dbt-spark Configuration Options (Apache Spark Adapter)

Model-Level Configurations (dbt-spark)

dbt-spark extends the standard dbt model configs with Spark-specific options for table storage formats, locations, partitioning, and more. These configs can be set in a model's {{ config(...) }} block or as defaults in dbt_project.yml (prefixed with +). The table below summarizes the key model-level configurations:

Config	Description	Possible Values (Default)	Notes / Format Specificity
materialized	How the model is materialized.	'table', 'view', 'incremental', 'ephemeral' ('view' default if not specified)	Standard dbt config. Use 'table' to apply Spark-specific table configs.
file_format	File format (storage format) to use when creating tables .	'parquet' (default), 'delta', 'iceberg', 'hudi', 'csv', 'json', 'text', 'jdbc', 'orc', 'hive', 'libsvm' .	Determines the underlying storage format. For example, 'delta' for Delta Lake tables, 'iceberg' for Apache Iceberg, etc. 'hive' uses Hive's native format (Hive SerDe) . If not set, Spark uses Parquet by default .
location_root	Root directory path for table data (for external tables) . The table name will be appended	Any valid file path (e.g. '/mnt/data/spark_tables'). Default is **unset**, meaning a managed table stored in the warehouse's default location.	Use to create *external tables* at a fixed location. The table directory is created as <location_root>/<table_name> . Ensure the Spark user has access to this path.

	to this path .		
partition_by	Partition the table by one or more columns . Spark will create a directory for each partition value.	A single column or list of columns (e.g. ['date_day']). Default **none** (no partitioning).	Improves read performance by pruning partitions. If used with incremental models, can enable partition-wise overwrites (see **insert_overwrite** strategy). Not applicable to views.
clustered_by	Column(s) to cluster (bucket) each partition by . Requires specifying **buckets**.	A single column or list of columns (e.g. 'country_code').	Used with **buckets** to create clustered/bucketed tables. Bucketing is mainly relevant for Hive/Parquet tables to distribute data evenly; not commonly used with Delta/Iceberg.
buckets	Number of buckets for clustering . Required if **clustered_by** is set.	Integer > 0 (e.g. 8).	Defines how many buckets to create per partition when using clustering. Ignored if **clustered_by** is not set.
tblproperties	Table properties to set on the table (key-value pairs) . These are stored in the metastore	Dict of property names to values. Example: \ntblproperties: \n commit.retry.num_retries: "10"\n delta.autoOptimize.optimizeWrite: "true"\n	Use for format-specific settings. For example, Iceberg table properties (e.g. split sizes) , Delta properties (e.g. auto-compaction) , or Hudi properties. Refer to format docs for valid keys (Spark

	to configure table behavior.		[Parquet][58], [Delta][59], [Iceberg][57], [Hudi][60]) .
options	Data source **OPTIONS** to use when creating the table. Allows setting write options like CSV headers, compression codecs, or JDBC connection info.	Dict of option keys to values. Example: options: {'header': 'true', 'compression': 'gzip'}.	This injects an OPTIONS (...) clause in the CREATE TABLE statement (supported by Spark for data sources). Useful for CSV/JSON (e.g. header option) or Parquet compression. **Note:** For Delta tables, options may be restricted (Delta uses table properties instead) . For file_format='jdbc', options **must** include the JDBC connection details (e.g. 'url', 'driver', 'dbtable', and credentials).

Example – Default Table Configs: In dbt_project.yml, you can set default configs for all Spark models. For instance, to default all models and seeds to Delta tables in an external location:

```
models:
  +file_format: delta       # use Delta Lake format for tables
  +location_root: /mnt/data/spark_tables  # default external table location
seeds:
  +file_format: delta       # also apply Delta format to seeds
```

. This ensures tables are created as Delta Lake by default, unlocking advanced features (e.g. merge, snapshots) .

Additional Spark Model Configs:

- **persist_docs:** If enabled, dbt will persist model descriptions to the metastore as table comments. Set in dbt_project.yml under models: as: persist_docs: {relation: true, columns: true}. In Spark, table-level comments will appear in DESCRIBE TABLE EXTENDED results . Column-level comments are only fully supported on **Delta** tables (Spark's Hive metastore may not store column comments for other formats) .

- **schema vs. database:** On Spark, **always use schema and never database** in configs or profiles . Spark uses "database" and "schema" interchangeably (they represent a Hive database), and dbt treats database as a higher-level concept not applicable to Spark. To control where models go, use the schema config (which corresponds to the Hive database/namespace) .

Incremental Model Configurations

When materialized='incremental', additional configs determine how dbt-spark builds and updates the table. Key incremental configs include the update **strategy**, how to match records, and schema change handling:

Config	Description	Allowed Values (Default)	Notes / Format Requirements
incremental_strategy	Specifies *how* to apply new data on incremental runs . Controls whether new data is appended, partitions are overwritten, or merges are done.	'append' (default), 'insert_overwrite', 'merge', 'microbatch' .	**append:** Insert new rows only (no updates) . Works on all file formats; simplest approach (may create duplicates) . **insert_overwrite:** Overwrite entire table or affected partitions with new data . Effective with **partition_by** (replaces only partitions in the incoming data) . *Not supported for Delta tables* . **merge:** Perform UPSERT (update existing rows, insert new) via an atomic *MERGE* statement . **Requires** file_format = **delta**, **iceberg**, or **hudi** (formats that support MERGE). Also requires a **unique_key** for matching records (if no unique_key, all new data is treated as inserts) . **microbatch:** New in dbt v1.9+, splits large time-based incremental loads into independent *batches*. See **Microbatch** configs below.

unique_key	Primary key used to match records in **merge** strategy (or other upsert strategies).	Column name (or multiple columns) in the model. No default (not required for append/overwrite).	Only applies when using incremental_strategy='merge' (or certain database-specific upsert strategies). For Spark's merge, if unique_key is provided, matching existing rows on this key allows updates; if omitted, merge will insert all rows without updates (effectively an append) . On Spark, **merge** (and thus unique_key) is available for Delta, Iceberg, or Hudi tables .

on_schema_change	How to handle *new or changed columns* in the incremental model schema.	'ignore' (default), 'fail', 'append_new_columns' (and in dbt-core ≥1.4: 'sync_all_columns').	**ignore:** (Default) Ignore schema differences – do not update the target table's schema for new columns . New columns will **not** appear in the existing table (no error) . **fail:** Throw an error if the model's columns don't match the table's schema . **append_new_columns:** Add new columns to the target table schema on incremental run (supported if the underlying Spark metastore allows ALTER TABLE ADD COLUMN). Spark's Delta and Iceberg support adding columns, so this is useful for those formats . **sync_all_columns:** (If supported) Make target table schema exactly match the model (adding new columns *and* dropping/reordering if necessary). This may trigger a full reload. **Note:** on_schema_change was introduced in dbt Core 1.0; full support in dbt-spark came later. In practice, **append_new_columns** is commonly used to evolve schemas on Delta Lake or Iceberg without full refresh .

Example – Incremental Config in a Model:

```
-- models/events_incremental.sql
{{ config(
   materialized='incremental',
   incremental_strategy='merge',
   unique_key='event_id',
   file_format='delta',
   partition_by=['event_date']
) }}
```

SELECT * FROM {{ ref('stg_events') }}

In this example, on incremental runs dbt will use a Delta Lake MERGE on the event_id key, partitioning the table by date.

Microbatch Incremental Configs

Microbatch is a special incremental strategy designed for large time-series data (available in dbt Core ≥ 1.9). Instead of processing all new data in one query, dbt splits the load into time-based “batches” and processes each batch separately, which can improve efficiency and allow parallel execution . When incremental_strategy: 'microbatch' is set, the following configs must be provided:

Config	Description	Example / Allowed Values	Notes
event_time	The timestamp or datetime column that defines the **time basis** for each batch . dbt uses this field to split data into sequential time ranges.	A column name present in the model (e.g. 'event_timestamp'). **Required** for microbatch.	Must be present in the model query and ideally in upstream sources/models. dbt will automatically apply filters on upstream refs/sources that also have an event_time config to constrain their data to the batch range .
batch_size	The length of each time batch interval .	'day' (default), 'hour', 'minute', etc. (Time grain strings).	Defines how dbt slices the time window. For example, 'day' means each batch covers one day’s worth of data.
begin	The start of the timeline for the first batch (inclusive) .	A timestamp or date string in **UTC** (e.g. '2024-01-01' or '2024-01-01 00:00:00').	Optional – if not set, you must specify --event-time-start on the first run to indicate where to begin. Using begin in the config fixes the historical start

			point for this incremental model .
lookback	How many recent batches to reprocess on each run (to catch late-arriving data) .	Integer (default 1) . For example, lookback: 3 to always reprocess the last 3 batches.	A lookback of 1 means the previous batch is reloaded every run (e.g. reprocess yesterday's data to catch late events) . Set to 0 to process only new batches.
concurrent_batches	Override for parallel batch execution. If set, dbt will run up to this many batches simultaneously.	Integer (optional). e.g. concurrent_batches: 4.	By default, dbt auto-detects concurrency based on thread count and adapter support . This config forces a specific degree of parallelism for microbatch loading.
full_refresh	Whether the model responds to a --full-refresh run by rebuilding from scratch.	false or true (default is true for incremental models).	**Best practice:** set full_refresh: false for microbatch models . A microbatch model should be backfilled via explicit --event-time-start/--event-time-end ranges instead of full-refresh, to avoid undefined behavior .

Microbatch usage: With the above configs, a model's SQL is written to process **one batch** at a time (i.e. covering a specific event_time range). dbt will handle splitting the overall time range into batches and will inject appropriate WHERE filters on event_time for each batch during execution . For example, if batch_size: 'day', dbt will run the model separately for each day's data not yet processed (plus lookback days), and use efficient partition replacement on Spark. On dbt-spark, the microbatch strategy uses the **insert_overwrite** mechanism under the hood for each batch (replacing partitions by date) , so **partition_by is required** in the model config when using microbatch on Spark (e.g. partition_by: ['event_date'] matching the event_time grain).

Snapshot Configurations (dbt-spark)

Snapshots in dbt track slowly changing records over time. On Spark, snapshot support is available **only when using file formats that support upserts** (Delta, Iceberg, or Hudi). In fact, the adapter will error if you attempt a snapshot without a supported format: *“Snapshot functionality requires file_format be set to ‘delta’ or ‘iceberg’ or ‘hudi’.”* . Key snapshot configs include:

Config	Description	Allowed Values / Example	Notes (Spark-specific)
strategy	Whether the snapshot uses a *timestamp* (append-only) approach or checks *all columns* for changes.	'timestamp' or 'check' (must be specified per snapshot).	In a snapshot .sql file, use e.g. {{ config(strategy='timestamp', ...) }}. **timestamp**: Requires an **updated_at** column. dbt records new rows where this timestamp is newer than last snapshot. **check**: Requires specifying which columns to monitor (see **check_cols**). Both strategies require a unique key.
unique_key	The primary key that identifies a record (used to detect new vs. existing rows).	Column name(s) (e.g. 'user_id').	Required for all snapshots – this key is how new records are matched to existing snapshot rows.
updated_at	(Timestamp strategy only) The column name that indicates the last update time of a record.	Column name (e.g. 'last_update_ts').	Must be a datetime or timestamp. dbt uses the max updated_at per unique key to decide if a new row represents a change.

check_cols	(Check strategy only) Which columns to compare to detect changes.	List of columns, or 'all'. Default is 'all' (all columns except the unique key) .	If 'all', any difference in any non-key column triggers a new snapshot record. You can list specific columns to limit the change detection scope.
file_format	Storage format for the snapshot table.	'delta', 'iceberg', or 'hudi' **required** for Spark.	As noted, Spark snapshots only work with these table formats that support merge/upsert operations. For example, set {{ config(file_format='delta') }} in the snapshot to use Delta Lake.
target_schema	Schema (database) to store the snapshot table.	Any existing schema name (by default, snapshots often go in a separate schema, e.g. "snapshots").	This can be set in dbt_project.yml (e.g. snapshots: target_schema: snapshots) or per snapshot. On Spark, this corresponds to a Hive database (same as “schema”).

Note: Snapshots on Spark are implemented via **MERGE** statements under the hood (to apply updates and inserts to the snapshot table). Therefore, they inherit the same requirements as incremental merges. Snapshots will fail or be ineffective if used on formats like Parquet that lack native merge support. Ensure file_format is set appropriately (Delta is commonly used for snapshots on Spark) . Also, as with incremental models, adding new columns to a snapshot might require on_schema_change handling (though often snapshots are rebuilt if schema changes).

Source-Level Configurations

Source definitions (in YAML files under models or sources) can also have configuration properties. While dbt-spark doesn’t introduce many adapter-specific source configs, you should be aware of general source config options:

- **Quoting:** Control whether dbt quotes the database/schema/identifier when referencing a source. Spark SQL is generally case-insensitive and defaults to lowercase unquoted names. If your source name or schema is case-sensitive or contains special characters, you may need quoting: true. Example YAML:

```
sources:
  - name: my_source
    schema: Analytics     # Spark will interpret unquoted as lowercased
    quoting:
      schema: false       # do not quote schema, to avoid case-sensitivity issues
      identifier: true    # quote table name if it has special chars
```

- By default, dbt will quote identifiers in sources if necessary. In Spark, leaving schema unquoted (lowercased) is often recommended since Hive metastore lowers names.

- **Freshness & Testing Configs:** These apply to sources for data quality checks. For example, loaded_at_field specifies a timestamp field to check source freshness, and freshness thresholds (warn_after, error_after) define staleness criteria. These are not Spark-specific, but they are part of source configuration. E.g.:

```
- name: my_source
  tables:
    - name: users
      loaded_at_field: last_update_ts
      freshness:
        warn_after: {hours: 24}
        error_after: {hours: 48}
```

- This would let you run dbt source freshness to ensure the source data is up-to-date. Spark's adapter fully supports source freshness checks.

Other general source configs like enabled (to disable/enable a source) and tags can be used as in any dbt project. There are no unique source config fields required exclusively for Spark beyond standard dbt.

Runtime Configurations (Connection/Profile Settings)

To use dbt-spark, you must configure a **profile** in profiles.yml with the connection details for Spark. The adapter supports multiple connection methods, each with its own required settings. Below are the supported method options and their configurations :

Connection Method	Description & Use Case	Required Profile Fields	Optional Fields / Notes

odbc	Connect via an ODBC driver (typically the Databricks Spark ODBC driver) over HTTPS . Use this for connecting to **Databricks SQL Warehouses or clusters** via ODBC.	type: spark method: odbc driver: ODBC driver path (Simba Spark driver) host: Databricks host (e.g. xxxx.cloud.databricks.com) **Either** endpoint (SQL Warehouse ID) **or** cluster (Cluster ID) token: Databricks personal access token (for auth) .	port: default 443 user: (Often not required, can use token) organization: (Azure Databricks only, your Org ID) . **Notes:** ODBC is specific to Databricks; for Spark on other platforms, use thrift or http. Make sure the ODBC driver is installed and accessible.
thrift	Connect to a Spark Thrift Server (HiveServer2) via **JDBC/Thrift**. Use for Spark clusters that expose a Thrift JDBC interface (e.g. Spark on EMR, HDInsight, or a local Spark Thrift server) .	type: spark method: thrift host: Hostname of the Thrift server schema: database (schema) name to use .	port: Thrift port (default **10001** for Spark Thrift; note: Spark's Thrift server often runs on 10001, while 10000 might be Hive CLI) . user: Username (if auth required) auth: Authentication type, e.g. 'KERBEROS' (and optional kerberos_service_name, e.g. 'hive') for Kerberized clusters . use_ssl: true/false for SSL (default false) . **Notes:** Ensure the Spark Thrift server is running on the cluster (for EMR, run start-thriftserver.sh) . This method uses PyHive/Thrift under the hood.

http	Connect via Spark's HTTP endpoint. **Databricks clusters** support a proprietary HTTP Spark API (Spark UI/statement execution). Use this for Databricks if not using ODBC, or other Spark services that provide an HTTP SQL interface .	type: spark method: http host: Databricks workspace host token: Databricks auth token cluster: Cluster ID (for all-purpose clusters) schema: database name to use.	port: default 443 organization: Org ID (Azure) user: (optional, not usually needed with token) connect_timeout: seconds to wait for connection (default 10) connect_retries: how many times to retry connecting (default 0) . **Notes:** The HTTP method will poll the Spark cluster via REST. Databricks interactive clusters can take time to start; you can increase connect_timeout and connect_retries to wait longer and retry .
session	Use an **existing local Spark session** (PySpark) already running. This method runs dbt *in the same process* as a Spark session, ideal for local development or integration with Spark unit tests .	type: spark method: session schema: default schema (database) name to use host: (not used, but required placeholder – can set to None or 'NA') .	server_side_parameters: A dict of Spark config settings to apply to the session . E.g. {"spark.sql.shuffle.partitions": "16"} to tweak session configs. **Notes:** The Spark session must be active (e.g. if you run dbt within pyspark or using spark-submit with an existing Spark context). This method bypasses network connections.

Optional Profile Settings: These can be added under the target profile to handle retries and Spark settings:

- **connect_timeout & connect_retries:** For http (and ODBC) connections, you can configure how long to wait for a connection and how many retry attempts. For example, connect_timeout: 60 and connect_retries: 5 will try for up to 5 minutes total . This is useful for clusters that take time to warm up.

- **retry_all:** When set to true, *any* query that fails will be retried automatically, not just the initial connection . dbt-spark will use the above timeout/retry settings for **each** query if you enable retry_all . This is a blunt but effective tool for transient Spark failures. It's recommended in production if you occasionally encounter flaky failures . Example profile config:

```
outputs:
  dev:
    type: spark
    method: http
    ... (connection details) ...
    retry_all: true
    connect_timeout: 5
    connect_retries: 3
```

- This would retry failed queries up to 3 times with a 5-second timeout each .

- **server_side_parameters:** A dictionary of Spark session config settings to apply on connection . This can be used with any method. It's equivalent to setting Spark properties like --conf on spark-submit. For example, in a profile:

```
outputs:
  dev:
    type: spark
    method: thrift
    host: my.spark.cluster
    schema: analytics
    server_side_parameters:
      "spark.sql.sources.partitionOverwriteMode": "DYNAMIC"
      "spark.driver.memory": "4g"
```

- This would configure the Spark session to use dynamic partition overwrite (required for insert_overwrite to work properly on some connections) and set driver memory to 4 GB. Use this to set any Spark [application properties][104] needed (e.g. to activate an Iceberg catalog, etc.) .

Databricks vs. Spark: Note that while dbt-spark can connect to Databricks, the **dbt-databricks** adapter is now recommended for Databricks-specific features (like Unity Catalog). dbt-spark's configs above will work for Databricks, but some capabilities (e.g. multiple catalogs, Unity Catalog, etc.) are only in dbt-databricks. If using Databricks, consider migrating . For pure Apache Spark (including EMR, Spark on Kubernetes, etc.), dbt-spark is the correct adapter.

Format-Specific Considerations (Hive vs. Delta vs. Iceberg)

To close, it’s important to understand how different file formats behave with these configs in dbt-spark:

- **Hive/Parquet Tables:** If you use the default parquet (or Spark’s built-in Hive format), incremental updates cannot use merge. You’ll rely on append or insert_overwrite strategies. Partitioning is supported (creates Hive-partitioned Parquet files), as are clustering/bucketing (for Hive). These tables use the Hive metastore for schema. Table properties (tblproperties) apply to the Hive table metadata. For example, a parquet table can set a compression codec via options or tblproperties. Hive-format tables (e.g. file_format: 'hive') use the old Hive serde default (textfile unless otherwise specified) – this is rarely used, but available for compatibility.

- **Delta Lake:** Using file_format: 'delta' unlocks the most features on Spark. Delta tables support **MERGE** (incremental_strategy='merge' with unique_key) , **microbatch** (with partitioning), and **snapshots**. Delta also allows **schema evolution** (e.g. on_schema_change='append_new_columns' will perform an ALTER TABLE for new columns). **Partitioning** can be used with Delta, but note that the insert_overwrite strategy is **not** supported for Delta tables (Delta has its own efficient upsert/replace mechanisms; use merge or full refresh for partition replacements). Delta tables also support **table properties** (e.g. delta.autoOptimize.optimizeWrite = true). Additionally, Delta is the only format on Spark that supports storing **column comments** in the metastore reliably . Overall, if you need slowly-changing data or merge/update capabilities on Spark, choosing Delta (or Iceberg/Hudi) as the file_format is recommended .

- **Iceberg:** Using file_format: 'iceberg' is supported in dbt-spark to create Apache Iceberg tables. Iceberg also supports merges and upserts (the adapter uses Spark’s Iceberg MERGE INTO under the hood for incremental merges). You should ensure the Spark environment is configured with the Iceberg engine (e.g. appropriate Spark packages and a catalog set via spark.sql.catalog... properties). Iceberg tables accept many custom tblproperties for things like snapshot retention, compaction, etc. (see [Iceberg docs][57]). Partitioning in Iceberg can be configured via partition_by as well – Iceberg will use those as partition fields (with its hidden partitioning). **Snapshots:** Iceberg tables can work with dbt snapshots (since Iceberg supports row-level inserts and updates). The dbt-databricks adapter recently added a table_format: iceberg option for Unity Catalog, but in open-source Spark you simply use file_format: iceberg . Note that if using Iceberg, you may need to configure a catalog in spark_conf (e.g. set spark.sql.catalog.spark_catalog to Iceberg’s catalog) .

- **Hudi:** file_format: 'hudi' is also available. Hudi tables support upserts and incremental pulls. dbt-spark treats it similarly to Delta/Iceberg for incremental models (allowing merge strategy). Hudi-specific table properties (e.g. precombine keys) can be set via tblproperties or options. Ensure the Hudi Spark bundles are installed on your cluster if using this format.

In summary, **dbt-spark** provides a rich set of configurations to control how models are materialized on Spark. By choosing the appropriate file_format and configs (e.g. Delta for merge capabilities, or Parquet for simplicity), and by tuning incremental strategies and cluster settings, you can adapt dbt to the nuances of Hive, Iceberg, or Delta Lake on Spark. The tables above serve as a reference for all major config options, their purposes, defaults, and any format-specific caveats – allowing you to get the most out of dbt on Spark's distributed platform ** **.

Sources:

- Official dbt documentation for Spark adapter configurations
- dbt Labs documentation on incremental strategies and table formats
- Apache Spark 3.x SQL reference for Hive/Delta/Iceberg table creation syntax
- Example usage from community and dbt project configs