

Apache Iceberg

An open table format that brings ACID transactions, schema evolution, and time travel to any data lake.

Last reviewed 29 Sep 2026 · Lab-tested with Apache Iceberg 1.11.0 ·

Prerequisites: [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) · [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/)

Related: [Lab 09](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/09-iceberg-lakehouse) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/09-iceberg-lakehouse) · [Databricks](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) · [Snowflake](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) · [Trino](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/trino-federation/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/trino-federation/) · [Azure and Fabric](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/azure-fabric/) · [DE Concepts](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/de-concepts/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

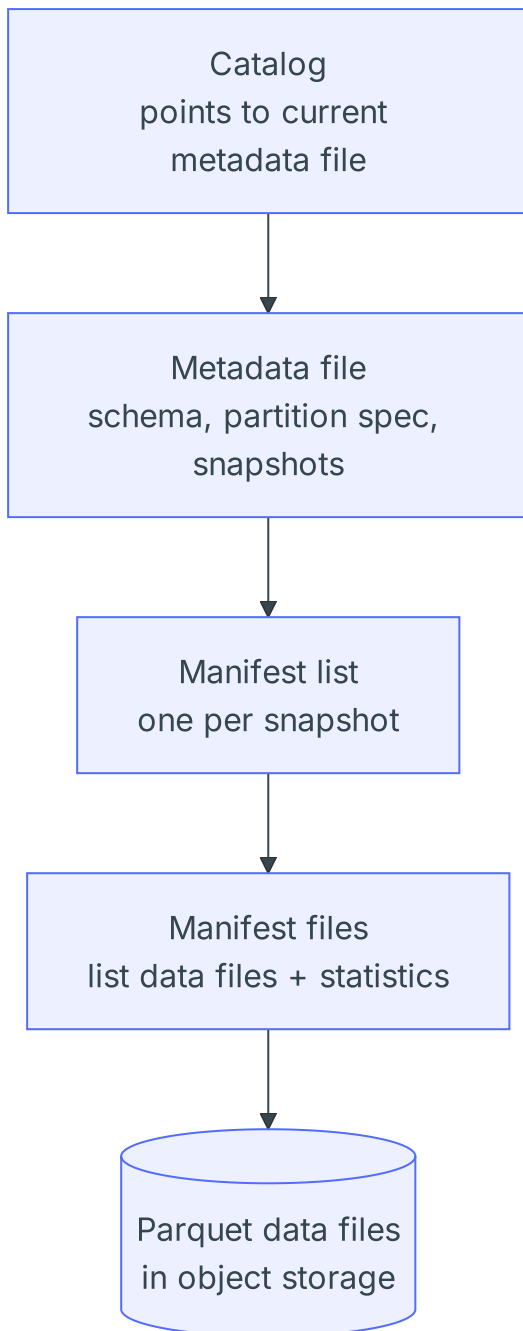
Challenge: A data lake of Parquet files in object storage has no concept of a table — only files. There is no safe way to update rows, roll back a bad write, or let two jobs write concurrently without corrupting results.

Solution: Apache Iceberg is an open **table format** that adds a metadata layer over those files. It tracks which files belong to the table, their schema, and the history of every change. The data stays in object storage; Iceberg provides transactional table semantics on top.

Without Iceberg:	With Iceberg:
s3://bucket/orders/ 2024-03-15.parquet snapshots, history) 2024-03-16.parquet files)	iceberg table "orders" ├ metadata/ (schema, └ data/ (the same Parquet files)

2024-03-17.parquet
(just files – no table) (a real table with ACID,
history, evolution)

Positioning: Iceberg is engine-neutral — Spark, Flink, Trino, and major cloud warehouses read and write it natively — which makes it a strong choice when several engines share the same tables. Delta Lake and Apache Hudi solve the same problem with different trade-offs (see the comparison later in this guide).



On this page

Basic - Core Concepts (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#core-concepts>) - **Setup with PySpark** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#setup-with-pyspark>) - **Creating and Writing Tables** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#creating-and-writing-tables>) - **Reading Data** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#reading-data>)

Intermediate - Schema Evolution (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#schema-evolution>) - **Partitioning** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#partitioning>) - **Time Travel** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#time-travel>) - **ACID Operations** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#acid-operations>)

Advanced - Table Maintenance (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#table-maintenance>) - **Iceberg on AWS (Glue + Athena)** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#iceberg-on-aws-glue--athena>) - **Iceberg vs Delta Lake vs Hudi** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#iceberg-vs-delta-lake-vs-hudi>) - **Common Pitfalls** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#common-pitfalls>)

Reference - Cheat Sheet (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#cheat-sheet>) - **Interview Questions** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#interview-questions>) - **Further Reading** (<https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/#further-reading>)

Core Concepts

Concept	Description
Table format	A specification for how to organize data files and metadata so any engine can read the "table"
Snapshot	A point-in-time version of the table — every write creates a new snapshot
Manifest	A file listing all data files in a snapshot with their statistics
Catalog	A service that maps table names to their metadata location (Hive Metastore, AWS Glue, Nessie, REST)
Hidden partitioning	Iceberg handles partition transforms automatically — no more <code>dt=2024-03-15</code> folder names in queries

Concept	Description
Metadata evolution	Add, rename, or drop columns without rewriting data

```
Iceberg metadata hierarchy:
  Table (name → metadata pointer)
    └─ Metadata file (schema, partition spec, current snapshot)
        └─ Snapshot (list of manifest files)
            └─ Manifest (list of data files + stats)
                └─ Data file (actual Parquet/ORC/Avro)
```

Setup with PySpark

```
# pip install 'pyspark==4.1.3'
# No Iceberg pip package needed – the runtime jar below is
# downloaded by Spark.
# Its name must match your Spark (4.1) and Scala (2.13)
# versions, and it is published only for
# some Spark versions: check Maven Central for iceberg-
# spark-runtime before upgrading Spark.

from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("iceberg-demo") \
    .config("spark.jars.packages",
            "org.apache.iceberg:iceberg-spark-runtime-
4.1_2.13:1.11.0") \
    .config("spark.sql.extensions",

"org.apache.iceberg.spark.extensions.IcebergSparkSessionExt
ensions") \
    .config("spark.sql.catalog.local",
            "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.local.type", "hadoop") \
    .config("spark.sql.catalog.local.warehouse",
```

```
"/tmp/iceberg-warehouse") \
    .getOrCreate()
```

```
# AWS Glue catalog (production setup)
spark = SparkSession.builder \
    .config("spark.sql.catalog.glue_catalog",
            "org.apache.iceberg.spark.SparkCatalog") \
    .config("spark.sql.catalog.glue_catalog.warehouse",
            "s3://my-bucket/iceberg/") \
    .config("spark.sql.catalog.glue_catalog.catalog-impl",
            "org.apache.iceberg.aws.glue.GlueCatalog") \
    .config("spark.sql.catalog.glue_catalog.io-impl",
            "org.apache.iceberg.aws.s3.S3FileIO") \
    .getOrCreate()
```

Creating and Writing Tables

```
from pyspark.sql import functions as F

# Create an Iceberg table via SQL
spark.sql("""
    CREATE TABLE IF NOT EXISTS local.db.orders (
        order_id      STRING,
        customer_id    STRING,
        amount         DOUBLE,
        status         STRING,
        order_date     DATE
    )
    USING iceberg
    PARTITIONED BY (order_date)
    TBLPROPERTIES (
        'format-version'           = '2',
        'write.format.default'     = 'parquet',
        'write.target-file-size-bytes' = '134217728' --
128MB
    )
""")

# Write via DataFrame
```

```
df = spark.createDataFrame([
    ("ORD-001", "CUST-1", 99.99, "placed", "2024-03-15"),
    ("ORD-002", "CUST-2", 149.50, "shipped", "2024-03-15"),
    ("ORD-003", "CUST-1", 29.00, "delivered", "2024-03-16"),
], ["order_id", "customer_id", "amount", "status", "order_date"])

df = df.withColumn("order_date",
F.col("order_date").cast("date"))

# Append (default)
df.writeTo("local.db.orders").append()

# Overwrite a specific partition
df.writeTo("local.db.orders") \
    .overwritePartitions()

# Full overwrite
df.writeTo("local.db.orders").overwrite(F.lit(True))
```

Reading Data

```
# Read the current snapshot
df = spark.table("local.db.orders")
df.show()

# Via SQL
spark.sql("SELECT * FROM local.db.orders WHERE status = 'placed'").show()

# Check table metadata
spark.sql("DESCRIBE EXTENDED local.db.orders").show(truncate=False)

# Show snapshots (history)
spark.sql("SELECT * FROM local.db.orders.snapshots").show()
```

```
# Show data files
spark.sql("SELECT * FROM local.db.orders.files").show()

# Show history
spark.sql("SELECT * FROM local.db.orders.history").show()
```

Schema Evolution

```
# Add a column (non-breaking)
spark.sql("ALTER TABLE local.db.orders ADD COLUMN
shipping_cost DOUBLE")

# Rename a column
spark.sql("ALTER TABLE local.db.orders RENAME COLUMN
shipping_cost TO shipping_fee")

# Change a column type – only safe widenings: int → long,
float → double,
# decimal(P,S) → decimal(P+n,S). DOUBLE → DECIMAL is NOT
allowed.
spark.sql("ALTER TABLE local.db.orders ADD COLUMN quantity
INT")
spark.sql("ALTER TABLE local.db.orders ALTER COLUMN
quantity TYPE BIGINT")

# Drop a column (data still in files, just not exposed in
schema)
spark.sql("ALTER TABLE local.db.orders DROP COLUMN
shipping_fee")

# After schema changes, old data files still work – Iceberg
handles nulls for
# missing columns automatically

# Schema evolution rules:
# Allowed:      Adding columns      – always safe
# Allowed:      Dropping columns    – safe (data stays, not
visible)
# Allowed:      Renaming columns    – safe (tracked by
column ID, not name)
```

```
# Allowed:      Widening types      – safe (int→long,
float→double)
# Not allowed: Narrowing types      – NOT allowed
(double→float loses precision)
# Not allowed: Changing semantics  – NOT safe (renaming to
mean something different)
```

Partitioning

Iceberg's hidden partitioning is one of its best features — no `WHERE dt='2024-03-15'` needed.

```
# Partition transforms – Iceberg applies these
automatically at write time
spark.sql("""
    CREATE TABLE local.db.events (
        event_id  STRING,
        user_id   STRING,
        event_type STRING,
        event_ts  TIMESTAMP,
        amount    DOUBLE
    )
    USING iceberg
    PARTITIONED BY (
        days(event_ts),          -- partition by day from a
timestamp
        bucket(16, user_id)     -- hash into 16 buckets (for
high-cardinality cols)
    )
""")

# Available transforms:
# year(ts)      → YYYY
# month(ts)     → YYYY-MM
# days(ts)      → YYYY-MM-DD (most common)
# hours(ts)     → YYYY-MM-DD-HH
# bucket(N, col) → hash into N buckets
# truncate(N, col) → string prefix or integer truncation

# Writing – Iceberg handles partitioning transparently
```

```
df.writeTo("local.db.events").append()

# Querying – no partition filter needed in WHERE; Iceberg
prunes automatically
spark.sql("""
    SELECT COUNT(*) FROM local.db.events
    WHERE event_ts BETWEEN '2024-03-01' AND '2024-03-15'
""").show()

# Iceberg reads only the day=2024-03-01 through day=2024-
03-15 partitions

# Evolve the partition spec without rewriting data
spark.sql("ALTER TABLE local.db.events ADD PARTITION FIELD
hours(event_ts)")
spark.sql("ALTER TABLE local.db.events DROP PARTITION FIELD
days(event_ts)")
```

Time Travel

```
# Query a specific snapshot
spark.sql("""
    SELECT * FROM local.db.orders
    VERSION AS OF 1234567890 -- snapshot ID
""").show()

# Query at a specific timestamp
spark.sql("""
    SELECT * FROM local.db.orders
    TIMESTAMP AS OF '2024-03-15 12:00:00'
""").show()

# Via DataFrame reader: Spark's own options. The Iceberg
options snapshot-id and
# as-of-timestamp raise an error in Spark 4.1 with Iceberg
1.11.
df = spark.read \
    .option("versionAsOf", 1234567890) \
    .table("local.db.orders")

df = spark.read \
```

```

.option("timestampAsOf", "2024-03-15 12:00:00") \
.table("local.db.orders")

# Rollback to a previous snapshot
spark.sql("CALL
local.system.rollback_to_snapshot('db.orders',
1234567890)")

# Rollback to a timestamp
spark.sql("CALL
local.system.rollback_to_timestamp('db.orders', TIMESTAMP
'2024-03-14 00:00:00')")

```

ACID Operations

```

# MERGE (upsert) – insert new, update existing
spark.sql("""
    MERGE INTO local.db.orders t
    USING updates s ON t.order_id = s.order_id
    WHEN MATCHED THEN UPDATE SET t.status = s.status,
t.amount = s.amount
    WHEN NOT MATCHED THEN INSERT *
""")

# UPDATE
spark.sql("""
    UPDATE local.db.orders
    SET status = 'cancelled'
    WHERE order_date < '2024-01-01' AND status = 'placed'
""")

# DELETE
spark.sql("""
    DELETE FROM local.db.orders
    WHERE status = 'cancelled' AND order_date < '2023-01-
01'
""")

# Copy-on-write (default) vs Merge-on-read:
# Copy-on-write: rewrites entire files on every

```

```
update/delete – fast reads, slow writes
# Merge-on-read: writes delete files; merges at read time –
fast writes, slower reads
spark.sql("""
    ALTER TABLE local.db.orders
    SET TBLPROPERTIES ('write.delete.mode' = 'merge-on-
read')
""")
```

Table Maintenance

```
# Expire old snapshots (reclaim storage)
spark.sql("""
    CALL local.system.expire_snapshots(
        table      => 'db.orders',
        older_than => TIMESTAMP '2024-03-01 00:00:00',
        retain_last => 5
    )
""")

# Remove orphan files (files not referenced by any
snapshot)
spark.sql("""
    CALL local.system.remove_orphan_files(table =>
'db.orders')
""")

# Compact small files (rewrite into larger files)
spark.sql("""
    CALL local.system.rewrite_data_files(
        table  => 'db.orders',
        options => map('target-file-size-bytes',
'134217728')
    )
""")

# Rewrite manifests (improves planning performance on large
tables)
spark.sql("""
    CALL local.system.rewrite_manifests(table =>
```

```
'db.orders')  
""")
```

Iceberg on AWS (Glue + Athena)

```
# AWS Glue Catalog + S3 + Athena – no Spark cluster needed   
for queries  
  
# Create the table with Athena DDL – Athena writes the  
Iceberg metadata and  
# registers the table in the Glue Data Catalog in one step:  
#  
# CREATE TABLE mydb.orders (  
#     order_id    string,  
#     customer_id string,  
#     amount       double,  
#     order_date   date  
# )  
# PARTITIONED BY (day(order_date))  
# LOCATION 's3://my-bucket/iceberg/orders/'  
# TBLPROPERTIES ('table_type' = 'ICEBERG');  
#  
# Tables created by Spark with the Glue catalog (setup  
above) show up in Athena  
# automatically – both engines share the same Glue  
metadata.  
  
# Query in Athena (after table registered in Glue)  
# SELECT * FROM mydb.orders WHERE order_date >= DATE '2024-  
03-01'  
  
# Athena time travel  
# SELECT * FROM mydb.orders FOR SYSTEM_TIME AS OF TIMESTAMP  
'2024-03-15 12:00:00'
```

Iceberg vs Delta Lake vs Hudi

	Apache Iceberg	Delta Lake	Apache Hudi
Creator	Netflix/Apple → Apache	Databricks	Uber
Open standard	Yes	Yes (open source)	Yes
Best engine	Any (Spark, Trino, Flink)	Databricks/Spark	Spark
ACID	Yes	Yes	Yes
Time travel	Yes	Yes	Yes
Schema evolution	Excellent (by column ID)	Good	Good
Partition evolution	Yes (no rewrite)	Limited	Limited
Hidden partitioning	Yes	No	No
Streaming	Good (via Flink)	Excellent	Excellent
Multi-engine	Best	Good	Fair
AWS native	Glue, Athena, EMR	EMR	EMR
Choose when	Multi-engine, cloud-agnostic	All-in on Databricks	Upsert-heavy, CDC workloads

Common Pitfalls

1. Forgetting to configure a catalog

Problem: Iceberg tables need a catalog – without one, nothing persists

Fix: Always configure Hive Metastore, AWS Glue, or Nessie catalog in SparkSession

2. Not running table maintenance

Problem: snapshots accumulate, storage balloons, planning slows down

Fix: Schedule weekly: `expire_snapshots + remove_orphan_files + rewrite_data_files`

3. Using `year()` or `month()` partition transforms on high-query-frequency tables

Problem: `year(ts)` creates partitions with millions of files – too coarse

Fix: Use `days(ts)` for daily-batch tables, `hours(ts)` for streaming tables

4. Over-partitioning with `bucket()`

Problem: `bucket(1000, user_id)` creates 1000 files per write – too many small files

Fix: Start with `bucket(16)` or `bucket(32)`; 128MB target file size

5. Creating tables as format V1

Problem: V1 has no row-level deletes, so `MERGE/UPDATE/DELETE` must rewrite whole files

Fix: V2 is the default since Iceberg 1.4; set `'format-version' = '2'` explicitly
on older engines/versions (V3 adds deletion vectors – check engine support first)

6. Skipping `OPTIMIZE` after heavy upserts

Problem: merge-on-read tables accumulate delete files → slower reads over time

Fix: Run `rewrite_data_files` after bulk upserts to compact

Cheat Sheet

Task	Spark SQL
Create table	<code>CREATE TABLE cat.db.t (...) USING iceberg PARTITIONED BY (days(ts))</code>

Task	Spark SQL
Append / overwrite partitions	<code>df.writeTo("cat.db.t").append() · .overwritePartitions()</code>
Upsert	<code>MERGE INTO cat.db.t t USING s ON t.id = s.id WHEN MATCHED THEN UPDATE SET * WHEN NOT MATCHED THEN INSERT *</code>
Add / rename column	<code>ALTER TABLE t ADD COLUMN c INT · RENAME COLUMN a TO b</code>
Change partitioning (no rewrite)	<code>ALTER TABLE t ADD PARTITION FIELD hours(ts) · DROP PARTITION FIELD days(ts)</code>
Time travel	<code>SELECT ... FROM t VERSION AS OF <snapshot_id> · TIMESTAMP AS OF '2024-03-15 12:00:00'</code>
Roll back	<code>CALL cat.system.rollback_to_snapshot('db.t', <id>)</code>
List snapshots	<code>SELECT * FROM cat.db.t.snapshots</code>
Files and sizes	<code>SELECT file_path, record_count, file_size_in_bytes FROM cat.db.t.files</code>
Partition stats	<code>SELECT * FROM cat.db.t.partitions</code>
Compact	<code>CALL cat.system.rewrite_data_files(table => 'db.t')</code>
Expire snapshots	<code>CALL cat.system.expire_snapshots(table => 'db.t', older_than => TIMESTAMP '...', retain_last => 5)</code>
Remove orphan files	<code>CALL cat.system.remove_orphan_files(table => 'db.t')</code>
Merge-on-read for upsert-heavy tables	<code>ALTER TABLE t SET TBLPROPERTIES ('write.merge.mode'='merge-on-read', 'write.update.mode'='merge-on-read', 'write.delete.mode'='merge-on-read')</code>
Branch for write-audit-publish	<code>ALTER TABLE t CREATE BRANCH audit → write with spark.wap.branch → CALL cat.system.fast_forward('db.t', 'main', 'audit')</code>

Partition transforms: `years · months · days · hours · bucket(N, col) · truncate(W, col)`

Maintenance schedule (typical): compact daily or after big loads · expire snapshots weekly (keep enough for your time-travel SLA) · remove orphan files weekly · rewrite manifests when query planning slows down

Catalog options: REST (Polaris, Unity Catalog, Nessie, Gravitino, and managed services) · AWS Glue · Hive Metastore · JDBC · Hadoop (local testing only)

Interview Questions

Q: What problem does Iceberg solve that raw Parquet files on S3 don't? A: Raw Parquet has no table concept — no ACID, no schema enforcement, no concurrent write safety, no history. Iceberg adds a metadata layer (snapshots, manifests) that makes S3 files into a real table with transactions, time travel, and schema evolution, while keeping the files open-format and readable by any engine.

Q: What is hidden partitioning and why does it matter? A: In Hive/traditional partitioning, the query must include a filter on the partition column (`WHERE dt='2024-03-15'`) or it scans everything. In Iceberg, you define partition transforms (`days(event_ts)`) and Iceberg applies them transparently — the query just filters on `event_ts` and Iceberg handles pruning. This prevents accidental full-table scans.

Q: How does Iceberg handle schema evolution without rewriting data? A: Iceberg tracks columns by a stable integer ID, not by name. When you rename a column, the ID stays the same — old files still map to the right column. New columns default to NULL for old files. Type changes are allowed only for safe widenings (`int` → `long`).

Q: What is copy-on-write vs merge-on-read? A: Copy-on-write rewrites entire data files when rows are updated/deleted — reads are fast because there's only one file per row, but writes are expensive. Merge-on-read writes small delete/delta files alongside data files and merges them at read time — writes are fast, reads are slightly slower. Choose copy-on-write for read-heavy, update-infrequent tables; merge-on-read for frequent upserts.

Q: What is an Iceberg catalog and why does it matter? A: The catalog maps a table name to the location of its *current* metadata file, and it's what makes commits atomic: a writer creates new metadata and then asks the catalog to swap the pointer from the old file to the new one, and that swap only succeeds if nobody else committed first (optimistic concurrency). Every engine that reads or writes the table must use the same catalog. The REST catalog spec has become the standard interface, so Spark, Trino, Flink, Snowflake, and others can all share one catalog.

Q: How does Iceberg handle two jobs writing to the same table at the same time? A: Optimistic concurrency. Each writer reads the current snapshot, writes its data files, and then tries to commit new metadata based on that snapshot. If another commit landed first, the catalog rejects the swap; the writer then checks whether the two changes conflict (for example, both touched the same files or partitions). If they don't, it retries the commit on top of the new snapshot; if they do, the commit fails. Data files are never modified in place, so readers always see a consistent snapshot.

Q: What maintenance does an Iceberg table need, and what happens if you skip it? A:

Every write adds a snapshot plus metadata and manifest files, and streaming or small batches add many small data files. Without maintenance, storage keeps growing (old snapshots pin deleted files), query planning slows down (too many manifests), and scans slow down (small files, accumulated delete files). The fix is scheduled compaction (`rewrite_data_files`), snapshot expiration, orphan file removal, and occasional manifest rewrites. Some managed catalogs run these for you.

Q: When would you choose Iceberg over Delta Lake? A: When several engines need to read and write the same tables (Spark, Trino, Flink, Snowflake, Athena, BigQuery), when you want to avoid tying storage to one vendor, or when partition evolution and hidden partitioning matter. Delta is the natural choice on Databricks, where features like Liquid Clustering and Predictive Optimization are tightly integrated. The gap is shrinking: Delta UniForm can expose Iceberg metadata, and Databricks and Snowflake both support Iceberg tables.

Further Reading

- [Apache Iceberg documentation](https://iceberg.apache.org/docs/latest/) (<https://iceberg.apache.org/docs/latest/>)
- [Iceberg table spec](https://iceberg.apache.org/spec/) (<https://iceberg.apache.org/spec/>) — how snapshots, manifests, and delete files actually work
- [Spark procedures reference](https://iceberg.apache.org/docs/latest/spark-procedures/) (<https://iceberg.apache.org/docs/latest/spark-procedures/>) — every maintenance `CALL` in one page
- [Using Iceberg tables in Athena](https://docs.aws.amazon.com/athena/latest/ug/querying-iceberg.html) (<https://docs.aws.amazon.com/athena/latest/ug/querying-iceberg.html>)
- [Pylceberg](https://py.iceberg.apache.org/) (<https://py.iceberg.apache.org/>) — read and write Iceberg from Python without Spark
- *Apache Iceberg: The Definitive Guide* — Tomer Shiran, Jason Hughes & Alex Merced (O'Reilly)

Previous: [Databricks](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/) (<https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/>) • **Next:** [Trino](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/trino-federation/) (<https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/trino-federation/>) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (<https://sarangambekar1997.github.io/data-engineering-handbook/>)

 September 29, 2026