Open Table Formats

Understanding What it is and why it is needed

The Need for Open Table Formats:
Raw files (CSV, Parquet) in data lakes lack structure and reliability.
Problems:

- No ACID, no schema evolution, no time travel, slow queries.
- Previously used Hive with rigid folder-based partitions.
- Hard to govern and scale efficiently.

What is an Open Table Format?

- A layer over data lake files that adds table-like behavior.
- Manages metadata, schema, partitions, and versioning.
- Makes data lakes behave like SQL tables.

Examples: Apache Iceberg, Delta Lake, Apache Hudi

Where Does It Fit in the Stack?

- Ingestion/ETL → Data Lake → Open Table Format → Query Engine
- OTF sits between raw file storage and tools like Spark, Trino, Flink.
- It enables transactional and structured access to lake data.

Apache Iceberg Architecture

- Query Engine → Iceberg Table Layer → Metadata & Snapshots → Parquet Files
- Iceberg manages metadata, schema, manifest lists, and snapshots.
- Optimized for query pruning, time travel, and schema tracking.

Key Features of Open Table Formats
ACID Transactions: Snapshot isolation for consistent updates.
Time Travel: Query older versions of data safely.
Schema Evolution: Add/drop/rename columns without breaking data.
Hidden Partitions: Partition logic stored in metadata.
Multi-Engine: Compatible with Spark, Trino, Flink, etc.

How Iceberg Works Internally

- Write data → Create Parquet files + Manifest files
- Update metadata.json → Create new snapshot version
- Table always points to latest snapshot
- Query engines read from optimized manifest metadata

Main Components

- **Meta Data File:** Stores table-level info. Tracks Current snapshot.

- **Manifest List:** Points to Manifest files, tracks current snapshot and table history.
- **Manifest File:** Stores actual file level data.
- (Level 1)Metadata —> snapshot(2)---> Manifest List—> Manifest file(3)--> Data Files(4)

Summary

- Open Table Formats bridge raw files and structured queries.
- Enable reliable, scalable, and governed data lakes.
- Apache Iceberg is ideal for complex schema management and multi-engine use.
- OTFs are essential for modern data lake architecture.

Demo

Main Tools:

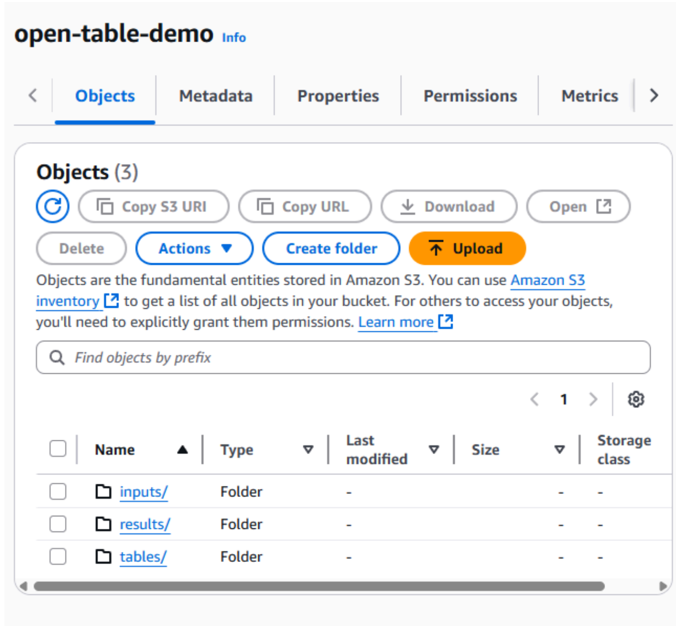
1. **S3 Storage:** Object storage where all the iceberg table files are stored.
2. **AWS Glue Catalog:** Lets PyIceberg find the table and its base location
3. **PyIceberg**: Creates Tables and Appends data, reads metadata and handles snapshots.
4. **PyArrow:** Reads parquet, holds data as tables and hands data to pyiceberg.

Starting with data:

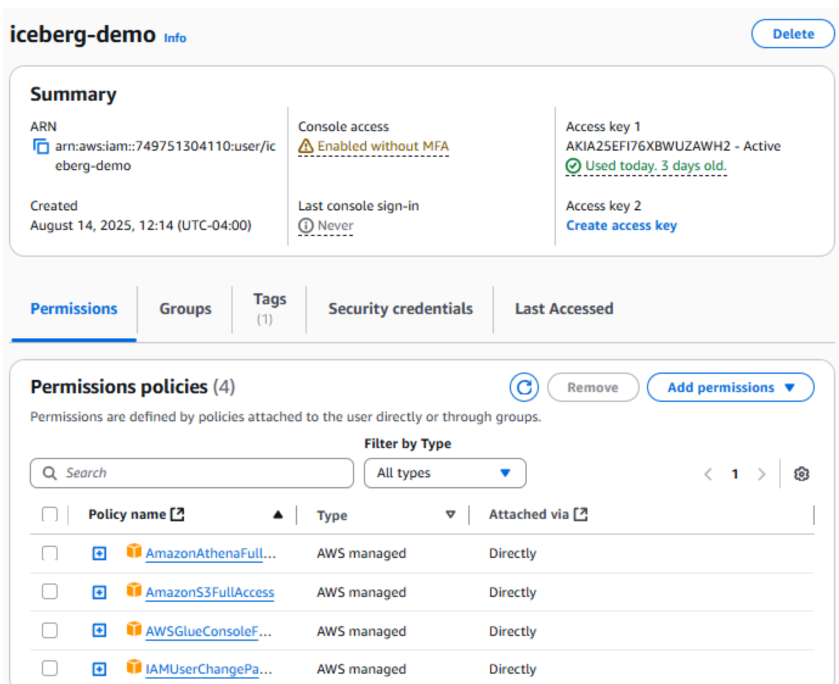
1. **Data Creation:** I have created 10 csv files using a random generator and then converted them into parquet files, with 11 fields in each parquet file.

   file_no → integer (int64)
   id → integer (int64)
   name → string
   category → string
   region → string
   value → integer (int64)
   price → decimal number (float64)
   is_active → true/false (boolean)
   ts → timestamp (date + time)
   event_date → date only (YYYY-MM-DD)
   note → string

2. **Creating a S3 Bucket:** Created a S3 bucket to hold the parquet data and Iceberg Metadata.
   I have created a bucket with folders, input, tables and results.
   Input is for loading the data.
   Tables are for iceberg table metadata and to store all the snapshots.
   Results location is save the query outputs.



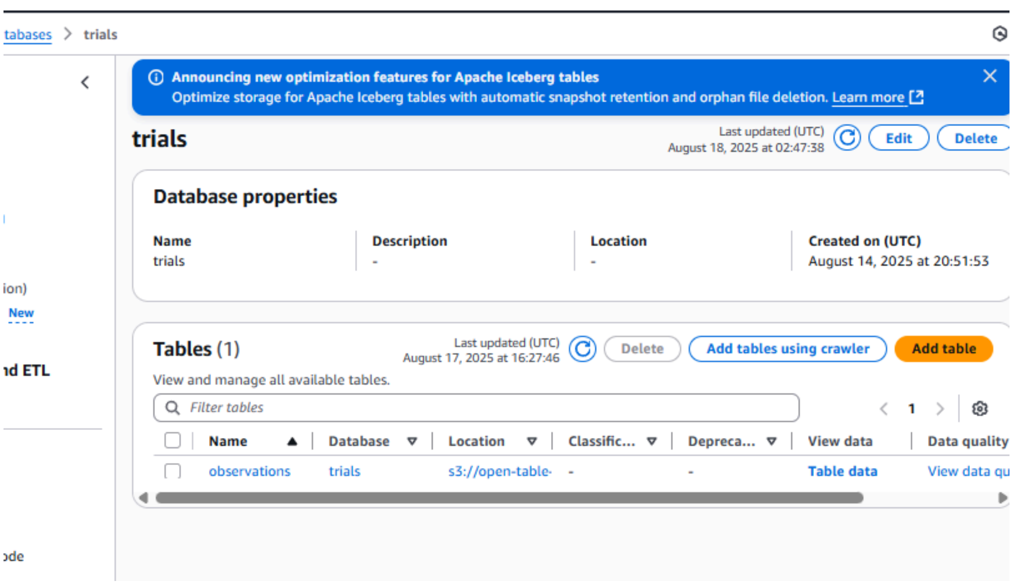
3. **Creating an IAM user:** I have created an IAM user and attached policies are S3 access and Glueconsole access.



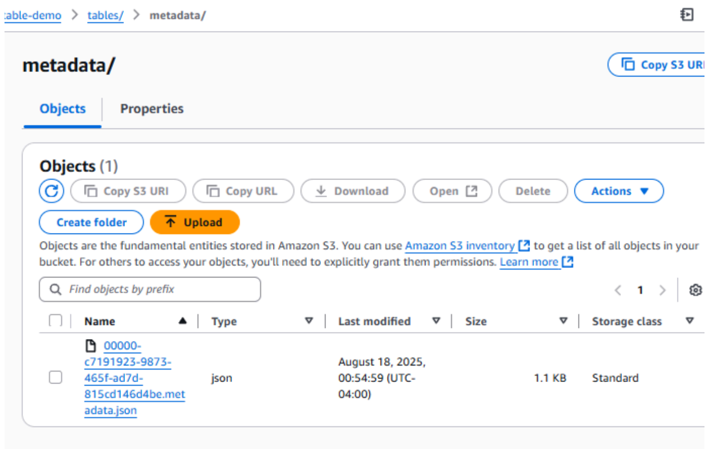
4. **Iceberg Table Creation:** Initialized pyiceberg and glue catalog in collab and then created Iceberg table with schema and S3 location. We have created the database name

```python
s3_bucket   = "open-table-demo/"
tables_path = "tables/"
table_name  = "observations"

schema = Schema(
    NestedField(1,  "file_no",      LongType(),     required=True),
    NestedField(2,  "id",           LongType(),     required=True),
    NestedField(3,  "name",         StringType(),   required=False),
    NestedField(4,  "category",     StringType(),   required=False),
    NestedField(5,  "region",       StringType(),   required=False),
    NestedField(6,  "value",        LongType(),     required=False),
    NestedField(7,  "price",        DoubleType(),   required=False),
    NestedField(8,  "is_active",    BooleanType(),  required=False),
    NestedField(9,  "ts",           TimestampType(),required=False),
    NestedField(10, "event_date",   DateType(),     required=False),
    NestedField(11, "note",         StringType(),   required=False),
)

table = catalog.create_table_if_not_exists(
    "{}.{}".format(namespace, table_name),
    schema,
    location="s3://{}{}".format(s3_bucket, tables_path),
)

print("Created/loaded table:", table.name, "->", table.location())
```

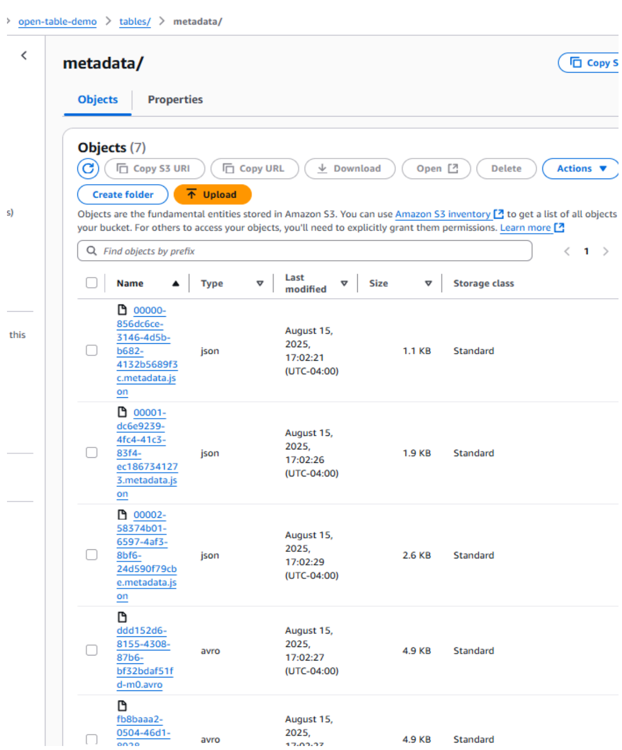
```
Created/loaded table: <bound method Table.name of observations(
  1: file_no: required long,
  2: id: required long,
  3: name: optional string,
  4: category: optional string,
  5: region: optional string,
  6: value: optional long,
  7: price: optional double,
  8: is_active: optional boolean,
  9: ts: optional timestamp,
  10: event_date: optional date,
  11: note: optional string
),
partition by: [],
sort order: [],
snapshot: null> -> s3://open-table-demo/tables
```



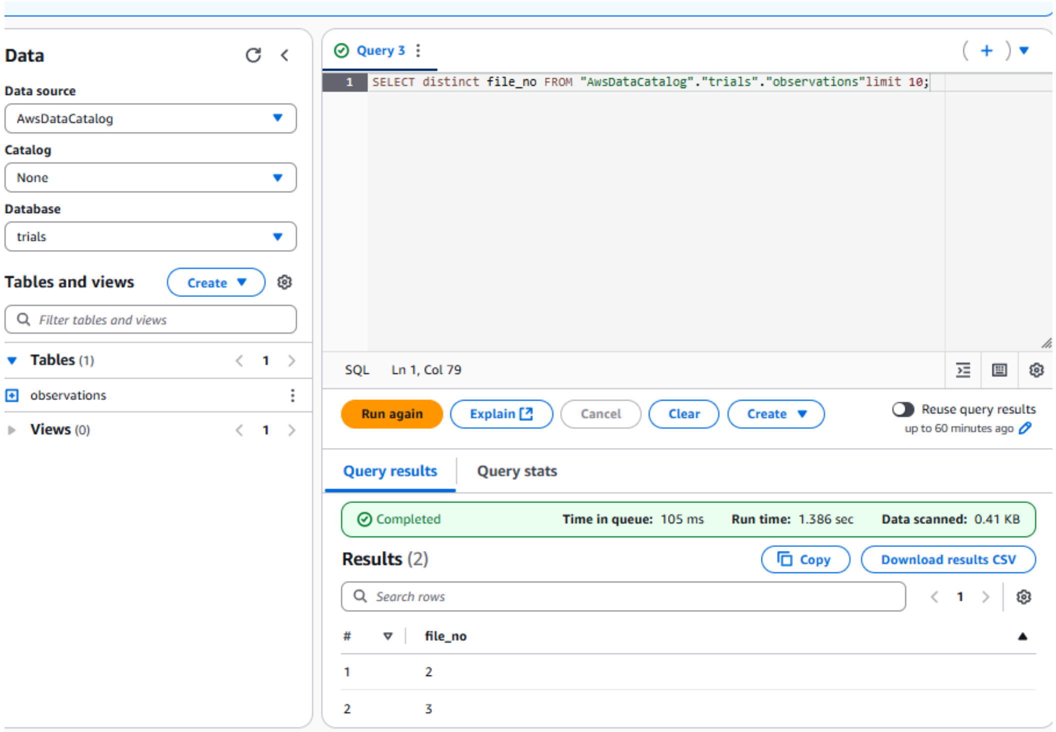
5. **Loading the Data(First Append) :** Always a metadata file is created when the table is created. So, ultimately when the table is created, one metadata file is created. We can see in the loop that the start and end variable for the file list variable is 1 and 3. Here it means that first and second files are appended. Because it's a loop, for each file each metadata file(JSON), Manifest file(AVRO) and snapshot file(Manifest List - AVRO) is created.



In total:
3 metadata files - 3 JSON files - One for creating a table, one metadata file for 1st file, one metadata file for the second file
2 Manifest Files - Lists all the data files that belong to a snapshot - One for each file
2 Snap shot file - one for each append



I have just queried the observations table using the query to know which files are actually loaded. From the loop - it's 1st and 2nd indexes in the file list - file 2 and file 3
SELECT distinct file_no FROM "AwsDataCatalog"."trials"."observations"limit 10;



We can see that we have loaded file 2 and file 3.

6. **ACID transactions:** I am appending file 9, 10 - index 8 and 9 from the file list.

```
start_snapshot = table.current_snapshot()

batch_a = load_as_arrow(file_list[8])
table.append(batch_a)
snap_after_a = table.current_snapshot()

batch_b = load_as_arrow(file_list[9])
table.append(batch_b)
snap_after_b = table.current_snapshot()

print("Start snapshot:", start_snapshot.snapshot_id if start_snapshot else None)
print("After A:", snap_after_a.snapshot_id)
print("After B:", snap_after_b.snapshot_id)
print("Total snapshots:", len(list(table.snapshots())))
```

```
Start snapshot: 3239556186961131551
After A: 6738247609746128565
After B: 7826614135507430187
Total snapshots: 4
```

Initially we have 2 snapshots for the old appends. Now for the 2 new appends, we are getting 2 snapshots. In total there are 4.

We can track which files are being added in each manifest file in each snapshot. In this case only one manifest file is created for each manifest list(snapshot file).

```python
snaps = snapshots_sorting(table)

print(f"Total there are {len(snaps)} snapshots\n")

for snap in snaps:
    print(f"\n Snapshot ID={snap.snapshot_id}  ts_ms={snap.timestamp_ms}")

    print(f"Manifest list: {snap.manifest_list}")

    manifests = snap.manifests(io=table.io)
    for i, m in enumerate(manifests, 1):
        path = getattr(m, "manifest_path", None) or getattr(m, "path", None)
        added   = getattr(m, "added_files_count", None)
        existing= getattr(m, "existing_files_count", None)
        deleted = getattr(m, "deleted_files_count", None)
        print(f"  Manifest {i}: {path}  (added={added}, existing={existing}, deleted={deleted})")

    files = file_paths_for_snapshot(table, snap.snapshot_id)
    print(f"Data files in this snapshot: {len(files)}")
    for p in files:
        print(f"  {p}")

print("\nAdded data files per snapshot")
prev_set = set()
for idx, snap in enumerate(snaps):
    curr_set = set(file_paths_for_snapshot(table, snap.snapshot_id))
    added = curr_set - prev_set if idx > 0 else curr_set
    print(f"\nSnapshot {snap.snapshot_id}: added {len(added)} file(s)")
    for p in sorted(added):
        print(f"  {p}")
    prev_set = curr_set
```

```
Found 4 snapshots


 Snapshot ID=5805377075394109O2  ts_ms=1755493012429
Manifest list: s3://open-table-demo/tables/metadata/snap-5805377075394109O2-0-929e201c-4cd1-4a55-ba4a-478c238e3e7c.avro
  Manifest 1: s3://open-table-demo/tables/metadata/929e201c-4cd1-4a55-ba4a-478c238e3e7c-m0.avro  (added=1, existing=0, deleted=0)
Data files in this snapshot: 1
  s3://open-table-demo/tables/data/00000-0-929e201c-4cd1-4a55-ba4a-478c238e3e7c.parquet

 Snapshot ID=3239556186961131551  ts_ms=1755493014719
Manifest list: s3://open-table-demo/tables/metadata/snap-3239556186961131551-0-bacbff33-1095-485d-80c6-17f589491abd.avro
  Manifest 1: s3://open-table-demo/tables/metadata/bacbff33-1095-485d-80c6-17f589491abd-m0.avro  (added=1, existing=0, deleted=0)
  Manifest 2: s3://open-table-demo/tables/metadata/929e201c-4cd1-4a55-ba4a-478c238e3e7c-m0.avro  (added=1, existing=0, deleted=0)
Data files in this snapshot: 2
  s3://open-table-demo/tables/data/00000-0-bacbff33-1095-485d-80c6-17f589491abd.parquet
  s3://open-table-demo/tables/data/00000-0-929e201c-4cd1-4a55-ba4a-478c238e3e7c.parquet
```

We can even track which data files are loaded in which snapshot and manifest files.

7. **Time Travel:** We can access the older version as well according to our need, very easily. Here's a simple example to showcase that. Here, we are getting the data from the older snapshot and also the latest snapshot and doing the operations required or just comparing them.

```python
old_idx = -2
old_id = snaps[old_idx].snapshot_id

print(f"\n Switching to older snapshot (ID: {old_id})")
old_data = table.scan(snapshot_id=old_id).to_arrow()
print(f"Row count at this snapshot: {old_data.num_rows}")

old_data.to_pandas().head()
```

```
 Switching to older snapshot (ID: 3937421198858816241)
Row count at this snapshot: 2500
```

	file_no	id	name	category	region	value	price	is_active	ts	event_date	note
0	9	4000	Name_4000	C	LATAM	771	559.12	False	2025-01-10 00:00:00	2025-01-10	note_0
1	9	4001	Name_4001	C	EMEA	444	700.13	True	2025-01-10 01:00:00	2025-01-10	note_1
2	9	4002	Name_4002	C	EMEA	964	711.50	False	2025-01-10 02:00:00	2025-01-10	None
3	9	4003	Name_4003	A	LATAM	629	185.47	True	2025-01-10 03:00:00	2025-01-10	note_3
4	9	4004	Name_4004	D	APAC	858	686.54	False	2025-01-10 04:00:00	2025-01-10	note_4

```python
latest_id = snaps[-1].snapshot_id

print(f"\n Switching to latest snapshot (ID: {latest_id})")
latest_data = table.scan(snapshot_id=latest_id).to_arrow()
print(f"Row count at this snapshot: {latest_data.num_rows}")

latest_data.to_pandas().head()
```

```
 Switching to latest snapshot (ID: 2752519333006923957)
Row count at this snapshot: 3000
```

	file_no	id	name	category	region	value	price	is_active	ts	event_date	note
0	10	4500	Name_4500	A	APAC	249	32.83	False	2025-01-11 00:00:00	2025-01-11	None
1	10	4501	Name_4501	D	EMEA	734	75.88	False	2025-01-11 01:00:00	2025-01-11	note_1
2	10	4502	Name_4502	D	APAC	558	402.19	False	2025-01-11 02:00:00	2025-01-11	note_2
3	10	4503	Name_4503	B	EMEA	727	817.48	True	2025-01-11 03:00:00	2025-01-11	note_3
4	10	4504	Name_4504	B	LATAM	441	945.71	True	2025-01-11 04:00:00	2025-01-11	note_4

8. **Schema Evolution:** we can always edit the schema on the go if needed.
   I am going to add a new column and test this out whether it’s working or not.

```
'note']
```

```python
from pyiceberg.types import StringType

us = table.update_schema()
us = us.add_column("new_column", StringType(), required=False)

us.commit()
print("Schema updated. New schema:")
for f in table.schema().fields:
    print(f"{f.field_id}: {f.name} - {type(f.field_type).__name__}, required={f.required}")
```

```
Schema updated. New schema:
1: file_no - LongType, required=True
2: id - LongType, required=True
3: name - StringType, required=False
4: category - StringType, required=False
5: region - StringType, required=False
6: value - LongType, required=False
7: price - DoubleType, required=False
8: is_active - BooleanType, required=False
9: ts - TimestampType, required=False
10: event_date - DateType, required=False
11: note - StringType, required=False
13: new_column - StringType, required=False
```

[43]
```python
print(table.current_snapshot())
```

```
Operation.APPEND: id=2752519333006923957, parent_id=3937421198858816241, schema_id=0
```

At any point we can add and delete columns. Here I have added a column.
Similarly we can even delete a column.

```
[44] us = table.update_schema()
     us = us.delete_column("new_column")
     us.commit()

     print("Column has been deleted.")
```

```
Column has been deleted.
```

```
[45] [f.name for f in table.schema().fields]
```

```
['file_no',
 'id',
 'name',
 'category',
 'region',
 'value',
 'price',
 'is_active',
 'ts',
 'event_date',
 'note']
```

We can even rename the column as well.

```
snaps_before = list(table.snapshots())
old_snapshot_id = snaps_before[-1].snapshot_id

us = table.update_schema()
us = us.rename_column("note", "comment")
us.commit()

snaps_after = list(table.snapshots())
new_snapshot_id = snaps_after[-1].snapshot_id

print("Old snapshot ID:", old_snapshot_id)
print("New snapshot ID:", new_snapshot_id)
```

```
Old snapshot ID: 2752519333006923957
New snapshot ID: 2752519333006923957
```

8. Hidden Partitioning: Hidden partitions let Iceberg organize data by year without adding extra columns. In our case, we want to partition data by the event_date. Instead of adding a new column like event_year, Iceberg allows us to define a partition transform like year(event_date). This transformation does not show up in the table schema, so the table stays clean, but Iceberg still uses it internally to organize files and optimize queries.

Here we can see that even for the new partition, we are able to upload the old schema files. Without manually changing the schema for all the old files.

```
[ ] from pyiceberg.partitioning import PartitionSpec, PartitionField
    # source_id : event_date - 10th one and field_id : virtual id for the new field
    spec = PartitionSpec(
        PartitionField(source_id=10, field_id=2001, transform="year", name="event_year")
    )

    partitioned_name = f"{table_name}_by_year"
    partitioned_loc = "s3://open-table-demo/results_by_year/"

    pt = catalog.create_table_if_not_exists(
        f"{namespace}.{partitioned_name}",
        schema=table.schema(),
        location=partitioned_loc,
        partition_spec=spec
    )

    print("Partitioned table created:", pt.name())
    print("Partition spec:", [ (f.name, f.transform) for f in pt.spec().fields ])
```

```
Partitioned table created: ('trials', 'observations_by_year')
Partition spec: [('event_year', YearTransform())]
```

```
[59] subset = [load_as_arrow(fp) for fp in file_list[4:6]]
     pt.append(pa.concat_tables(subset, promote=True))
     print("Appended 2 files to partitioned table")
```

```
Appended 2 files to partitioned table
```

Here we have created a hidden partition that gives results by year. We can test the same using AWS Athena as well. As shown below.

Query Used: SELECT * FROM "AwsDataCatalog"."trials"."observations_by_year" WHERE year(event_date) = 2025;