Cloudera Open Data Lakehouse

NDA & SAFE HARBOR COMPLIANCE REQUIRED

The information in this document is our currently intended developments and functionalities of Cloudera products which may change without notice at Cloudera's discretion. Cloudera makes no commitments about any future developments or functionality in any Cloudera product. The development, release, and timing of release of any software features or functionality described in this document remains at the discretion of Cloudera and you should not rely on any statements about development plans or anticipated functionality in this document when making any purchasing decisions.

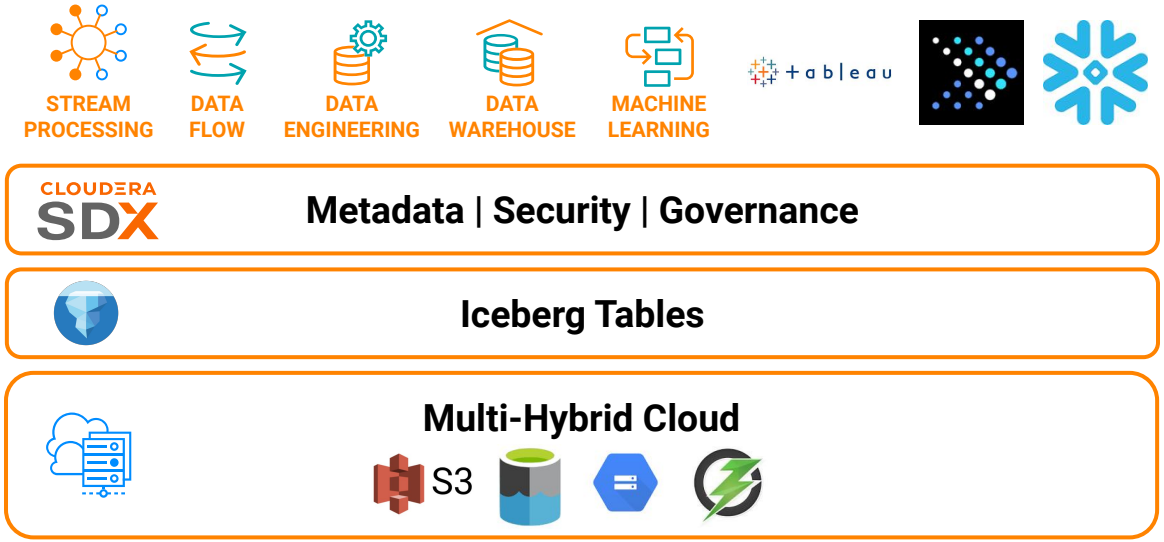
Agenda

- Cloudera Open Data Lakehouse on Private Cloud
- Iceberg features
- Roadmap
- Live Demo
- Q&A (10 mins)

Cloudera Open Data Lakehouse BI, AI, and Beyond

Platform architecture delivers secure interoperability to any infrastructure cost-effectively

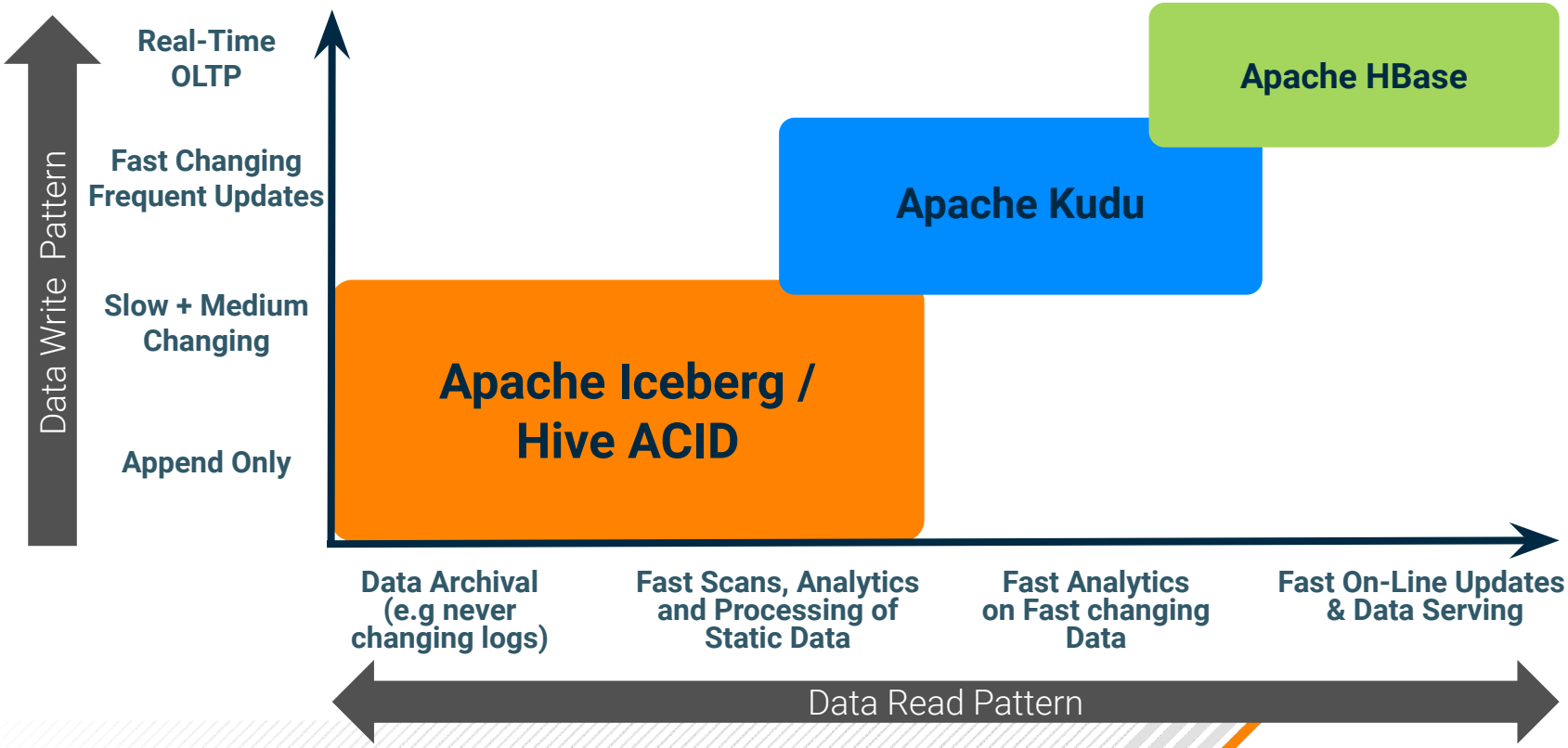
- Multi-function analytics for Streaming, Data Engineering, BI, & AI/ML
- Choose your engine: Cloudera integrated data services or 3rd party technologies
- SDX: Common security and governance with data lineage
- One Dataset: With all Compute engines. No data duplication or movement
- Deployment freedom with Multi-Hybrid Cloud on CPU & GPU Containers



APACHE ICEBERG
A Flexible, Performant & Scalable Table Format

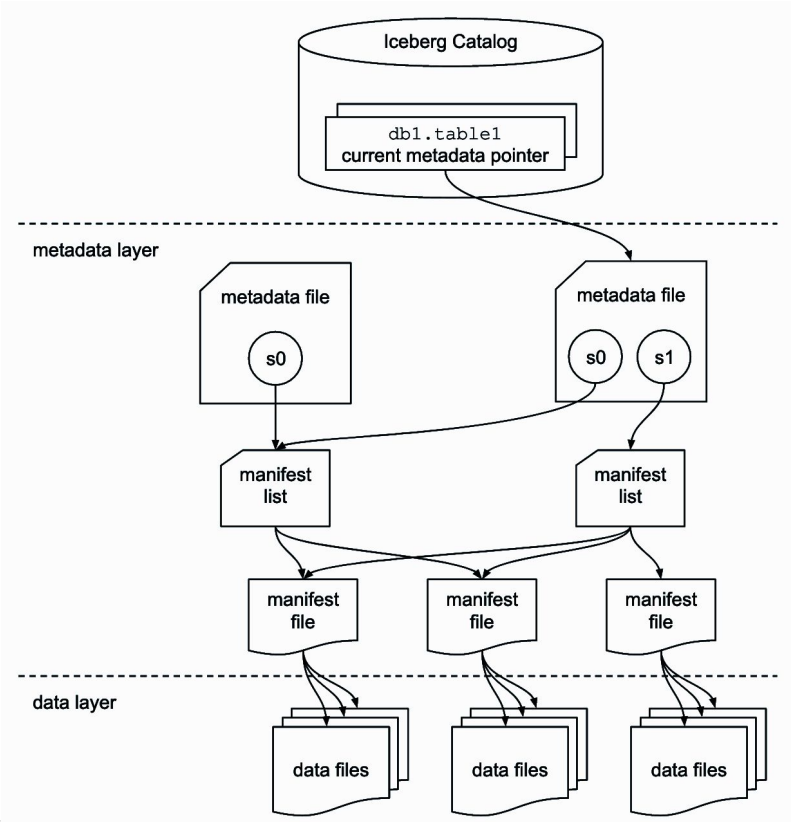
- Donated by Netflix to the Apache Foundation in 2018
- Flexibility
  - Hidden partitioning
  - Full schema evolution
- Data Warehouse Operations
  - ACID Transactions
  - Time travel and rollback
- Supports best in class SQL performance
  - High performance at Petabyte scale

Where Apache Iceberg fits in the CDP Ecosystem

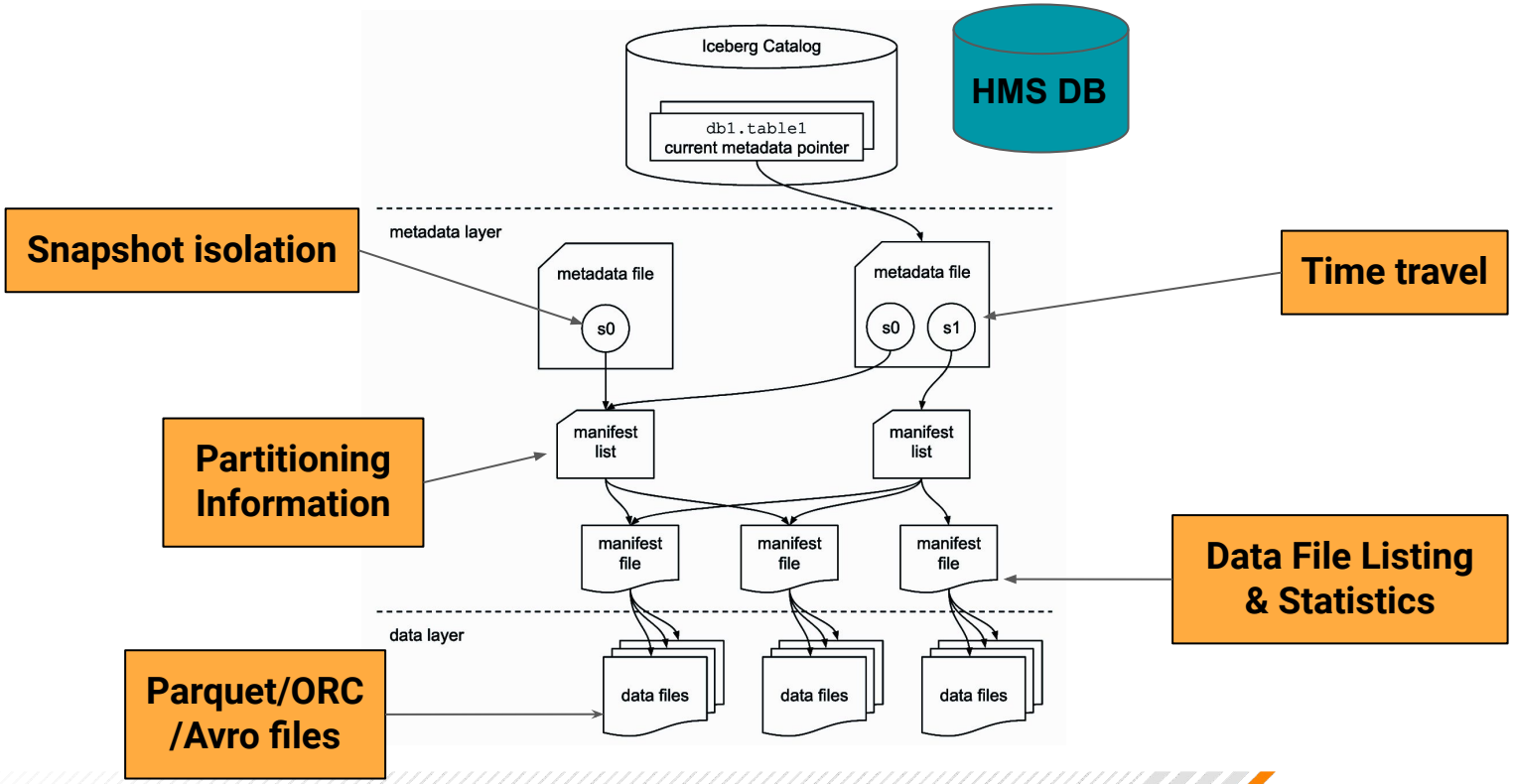


What is Iceberg Table Format?

- Iceberg is a new Big Data scale **Table Format** that specifies how a table's files are organized in a file system or object store
- Iceberg offers advanced features and performance improvements over the default table format
- 3 file formats supported - Parquet, Avro, and ORC
    - Supported by Impala, Hive, Spark, Fink, etc

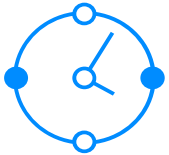


How Does Iceberg Format Solve the Issues

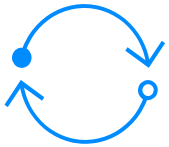


KEY FEATURES

ACID Transaction



Time Travel / Table Rollback



In-place Table Evolution

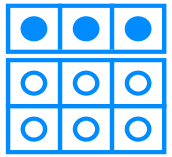
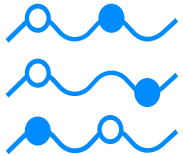
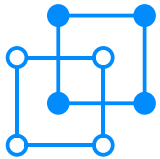


Table Maintenance

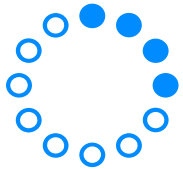


Streaming & Batch Ingestion



Iceberg Replication

SDX Integration



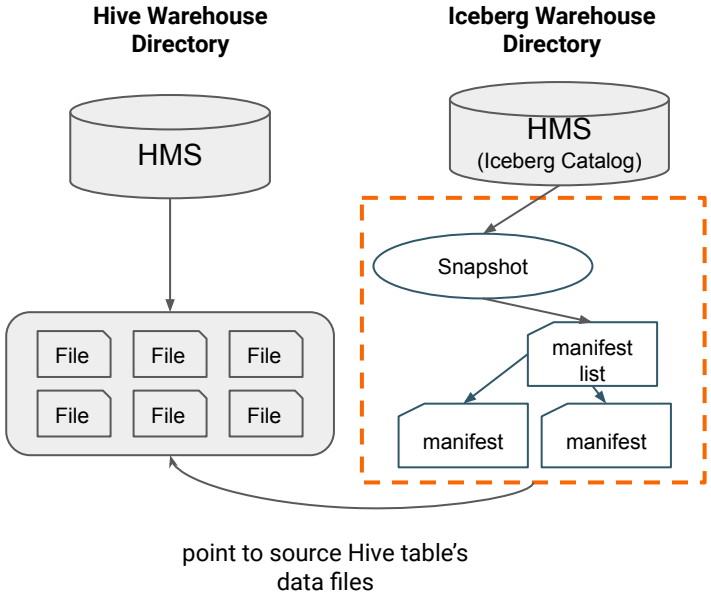
Ease of Adoption

Multi-Hybrid Cloud

Performance / Scalability

Table Migration

Table Migration In-Place



Avoids rewriting data files, just **re-write** the metadata

- **Hive table migration**:

```
ALTER TABLE tbl SET TBLPROPERTIES
('storage_handler'='org.apache.iceberg.mr.hive.Hive
IcebergStorageHandler')
```

- **Spark 3:**

  a. Import Hive tables into Iceberg

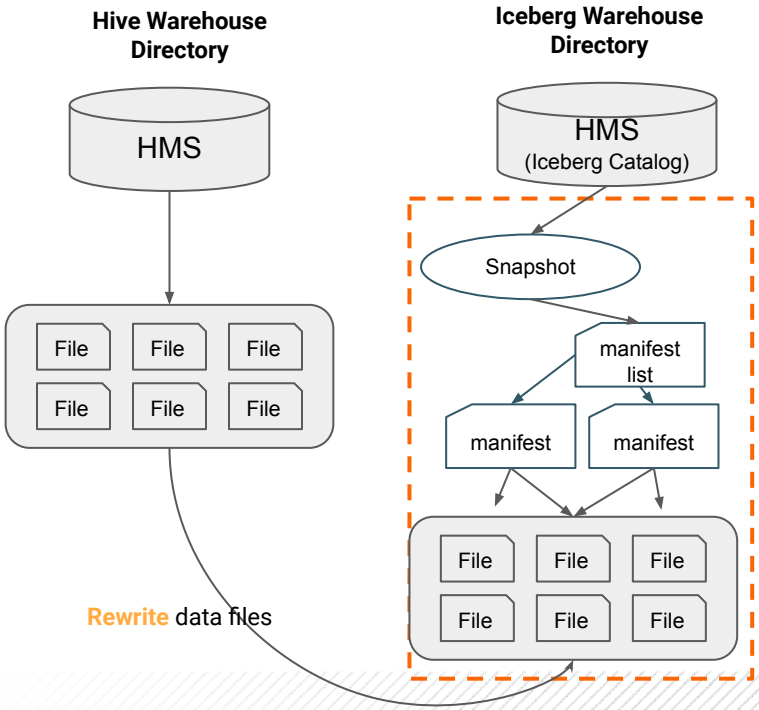
```
spark.sql("CALL
<catalog>.system.snapshot('<src>', '<dest>')")
```

  b. Migrate Hive tables to Iceberg tables

```
spark.sql("CALL
 <catalog>.system.migrate('<src>')")
```

Table Migration CTAS / RTAS

Data files **recreated** in addition to creation of Iceberg tables and corresponding metadata



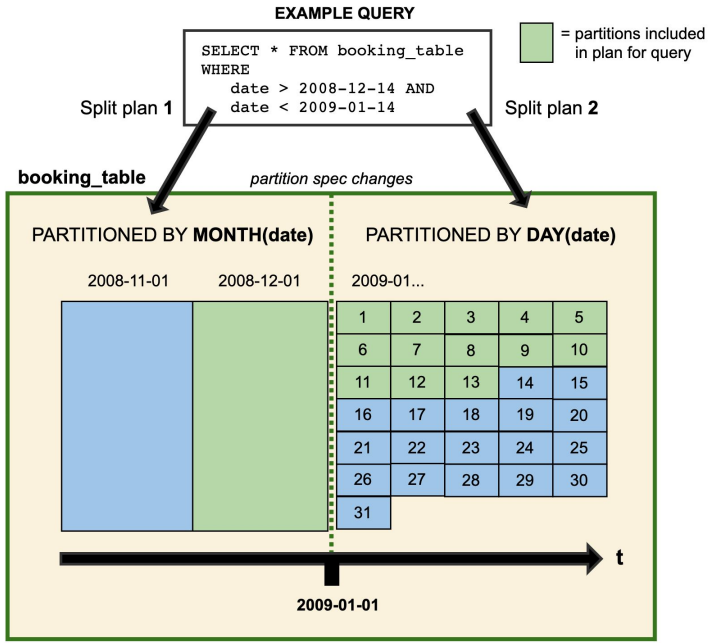

```
CREATE TABLE ctas PARTITIONED BY(z) STORED
BY ICEBERG AS SELECT x, y FROM t;
```

```
spark.sql("CREATE TABLE ctas PARTITIONED
BY(z) USING iceberg AS (SELECT x, y FROM
t)")
```

```
spark.sql("REPLACE TABLE rtas PARTITIONED
BY(z) USING iceberg AS (SELECT x, y FROM
t)")
```

Partition Evolution

Iceberg Partition Evolution



ALTER TABLE t1 SET PARTITION SPEC (hour(ts))
ALTER TABLE t1 ADD PARTITION FIELD (month(ts))

- Existing big data solution doesn't support **in-place** partition evolution. Entire table must be completely rewritten with new partition column
- With Iceberg's **hidden partition**, a separation between physical and logical, users are not required to maintain partition columns.
- Iceberg tables can **evolve partition** schemas over time as data volume changes.
- **Benefits**:
  - No costly table rewrites or table migration
  - No query rewrites
  - Reduce downtime and improve SLA

In-Place Partition Evolution SQL examples

Engine	SQL Examples
Impala	`// Partition evolution to hour` `ALTER TABLE t SET PARTITION SPEC (hour(ts))`
Spark SQL	`// Partition evolution to hour` `ALTER TABLE t ADD PARTITION FIELD (hour(ts))`

TIME TRAVEL

Two rollback and time travel options: snapshot-id and as-of-timestamp

Audit data changes (BMS)

- Creates automatic snapshots of past data
- History of all operations are recorded for audits

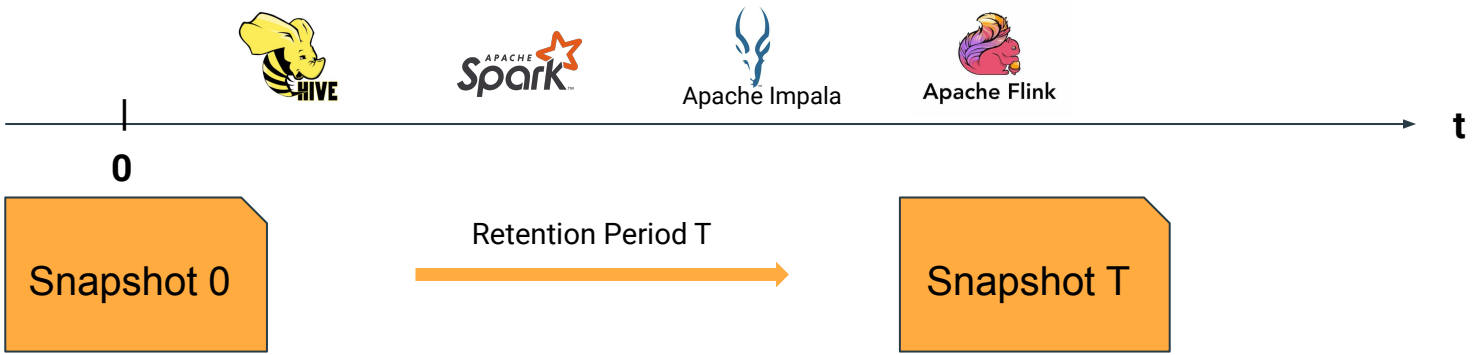
Data Reproducibility

- Retrain ML models with past data
- Generate financial reports faithfully, on past data

Rollback: Eliminate Data Errors

- Data pipeline debugging
- Rollback to any point to find source of bad data
- Root cause analysis
- Update, replace, or delete bad data

Time Travel & Table Rollback



- **Time Travel** enables reproducible queries that use exactly the same table snapshot, or lets users easily examine changes
- **Rollback** allows users to quickly correct problems by resetting tables to a good state

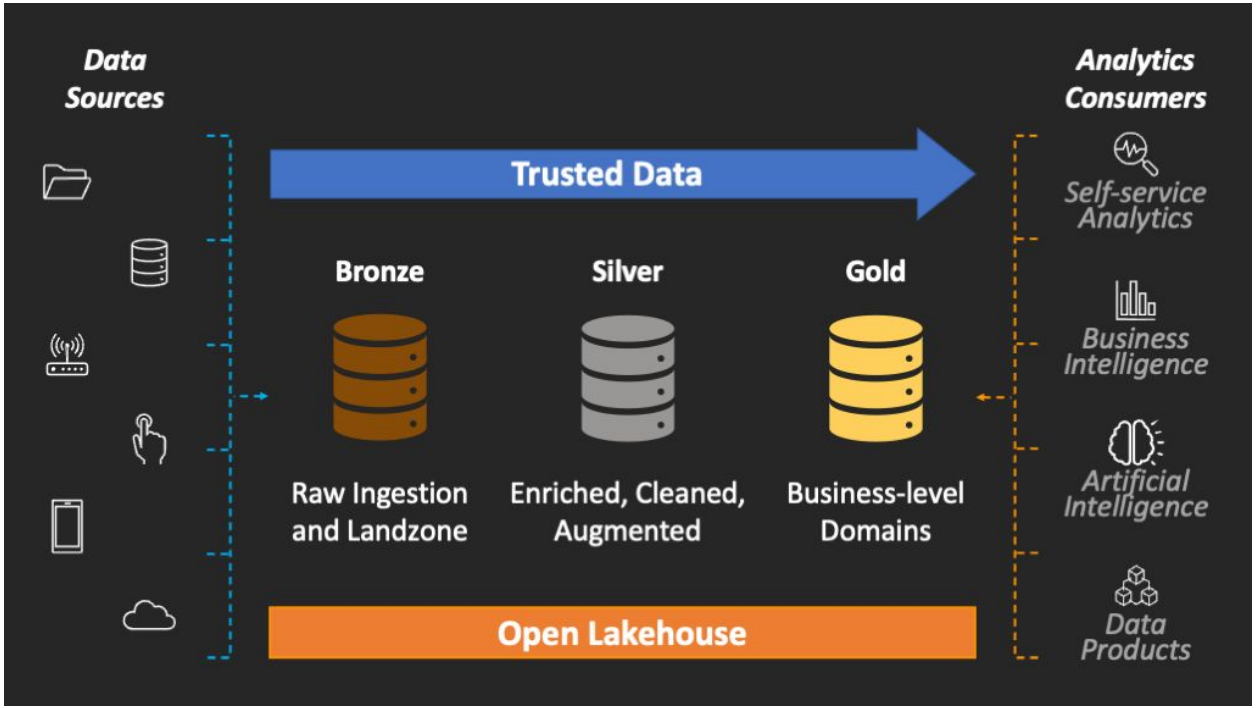
- `SELECT * FROM table FOR SYSTEM_TIME AS OF '2021-08-09 10:35:57';`
- `SELECT * FROM table FOR SYSTEM_VERSION AS OF 1234567;`
- `ALTER TABLE table EXECUTE ROLLBACK ('2021-08-09 10:35:57')`
- `ALTER TABLE table EXECUTE ROLLBACK (1234567)`

CLOUDERA

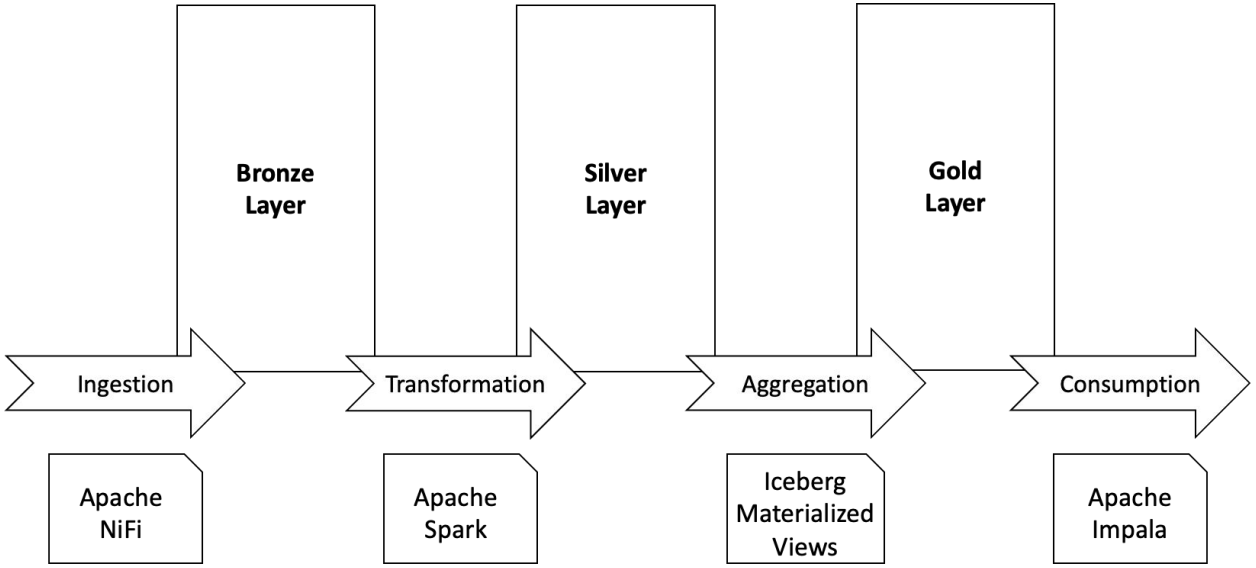
Lakehouse Best Practices with Apache Iceberg

Medallion Architecture

MEDALLION ARCHITECTURE



MEDALLION ARCHITECTURE



MEDALLION ARCHITECTURE

Icetip

Leverage Medallion Architecture pattern to simplify the maintenance of the different stages of the data that will be ingested into the Lakehouse.

Keep the data format closer to the origin data source in the Bronze layer and perform the additional transformations and enrichment for the upper layers.

Materialized Views can also be leveraged for the Gold layer as a mechanism to facilitate multiple table aggregation creation and maintenance.

Partitioning

PARTITIONING

Data Management	Data Consumption

• There is no need to create a specific column for the partition strategy
 	
`CREATE TABLE tbl1 (i int, s string, ts timestamp, d date)`
`PARTITIONED BY SPEC (YEAR(ts))`
`STORED by ICEBERG;`
 	
• Insert can leverage built-in transforms to allocate data into partitions
 	
`PARTITIONED BY SPEC (TRUNCATE(10, i), BUCKET(11, s), YEAR(ts))`
 	
• Storage efficiencies, as the additional columns for partitions are not required anymore
	• End-users don't need to specify the filter corresponding to the partition column
	
	`SELECT * FROM tbl1 WHERE ts BETWEEN '2022-07-01 00:00:00' AND '2022-07-31 00:00:00'` ~~`AND month = ?`~~`;`
	
	• More intuitive and natural data access, using the original value of the partition columns
	
	• Minimize the risk of full scans, as Iceberg statistics are leveraged during query execution

PARTITIONING

Icetip

Whenever possible, leverage the Hidden Partitioning capabilities Iceberg tables deliver. In most cases, tables are partitioned by an attribute that represents time.

Using the transforms to allocate data rows to different partitions will make data ingestion and data consumption easier.

PARTITIONING

Transforms

Transformation	Spec	Supported by SQL Engine
Partition by year	years(time_stamp) | year(time_stamp)	Hive and Impala
Partition by month	months(time_stamp) | month(time_stamp)	Hive and Impala
Partition by a date value stored as int (dateint)	days(time_stamp) | date(time_stamp)	Hive
Partition by hours	hours(time_stamp)	Hive
Partition by a dateint in hours	date_hour(time_stamp)	Hive
Partition by hashed value mod N buckets	bucket(N, col)	Hive and Impala
Partition by value truncated to L, which is a number of characters	truncate(L, col)	Hive and Impala

Partition Evolution

PARTITION EVOLUTION

From less to more granular

Partitioned by year:

- 2015
- 2016
- 2017
- 2018
- 2019
- 2020
- 2021
- 2022
- 2023

Partition Evolution

Partitioned by year, month:

- 2015
- 2016
- 2017
- 2018
- 2019
- 2020
- 2021
- 2022
- 2023
  - June
  - July
  - August
  - September
  - October
  - November
  - December

Partition Evolution

Partitioned by year, month, day:

- 2015
- 2016
- 2017
- 2018
- 2019
- 2020
- 2021
- 2022
- 2023
  - June
  - July
  - August
  - September
  - October
  - November
  - December
    - 20
    - 21
    - 22
    - 23
    - 24
    - 25

Table structure, partitioned by year:

```
CREATE TABLE transaction (id INT,
time TIMESTAMP, ...)
PARTITIONED BY YEARS(time);
```

Adding new partitions. First, break down by month, then by day:

```
ALTER TABLE transaction ADD
PARTITION FIELD MONTH(time)

ALTER TABLE transaction ADD
PARTITION FIELD DAY(time)
```

PARTITION EVOLUTION

Icetip

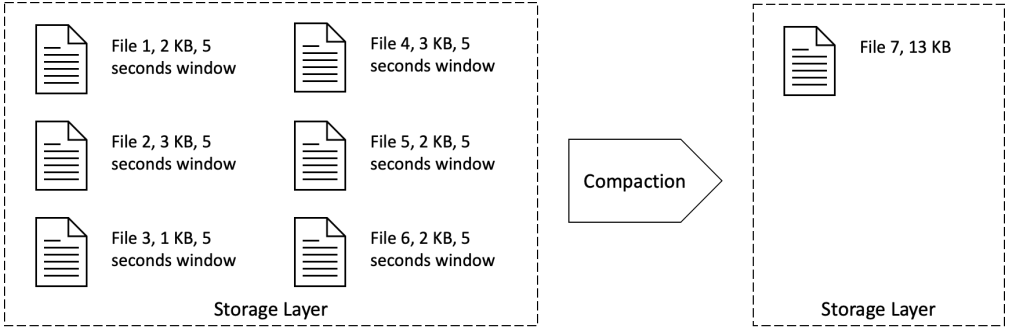
Monitor the amount of data ingested in each partition, and the query's patterns, to define when a new partition is needed to keep consumption jobs performant. Leverage in-place partition evolution to break down data into more granular partitions easily.

End-user doesn't need to be aware of the partition changes, even changing how to query the table.

Data Compaction

DATA COMPACTION

Optimize data file size



1. Strategy

binPack, order, zorder

2. Filters

Run compaction on specific parts of the data

3. Options

partial-progress-enabled, rewrite-job-order, target-file-size-bytes, etc…

4. Automation

CDE => Spark job orchestrated by Airflow

DATA COMPACTION

Optimize data file size

Icetip

Accordingly to the row-level mutations strategy mutation defined for the Iceberg table, data compactions should be run more or less frequently.

As COW already creates new files with the changes, this strategy will eventually require less frequent compactions.

On the other hand, MOR creates delta files to record the changes, and to keep good query performance, compactions will create new files, therefore, better access performance.

Row-level Mutation

COW vs MOR

ROW-LEVEL MUTATION

	Change actions		Speed		
Mutation Strategy	**DELETIONS**	**UPDATES**	**Read**	**Write**	**Recommendations**
COW	The entire file is rewritten excluding the deleted rows	The whole file is rewritten with the updated rows	Fastest	Slowest	Recommended where changes to data are low frequency and reads performance is significantly more important
MOR (Equality deletes)	New files are created containing the row conditions to be deleted	New files are created just with the updated rows	Slow	Fastest	Requires frequent complications depending on how frequent are the updates/deletes.
MOR (Position deletes)	New files are created containing the row positions to be deleted	New files are created just with the updated rows	Fast	Fast	Requires frequent complications depending on how frequent are the updates/deletes.

ROW-LEVEL MUTATION

Icetip

Understand the use case needs - how data will be ingested, processed and consumed

Use case	Description	Requirements	Strategy
Batch processing	Process large chunks of files	Data can be consumed minutes/hours after processed	COW
Streaming analytics	(Near) real-time ingestion and processing	Data should be available immediately after ingestions	MOR, and run compactions to optimize reads; or consider Hbase or Kudu as better alternatives
Machine Learning	Run advanced analytics on static/batch data	Train ML models on vast datasets, creating new ones	COW
Real-time Business Intelligence	BI analytics for reporting and dashboards	Build snowflake or star schemas in 3NF	MOR, and run compactions to optimize reads

Miscellaneous

STORAGE OPTIMIZATION

Expire snapshots

Icetip

Understand the data retention requirements accordingly to the use case, project, business needs or local regulations. Then automate jobs to expire the snapshots that are no longer needed periodically.

```
ALTER TABLE test_table EXECUTE
expire_snapshots('2021-12-09
05:39:18);
```

STATISTICS COLLECTION

Control columns' metrics

Icetip

When creating an Iceberg table, understand the most common attributes end-users will use to filter out the table. That will indicate which columns are worth computing statistics and the appropriate metric mode. This task can be performed after table creation if the access pattern changes.

Metric Mode	Description
none	Doesn't compute any column statistics
counts	Only compute counts for the column
truncate(length)	Similar to counts, but truncate column value to a certain number of characters
full	Compute counts, lower and upper bounds with the column's full value

MATERIALIZED VIEW

```
CREATE MATERIALIZED VIEW transaction_aggregation
PARTITIONED ON (col4)
STORED BY ICEBERG STORED AS PARQUET
AS
SELECT tbl1.col1, tbl2.col4, count(tbl1.amount) as total_amt
FROM tbl1 JOIN tbl2 ON tbl1.id = tbl2.id
GROUP BY tbl1.col1, tbl2.col4;
```

Icetip

Leverage Materialized Views to pre-compute most common aggregation for end-user and BI projects. With MV, creating, maintaining, and evolving those business views will be much easier from the data management perspective.

Only INSERT transactions will perform an incremental rebuild. Under other conditions, will force a full rebuild.

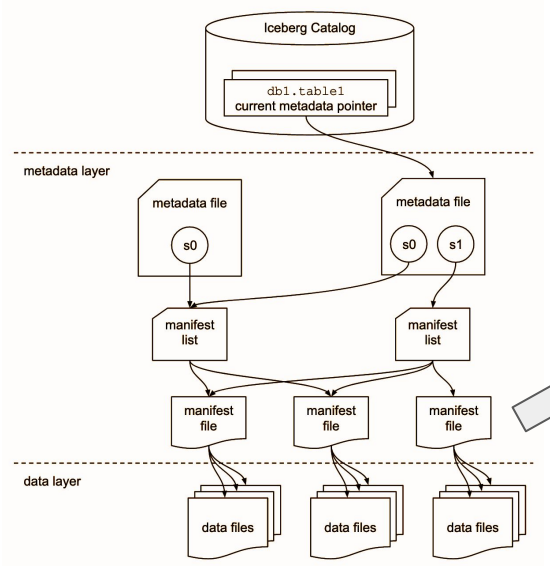
FILE FORMATS

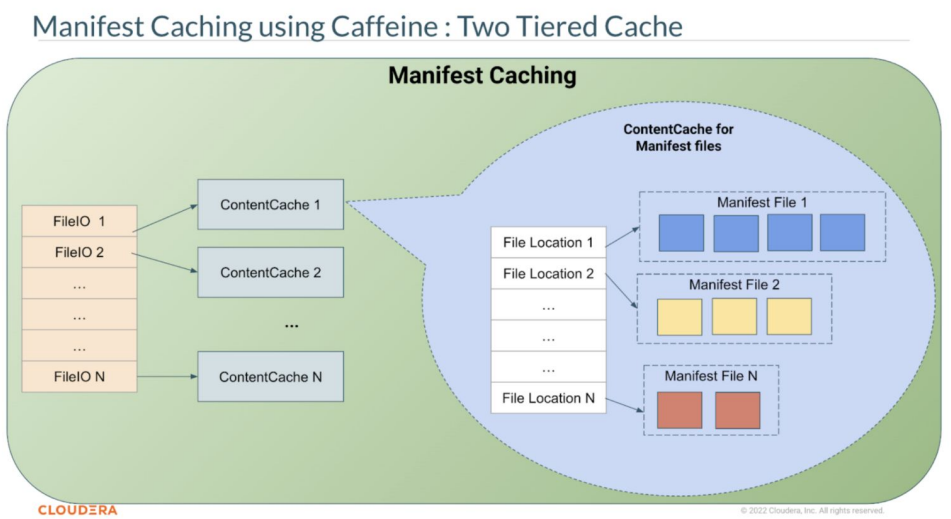
File Format	Description	Suitable for
Apache ORC	Columnar data format, with good compression ratio performance.	Writing-intensive tasks, like streaming and real-time ingestion.
Apache Parquet	Columnar data format achieves storage efficiencies and processing performance.	Reading-intensive tasks, like analytics and OLAP workloads.
Apache Avro	A popular format for row-based data serialization.	Low-latency streaming analytics use cases.

Icetip

Understanding the use case for the Iceberg table to choose the proper file format will directly impact the transactions (updates and deletes) and the access performance.

METADATA CACHING





Icetip

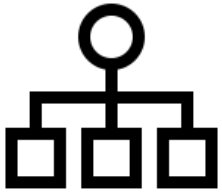
Enabling the manifest caching feature helps to reduce repeated reads of small Iceberg manifest files from remote storage by Impala Coordinators and Catalogd.

Checklist

TABLE CREATION

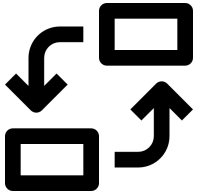
File Format

Choose the appropriate file format for the use case - Parquet, Avro, or ORC



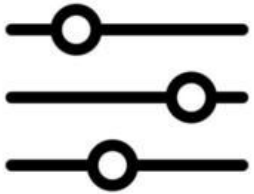
Partitioning Strategy

Choose the appropriate way to start the table partitioning from less to more granular, according to the most frequent access pattern



Row-level Mutation Strategy

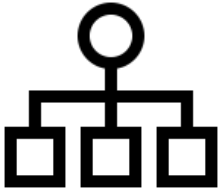
According to the frequency of updates and deleted transactions, the table will require a choice of COW or MOR row-level mutation strategy



Access Pattern

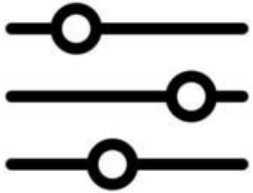
Define the proper metric mode for which table's columns

TABLE MAINTENANCE



Revisit Partitioning Strategy

Review patterns of how data consumers are accessing and querying the table. Consider updating the partitioning strategy to improve query performance

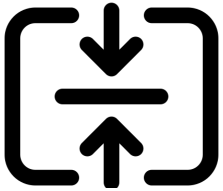


Revisit Access Pattern

Analyse if the access pattern changed, so update the metric mode for the impacted table's columns.

Snapshot Expiration

If snapshots are no longer required, consider deleting them, saving storage space and simplifying metadata files.



Data Compaction

Run file compactions more or less frequently. To keep performant data access, MOR row-level mutation requires more frequent compactions than COW.

INSERT OVERWRITE VS. MERGE INTO

With Iceberg v2 format support (row-level deletes / updates), for **slow-changing** tables' **change data capture (CDC)**, **MERGE INTO** is recommended instead of INSERT OVERWRITE as **MERGE INTO** can replace only the affected data files instead of partitions.

SQL MERGE FOR CDC IN SCD TABLES

	SQL Examples
HIVE	MERGE INTO <target table> AS T USING <source expression/table> AS S ON <boolean expression1> WHEN MATCHED [AND <boolean expression2>] THEN UPDATE SET <set clause list> WHEN MATCHED [AND <boolean expression3>] THEN DELETE WHEN NOT MATCHED [AND <boolean expression4>] THEN INSERT VALUES <value list>
SPARK	MERGE INTO <target table> t USING <source expression/table> s ON <boolean expression1> WHEN MATCHED [AND <boolean expression2>] THEN UPDATE SET <set clause list> WHEN MATCHED [AND <boolean expression3>] THEN DELETE WHEN NOT MATCHED [AND <boolean expression4>] THEN INSERT VALUES <value list>

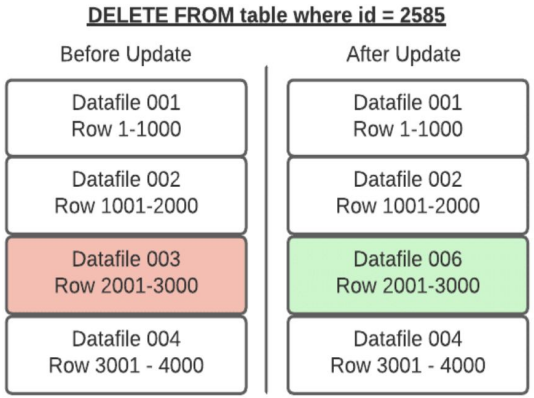
Row-Level Changes on the Lakehouse: COW vs MOR

Iceberg table has two different table formats v1 & v2.

- **v1** format — defaults **copy-on-write**
- **v2** format — **copy-on-write** or **merge-on-read.**

Copy-on-Write (COW)

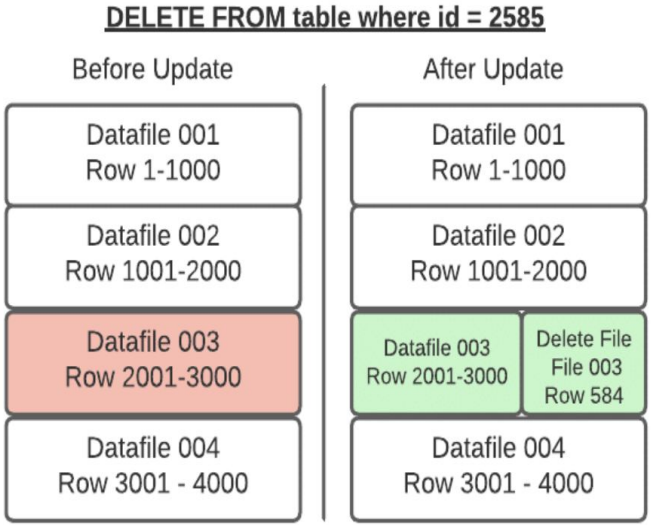
- With COW, when a change is made to delete or update a particular row or rows, the datafiles with those rows are duplicated, but the new version has the updated rows.
- This makes writes slower depending on how many data files must be re-written.
- If updating a large number of rows, COW is ideal.
- However, if updating just a few rows you still have to rewrite the entire data file, making small or frequent changes expensive.



Summary of COW (Copy-On-Write)	
PROS	CONS
Fastest reads	Expensive writes
Good for infrequent updates/deletes	Bad for frequent updates/deletes

Merge-on-Read (MOR)

- With merge-on-read, **the file is not rewritten, instead the changes are written to a new file.**
- Then when the data is read, **the changes are applied or merged to the original data file** to form the new state of the data during processing.
- This makes writing the changes much quicker, but also means more work must be done when the data is read.
- If you **delete a row**, it gets added to **a delete file** and reconciled on each subsequent read till the files undergo compaction.
- If you **update a row**, that row is tracked **via a delete file** so future reads ignore it from the old data file and the updated row is added to a new data file.
- Again, once **compaction** is run, **all the data will be in fewer data files and the delete files will no longer be needed.**



When to Use COW and When to Use MOR

Approach	Update/Delete Frequency	Insert Performance	Update/Delete Latency / Performance	Read Latency/ Performance
Copy-On-Write	Infrequent updates/deletes	Same	Slowest	Fastest
Merge-On-Read Position Deletes	Frequent updates/deletes	Same	Fast	Fast
Merge-On-Read Equality Deletes	Frequent updates/deletes where position delete MOR is still not fast enough on the write side	Same	Fastest	Slowest

Table Maintenance with CDE / CML (Spark 3 SQL)

Table Maint.	Examples
Snapshot Management	// **Remove snapshots** older than one day, but retain the last 100 snapshots spark.sql("CALL <catalog>.system.expire_snapshots('db.sample',TIMESTAMP '2021-06-30 00:00:00.000', 100)") // **Rollback** a table to specific snapshot ID spark.sql("CALL <catalog>.system.rollback_to_snapshot('db.sample',12345)") // **Rollback** a table to one day spark.sql("CALL <catalog>.system.rollback_to_timestamp('db.sample', TIMESTAMP '2021-06-30 00:00:00.000')")
Compaction	// **Rewrite the data files** in db.sample using the default rewrite // algorithm of bin-packing to combine small files and also split large // files according to the default write size of the table spark.sql("CALL <catalog>.system.rewrite_data_files('db.sample')")

Expiring old snapshots removes them from metadata, so they are no longer available for time travel operations. Data files are not deleted until they are no longer referenced by a snapshot that may be used for time travel. Regularly expiring snapshots deletes unused data files.

Table Maintenance supported by CDE / CML Spark 3 SQL

	CDE / CML Spark 3 SQL
Expire Snapshots	●
Remove old metadata files	●
Delete orphan files	●
Compact data files	●
Rewrite manifests	●

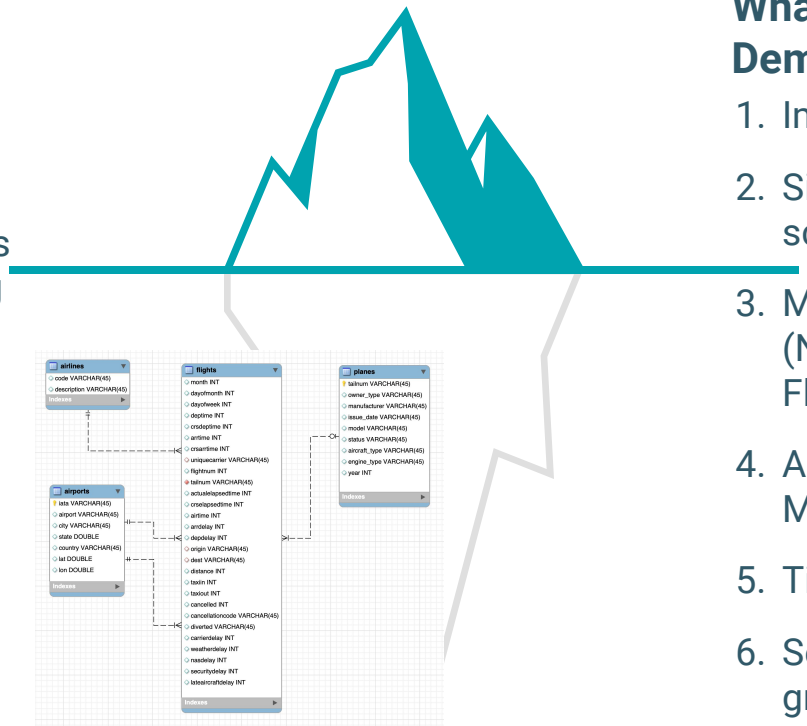
Lakehouse Hands-on Lab

OVERVIEW OF DEMO

Try it out, what could it hurt?????

Airline Logistics Analytics Data Lakehouse

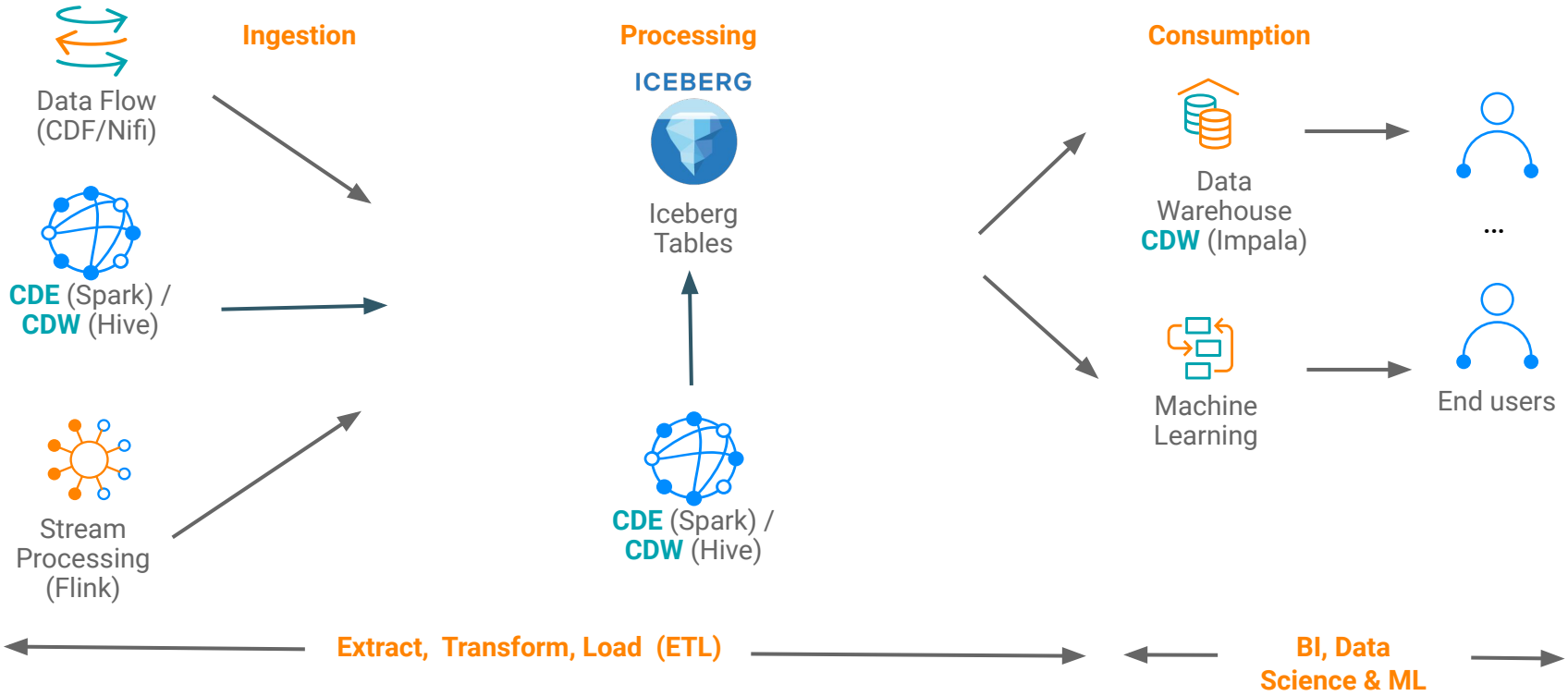
- Concerns with Flight Delays & Cancellations
- Data volumes growing fast, stressing current solution(s)
- Stream in additional data - Fare data to augment analytics
- Need to prepare data for AI use cases



What to Expect in the Demo

1. Improved performance
2. Simplified management - schema/partition evolution
3. Multi-function analytics (NiFi, Spark, Impala, Flink/SSB)
4. ACID - Insert, Update, & Merge
5. Time Travel
6. Security & Lineage - incl. fine grained access control

Open Data Lakehouse Reference Architecture



Handy Links

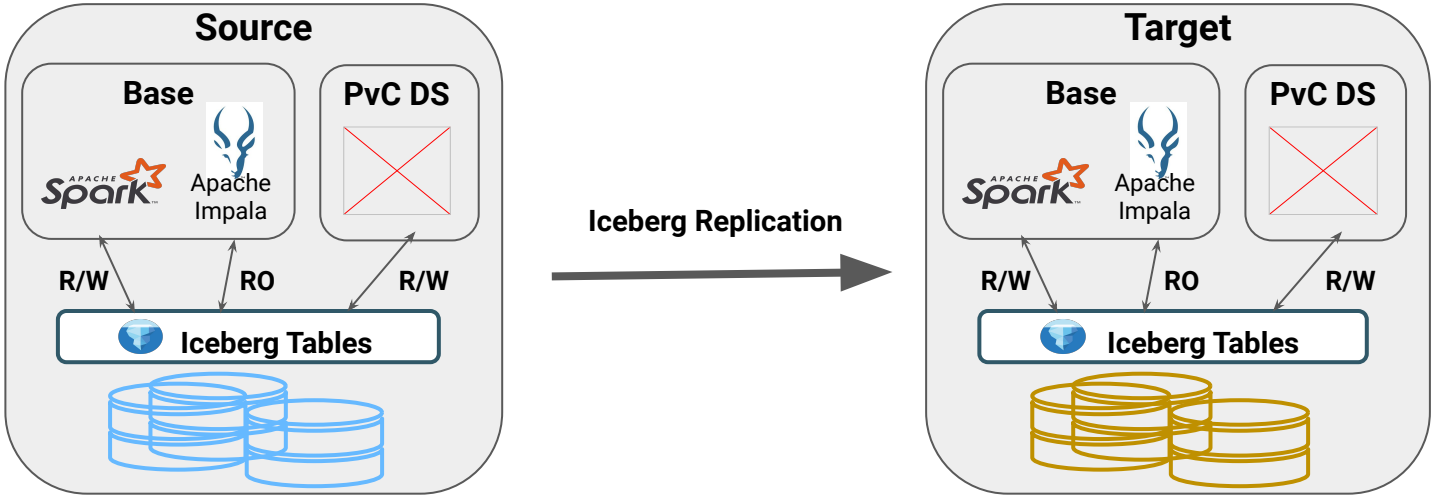
- Hive Iceberg Cheatsheet
- Impala Iceberg Cheatsheet for 719
- Cloudera Open Data Lakehouse - Internal FAQ [DRAFT]
- Cloudera Open Data Lakehouse Private Cloud - Train the Trainer HOL

Q&A

Thank You

Iceberg Replication

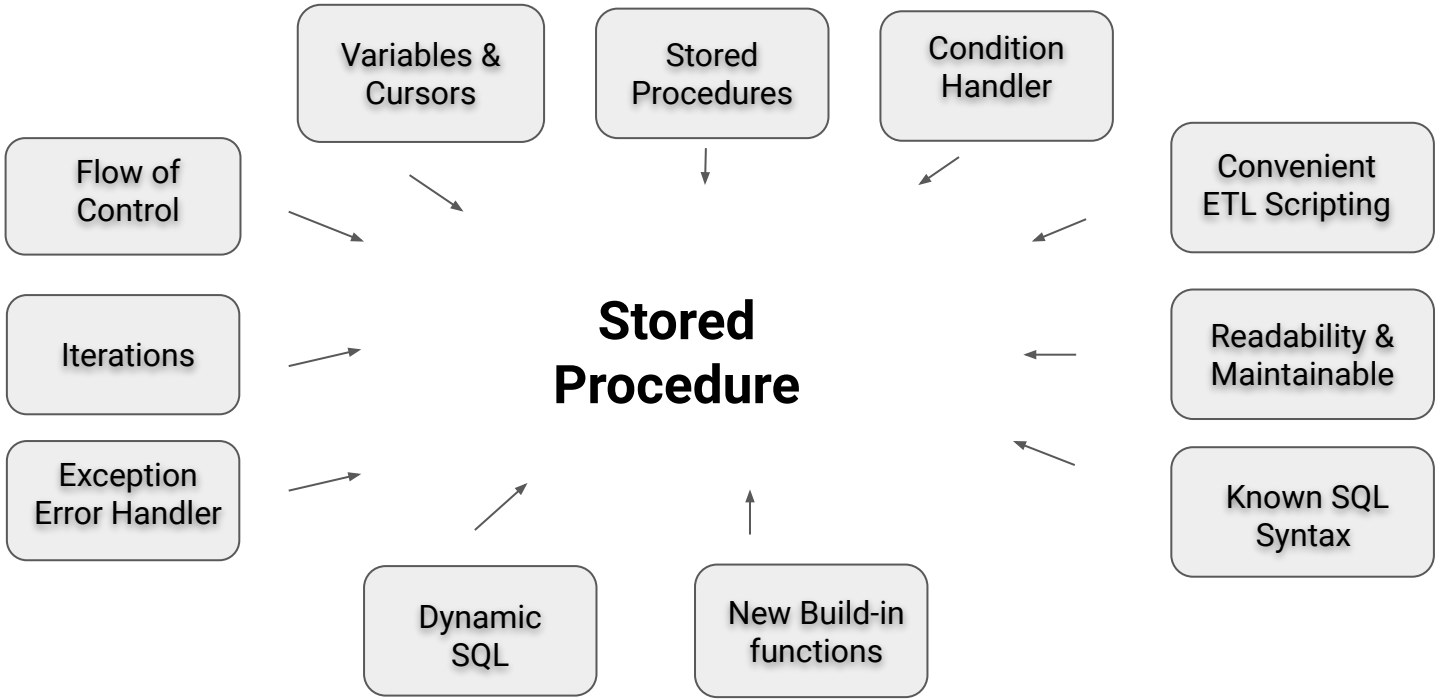
Iceberg Replication



- Bootstrap and Incremental Replication Support
- Iceberg Version V1/V2 support
- Table level replication scope with inclusion and exclusion filters

CDW Stored Procedure - HPL/SQL

CDW Stored Procedures - HPL/SQL

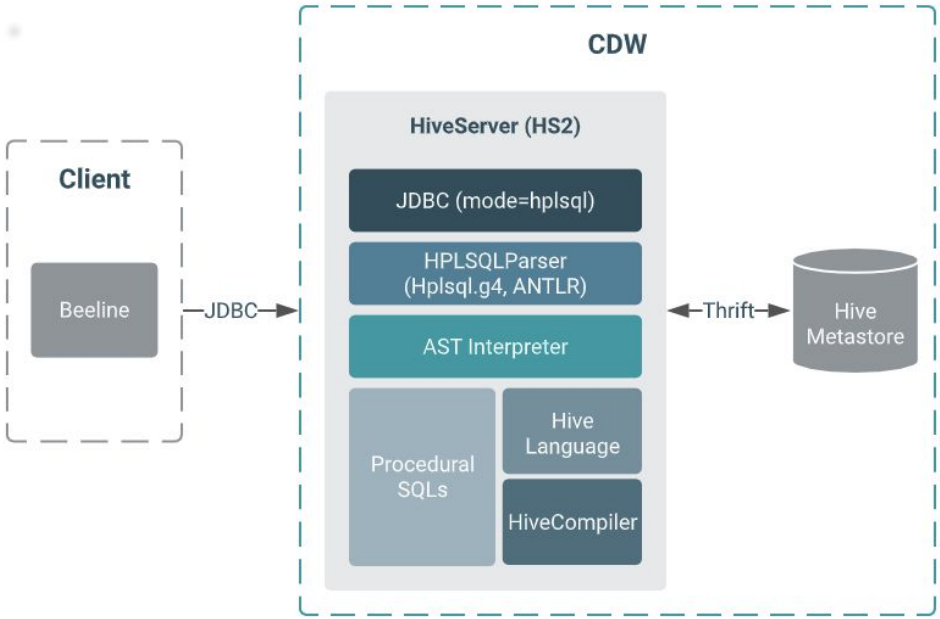


Stored Procedure Architecture

CDW Stored Procedure has been re-architected from a command line tool to an **integrated part of HiveServer (HS2).**

The **interpreter** executes the abstract syntax tree (AST) from the parser.

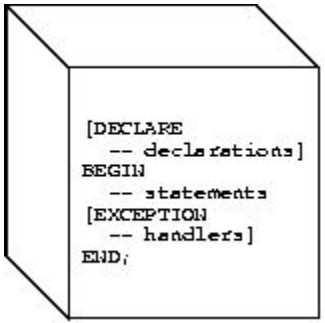
Hive metastore securely stores the function and procedure code permanently.



The procedure is loaded and cached on demand to the interpreter's memory when needed.

Procedural Language & SQL

CDW Stored Procedure is (partial) compatible with Oracle PL/SQL, IBM DB2, MySQL and Teradata.



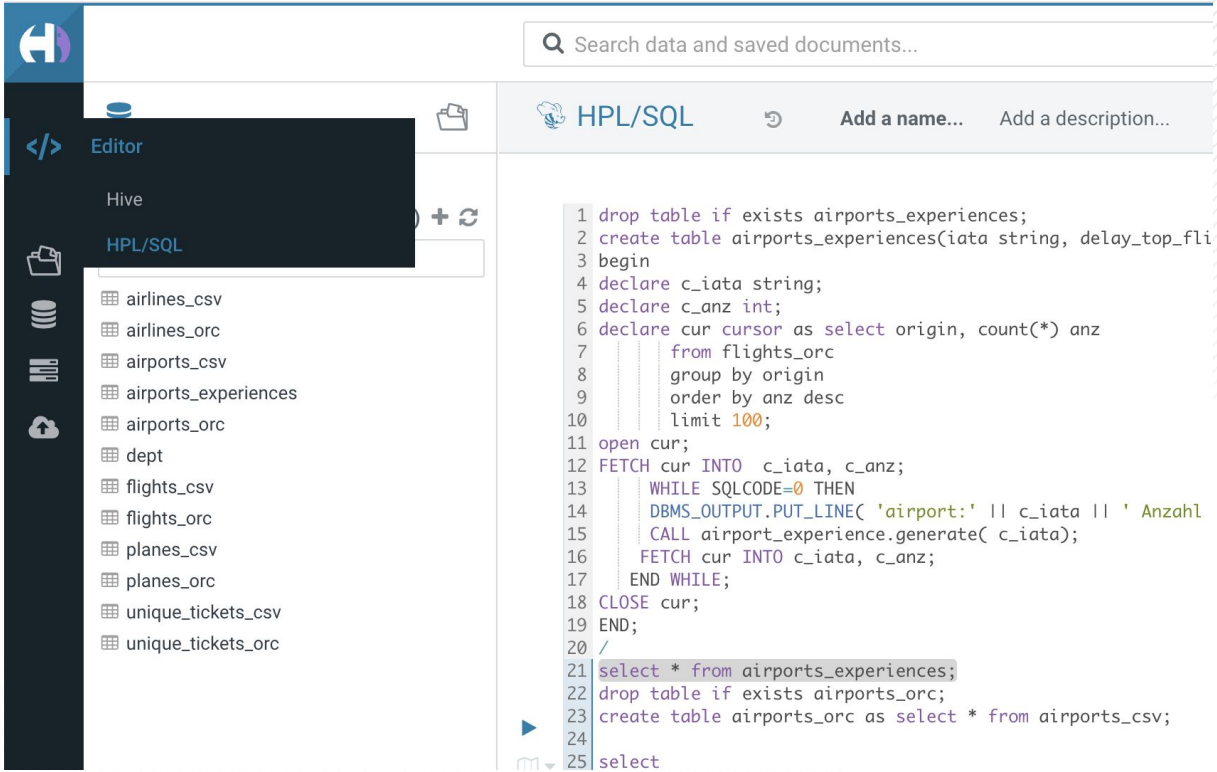

```
create procedure hello(name STRING)
begin
  print 'Hello ' || name;
end;

call hello ('World!');
```

```
declare row_cnt    int = 0;
declare rows       flights_csv%rowtype;
declare cur cursor for SELECT * FROM flights_csv;

OPEN cur;
FETCH cur INTO rows;
WHILE SQLCODE=0 THEN
  row_cnt = row_cnt + 1;
  FETCH cur INTO rows;
END WHILE;
CLOSE cur;

PRINT 'row count ' || row_cnt;
```


Search data and saved documents...
Editor
Hive
HPL/SQL
HPL/SQL
Add a name...
Add a description...
airlines_csv
airlines_orc
airports_csv
airports_experiences
airports_orc
dept
flights_csv
flights_orc
planes_csv
planes_orc
unique_tickets_csv
unique_tickets_orc
1 drop table if exists airports_experiences;
2 create table airports_experiences(iata string, delay_top_fli
3 begin
4 declare c_iata string;
5 declare c_anz int;
6 declare cur cursor as select origin, count(*) anz
7 from flights_orc
8 group by origin
9 order by anz desc
10 limit 100;
11 open cur;
12 FETCH cur INTO c_iata, c_anz;
13 WHILE SQLCODE=0 THEN
14 DBMS_OUTPUT.PUT_LINE('airport:' || c_iata || ' Anzahl
15 CALL airport_experience.generate(c_iata);
16 FETCH cur INTO c_iata, c_anz;
17 END WHILE;
18 CLOSE cur;
19 END;
20 /
21 select * from airports_experiences;
22 drop table if exists airports_orc;
23 create table airports_orc as select * from airports_csv;
24
25 select