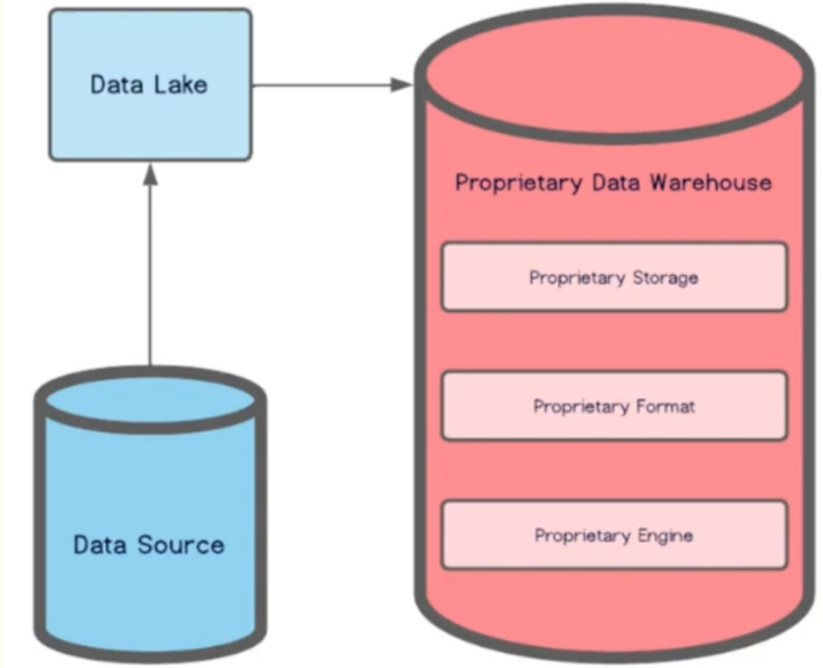
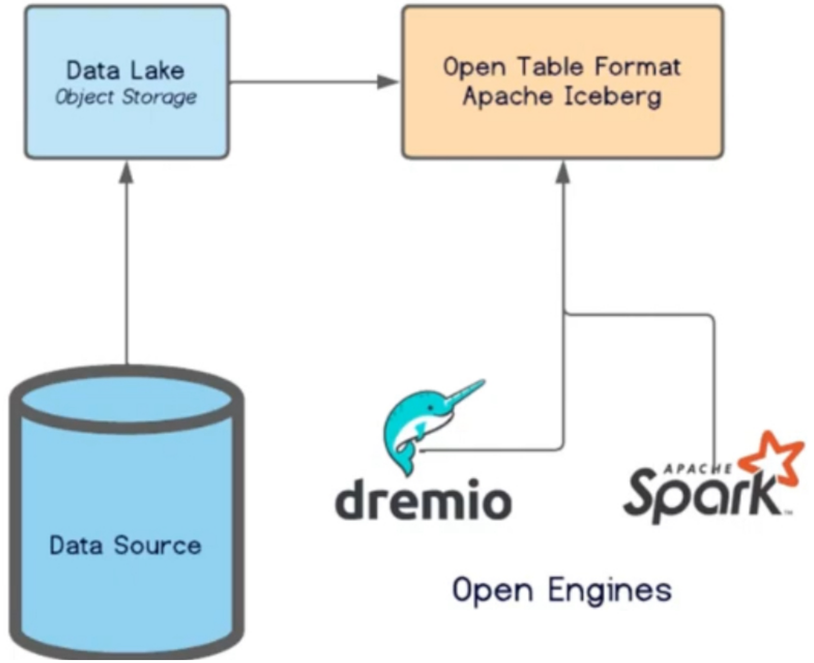
Lakehouses and Tables

Rahul Dave(@rahuldave), Univ.Ai

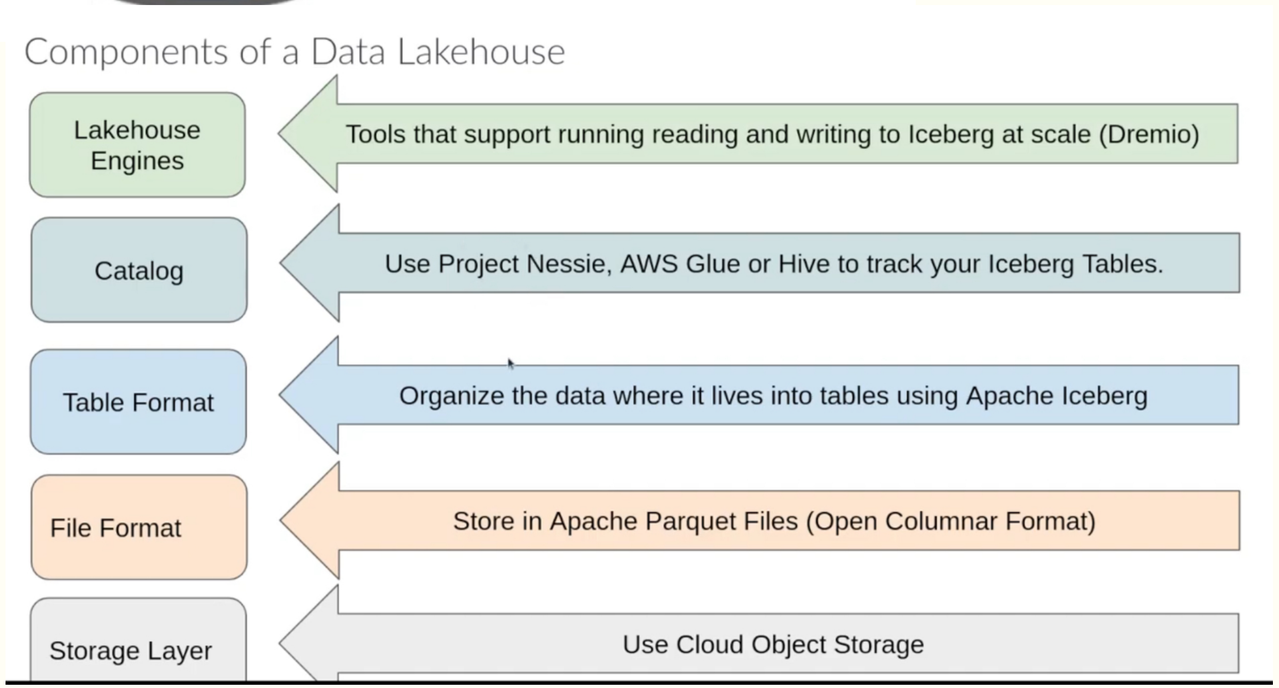
Univ.AI



Warehouse



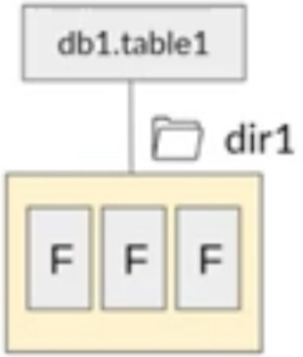
Open Lakehouse



What's the trouble with the Hive Data Format?

Hive table format

- A table's contents is all files in that table's directories
- The old de-facto standard



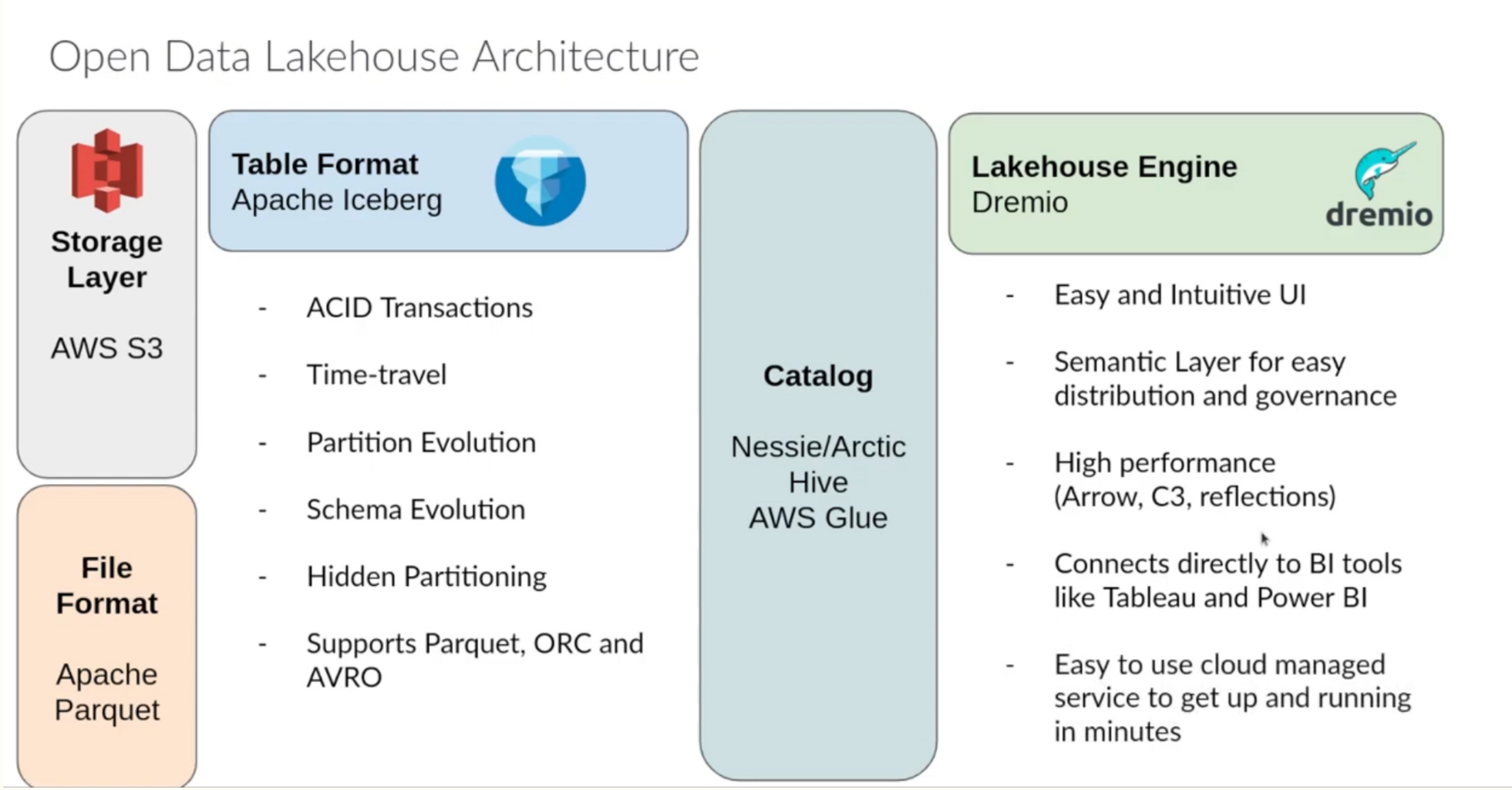
Pros

- Works with basically every engine since it's been the de-facto standard for so long
- More efficient access patterns than full-table scans for every query
- File format agnostic
- Atomically update a whole partition
- Single, central answer to "what data is in this table" for the whole ecosystem

Cons

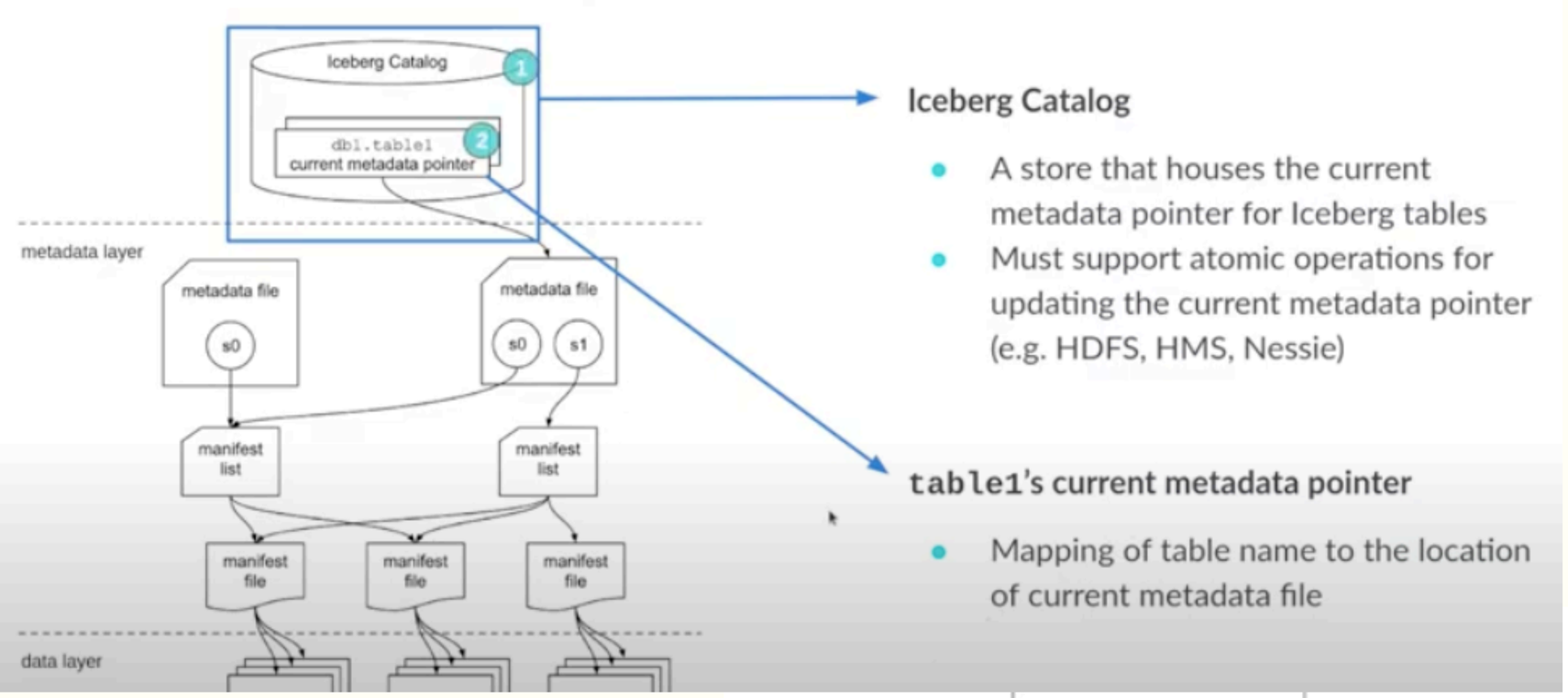
- Smaller updates are very inefficient
- No way to change data in multiple partitions safely
- In practice, multiple jobs modifying the same dataset don't do so safely
- All of the directory listings needed for large tables take a *long* time
- Users have to know the physical layout of the table

An example of a Lakehouse



Iceberg tables

Similar systems exist in Apache Hudi and Databricks delta-lake.



Iceberg Catalog

- A store that houses the current metadata pointer for Iceberg tables
- Must support atomic operations for updating the current metadata pointer (e.g. HDFS, HMS, Nessie)

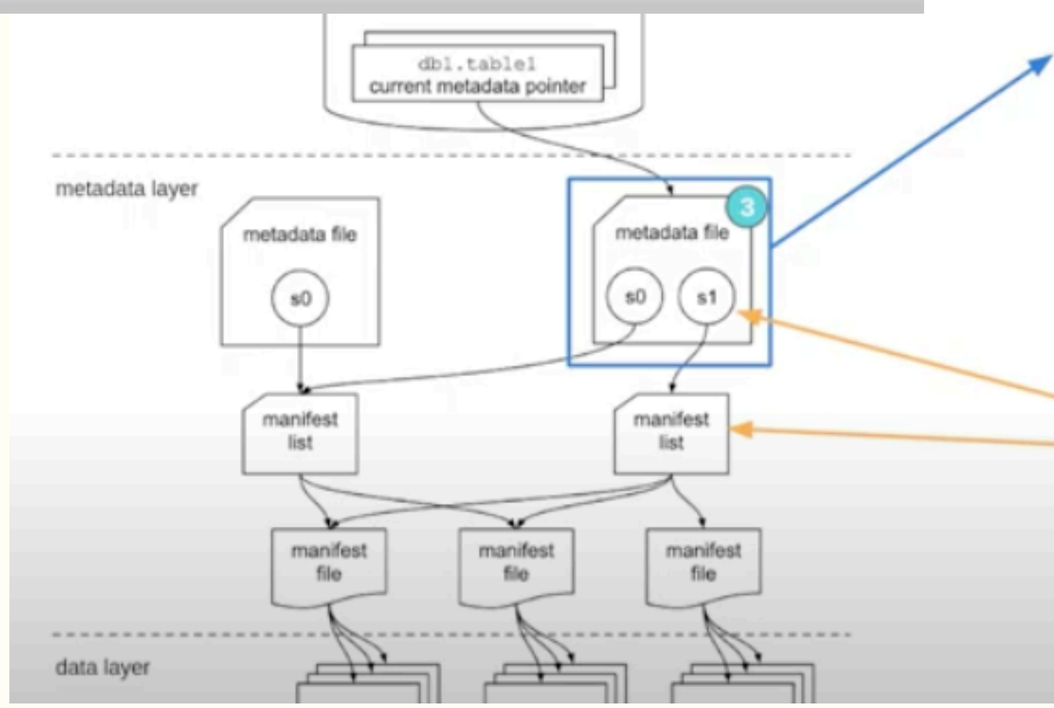
`table1`'s current metadata pointer

- Mapping of table name to the location of current metadata file

Components of a Lakehouse

- Catalog: an external database, or a hive metastore
- in some cases provides multi-process transactional guarantees
- keeps a pointer to the current metadata file

- metadata is arranged on a per snapshot basis
- consists of location and schema information, and snapshots
- keeps a pointer to the current snapshot

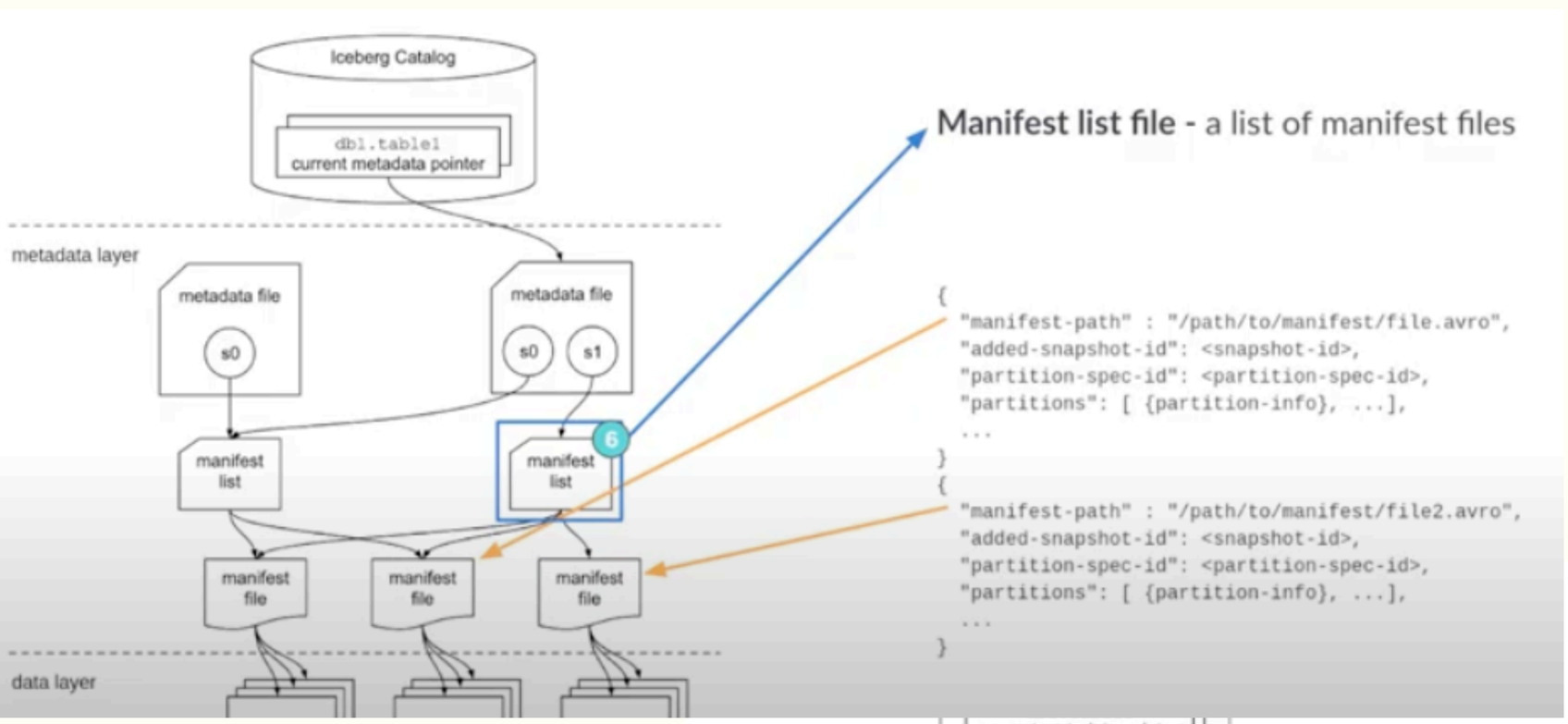


Metadata file - stores metadata about a table at a certain point in time

```
{
  "table-uuid" : "<uuid>",
  "location" : "/path/to/table/dir",
  "schema": {...},
  "partition-spec": [ {<partition-details>}, ...],
4 "current-snapshot-id": <snapshot-id>,
  "snapshots": [ {
      "snapshot-id": <snapshot-id>
5     "manifest-list": "/path/to/manifest/list.avro"
  }, ...],
  ...
}
```

Manifests

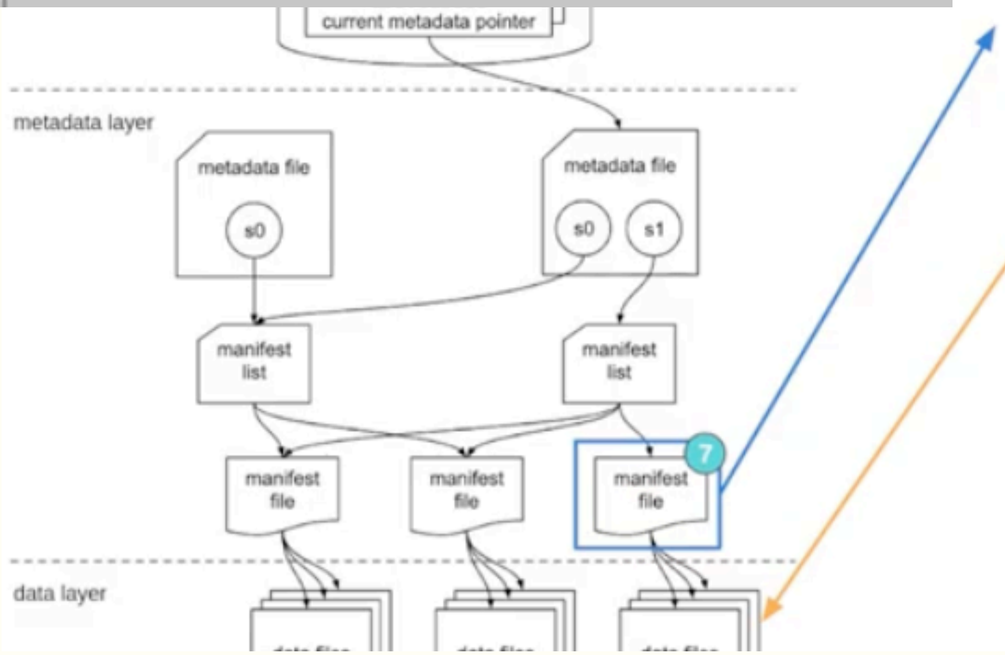
- metadata file points to a manifest list file
- the manifest list file has a set of manifest files



Manifest list file - a list of manifest files

```
{
  "manifest-path" : "/path/to/manifest/file.avro",
  "added-snapshot-id": <snapshot-id>,
  "partition-spec-id": <partition-spec-id>,
  "partitions": [ {partition-info}, ...],
  ...
}
{
  "manifest-path" : "/path/to/manifest/file2.avro",
  "added-snapshot-id": <snapshot-id>,
  "partition-spec-id": <partition-spec-id>,
  "partitions": [ {partition-info}, ...],
  ...
}
```

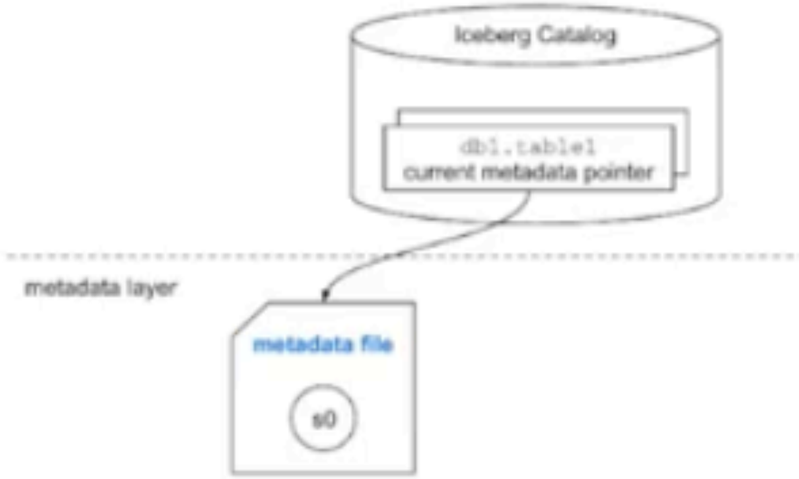
- each manifest file has a list of parquet, orc or avro files
- the manifest file also keeps statistics about the columns in the schema

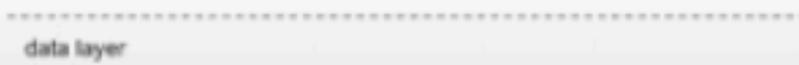


Manifest file - a list of data files, along with details and stats about each data file

```
{
  "data-file": {
    "file-path": "/path/to/data/file.parquet",
    "file-format": "PARQUET",
    "partition": {"<part-field>":{"<data-type>":<value>}},
    "record-count": <num-records>,
    "null-value-counts": [{
      "column-index": "1", "value": 4
    }, ...],
    "lower-bounds": [{
      "column-index": "1", "value": "aaa"
    }, ...],
    "upper-bounds": [{
      "column-index": "1", "value": "eee"
    }, ...],
  }
  ...
}
```

```
✅ CREATE TABLE db1.table1 (
        order_id bigint,
        customer_id bigint,
        order_amount DECIMAL(10, 2),
        order_ts TIMESTAMP
    )
    USING iceberg
    PARTITIONED BY ( hour(order_ts) );
```

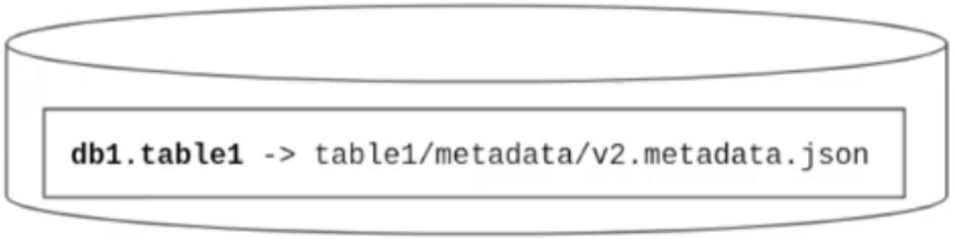




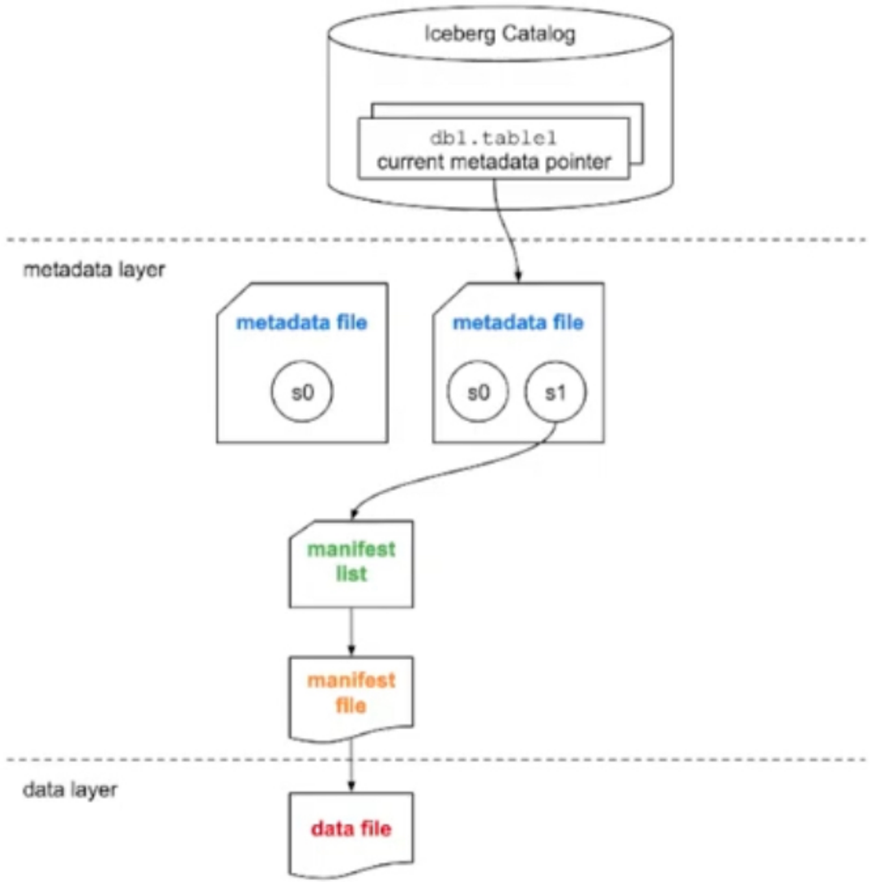

```
table1/
 |- metadata/
 |    |- v1.metadata.json
 |- data/
```

The first thing we do is create a schema. The schema is written into a json file. The metadata pointer is then pointed to it.

```
✅ INSERT INTO db1.table1 VALUES (
        123,
        456,
        36.17,
        '2021-01-26 08:10:23'
   );
```



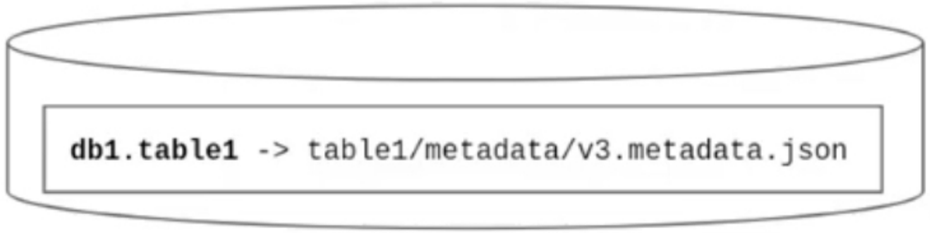

```
table1/
 |- metadata/
 |    |- v1.metadata.json
 |    |- v2.metadata.json
 |    |- snap-2938-1-4103.avro
 |    |- d8f9-ad19-4e.avro
 |- data/
 |    |- order_ts_hour=2021-01-26-08/
 |    |    |- 00000-5-cae2d.parquet
```



Now we insert. This creates a new snapshot with a manifest list with a single manifest file. The tables values are stored in a hive-like partitioned format. A new metadata file is created for the transaction and the metadata pointer updated.

Univ.AI

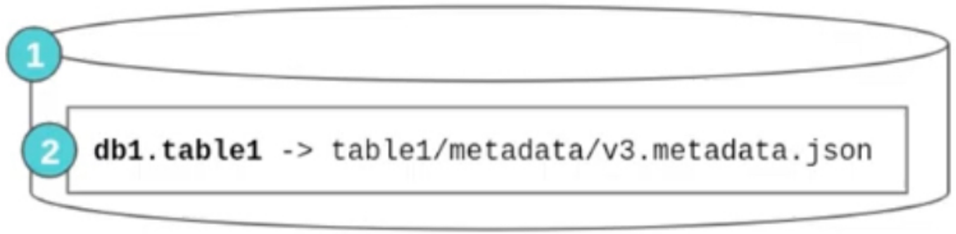
✅
```
MERGE INTO db1.table1
USING ( SELECT * FROM table1_stage ) s
ON table1.order_id = s.order_id
WHEN MATCHED THEN UPDATE table1.order_amount = s.order_amount
WHEN NOT MATCHED THEN INSERT *
```



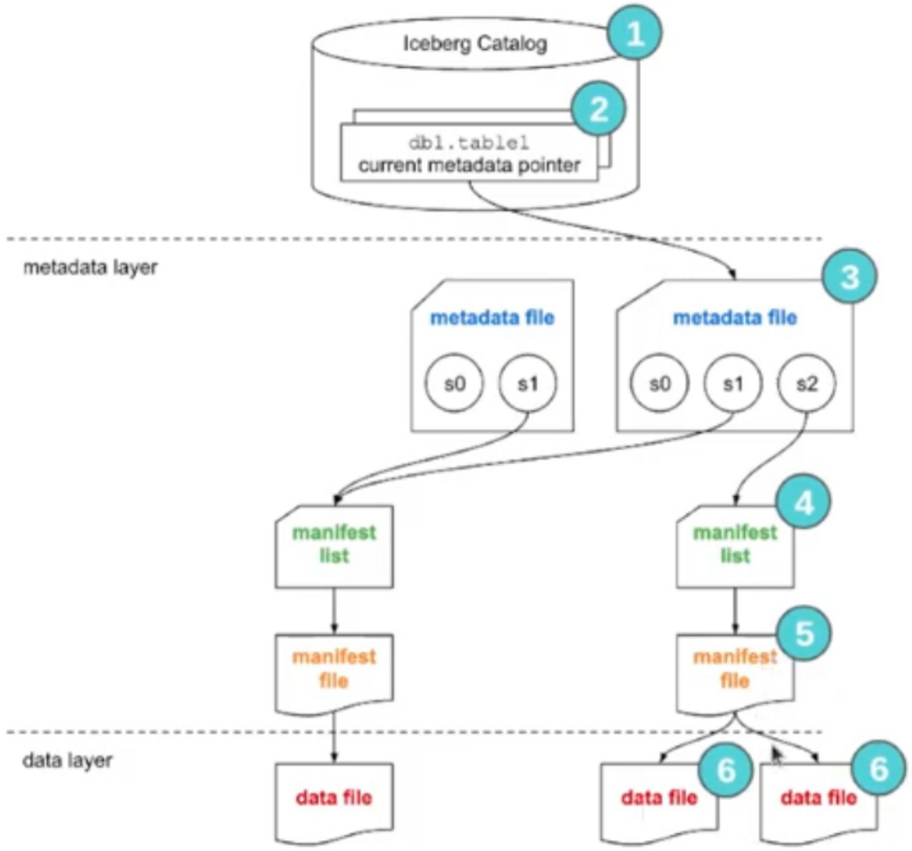

```
table1/
 |- metadata/
 |    |- v1.metadata.json
 |    |- v2.metadata.json
 |    |- v3.metadata.json
 |    |- snap-29c8-1-b103.avro
 |    |- snap-9fa1-3-16c3.avro
 |    |- d8f9-ad19-4e.avro
 |    |- 0d9a-98fa-77.avro
 |- data/
 |    |- order_ts_hour=2021-01-26-08/
 |    |    |- 00000-5-cae2d.parquet
 |    |    |- 00000-1-aef71.parquet
 |    |- order_ts_hour=2021-01-27-10/
 |    |    |- 00000-3-0fa3a.parquet
```

Now we are making a new transaction in which we merge from a staging table. This adds a new snapshot, with its own manifest list, new manifest file, and partitioned parquet files.

✅ SELECT *
FROM db1.table1

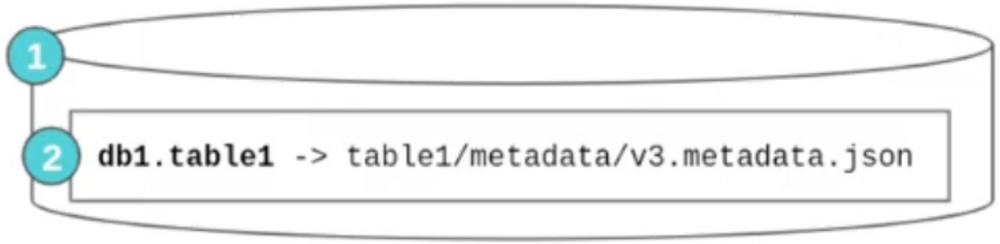



```
table1/
 |- metadata/
 |    |- v1.metadata.json
 |    |- v2.metadata.json
 |    |- v3.metadata.json
 |    |- snap-29c8-1-b103.avro
 |    |- snap-9fa1-3-16c3.avro
 |    |- d8f9-ad19-4e.avro
 |    |- 0d9a-98fa-77.avro
 |- data/
 |    |- order_ts_hour=2021-01-26-08/
 |    |    |- 00000-5-cae2d.parquet
 |    |    |- 00000-1-aef71.parquet
 |    |- order_ts_hour=2021-01-27-10/
 |    |    |- 00000-3-0fa3a.parquet
```

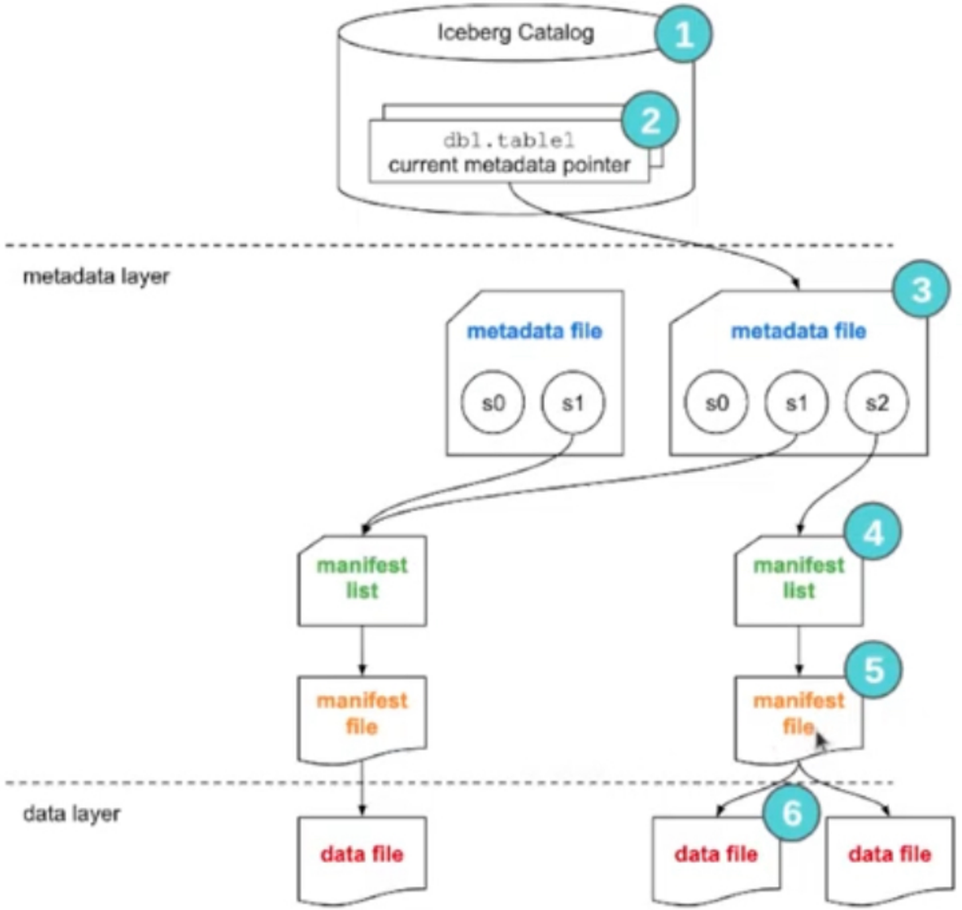


What if I now do a select? The select gets the latest metadata file, the latest snapshot, the latest manifest list, the appropriate manifest files, and the latest parquet files for this query.

✅
```
SELECT *
FROM db1.table1
WHERE order_ts = DATE '2021-01-26'
```




```
table1/
 |- metadata/
 |    |- v1.metadata.json
 |    |- v2.metadata.json
 |    |- v3.metadata.json
 |    |- snap-29c8-1-b103.avro
 |    |- snap-9fa1-3-16c3.avro
 |    |- d8f9-ad19-4e.avro
 |    |- 0d9a-98fa-77.avro
 |- data/
 |    |- order_ts_hour=2021-01-26-08/
 |    |    |- 00000-5-cae2d.parquet
 |    |    |- 00000-1-aef71.parquet
 |    |- order_ts_hour=2021-01-27-10/
 |    |    |- 00000-3-0fa3a.parquet
```

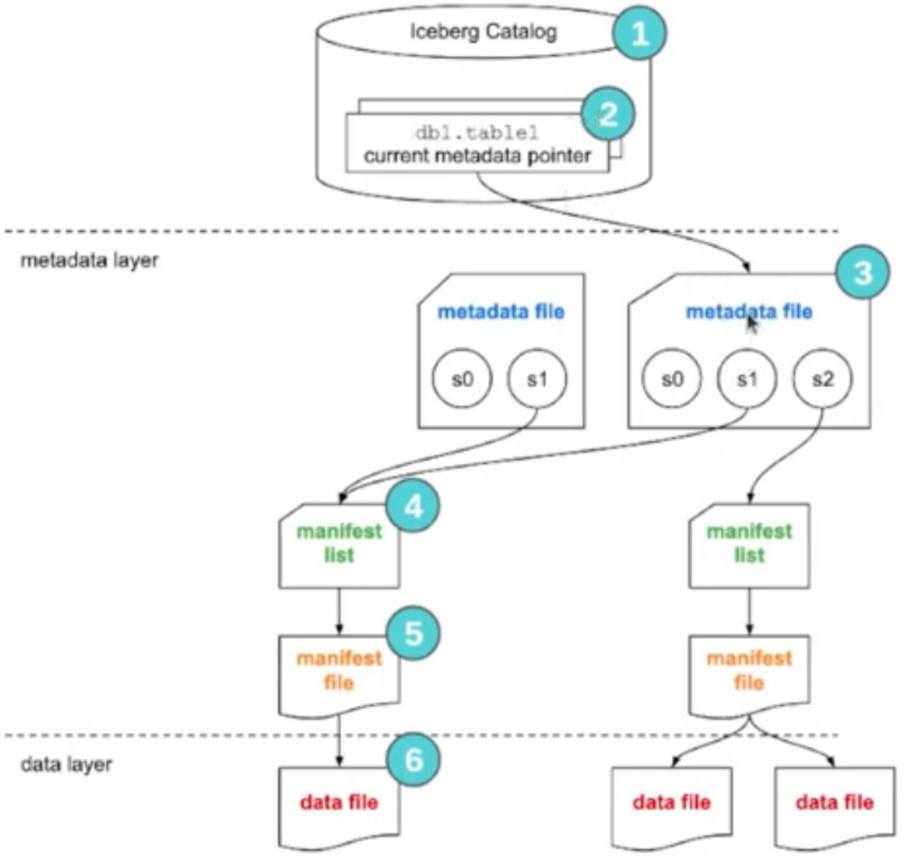


A select with a particular date will take me to the latest snapshot and in this snapshot give me the appropriate rows for that date.

✅
```
SELECT *
FROM db1.table1 AS OF '2021-05-26 09:30:00'
 -- (timestamp is from before MERGE INTO operation)
```

1
2 **db1.table1** -> table1/metadata/v3.metadata.json

```
table1/
 |- metadata/
 |    |- v1.metadata.json
 |    |- v2.metadata.json
 |    |- v3.metadata.json
 |    |- snap-29c8-1-b103.avro
 |    |- snap-9fa1-3-16c3.avro
 |    |- d8f9-ad19-4e.avro
 |    |- 0d9a-98fa-77.avro
 |- data/
 |    |- order_ts_hour=2021-01-26-08/
 |    |    |- 00000-5-cae2d.parquet
 |    |    |- 00000-1-aef71.parquet
 |    |- order_ts_hour=2021-01-27-10/
 |    |    |- 00000-3-0fa3a.parquet
```



A select "as of" is a different query. It is asking for a previous snapshot. So we need to locate that snapshot and get the information from the appropriate files there.

Catalogs

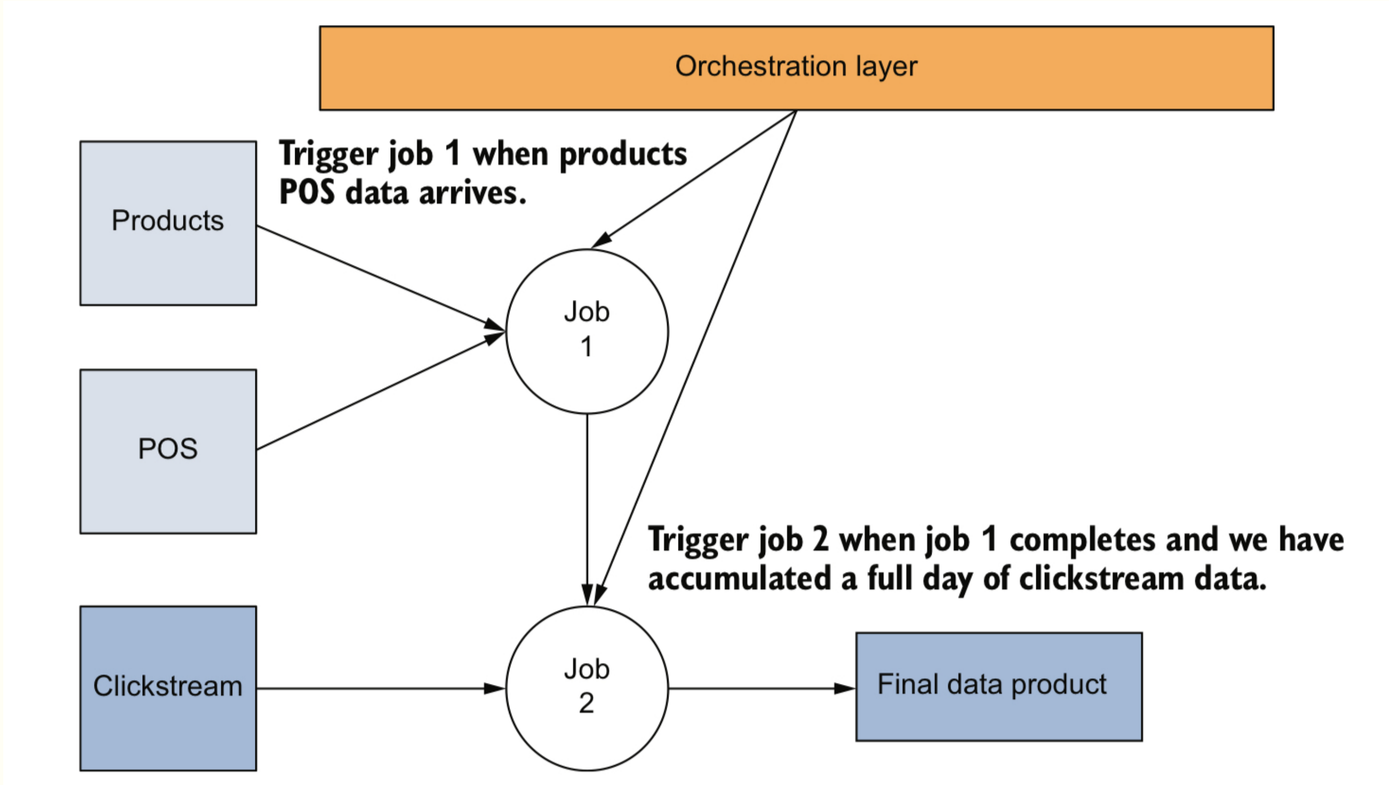
Type of Catalog	Pros	Cons
Project Nessie	- Git Like Functionality - Cloud Managed Service (Arctic)	- Support from engines beyond Spark & Dremio - If not using arctic must deploy and maintain Nessie server
Hive Metastore	- Can use existing Hive Metastore	- You have to deploy and maintain a hive metastore
AWS Glue	- Interop with AWS Services	- Support outside of AWS, Spark and Dremio

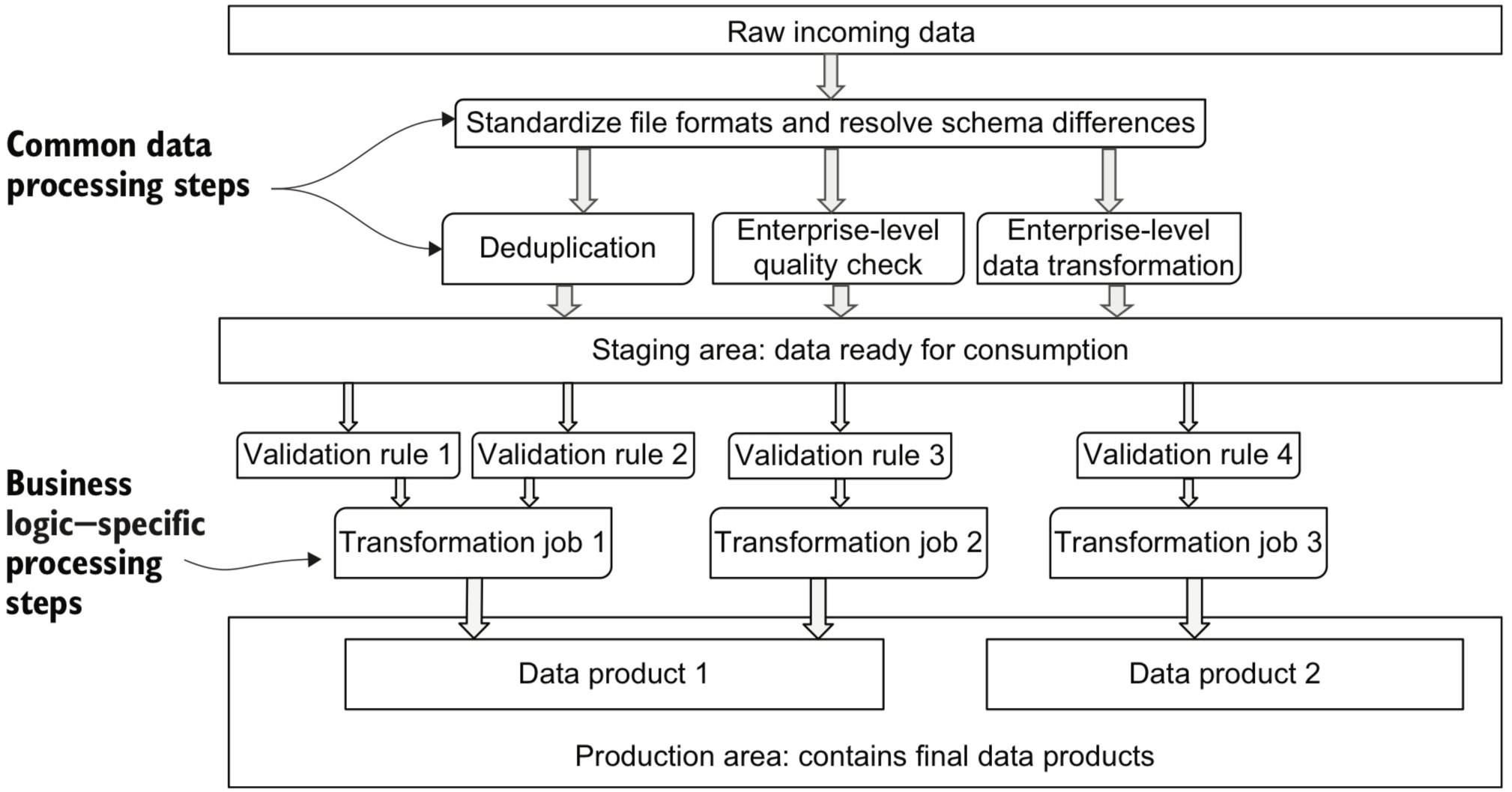
- HDFS, JDBC and REST catalogs also available

- Catalogs keep the metadata pointer
- they provide locking for ACID guarantees

Orchestration Systems

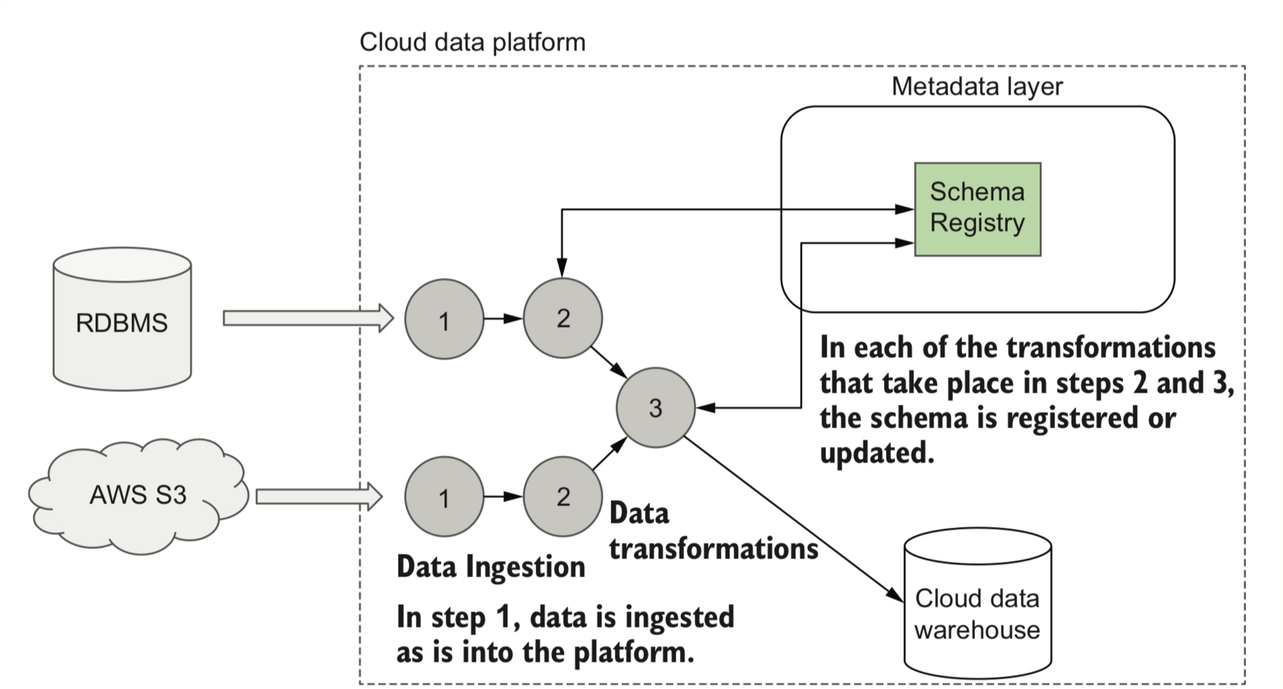
- These system run jobs in pipelines.
- Examples include airflow, prefect, and dagster. Some software also builds in its own DAG running.
- They provide schedules on which job runs and sensors which cause jobs to run
- Dagster supports Software Defined Assets, which auto-schedule jobs to make sure a particular final or intermediate data product is up-to-date at some cadence.

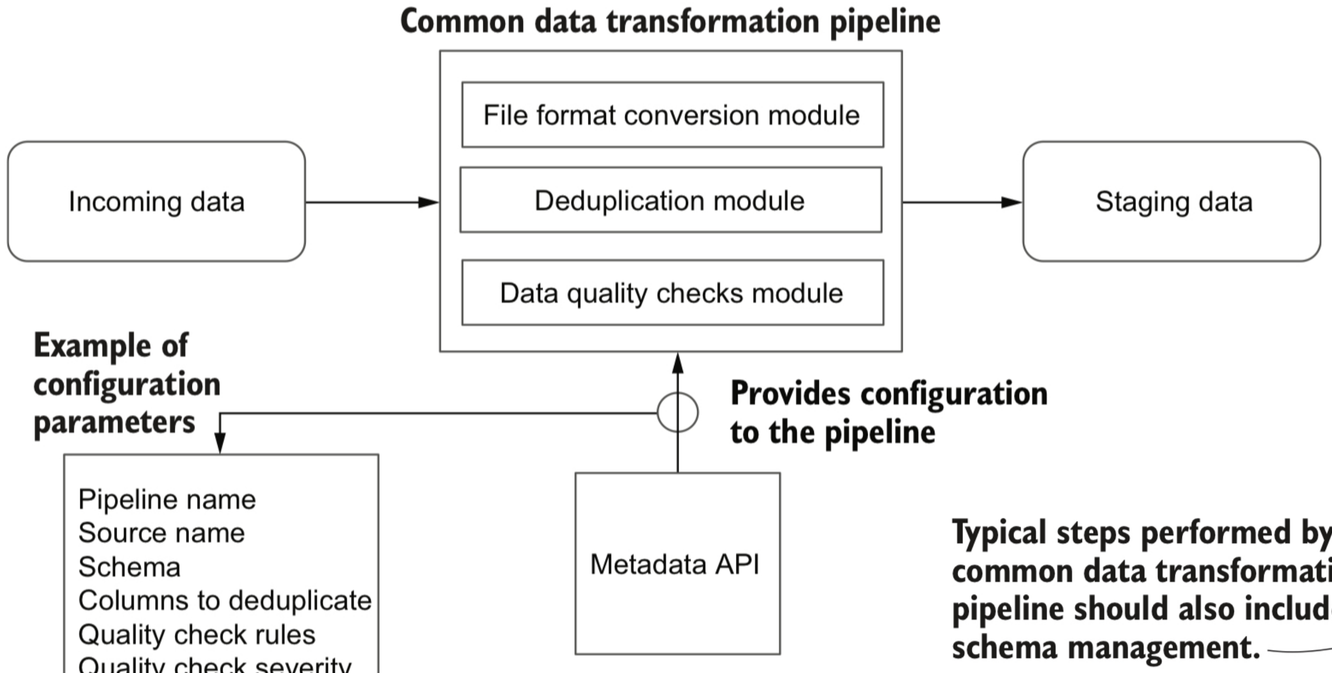


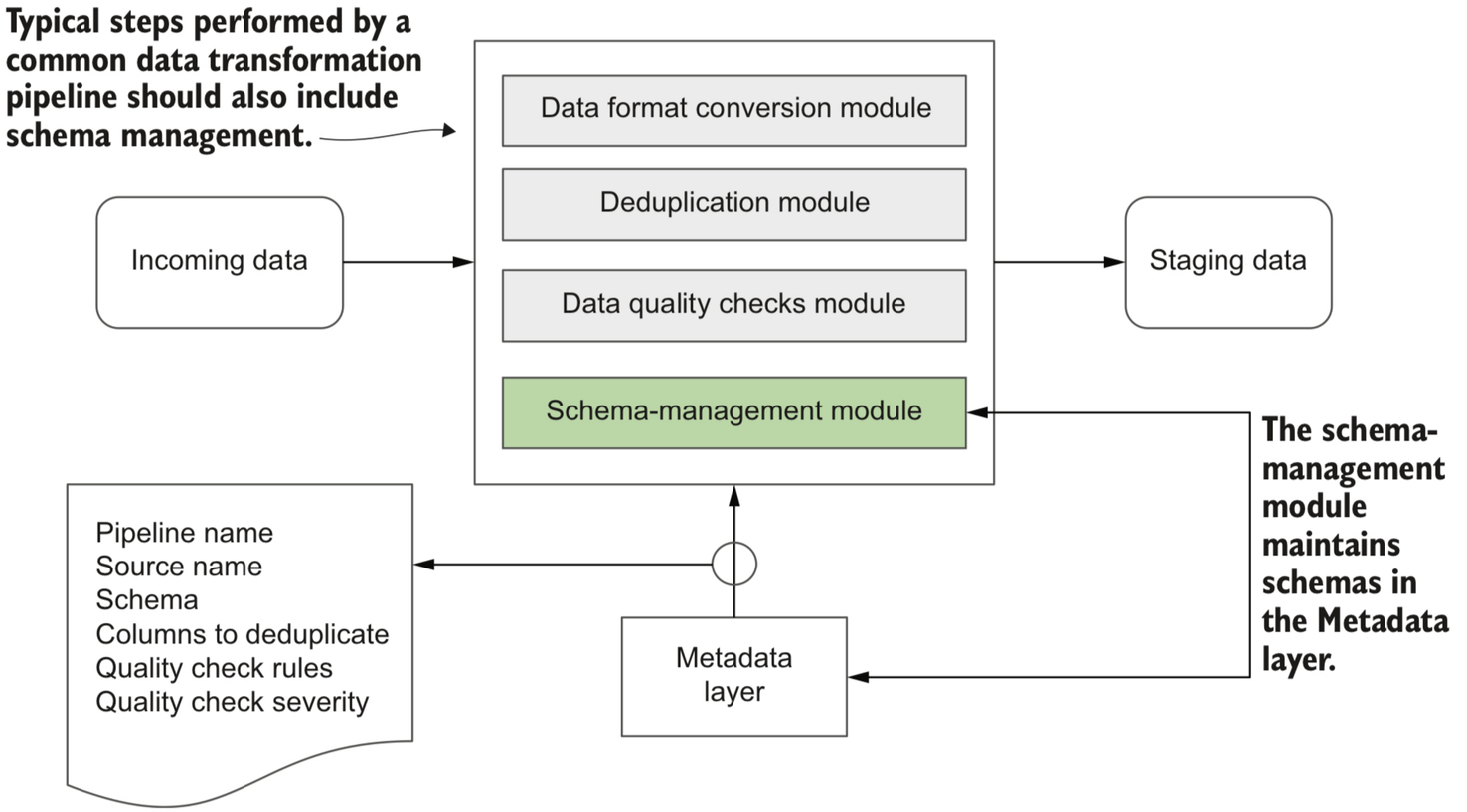

Raw incoming data
Common data processing steps
Standardize file formats and resolve schema differences
Deduplication
Enterprise-level quality check
Enterprise-level data transformation
Staging area: data ready for consumption
Validation rule 1
Validation rule 2
Validation rule 3
Validation rule 4
Business logic–specific processing steps
Transformation job 1
Transformation job 2
Transformation job 3
Data product 1
Data product 2
Production area: contains final data products

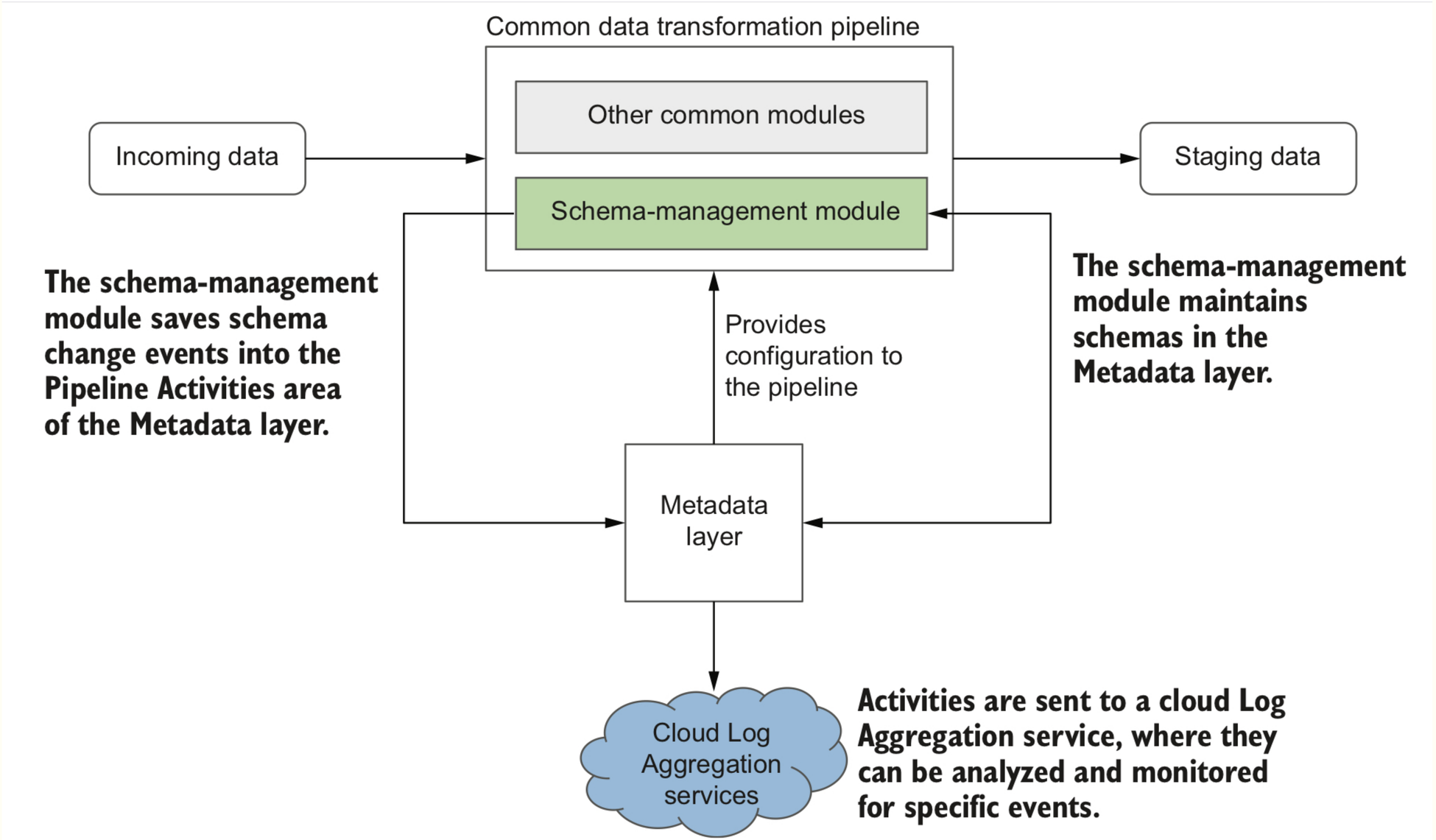
Metadata Layers

- Metadata is not just needed for the table file layer
- There needs to be operational metadata for the state of the ingest/transform/etc pipelines
- There needs to be metadata that tells us which tables exist where: an asset catalog. These lead to dataset registries.



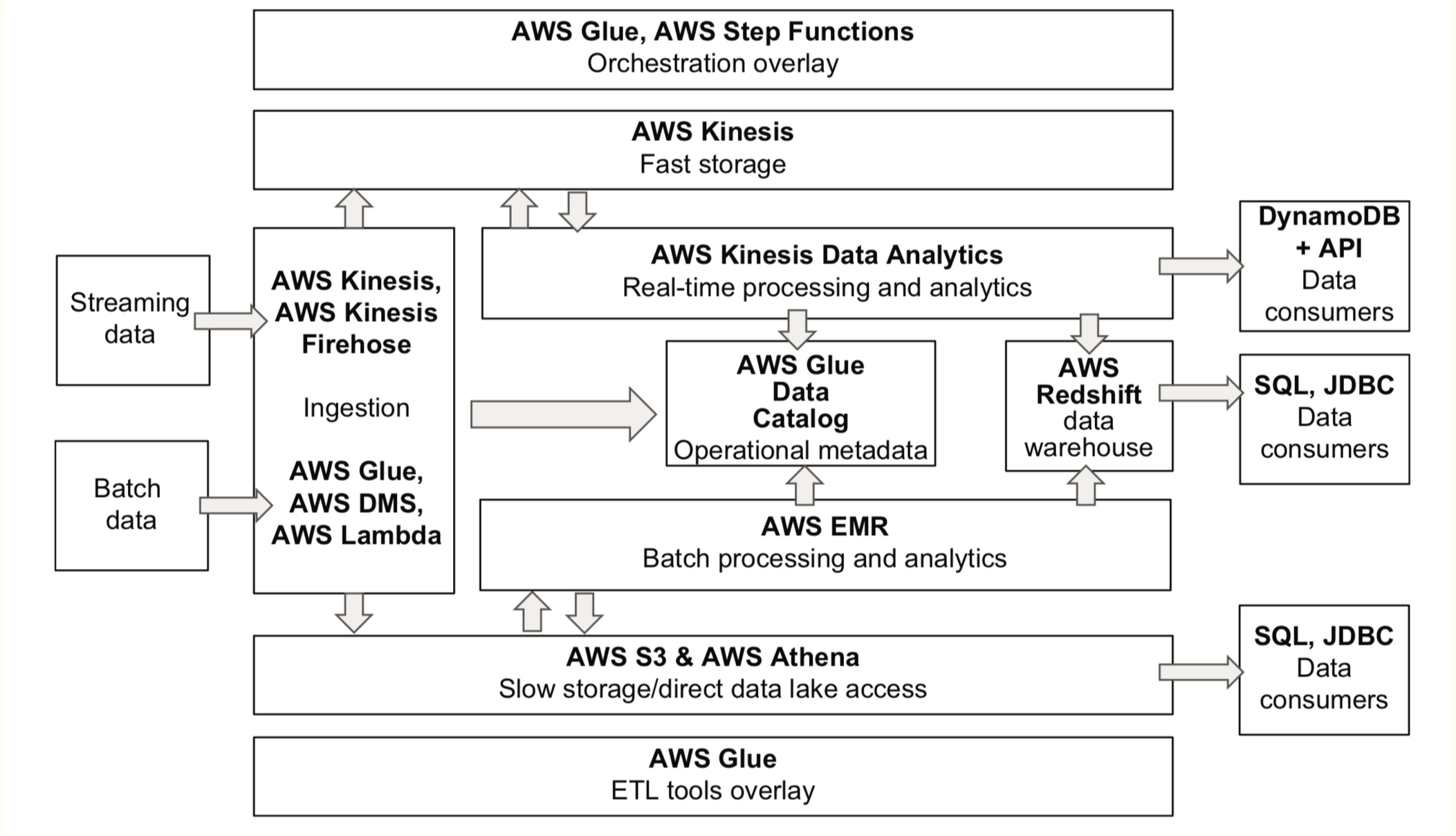

Common data transformation pipeline
Incoming data
File format conversion module
Deduplication module
Data quality checks module
Staging data
Example of configuration parameters
Pipeline name
Source name
Schema
Columns to deduplicate
Quality check rules
Quality check severity
Metadata API
Provides configuration to the pipeline


Typical steps performed by a common data transformation pipeline should also include schema management.
Incoming data
Data format conversion module
Deduplication module
Data quality checks module
Schema-management module
Staging data
Pipeline name
Source name
Schema
Columns to deduplicate
Quality check rules
Quality check severity
Metadata layer
The schema-management module maintains schemas in the Metadata layer.


Common data transformation pipeline
Other common modules
Incoming data
Staging data
Schema-management module
The schema-management module saves schema change events into the Pipeline Activities area of the Metadata layer.
Provides configuration to the pipeline
The schema-management module maintains schemas in the Metadata layer.
Metadata layer
Cloud Log Aggregation services
Activities are sent to a cloud Log Aggregation service, where they can be analyzed and monitored for specific events.

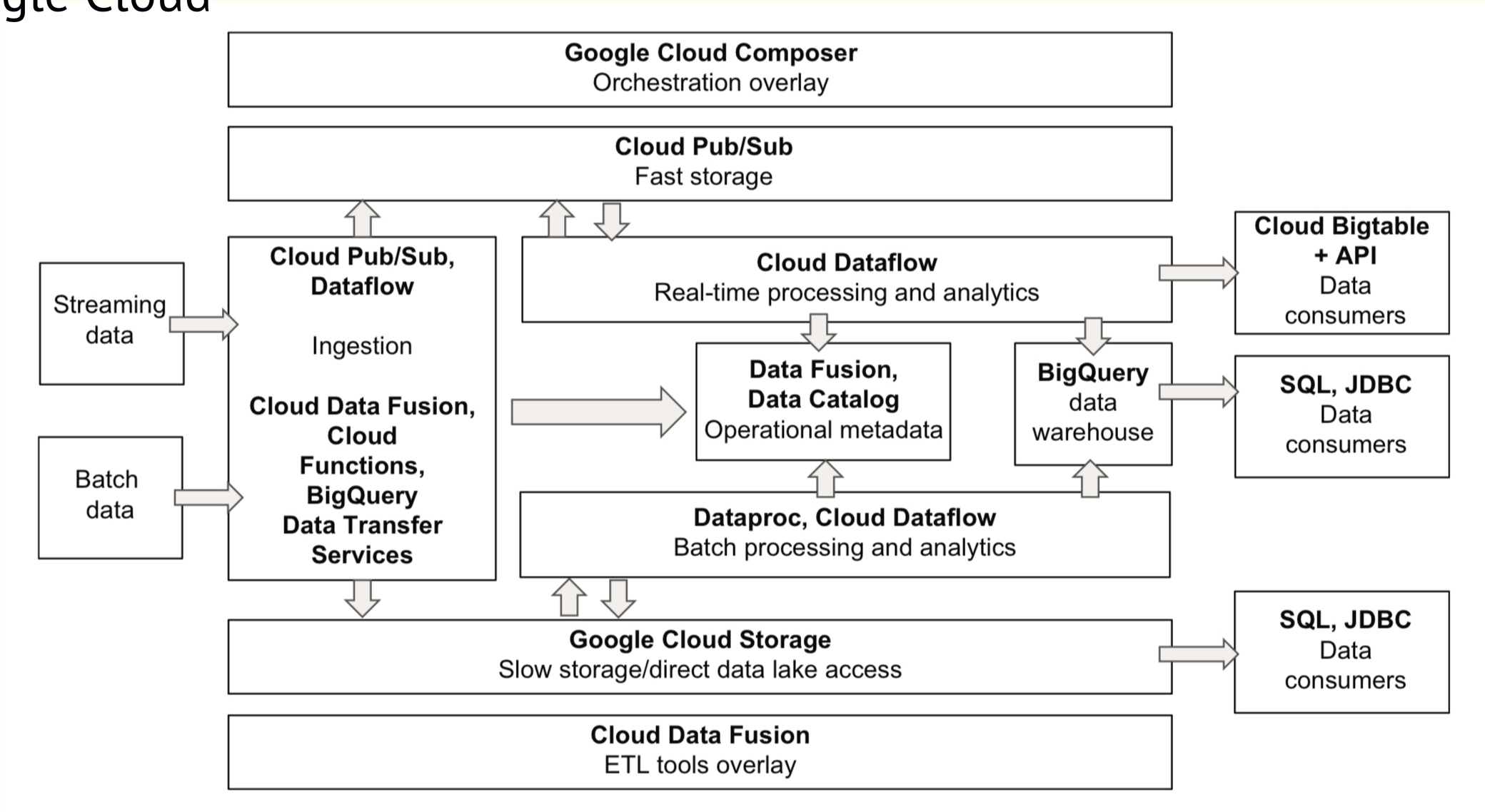
From Lakehouse to Platform

AWS



Univ.AI

Google Cloud



Azure