guide, detailing exactly how to get from the table's original state to its current state.
- Thus, we can recreate the state of a table at any point in time.

Use Cases - Time Travel

- Every table is the result of the sum of all of the commits recorded in the Delta Lake DeltaLog.
- The DeltaLog provides a step-by-step instruction guide, detailing exactly how to get from the table's original state to its current state.
- Thus, we can recreate the state of a table at any point in time.
  - Starting with an original table, and processing only commits made prior to that point.

Use Cases - Time Travel

- Every table is the result of the sum of all of the commits recorded in the Delta Lake DeltaLog.
- The DeltaLog provides a step-by-step instruction guide, detailing exactly how to get from the table's original state to its current state.
- Thus, we can recreate the state of a table at any point in time.
    - Starting with an original table, and processing only commits made prior to that point.
- This ability is known as time travel or data versioning.

Use Cases - Data Lineage and Debugging

- The Delta Lake DeltaLog offers users a verifiable data lineage.

Use Cases - Data Lineage and Debugging

- The Delta Lake DeltaLog offers users a verifiable data lineage.
- It is is useful for governance, audit and compliance purposes.

Use Cases - Data Lineage and Debugging

- The Delta Lake DeltaLog offers users a verifiable data lineage.
- It is is useful for governance, audit and compliance purposes.
- It can also be used to trace the origin of an inadvertent change or a bug in a pipeline back to the exact action that caused it.

Schema Enforcement and Evolution

Schema Enforcement and Evolution

- Data is always evolving and accumulating.

Schema Enforcement and Evolution

- Data is always evolving and accumulating.
- So, structure of data evolves over time.

Schema Enforcement and Evolution

- Data is always evolving and accumulating.
- So, structure of data evolves over time.
- With Delta Lake, as the data changes, incorporating new dimensions is easy.

Schema Enforcement and Evolution

- Data is always evolving and accumulating.
- So, structure of data evolves over time.
- With Delta Lake, as the data changes, incorporating new dimensions is easy.
- Schema enforcement: prevents users from accidentally polluting their tables with mistakes or garbage data.

Schema Enforcement and Evolution

- Data is always evolving and accumulating.
- So, structure of data evolves over time.
- With Delta Lake, as the data changes, incorporating new dimensions is easy.
- Schema enforcement: prevents users from accidentally polluting their tables with mistakes or garbage data.
- Schema evolution: enables automatic addition of columns when desired.

Understanding Table Schemas

- Spark DataFrames contain the schema.
- With Delta Lake, the table's schema is saved in JSON format inside the DeltaLog.

```
schemaString: {"type":"struct","fields":[
  {"name":"loan_id","type":"long","nullable":false,"metadata":{}},
  {"name":"funded_amnt","type":"integer","nullable":true,"metadata":{}},
  {"name":"paid_amnt","type":"double","nullable":true,"metadata":{}},
  {"name":"addr_state","type":"string","nullable":true,"metadata":{}}
]}
```

Schema Enforcement

- Schema enforcement (a.k.a schema validation) occurs on write.

Schema Enforcement

- Schema enforcement (a.k.a schema validation) occurs on write.
- If the schema is not compatible, Delta Lake cancels the transaction, i.e., no data is written.

Schema Enforcement

- Schema enforcement (a.k.a schema validation) occurs on write.
- If the schema is not compatible, Delta Lake cancels the transaction, i.e., no data is written.
- As well, Delta Lake raises an exception to let the user know about the mismatch.

Schema Enforcement Rules

- Rule 1: cannot contain any additional columns that are not present in the target table's schema.

Schema Enforcement Rules

- Rule 1: cannot contain any additional columns that are not present in the target table's schema.
- Rule 2: cannot have column data types that differ from the column data types in the target table.

Schema Enforcement Rules

- Rule 1: cannot contain any additional columns that are not present in the target table's schema.
- Rule 2: cannot have column data types that differ from the column data types in the target table.
- Rule 3: Can not contain column names that differ only by case.

Schema Evolution

- Schema evolution allows users to change a table's current schema to accommodate data that is changing over time.

Schema Evolution

- Schema evolution allows users to change a table's current schema to accommodate data that is changing over time.
- Most commonly used operations for append and overwrite.

Delta Lake and Spark

Loading Data into a Delta Lake Table (1/2)

- All you need to migrate any of the structured data formats (e.g., Parquet) to Delta Lake is to use `format("delta")`.

```scala
// Configure source data and Delta Lake path
val sourcePath = "loan-risks.snappy.parquet"
val deltaPath = "loans_delta"

// Create the Delta table with the same loans data
spark.read.format("parquet").load(sourcePath).write.format("delta").save(deltaPath)

// Create a view on the data called loans_delta
spark.read.format("delta").load(deltaPath).createOrReplaceTempView("loans_delta")
```

Loading Data into a Delta Lake Table (2/2)

```
// Read and explore the data
spark.sql("SELECT count(*) FROM loans_delta").show()

+--------+
|count(1)|
+--------+
|   14705|
+--------+

// First 3 rows of loans table
spark.sql("SELECT * FROM loans_delta LIMIT 3").show()

+-------+-----------+---------+----------+
|loan_id|funded_amnt|paid_amnt|addr_state|
+-------+-----------+---------+----------+
|      0|       1000|   182.22|        CA|
|      1|       1000|   361.19|        WA|
|      2|       1000|   176.26|        TX|
+-------+-----------+---------+----------+
```

Loading Data Streams into a Delta Lake Table

- You can modify your existing Structured Streaming jobs to write to and read from a Delta Lake table by setting the format to `"delta"`.

Loading Data Streams into a Delta Lake Table

- You can modify your existing Structured Streaming jobs to write to and read from a Delta Lake table by setting the format to `"delta"`.

```
import org.apache.spark.sql.streaming._

// Streaming DataFrame with new loans data
val newLoanStreamDF = ...

// Directory for streaming checkpoints
val checkpointDir = ...

val streamingQuery = newLoanStreamDF.writeStream
.format("delta")
.option("checkpointLocation", checkpointDir)
.trigger(Trigger.ProcessingTime("10 seconds"))
.start(deltaPath)
```

Schema Enforcement

- All writes to a Delta Lake table can verify whether the data being written has a schema compatible with that of the table.

```
val loanUpdates = Seq(
   (1111111L, 1000, 1000.0, "TX", false),
   (2222222L, 2000, 0.0, "CA", true))
 .toDF("loan_id", "funded_amnt", "paid_amnt", "addr_state", "closed")

loanUpdates.write.format("delta").mode("append").save(deltaPath)

// The exception message:
// This write will fail with the following error message:
// org.apache.spark.sql.AnalysisException: A schema mismatch detected when writing
// to the Delta table (Table ID: 48bfa949-5a09-49ce-96cb-34090ab7d695).
```

Schema Enforcement

- All writes to a Delta Lake table can verify whether the data being written has a schema compatible with that of the table.
- If it is not compatible, Spark will throw an error before any data is written and committed to the table.

```
val loanUpdates = Seq(
   (1111111L, 1000, 1000.0, "TX", false),
   (2222222L, 2000, 0.0, "CA", true))
 .toDF("loan_id", "funded_amnt", "paid_amnt", "addr_state", "closed")

loanUpdates.write.format("delta").mode("append").save(deltaPath)

// The exception message:
// This write will fail with the following error message:
// org.apache.spark.sql.AnalysisException: A schema mismatch detected when writing
// to the Delta table (Table ID: 48bfa949-5a09-49ce-96cb-34090ab7d695).
```

Schema Evolution

- A new column can be explicitly added by setting the option `mergeSchema` to `true`.

```
loanUpdates.write.format("delta").mode("append")
  .option("mergeSchema", "true")
  .save(deltaPath)
```

Transforming Existing Data - Updating Data

- Delta Lake supports UPDATE, DELETE, and MERGE commands

Transforming Existing Data - Updating Data

- Delta Lake supports UPDATE, DELETE, and MERGE commands
- They ensure ACID guarantees.

Transforming Existing Data - Updating Data

- Delta Lake supports UPDATE, DELETE, and MERGE commands
- They ensure ACID guarantees.
- Assume we want to change all `addr_state = 'OR'` to `addr_state = 'WA'` in a table.

```
import io.delta.tables.DeltaTable
import org.apache.spark.sql.functions._

val deltaTable = DeltaTable.forPath(spark, deltaPath)

deltaTable.update(
    col("addr_state") === "OR",
    Map("addr_state" -> lit("WA")))
```

Transforming Existing Data - Deleting Data

- Deleting user data from all tables.

```
val deltaTable = DeltaTable.forPath(spark, deltaPath)

deltaTable.delete("funded_amnt >= paid_amnt")
```

Auditing Data Changes with Operation History

- All of the changes are recorded as commits in the table's DeltaLog.
- Every operation is automatically versioned.
- You can query the table's operation history.

```
deltaTable
    .history(3)
    .select("version", "timestamp", "operation", "operationParameters")
    .show(false)
```

Querying Previous Snapshots of a Table with Time Travel

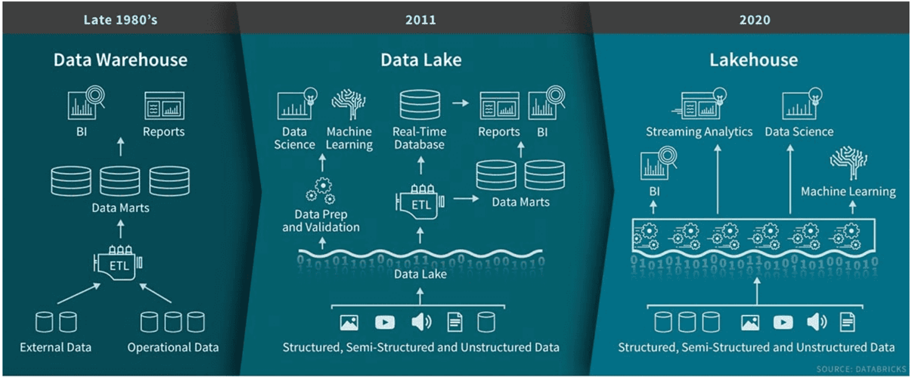
- You can query previous versioned snapshots of a table by using the `DataFrameReader` options `versionAsOf` and `timestampAsOf`.

```
spark.read.format("delta")
    .option("timestampAsOf", "2020-01-01") // timestamp after table creation
    .load(deltaPath)

spark.read.format("delta")
    .option("versionAsOf", "4")
    .load(deltaPath)
```

Summary

Summary



Late 1980's
2011
2020
Data Warehouse
BI
Reports
Data Marts
ETL
External Data
Operational Data
Data Lake
Data Science
Machine Learning
Real-Time Database
Reports
BI
Data Prep and Validation
ETL
Data Marts
Data Lake
Structured, Semi-Structured and Unstructured Data
Lakehouse
Streaming Analytics
Data Science
BI
Machine Learning
Structured, Semi-Structured and Unstructured Data
SOURCE: DATABRICKS

References

- J. S. Damji et al., "Learning Spark - Lightning-Fast Data Analytics", O'Reilly Media, 2020 - Chapters 9
- M. Armbrust et al., "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics", CIDR 2021
- M. Armbrust et al., "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores", VLBD 2020

Questions?

Acknowledgements

Some content and images are derived from Jules S. Damji, Andreas Neumann, Burak Yavuz, and Denny Lee slides from Databricks.