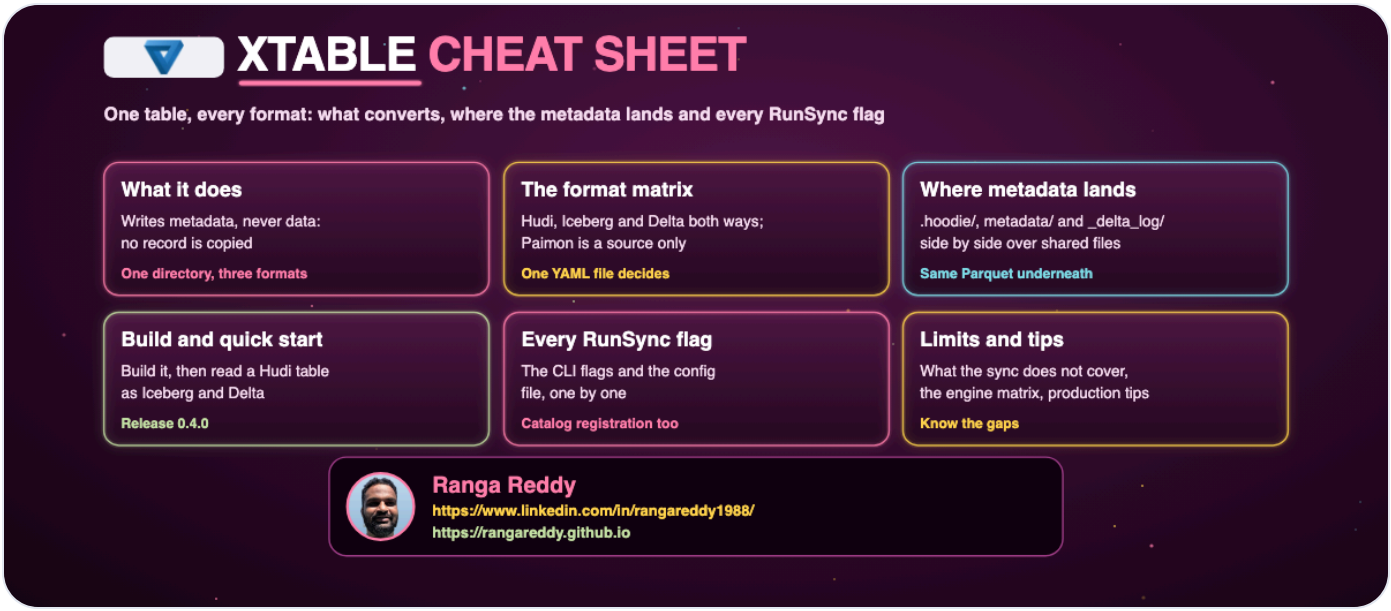
← All posts



Apache XTable cheat sheet: one table, every format

One page to keep open while you work: what XTable converts, where the metadata lands, the build, the config file, every RunSync flag, catalog registration and the engine matrix.

Jul 29, 2026 · 20 min read · 4 · Lakehouse

#XTable #Hudi #Iceberg #Delta #Paimon #Reference

TL;DR

- XTable writes metadata, never data. One directory of Parquet files ends up carrying `.hoodie/`, `metadata/` and `_delta_log/` side by side, and no record is copied.
- The authoritative format list is one file, `xtable-conversion-defaults.yaml`. Hudi, Iceberg and Delta convert in both directions; Paimon is source only;

Parquet has a source provider but is not in the defaults, so it needs an explicit converters config.

- The bundled jar is runnable again, so `java -jar` works and the old `jol-core` classpath workaround is gone.
- `xtable-utilities` is not published to Maven Central. You build it. `xtable-core`, `xtable-api`, `xtable-service` and the new `xtable-spark-runtime` are published.
- Only Copy-on-Write or read-optimized views convert. Hudi log files and Delta or Iceberg deletion vectors are not captured, so on a Merge-on-Read source the converted formats are as fresh as the last compaction.

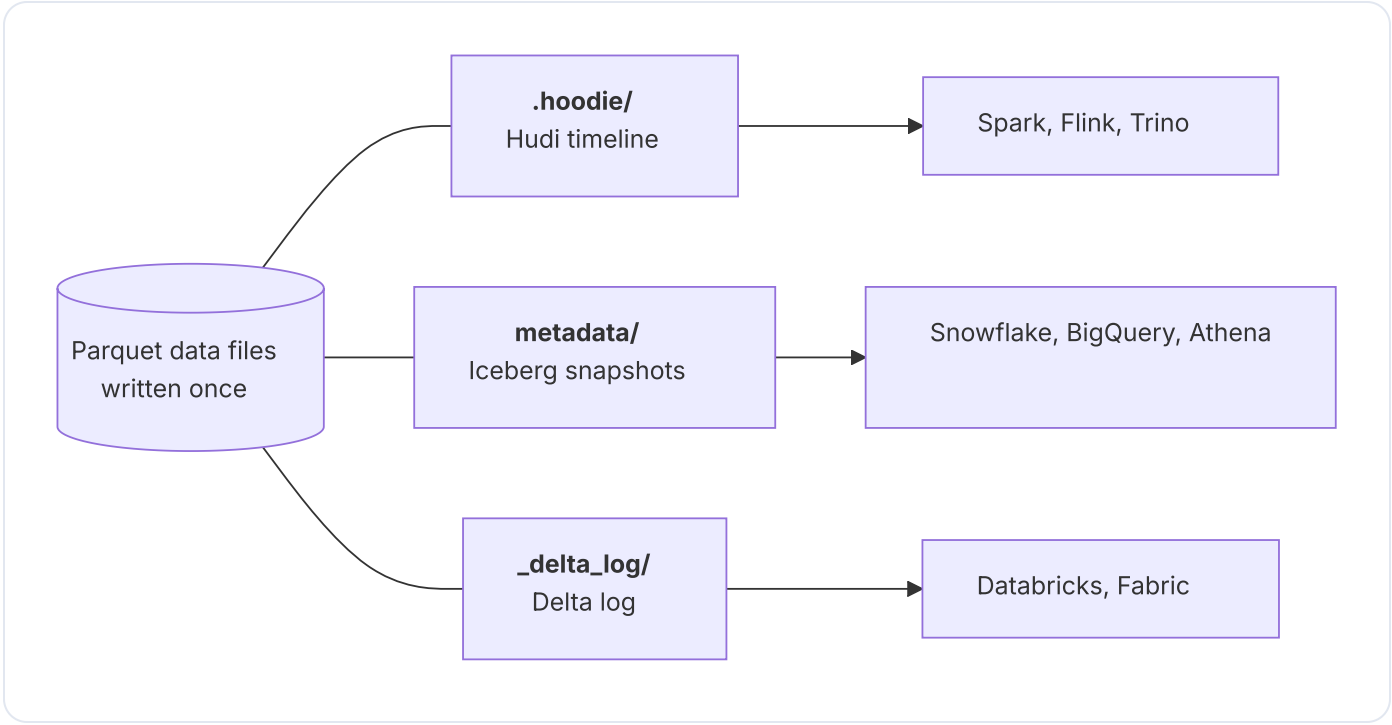
There is a particular kind of meeting where someone says the data is in Hudi and someone else says their warehouse only reads Iceberg, and the next forty minutes go on who is going to maintain the copy. XTable removes the meeting. The Parquet files stay exactly where they are, and a second set of metadata appears beside the first describing the same files in another format.

This is the reference page for doing that: what converts to what, where the output lands, how to build the tool, every flag it takes, and how to register the result so an engine can find it by name. It is written to be kept open in a tab rather than read once.

Assumed knowledge: you have at least one lakehouse table already. For how XTable decides between a full and an incremental pass, see Apache XTable incremental sync.

What does XTable actually do?

It reads the metadata of a table you already have, builds a format-neutral description of it in memory, and writes that description out in the metadata language of another format. The Parquet files are never opened for their contents and never rewritten.



Three consequences follow, and they are the whole value proposition:

- **A conversion costs metadata time, not data time.** The work scales with how many commits and files the table has, not with how many rows.
- **There is one copy of the truth.** Nothing can drift, because there is nothing to drift from. The formats disagree only about how they describe the same files.
- **It is not a runtime.** XTable is a tool you invoke, on a schedule or after an ingest job. It does not sit in the query path, and it is not a new storage format to adopt.

Which formats convert to which?

The honest answer is one file in the repository, `xtable-conversion-defaults.yaml`, which maps each format name to the provider classes that can read and write it. Read it as the matrix rather than trusting a diagram:

FORMAT	CAN BE A SOURCE	CAN BE A TARGET	PROVIDER
Hudi	Yes	Yes	`HudiConversionSourceProvider`, `HudiConversionTarget`

FORMAT	CAN BE A SOURCE	CAN BE A TARGET	PROVIDER
Iceberg	Yes	Yes	`IcebergConversionSourceProvider`, `IcebergConversionTarget`
Delta	Yes	Yes	`DeltaConversionSourceProvider`, `DeltaConversionTarget`
Paimon	Yes	No	`PaimonConversionSourceProvider`, added in 0.4.0
Parquet	Yes, with configuration	No	`ParquetConversionSourceProvider`, added in 0.4.0

Two entries need a note, because both are new and neither behaves like the first three.

Paimon is a source only. A Paimon table can be exposed as Hudi, Iceberg or Delta. Nothing converts back into Paimon, because no Paimon target is registered.

Parquet is a source that the defaults do not list. The provider class exists and the release notes describe partition extraction and incremental sync for it, but `xtable-conversion-defaults.yaml` has no `PARQUET` entry, so `RunSync` will reject it with "Source format PARQUET is not supported" unless you pass your own converters config with `--convertersConfig`. The constant class tells the same story from the other side: `TableFormat` declares five names, and its `values()` helper returns four, leaving `PARQUET` out.

There is also a second implementation of the Delta target. `DeltaKernelConversionTarget` is registered alongside the original and its `getTableFormat()` returns `TableFormat.DELTA`, so it is not a new format but an alternative writer built on Delta Kernel `4.0.0`. You select it by overriding `conversionTargetProviderClass` for `DELTA` in a converters config.

Where does the converted metadata land?

In the place each format expects, inside the table directory. Nothing moves.

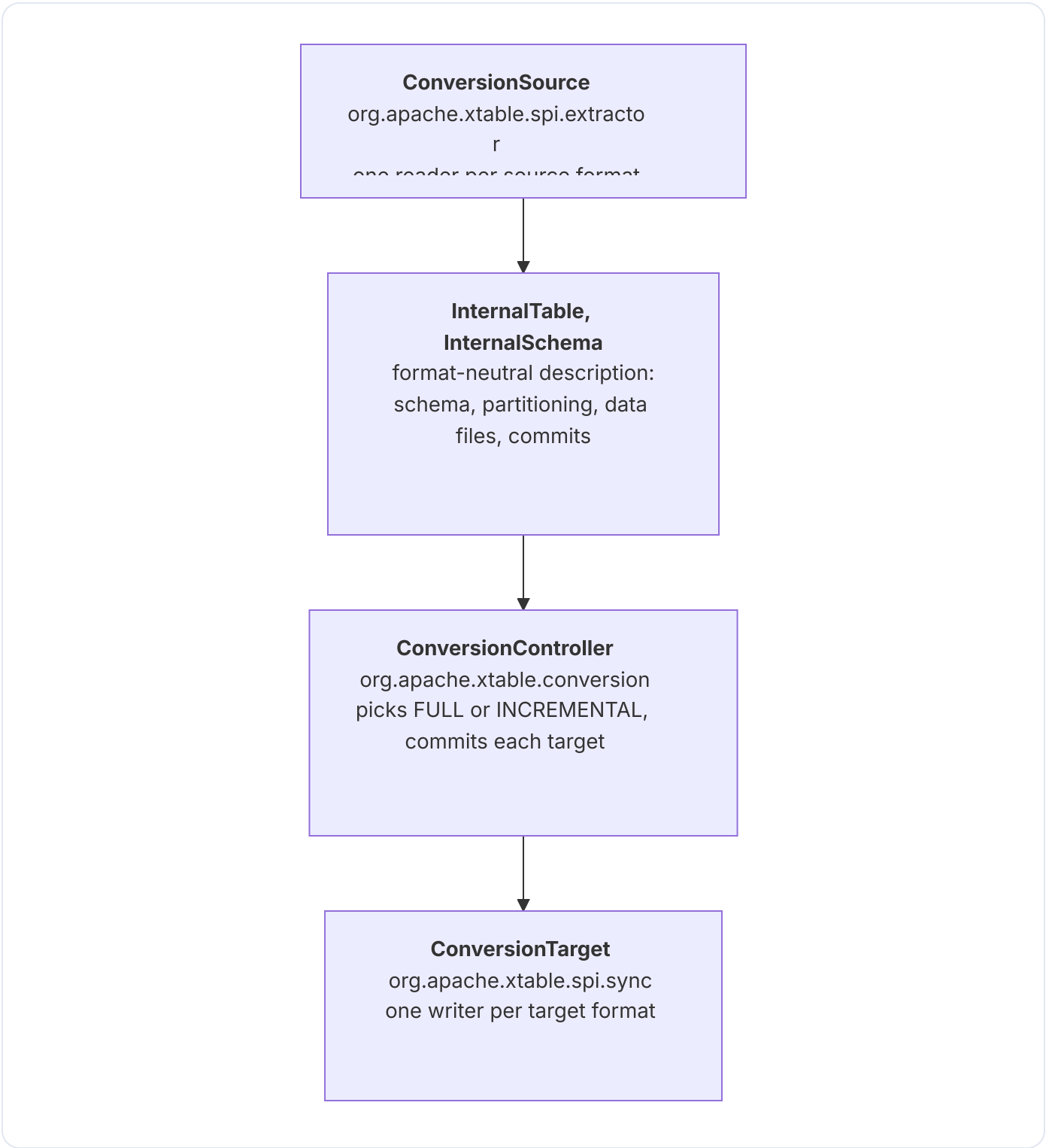
```
s3a://lakehouse-prod/warehouse/trips/
├── city_id=sf/
│   └── 8f3a1c92-...-0_0-24-1893_20260916090000123.parquet
├── city_id=nyc/
│   └── b2e7d410-...-0_0-25-1894_20260916090000123.parquet
├── .hoodie/          <- the source, written by your Hudi pipeline
├── metadata/         <- written by XTable, Iceberg reads this
└── _delta_log/       <- written by XTable, Delta reads this
```

FORMAT	METADATA DIRECTORY	WHAT IS INSIDE
Hudi	`.hoodie/`	The instant timeline and `hoodie.properties`
Iceberg	`metadata/`	`v<N>.metadata.json`, manifest lists, manifests
Delta	`_delta_log/`	Numbered JSON commits and checkpoints

The practical consequence is that a table which has been converted looks, to any engine you point at it, like a native table of that format. There is no shim to install on the read side and no XTable process running when the query runs.

Architecture: three pieces and an internal model

XTable's design is a hub and spoke, and the hub is what makes it N formats rather than N times N converters.


ConversionSource
org.apache.xtable.spi.extracto
r
InternalTable,
InternalSchema
format-neutral description:
schema, partitioning, data
files, commits
ConversionController
org.apache.xtable.conversion
picks FULL or INCREMENTAL,
commits each target
ConversionTarget
org.apache.xtable.spi.sync
one writer per target format

PIECE	INTERFACE	RESPONSIBILITY
Source	`org.apache.xtable.spi.extractor.ConversionSource`	Read the source table's schema, partitioning and commit history into the internal model
Model	`org.apache.xtable.model.InternalTable`, `InternalSchema`	Describe a table without reference to any format
Controller	`org.apache.xtable.conversion.ConversionController`	Decide the sync mode per target, drive the conversion, commit each target
Target	`org.apache.xtable.spi.sync.ConversionTarget`	Write the model out as that format's metadata

Adding a format means writing one source or one target against the model, not a converter for every other format. That is why Paimon arriving in 0.4.0 made it readable as three formats at once rather than as one.

What ships in 0.4.0?

XTable does the conversion using its own bundled copies of the format libraries, so these versions are not trivia. They are the compatibility contract for your source table. This table is what XTable itself is built against, read from the root `pom.xml` at the `0.4.0-incubating` tag. It is not the Spark you run your own jobs on: the quick start below writes and reads the table from Spark 3.5.9.

XTABLE BUILD DEPENDENCY	VERSION IN 0.4.0
Apache Spark	`3.4.2`
Apache Hudi	`0.14.0`
Apache Iceberg	`1.9.2`
Delta Lake	`2.4.0`
Delta Kernel	`4.0.0`
Apache Avro	`1.12.0`
Apache Parquet	`1.15.2`
Scala	`2.12.20`

Check your source table against the matching row before assuming a conversion will work, particularly for Hudi, where table versions move between releases.

What is new in this release, in the order it will matter to you:

- **The bundled jar is runnable again.** The shade plugin sets `org.apache.xtable.utilities.RunSync` as the main class, so `java -jar` works without naming the class.
- **`jol-core` is now inside the jar.** Hudi's `ObjectSizeCalculator` loads it at runtime, and its absence used to surface as `NoClassDefFoundError: org/openjdk/jol/info/GraphLayout`. The pom now pulls it into the shaded artifact deliberately, with a comment saying why.
- **`xtable-spark-runtime` is a new module**, published as `xtable-spark-runtime_2.12`, described in the release notes as a thin drop-in Spark bundle for Spark 3.4 and 3.5.
- **Paimon and Parquet sources**, as covered above.
- **A Maven wrapper.** Build with `./mvnw` and you get the version the project expects.

How do I build it?

There is no published `xtable-utilities` artifact on Maven Central, so the CLI is something you build. Java 11 is what the project asks for.

Clone the release tag, not the default branch. `main` has moved on to a `0.5.0-SNAPSHOT` that bundles Hudi `1.2.0`, so a plain clone builds something other than the 0.4.0 this page describes. At the tag, `xtable-utilities` is also left out of the root pom's `<modules>` list, so a root `./mvnw package` never builds the CLI. Install the modules it depends on, then build it from its own pom:

```
git clone --branch 0.4.0-incubating https://github.com/apache/incubator-xtable.git
cd incubator-xtable
./mvnw clean install -DskipTests \
  -pl xtable-core,xtable-aws,xtable-hive-metastore,xtable-hudi-support/xtable-hudi
./mvnw -f xtable-utilities/pom.xml clean package -DskipTests
```

The shaded jar is large, 1,111,900,566 bytes in this build. With `MAVEN_OPTS=-Xmx1200m` the second command wrote that jar and then ended in `BUILD FAILURE` with `Java heap space`, so give Maven more heap than that rather than trusting a jar from a failed build.

The shaded artifact carries the `bundled` classifier, so glob for it rather than typing a version that will be stale next release:

```
export XTABLE_JAR=$(ls xtable-utilities/target/xtable-utilities_*-bundled.jar)
java -jar "$XTABLE_JAR" --help
```

If you are embedding XTable in your own job instead of shelling out to the CLI, these are on Maven Central at `0.4.0-incubating`:

```
<dependency>
  <groupId>org.apache.xtable</groupId>
  <artifactId>xtable-core_2.12</artifactId>
```

```
  <version>0.4.0-incubating</version>
</dependency>
```

`xtable-api`, `xtable-service` and `xtable-spark-runtime_2.12` are published at the same version. Only the Scala 2.12 build is published, so there is no 2.13 artifact to pick.

Quick start: a Hudi table read as Iceberg and Delta

Every step below was run on Spark 3.5.9 (`apache/spark:3.5.9-python3`, Scala 2.12, Java 11), with the XTable CLI built from the `0.4.0-incubating` tag as above. The writer is Hudi `0.15.1`, the Spark 3.5 bundle, which writes table version 6 (`hoodie.table.version=6` in `hoodie.properties`), the same table version XTable's bundled Hudi `0.14.0` writes. The Iceberg reader is `1.9.2`, matching the Iceberg XTable bundles.

1. Write a Hudi table.

```
pyspark \
  --packages org.apache.hudi:hudi-spark3.5-bundle_2.12:0.15.1 \
  --conf "spark.serializer=org.apache.spark.serializer.KryoSerializer" \
  --conf "spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtensi
  --conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.Hoodie
```

```
table_path = "file:///tmp/lakehouse/trips"

trips = spark.createDataFrame(
    [("trip-1001", "rider-88", 24.50, 1758000000, "sf"),
     ("trip-1002", "rider-12", 61.75, 1758000060, "nyc"),
     ("trip-1003", "rider-88", 18.20, 1758000120, "sf")],
    ["trip_id", "rider_id", "fare_amount", "updated_at", "city_id"])

(trips.write.format("hudi")
    .option("hoodie.table.name", "trips")
    .option("hoodie.datasource.write.recordkey.field", "trip_id")
    .option("hoodie.datasource.write.precombine.field", "updated_at")
```

```
    .option("hoodie.datasource.write.partitionpath.field", "city_id")
    .option("hoodie.datasource.write.hive_style_partitioning", "true")
    .mode("overwrite")
    .save(table_path))

spark.read.format("hudi").load(table_path).show()
```

Leave this shell with `exit()` before the next step. Each format wants its own runtime on the classpath.

2. Describe the conversion. Save as `trips_sync.yaml`:

```
sourceFormat: HUDI
targetFormats:
  - ICEBERG
  - DELTA
datasets:
  - tableBasePath: file:///tmp/lakehouse/trips
    tableName: trips
    partitionSpec: city_id:VALUE
```

`sourceFormat` can be auto-detected, but the javadoc recommends setting it explicitly, because a directory that has already been converted contains metadata for several formats and the guess becomes ambiguous. `partitionSpec` is only needed when the source table is partitioned.

3. Run it.

```
export XTABLE_JAR=$(ls xtable-utilities/target/xtable-utilities_*-bundled.jar)
java -jar "$XTABLE_JAR" --datasetConfig trips_sync.yaml
```

The line to grep for comes from `ConversionController`:

```
2026-09-27 12:37:39 INFO  org.apache.xtable.conversion.ConversionController:215 -
```

The formats are a comma-joined string, not a bracketed list, so match on `Sync is successful for the following formats` rather than on the list.

There is a matching failure line, `Sync failed for the following formats`, and a run can produce both when one target succeeds and another does not. Check for the failure line rather than only for the success one.

4. Read the same directory as Iceberg.

```
pyspark \
  --packages org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.9.2 \
  --conf "spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSes
  --conf "spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCat
```

```
spark.read.format("iceberg").load("file:///tmp/lakehouse/trips").show()
```

5. Read the same directory as Delta. On Spark 3.5 the Delta artifact is `delta-spark`, which replaced `delta-core` from Delta 3.0. It reads the log XTable wrote with its bundled Delta `2.4.0`.

```
pyspark \
  --packages io.delta:delta-spark_2.12:3.3.2 \
  --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" \
  --conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.Delta
```

```
spark.read.format("delta").load("file:///tmp/lakehouse/trips").show()
```

Same three rows, same Parquet files, three formats.

Every RunSync flag

```
java -jar "$XTABLE_JAR" \
  --datasetConfig trips_sync.yaml \
  --hadoopConfig core-site.xml \
  --convertersConfig my_converters.yaml \
  --icebergCatalogConfig iceberg_catalog.yaml \
  --continuousMode \
  --continuousModeInterval 60
```

FLAG	SHORT	REQUIRED	WHAT IT DOES
`--datasetConfig`	`-d`	Yes	The YAML describing source format, targets and datasets
`--hadoopConfig`	`-p`	No	Hadoop XML with filesystem credentials, overriding the defaults
`--convertersConfig`	`-c`	No	Override the provider classes per format. This is how you reach Parquet or the Delta Kernel target
`--icebergCatalogConfig`	`-i`	No	Iceberg catalog configuration, used for any Iceberg source or target
`--continuousMode`	`-m`	No	Run on a loop instead of once, reloading the config file each pass
`--continuousModeInterval`	`-t`	No	Loop interval in seconds. Defaults to 5
`--help`	`-h`	No	Print the usage

`--continuousMode` is the one worth a second look. Because it re-reads the dataset config on every pass, you add or remove tables by editing a file rather than by restarting the job, which makes it a small service instead of a cron entry.

And the dataset config has more fields than most examples show:

FIELD	MEANING
`tableBasePath`	The table root, where metadata is read and written
`tableDataPath`	The data location, when it differs from the base path
`tableName`	The name given to the converted table
`partitionSpec`	Required only when the source is partitioned, for example `city_id:VALUE`
`namespace`	The namespace to place the table in

How do I register the result in a catalog?

Conversion makes the metadata. Registration makes an engine able to find it by name. They are separate steps, and you need both.

For Iceberg, XTable writes through Iceberg's `HadoopTables` API, so `version-hint.text` holds the current metadata version and the file to register is the matching `v<N>.metadata.json`:

```
cat /tmp/lakehouse/trips/metadata/version-hint.text
```

Then register it from a session that defines the catalog you are registering into:

```
spark-sql \
  --packages org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.9.2 \
  --conf "spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSes
  --conf "spark.sql.catalog.hive_prod=org.apache.iceberg.spark.SparkCatalog" \
  --conf "spark.sql.catalog.hive_prod.type=hive" \
  --conf "spark.sql.catalog.hive_prod.uri=thrift://metastore.internal:9083"
```

```
CREATE SCHEMA IF NOT EXISTS hive_prod.iceberg_db;
```

```
-- version-hint.text printed the N to use here
CALL hive_prod.system.register_table(
  table         => 'hive_prod.iceberg_db.trips',
  metadata_file => 'file:///tmp/lakehouse/trips/metadata/v3.metadata.json'
);

SELECT city_id, count(*) FROM hive_prod.iceberg_db.trips GROUP BY city_id;
```

Copy `v<N>.metadata.json` somewhere safe before the `CALL`. In two runs where `register_table` failed inside the Hive commit (a local metastore without its lock tables, first with `CommitFailedException: org.apache.iceberg.hive.LockException`, then with `IllegalStateException: Bug: operationType=UNSET`), Iceberg `1.9.2` deleted the very `v3.metadata.json` it had been asked to register. `version-hint.text` still said `3`, pointing at a file that no longer existed. Both times the table was rebuilt and re-synced before the next attempt.

Delta is simpler, because a location is enough:

```
CREATE SCHEMA IF NOT EXISTS delta_db;

CREATE TABLE IF NOT EXISTS delta_db.trips
USING DELTA LOCATION 'file:///tmp/lakehouse/trips';

SELECT city_id, count(*) FROM delta_db.trips GROUP BY city_id;
```

Doing it for many tables with RunCatalogSync

`RunCatalogSync` registers converted tables into external catalogs in one run, so nobody runs DDL by hand. Two catalog clients are registered through the service loader in 0.4.0: `HMSCatalogSyncClient` for Hive Metastore and `GlueCatalogSyncClient` for AWS Glue. The docs describe those two as the current support, with Unity Catalog, Apache Polaris, Apache Gravitino and DataHub listed as coming.

```
sourceCatalog:
  catalogId: "hms-source"
  catalogType: "HMS"
```

```
    catalogProperties:
      externalCatalog.hms.serverUrl: "thrift://metastore.internal:9083"
  targetCatalogs:
    - catalogId: "glue-target"
      catalogSyncClientImpl: "org.apache.xtable.glue.GlueCatalogSyncClient"
      catalogProperties:
        externalCatalog.glue.region: "us-west-2"
  datasets:
    - sourceCatalogTableIdentifier:
        tableIdentifier:
          hierarchicalId: "hudi_db.trips"
          partitionSpec: "city_id:VALUE"
      targetCatalogTableIdentifiers:
        - catalogId: "glue-target"
          tableFormat: "ICEBERG"
          tableIdentifier:
            hierarchicalId: "iceberg_db.trips"
```

```
java -cp "$XTABLE_JAR" \
  org.apache.xtable.utilities.RunCatalogSync \
  --catalogSyncConfig trips_catalog.yaml
```

Note the `-cp` and explicit class: only `RunSync` is the jar's main class, so `RunCatalogSync` is named. Its flags are `--catalogSyncConfig` (required), `--hadoopConfig` and `--convertersConfig`. Every `catalogId` under `datasets` must match one declared in `sourceCatalog` or `targetCatalogs`, or the config is rejected.

Is there an API instead of a CLI?

Yes, and it is the part of the project moving fastest. `xtable-service` is published on Maven Central at `0.4.0-incubating`, and the repository carries an OpenAPI description at `spec/rest-service-open-api.yaml` defining two operations:

OPERATION	PURPOSE
`POST /v1/conversion/table`	Start a conversion, the same work `RunSync` does
`GET /v1/conversion/table/{conversion-id}`	Poll that conversion for completion

The shape tells you it is asynchronous: you submit and then poll, rather than holding a connection open for the length of a sync. The spec is versioned `0.0.1`, so treat it as early and pin what you build against it.

What does the sync not cover?

This is the section to read before anything else, because one line in the project's own limitations page decides whether XTable fits your table at all:

> Only Copy-on-Write or Read-Optimized views of tables are currently supported. This means that only the underlying parquet files are synced but log files from Hudi and delete vectors from Delta and Iceberg are not captured by the sync.

A Merge-on-Read Hudi table converts as its read-optimized view. Everything sitting in log files since the last compaction is not in the Iceberg or Delta metadata XTable writes. The conversion is not wrong, it is as of the last compaction, and a reader on the converted side sees exactly what a `read_optimized` query would see. The same applies to Delta and Iceberg deletion vectors on the source side.

If your Hudi table is Copy-on-Write, this costs you nothing. If it is Merge-on-Read, compaction frequency becomes the freshness bound on every converted format, which is a scheduling decision rather than a configuration one.

The other documented caveats, all of which are cheap to satisfy once you know:

SOURCE	REQUIREMENT
Hudi	Reading a Hudi target needs Hudi `0.14.0`, with `hoodie.metadata.enable=true` and `hoodie.datasource.write.hive_style_partitioning=true`
Hudi to Iceberg	Set `parquet.avro.write-old-list-structure=false` so list types survive, and field IDs may be needed in the Parquet schema
Delta to Iceberg	Field IDs may be needed, which on the Delta side means enabling column mapping
Delta	Generated columns are not synced to the target schema, and partitioning on them is only partly supported

XTable also does target-side housekeeping rather than leaving it to you: Hudi targets get unreferenced files marked for cleaning, Iceberg targets get snapshots expired after a configured age, and Delta targets get log retention.

Which engines can read the result?

Once the metadata is written, this stops being an XTable question. A converted table is an ordinary table of that format, and the project says so plainly: synced tables “behave the similarly to native tables which means you do not need any additional configurations on query engines’ side”.

So the useful list is not a support matrix but the set of engines the project documents, and which formats each page actually covers:

ENGINE	FORMATS COVERED BY ITS PAGE
Apache Spark	Hudi, Iceberg, Delta
Trino	Hudi, Iceberg, Delta
Presto	Hudi, Iceberg, Delta

ENGINE	FORMATS COVERED BY ITS PAGE
Amazon Athena	Hudi, Iceberg, Delta
Amazon Redshift	Hudi, Iceberg, Delta
Google BigQuery	Hudi, Iceberg, Delta
Microsoft Fabric	Hudi, Iceberg, Delta
StarRocks	Hudi, Iceberg, Delta
Snowflake	Iceberg only

Snowflake being the one Iceberg-only page is the shape of the whole argument for this tool: the warehouse dictates the format, your ingestion pipeline does not have to change to satisfy it, and the gap between the two is metadata XTable can write. Check the engine docs for current, engine-specific instructions before you commit to one.

Trying it without installing anything

The repository ships a Docker demo with Spark on Jupyter, a Hive Metastore, Trino and Presto already wired together:

```
git clone https://github.com/apache/incubator-xtable.git
cd incubator-xtable/demo
./start_demo.sh
```

SERVICE	HOW TO REACH IT
Jupyter	`http://127.0.0.1:8888/`, notebook at `work/demo.ipynb`
Trino	`docker exec -it trino trino`
Presto	`docker exec -it presto presto-cli --server localhost:8082`

Querying the same table from Trino as Iceberg and from Presto as Delta, in two terminals, makes the point faster than any diagram.

This demo is the project's, not this page's, and it was not re-run here. Its notebook takes Spark and Hudi from the root pom's `spark.version` and `hudi.version` (`demo/build_demo.sh` writes them to `versions.properties`), so it runs the Spark XTable builds against, currently `3.4.2`, rather than the 3.5.9 used in the quick start.

Production tips

- **Check your source table type first.** A Merge-on-Read Hudi source converts as its read-optimized view, so compaction frequency, not sync frequency, bounds freshness on every target format.
- **Pin the XTable version and the format libraries together.** The conversion uses XTable's bundled Hudi, Iceberg and Delta, not yours.
- **Run the sync as the last step of ingestion,** not on an independent timer, so a converted table is never newer or older than the commit it claims to describe.
- **Grep for the failure line, not just the success line.** A partially successful run logs both.
- **Prefer `--continuousMode` over cron** when you have more than a handful of tables, since it reloads its config without a restart.
- **Set `sourceFormat` explicitly.** After the first conversion the directory holds several formats and auto-detection has more than one right answer.
- **Treat catalog registration as a separate job** with its own alerting. A converted table nobody registered is invisible, and it fails silently.
- **Remember XTable is incubating.** The disclaimer in the repository is not a formality: config shapes and module names have moved between releases, and the REST spec is at `0.0.1`.

Treat one format as the writer and the others as read-only projections of it. Two writers committing to the same files through different metadata trees have no shared concurrency control, and nothing coordinates them.

To tell whether a run was incremental or full, read the controller log: it records its fallback, including “No previous InternalTable sync for target. Falling back to snapshot sync.” The incremental sync post goes through the decision in detail.

Key takeaways

The idea underneath all of this is smaller than the tooling around it. A table format is a description of a set of files, the files are ordinary Parquet, and nothing stops one set of files from carrying several descriptions at once. Once you accept that, the copy everyone was arguing about in the meeting turns out to be metadata, and metadata is cheap to produce twice.

What 0.4.0 adds is mostly the unglamorous kind of progress that makes a tool usable: a jar that runs with `java -jar`, a dependency that no longer has to be bolted onto the classpath by hand, a Maven wrapper, a Spark runtime bundle. The new sources matter too, because a Paimon or Parquet table becoming readable as three formats is the clearest demonstration that the hub-and-spoke model was worth building.

Where it does not fit: if you only ever use one engine, you are adding a moving part for a problem you do not have. XTable earns its keep when the write side and the read side want different formats and neither is willing to move. That is a common situation, and paying for it in metadata rather than in storage and a sync pipeline is a good trade.

References

- Apache XTable documentation for the quickstart and catalog guides
- XTable release notes for the full change list behind this page
- `xtable-conversion-defaults.yaml`, the authoritative source and target matrix
- `RunSync.java` for every flag and config field
- Apache XTable incremental sync for how the sync mode is chosen

Trademarks

Apache XTable (incubating), Apache Hudi, Apache Iceberg, Apache Paimon, Apache Parquet, Apache Spark, Apache Flink, Apache Avro, Apache Hive and Apache are either registered trademarks or trademarks of The Apache Software Foundation in the United States and other countries. Delta Lake is a trademark of the Linux Foundation. All other marks are the property of their respective owners.