

dbt Reference

From first model to production-grade ELT transformation layer.

Last reviewed 29 Sep 2026 · Lab-tested with dbt-core 1.12.5 ·

Prerequisites: [SQL](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/sql-reference/) · [Data Modeling](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/data-modeling/)

Related: [Snowflake](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/snowflake-reference/) · [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) · [Git for DE](https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/) (https://sarangambekar1997.github.io/data-engineering-handbook/00-foundations/git-for-de/) · [Airflow](https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/03-orchestration/airflow-reference/) · [BI Tools](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/bi-tools/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/bi-tools/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Practice: [Lab 02 — dbt Transformations](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/02-dbt-transformations) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/02-dbt-transformations)

Overview

Purpose: dbt (data build tool) runs SQL `SELECT` statements and materializes the results as tables or views in your warehouse.

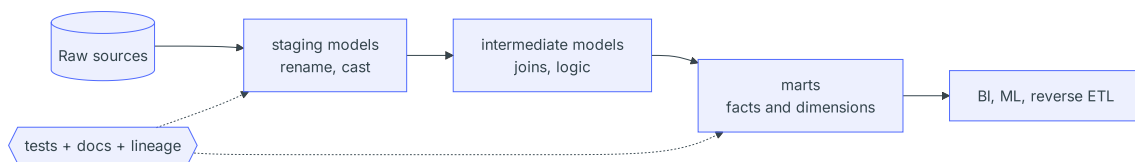
Around that core idea, it provides what a maintainable transformation layer needs:

Without dbt:	With dbt:
- Raw SQL files in a folder	- SQL files with dependency tracking
- Run manually or via random scripts	- <code>`dbt run`</code> runs everything in the right order
- No tests	- <code>`dbt test`</code> validates data quality
- No documentation	- <code>`dbt docs generate`</code> creates a searchable catalog

- No versioning
- reviewable SQL
- Copy-paste table references
- auto-resolves dependencies
- Git-native, PR-reviewable
- `ref('model_name')`

In short: dbt manages SQL transformations the way an orchestrator manages pipeline scheduling.

Out of scope for dbt: - Moving data from source systems into the warehouse — handled by ingestion tools or custom pipelines - Scheduling — handled by an orchestrator or dbt Cloud's scheduler - General-purpose Python processing — dbt is SQL-first (Python models exist for specific cases)



On this page

Basics - [What is dbt?](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#what-is-dbt) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#what-is-dbt) - [Project Structure](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#project-structure) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#project-structure) - [Models](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#models) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#models) - [Materializations](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#materializations) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#materializations) - [Sources](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#sources) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#sources) - [Seeds](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#seeds) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#seeds)

Intermediate - [ref\(\) and source\(\)](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#ref-and-source) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#ref-and-source) - [Tests](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#tests) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#tests) - [Documentation](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#documentation) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#documentation) - [Jinja & Macros](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#jinja--macros) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#jinja--macros) - [dbt CLI Commands](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#dbt-cli-commands) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#dbt-cli-commands)

Advanced - [Incremental Models](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#incremental-models) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#incremental-models) - [Snapshots](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#snapshots) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#snapshots) - [Packages](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#packages) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#packages) - [Hooks & Operations](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#hooks--operations) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#hooks--operations) - [Advanced Macros](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#advanced-macros) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#advanced-macros)

reference/#advanced-macros) - [Exposures & Metrics](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#exposures--metrics) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#exposures--metrics) - [Best Practices](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#best-practices) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#best-practices)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#further-reading)

What is dbt?

dbt (data build tool) is the **T in ELT**. It takes raw data already loaded into your warehouse and transforms it into analytics-ready tables using SQL + a Python project structure.

```
Source DB → [Extract + Load] → Raw tables in warehouse → [dbt transforms] → Analytics tables
                                     (Fivetran/Airbyte)
(dbt)
```

What dbt does: - Compiles `.sql` model files and runs them against your warehouse - Manages model dependencies (builds in the right order) - Runs data tests to validate output - Generates documentation and a data lineage graph - Handles incremental loading patterns - Manages SCD Type 2 via snapshots

What dbt does NOT do: - Extract or load data (that's Fivetran, Airbyte, Stitch, custom pipelines) - Move data between systems - Run Python (dbt-core runs SQL; dbt Python models are a separate feature)

Project Structure

```
my_dbt_project/
|
├─ dbt_project.yml          # project config: name,
version, model paths, vars
├─ profiles.yml            # warehouse connections
(usually in ~/.dbt/)
├─ packages.yml            # external package
dependencies
```

```

├── models/                                # SQL transformation files
│   ├── staging/                          # clean raw source data, 1-to-
1 with source tables
│   │   ├── _sources.yml                 # source definitions
│   │   ├── _staging.yml                 # model documentation and
tests
│   │   ├── stg_orders.sql
│   │   └── stg_customers.sql
│   ├── intermediate/                    # business logic that combines
staging models
│   │   └── int_order_items.sql
│   └── marts/                           # final analytics-ready tables
│       ├── core/
│       │   ├── dim_customer.sql
│       │   ├── fct_orders.sql
│       │   └── _mart_core.yml
│       └── finance/
│           └── fct_revenue.sql
├── tests/                               # custom SQL tests (singular
tests)
│   └── assert_positive_revenue.sql
├── macros/                              # reusable Jinja macros
│   └── utils.sql
├── seeds/                               # CSV files loaded as static
tables
│   └── country_codes.csv
├── snapshots/                           # SCD Type 2 history tracking
│   └── orders_snapshot.sql
└── analyses/                            # ad-hoc SQL (not
materialized)
    └── revenue_deep_dive.sql

```

dbt_project.yml

```

name: 'my_dbt_project'
version: '1.0.0'
config-version: 2

profile: 'my_snowflake_profile'

model-paths: ["models"]
test-paths:  ["tests"]
seed-paths:  ["seeds"]
macro-paths: ["macros"]

# Default materializations by folder
models:
  my_dbt_project:
    staging:
      +materialized: view
      +schema: staging
    intermediate:
      +materialized: ephemeral
  marts:
    +materialized: table
  core:
    +schema: core
  finance:
    +schema: finance

vars:
  start_date: '2024-01-01'
  environment: 'dev'

```

profiles.yml (~/.dbt/profiles.yml)

```

my_snowflake_profile:
  target: dev
  outputs:
    dev:
      type: snowflake
      account: myaccount.us-east-1
      user: "{{ env_var('SNOWFLAKE_USER') }}"
      password: "{{ env_var('SNOWFLAKE_PASSWORD') }}"
      role: transformer

```

```

    database: analytics_dev
    warehouse: transform_wh
    schema: dbt_myname          # developer-specific
schema
    threads: 4

prod:
    type: snowflake
    account: myaccount.us-east-1
    user: "{{ env_var('SNOWFLAKE_USER') }}"
    password: "{{ env_var('SNOWFLAKE_PASSWORD') }}"
    role: transformer
    database: analytics
    warehouse: transform_wh
    schema: marts
    threads: 8

```

Models

A model is a single `.sql` file containing a `SELECT` statement. dbt wraps it in `CREATE TABLE AS` or `CREATE VIEW AS` automatically.

```

-- models/staging/stg_orders.sql
-- Clean and rename columns from the raw source table

SELECT
    order_id,
    customer_id,
    LOWER(TRIM(status))           AS status,
    TRY_CAST(amount AS NUMBER(12, 2)) AS amount,
    TO_TIMESTAMP_NTZ(created_at_epoch) AS created_at,
    _loaded_at                   AS ingested_at
FROM {{ source('raw', 'orders') }}
WHERE order_id IS NOT NULL

```

```

-- models/marts/core/fct_orders.sql
-- Fact table joining staging models

SELECT
    o.order_id,

```

```

    o.customer_id,
    c.full_name      AS customer_name,
    c.country,
    o.status,
    o.amount,
    o.created_at,
    DATE(o.created_at) AS order_date
FROM {{ ref('stg_orders') }} o
JOIN {{ ref('stg_customers') }} c USING (customer_id)

```

Model-level configuration (in the SQL file)

```

-- Set config at the top of the model
{{ config(
    materialized = 'table',
    schema       = 'core',
    tags         = ['daily', 'finance'],
    post_hook    = "GRANT SELECT ON {{ this }} TO ROLE
analyst"
) }}

SELECT ...

```

Materializations

Materialization	What dbt creates	When to use
view	A SQL view (no data stored)	Staging layer, lightweight transforms; fast to build
table	A full table rebuilt every run	Mart/reporting tables that are manageable in size
incremental	Appends/merges only new rows	Large tables where full rebuild is too slow
ephemeral	A <u>CTE</u> inlined into downstream models — nothing persisted	Intermediate logic you don't need to query directly

```
# Set in dbt_project.yml or per-model config block
models:
  my_project:
    staging:
      +materialized: view      # all staging models →
views
  marts:
    +materialized: table      # all mart models → tables
```

Sources

Sources define your raw tables — the data dbt does NOT own. Declaring them enables: -
 {{ source() }} references in models - Source freshness checks - Lineage from raw
 tables through to marts

```
# models/staging/_sources.yml
version: 2

sources:
  - name: raw                # logical name
    database: raw_db         # actual Snowflake
    database
    schema: public           # actual schema
    loaded_at_field: _loaded_at # column to check for
    freshness
    freshness:
      warn_after: {count: 6, period: hour}
      error_after: {count: 24, period: hour}

    tables:
      - name: orders
        description: "Raw orders from the transactional
database"
        columns:
          - name: order_id
            description: "Unique order identifier"
            data_tests:
              - not_null
              - unique
```

```
- name: customers
  description: "Raw customer records"
```

```
# Check source freshness
dbt source freshness
```

Seeds

Seeds are CSV files in the `seeds/` directory that dbt loads as static tables. Use for small, slow-changing reference data (country codes, status mappings, etc.).

```
# seeds/order_status_map.csv
status_code,status_label,is_terminal
placed,Order Placed,false
shipped,Shipped,false
delivered,Delivered,true
cancelled,Cancelled,true
returned,Returned,true
```

```
dbt seed # load all seeds
dbt seed --select order_status_map
```

```
-- Reference a seed in a model
SELECT o.order_id, m.status_label
FROM   {{ ref('stg_orders') }} o
JOIN   {{ ref('order_status_map') }} m ON o.status =
m.status_code
```

```
# seeds/schema.yml – configure types and tests
seeds:
  - name: order_status_map
    config:
      column_types:
        status_code: varchar(20)
        is_terminal: boolean
```

ref() and source()

These two functions are the foundation of dbt's dependency graph.

```
-- source() — reference a raw source table
FROM {{ source('raw', 'orders') }}
-- Compiles to: FROM raw_db.public.orders

-- ref() — reference another dbt model
FROM {{ ref('stg_orders') }}
-- Compiles to: FROM analytics_dev.dbt_alice.stg_orders (in dev)
--                  FROM analytics.marts.stg_orders (in prod)
-- AND adds an edge in the DAG: this model depends on stg_orders
```

Why ref() matters: 1. dbt resolves the correct database + schema per environment automatically 2. dbt builds models in the correct dependency order 3. dbt generates the lineage graph from `ref()` calls

Tests

Tests are SQL queries that return rows when they **fail**. Zero rows = test passed.

dbt 1.8+: the YAML key is `data_tests:` (the older `tests:` still works but is deprecated), and `unit_tests:` were added for testing model logic against mock inputs — see [Unit tests](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#unit-tests-dbt-18) (<https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/dbt-reference/#unit-tests-dbt-18>) below.

Generic tests (built-in)

```
# models/staging/_staging.yml
version: 2

models:
  - name: stg_orders
    description: "Cleaned orders from the raw source"
    columns:
      - name: order_id
        data_tests:
          - not_null
```

```

    - unique

    - name: status
      data_tests:
        - not_null
        - accepted_values:
            values: ['placed', 'shipped', 'delivered',
'cancelled', 'returned']

    - name: customer_id
      data_tests:
        - not_null
        - relationships:
            to: ref('stg_customers')
            field: customer_id

    - name: amount
      data_tests:
        - not_null

```

Singular tests — custom SQL

```

-- tests/assert_positive_revenue.sql
-- Fails if any order has a negative amount

SELECT order_id, amount
FROM   {{ ref('fct_orders') }}
WHERE  amount < 0

```

Unit tests (dbt 1.8+)

Data tests check the data a model produced; unit tests check the model's *logic* using small, hand-written inputs — no real data needed.

```

# models/marts/_marts.yml
unit_tests:
  - name: test_is_first_order_flag
    model: fct_orders
    given:
      - input: ref('stg_orders')
        rows:

```

```

        - {order_id: 1, customer_id: 10, created_at:
'2024-01-01'}
        - {order_id: 2, customer_id: 10, created_at:
'2024-02-01'}
    expect:
    rows:
        - {order_id: 1, is_first_order: true}
        - {order_id: 2, is_first_order: false}

```

```

dbt test --select "test_type:unit"      # run unit tests
only (typically in CI)

```

Running tests

```

dbt test                                # run all tests
dbt test --select stg_orders            # tests for one model
dbt test --select tag:daily             # tests for models with
'daily' tag
dbt test --select source:raw            # tests for source
definitions

```

dbt-expectations (extended tests via package)

```

- name: amount
  data_tests:
    - dbt_expectations.expect_column_values_to_be_between:
        min_value: 0
        max_value: 100000
    - dbt_expectations.expect_column_to_exist
- name: created_at
  data_tests:
    - dbt_expectations.expect_column_values_to_be_of_type:
        column_type: timestamp_ntz

```

Documentation

```
# models/marts/core/_mart_core.yml
version: 2

models:
  - name: fct_orders
    description: >
      One row per order. Central fact table for order
      analytics.
      Joins customers, products, and status mappings.
    columns:
      - name: order_id
        description: "Surrogate key – unique per order"
      - name: customer_id
        description: "FK to dim_customer"
      - name: amount
        description: "Order total in USD, inclusive of tax"
      - name: status
        description: "Current order status. See
        order_status_map seed for definitions"
```

```
# Generate and serve documentation
dbt docs generate    # builds the docs site from
model/test/source yml files
dbt docs serve       # opens in browser at localhost:8080

# Docs include:
# - Model descriptions and column descriptions
# - Auto-generated DAG / lineage graph
# - Source freshness
# - Test results
```

Jinja & Macros

dbt models are Jinja templates — you can use variables, conditionals, loops, and macros inside SQL files.

Built-in Jinja

```

-- Variables
{{ var('start_date') }}      -- from dbt_project.yml vars or
--vars CLI flag
{{ env_var('MY_SECRET') }}  -- from environment variable

-- Conditional logic
{% if target.name == 'dev' %}
    WHERE created_at >= DATEADD(day, -7, CURRENT_DATE()) -
- dev: last 7 days
{% else %}
    WHERE created_at >= '{{ var("start_date") }}' -
- prod: full history
{% endif %}

-- this – reference the current model's name
{{ this }}                  --
analytics_dev.staging.stg_orders
{{ this.schema }}          -- staging
{{ this.name }}            -- stg_orders

-- loop
{% set columns = ['col_a', 'col_b', 'col_c'] %}
SELECT
{% for col in columns %}
    {{ col }}{% if not loop.last %},{% endif %}
{% endfor %}
FROM my_table

```

Macros

Macros are reusable Jinja functions. Defined in `macros/*.sql`, called anywhere in the project.

```

-- macros/clean_string.sql
{% macro clean_string(col) %}
    LOWER(TRIM('{{ col }}'))
{% endmacro %}

-- macros/cents_to_dollars.sql
{% macro cents_to_dollars(col, precision=2) %}
    ROUND('{{ col }} / 100.0, {{ precision }})

```

```
{% endmacro %}

-- Use in a model
SELECT
    {{ clean_string('email') }}      AS email,
    {{ cents_to_dollars('amount') }} AS amount_usd
FROM {{ source('raw', 'orders') }}
```

```
-- macros/generate_schema_name.sql
-- Override the default schema naming (runs in every
project)
{% macro generate_schema_name(custom_schema_name, node) -%}
    {%- set default_schema = target.schema -%}
    {%- if custom_schema_name is none -%}
        {{ default_schema }}
    {%- else -%}
        {{ custom_schema_name | trim }}
    {%- endif -%}
{% endmacro %}
```

dbt CLI Commands

```
# — Core workflow —————
dbt debug          # test warehouse connection
dbt compile        # compile SQL without running (check
for errors)
dbt run            # run all models
dbt test           # run all tests
dbt build          # run + test + seed + snapshot in
one command (recommended)

# — Targeting specific models —————
dbt run --select stg_orders          # one model
dbt run --select staging.*           # all models in
staging/ folder
dbt run --select tag:daily            # models with 'daily'
tag
dbt run --select +fct_orders          # fct_orders + all
its ancestors
```

```

dbt run --select fct_orders+          # fct_orders + all
its descendants
dbt run --select +fct_orders+         # full upstream and
downstream graph

# — Node selection operators —————
# model_name          — exact model
# staging.*           — all in folder
# tag:daily           — by tag
# source:raw          — all models sourcing from 'raw'
# +model              — model + parents
# model+              — model + children
# @model              — model + all ancestors + all
descendants of ancestors

# — Other commands —————
dbt seed                # load CSV seeds
dbt snapshot            # run snapshots
dbt source freshness    # check source table
freshness
dbt docs generate && dbt docs serve
dbt run --vars '{"start_date": "2024-01-01"}' # pass
variables at runtime
dbt run --target prod    # use prod profile
dbt run --full-refresh   # rebuild incremental
models from scratch
dbt ls                   # list all models, tests,
sources
dbt ls --select staging.* # list models in a folder

```

Incremental Models

Incremental models only process new or changed rows on each run, rather than rebuilding the full table.

```

-- models/marts/core/fct_orders.sql
{{ config(
    materialized = 'incremental',
    unique_key   = 'order_id',      -- used for upsert
    (merge)

```

```

        incremental_strategy = 'merge'      -- merge | append |
delete+insert
    ) }}

SELECT
    order_id,
    customer_id,
    amount,
    status,
    created_at,
    updated_at
FROM {{ ref('stg_orders') }}

{% if is_incremental() %}
    -- Only runs on incremental runs (not --full-refresh)
    WHERE updated_at > (SELECT MAX(updated_at) FROM {{ this
}})
{% endif %}

```

Incremental strategies

Strategy	How it works	Use when
append	INSERT new rows only — no deduplication	Append-only data (events, logs)
merge	MERGE based on <code>unique_key</code> — upsert	Data can change (orders, customers)
delete+insert	DELETE matching keys, then INSERT	Warehouses without MERGE support
insert_overwrite	Overwrite entire partitions (Spark/BigQuery)	Partitioned tables

```
# Full rebuild – ignore the incremental filter
dbt run --select fct_orders --full-refresh
```

Always test incremental models with `--full-refresh` first to verify the full-load logic works before relying on the incremental path.

Snapshots

Snapshots implement **SCD Type 2** — track the full history of slowly changing dimension data.

```
-- snapshots/customers_snapshot.sql
{% snapshot customers_snapshot %}

{{ config(
    target_schema = 'snapshots',
    unique_key     = 'customer_id',
    strategy       = 'timestamp',      -- 'timestamp' or
    'check'
    updated_at     = 'updated_at',    -- for timestamp
    strategy
) }}

SELECT *
FROM {{ source('raw', 'customers') }}

{% endsnapshot %}
```

```
dbt snapshot      # run all snapshots
```

What dbt adds automatically: - `dbt_scd_id` — unique identifier for each snapshot row - `dbt_updated_at` — when this row was last changed - `dbt_valid_from` — when this version became active - `dbt_valid_to` — when this version was superseded (NULL = current row)

```
-- Query snapshot: current state
SELECT * FROM analytics.snapshots.customers_snapshot
WHERE dbt_valid_to IS NULL;

-- Historical state: what did this customer look like on
March 1?
SELECT * FROM analytics.snapshots.customers_snapshot
WHERE customer_id = 42
    AND dbt_valid_from <= '2024-03-01'
    AND (dbt_valid_to > '2024-03-01' OR dbt_valid_to IS
    NULL);
```

Snapshot strategies

```
-- timestamp strategy: check updated_at column for changes
{{ config(strategy='timestamp', updated_at='updated_at') }}

-- check strategy: compare specific columns for any change
{{ config(
    strategy = 'check',
    check_cols = ['email', 'address', 'plan_tier']
) }}

-- check all columns
{{ config(strategy='check', check_cols='all') }}
```

Packages

dbt packages add reusable macros, tests, and models. Defined in `packages.yml`.

```
# packages.yml
packages:
  - package: dbt-labs/dbt_utils
    version: [">=1.0.0", "<2.0.0"]      # ranges get patch
    fixes; exact pins go stale

  - package: metaplane/dbt_expectations  # moved from
    calogica/ (no longer maintained)
    version: [">=0.10.0", "<0.11.0"]

  - package: dbt-labs/codegen
    version: [">=0.12.0", "<1.0.0"]
```

```
dbt deps    # install packages (like npm install)
```

dbt_utils — most commonly used

```
-- Generate a surrogate key from multiple columns
{{ dbt_utils.generate_surrogate_key(['order_id',
'product_id']) }}
```

```

-- Date spine — generate a row for every date in a range
{{ dbt_utils.date_spine(
    datepart = "day",
    start_date = "cast('2024-01-01' as date)",
    end_date = "current_date()"
) }}

-- Star — select all columns except some
{{ dbt_utils.star(from=ref('stg_orders'), except=
["_loaded_at", "_file_name"]) }}

-- Get column values
{% set statuses =
dbt_utils.get_column_values(table=ref('stg_orders'),
column='status') %}
{% for status in statuses %}
    SUM(CASE WHEN status = '{{ status }}' THEN amount ELSE
0 END) AS revenue_{{ status }},
{% endfor %}

```

Hooks & Operations

Hooks — run SQL before or after a model

```

# In dbt_project.yml
models:
  my_project:
    marts:
      +post_hook:
        - "GRANT SELECT ON {{ this }} TO ROLE analyst"
        - "GRANT SELECT ON {{ this }} TO ROLE reporting"

```

```

-- In a model config block
{{ config(
    post_hook = [
        "GRANT SELECT ON {{ this }} TO ROLE analyst",
        "ALTER TABLE {{ this }} CLUSTER BY (order_date)"
    ]
) }}

```

```
]
) }}
```

Operations — run arbitrary SQL via CLI

```
# Run a macro as a one-off operation (doesn't create a
model)
dbt run-operation grant_all_schemas --args '{"role":
"analyst"}'
dbt run-operation clone_schema --args '{"source": "prod",
"target": "dev"}'
```

```
-- macros/grant_all_schemas.sql
{% macro grant_all_schemas(role) %}
    {% set schemas =
dbt_utils.get_schemas_in_database(database=target.database)
%}
    {% for schema in schemas %}
        GRANT USAGE ON SCHEMA {{ target.database }}.{{
schema }} TO ROLE {{ role }};
        GRANT SELECT ON ALL TABLES IN SCHEMA {{
target.database }}.{{ schema }} TO ROLE {{ role }};
    {% endfor %}
{% endmacro %}
```

Advanced Macros

Generic test as a macro

```
-- macros/test_not_negative.sql
{% test not_negative(model, column_name) %}
SELECT {{ column_name }}
FROM    {{ model }}
WHERE   {{ column_name }} < 0
{% endtest %}
```

```
# Use it like a built-in test
columns:
```

```
- name: amount
  data_tests:
    - not_negative
```

Conditional compilation

```
-- macros/limit_in_dev.sql
{% macro limit_in_dev(n=1000) %}
  {% if target.name != 'prod' %}
    LIMIT {{ n }}
  {% endif %}
{% endmacro %}

-- Use in a model
SELECT * FROM {{ ref('stg_large_table') }}
{{ limit_in_dev(500) }}
```

Exposures & Metrics

Exposures — document downstream consumers

```
# models/exposures.yml
exposures:
  - name: orders_dashboard
    type: dashboard
    owner:
      name: Analytics Team
      email: analytics@company.com
    depends_on:
      - ref('fct_orders')
      - ref('dim_customer')
    description: >
      The main orders dashboard in Looker.
      Shows daily revenue, order counts, and customer
      breakdown.
    url: https://looker.mycompany.com/dashboards/42
```

Best Practices

Naming conventions

```
stg_<source>__<table>      - staging models
(stg_postgres__orders)
int_<entity>_<verb>         - intermediate
(int_orders_joined)
fct_<event>                 - fact tables      (fct_orders)
dim_<entity>                - dimensions       (dim_customer)
```

Model layers

```
Raw (source) → Staging → Intermediate → Marts

Staging:      1-to-1 with source tables; clean + rename
              only
              Always a view; no joins; no business logic
Intermediate: Join and combine staging models
              Ephemeral or view; business logic lives here
Marts:        Aggregated, business-ready tables
              Always a table; named for business use cases
```

One model, one file

```
-- Bad: two concepts in one model
SELECT ... FROM orders
JOIN customers ...

-- Good: separate models
-- stg_orders.sql    - clean orders
-- stg_customers.sql - clean customers
-- fct_orders.sql    - join them
```

Testing every model

```
# Minimum: every model's primary key should be tested
- name: fct_orders
  columns:
```

```
- name: order_id
  data_tests:
    - not_null
    - unique
```

Slim CI — only test what changed

```
# In CI/CD: only run models that changed since last
production run
dbt run --select state:modified+    # run changed models
and their children
dbt test --select state:modified+    # test them

# Requires a manifest.json from the last production run
dbt run --defer --state ./prod_artifacts/ --select
state:modified+
```

Common Pitfalls

Pitfall	Symptom	Fix
Hardcoding table names instead of <code>ref()</code> / <code>source()</code>	Wrong build order; dev reads prod tables; lineage graph is incomplete	Always <code>ref()</code> models and <code>source()</code> raw tables
Incremental filter <code>> MAX(updated_at)</code> with late-arriving data	Rows that arrive late with older timestamps are never loaded	Add a lookback window (<code>>= MAX(updated_at) - INTERVAL '3 days'</code>) together with a <code>unique_key</code> merge
Incremental model without <code>unique_key</code>	Duplicates on every rerun or backfill	Set <code>unique_key</code> (or use <code>insert_overwrite</code> / <code>delete+insert</code> by partition)
Changing an incremental model's columns	New columns never appear, or the run fails	<code>on_schema_change='append_new_columns'</code> (or <code>sync_all_columns</code>), or <code>--full-refresh</code>
Business logic in staging models	The same logic duplicated across marts; hard to change	Staging = rename, cast, light cleaning; logic goes in intermediate/marts

Pitfall	Symptom	Fix
Joins in staging / one huge 500-line model	Slow, untestable, impossible to review	Split into staging → intermediate → mart models, one concept each
Primary keys never tested	Fan-out duplicates silently inflate metrics	<code>unique</code> + <code>not_null</code> on every model's grain
<code>dbt run</code> in CI without tests	Broken models reach production	<code>dbt build</code> (runs and tests in DAG order) with slim CI (<code>state:modified+ --defer</code>)
Everything materialized as <code>table</code>	Long runs and high warehouse cost	Views for staging, ephemeral/views for light intermediates, tables/incremental for marts
Jinja-heavy macros for simple SQL	Nobody can read the compiled SQL	Keep models as plain SQL; use macros for true repetition
Snapshotting from staging models or transformed data	History reflects your logic changes, not the source's	Snapshot raw sources directly; the snapshot is itself a source of truth
Sources without freshness checks	Stale data flows through with every test passing	<code>loaded_at_field</code> + <code>freshness</code> in <code>sources.yml</code> ; <code>run dbt source freshness</code>

Cheat Sheet

Task	Command
Run + test everything in DAG order	<code>dbt build</code>
One model and everything downstream	<code>dbt build -s fct_orders+</code>
Everything upstream of a model	<code>dbt build -s +fct_orders</code>
A folder / tag	<code>dbt build -s marts.core -s tag:daily</code>
Only changed models (slim CI)	<code>dbt build -s state:modified+ --defer --state prod-artifacts/</code>

Task	Command
Rebuild an incremental model from scratch	<code>dbt run -s fct_orders --full-refresh</code>
Data tests / unit tests only	<code>dbt test -s fct_orders</code> · <code>dbt test -s "test_type:unit"</code>
Source freshness	<code>dbt source freshness</code>
Snapshots	<code>dbt snapshot</code>
See compiled SQL	<code>dbt compile -s model → target/compiled/...</code> · <code>dbt show -s model</code> (preview rows)
Install packages	<code>dbt deps</code>
Docs site	<code>dbt docs generate</code> && <code>dbt docs serve</code>
Check connection	<code>dbt debug</code>
Retry only what failed	<code>dbt retry</code>
Pass variables	<code>dbt run --vars '{"start_date": "2024-01-01"}'</code>

Jinja you'll use constantly

```

{{ config(materialized='incremental',
unique_key='order_id',
on_schema_change='append_new_columns') }}

select * from {{ ref('stg_orders') }}
{% if is_incremental() %}
    where updated_at >= (select max(updated_at) - interval '3
days' from {{ this }})
{% endif %}

{{ source('stripe', 'charges') }}           -- raw table
{{ var('start_date', '2020-01-01') }}       -- variable
with default
{{ target.name }}                           -- dev / prod
{{ dbt_utils.generate_surrogate_key(['order_id',
'line_no']) }}

```

Materialization picker: staging → view · light intermediate → ephemeral / view · marts
→ table · big, append-heavy facts → incremental · SCD Type 2 history → snapshot

Selectors: +model (parents) · model+ (children) · @model (children and their parents) ·
tag:x · path:models/marts · state:modified · result:error · source:stripe+

Interview Questions

Q: What is the difference between ref() and source() in dbt? A: `ref('model_name')` references another dbt model — it resolves to the correct schema/table for the current environment (dev vs prod) and builds the dependency graph. `source('source_name', 'table_name')` references a raw table that dbt doesn't own — it's defined in a `sources.yml` file and enables source freshness testing. Use `ref()` for dbt-managed models, `source()` for raw/ingested tables.

Q: What are the four materializations in dbt and when do you use each? A: (1) `view` — creates a SQL view, always fresh, no storage cost — use for staging models; (2) `table` — creates a physical table, fast to query — use when many models read it or the underlying query is expensive; (3) `incremental` — appends or merges only new/changed rows — use for large tables where reprocessing everything is too slow; (4) `ephemeral` — a CTE that gets inlined into the calling model, not materialized at all — use for reusable logic that's only needed once.

Q: How does dbt handle incremental models and what is the unique_key? A: With `materialized='incremental'`, dbt only processes rows where the filter condition is true (usually `where updated_at > max(updated_at)`). On first run it builds the full table; on subsequent runs it adds only new data. The `unique_key` tells dbt to MERGE instead of INSERT — if a row with that key already exists, it's updated rather than duplicated. This makes the model idempotent.

Q: What is the difference between a generic test and a singular test in dbt? A: A generic test is a reusable test applied in `schema.yml` using YAML — like `not_null`, `unique`, `accepted_values`. It can be applied to many columns across many models. A singular test is a custom SQL file in `tests/` that returns rows on failure — use for complex multi-table or business-logic checks that don't fit a generic pattern.

Q: What is slim CI and why is it important for large dbt projects? A: In a large dbt project with 500+ models, running `dbt build` on every PR takes 2+ hours. Slim CI uses `state:modified+` to run only the models that changed in this PR plus their downstream dependents. It compares the current code against a `manifest.json` from the last production run. This cuts CI time from hours to minutes and enables fast PR feedback loops.

Q: How does dbt handle environments (dev vs prod)? A: dbt uses `profiles.yml` to define target environments. In dev, models are built in a personal schema (`dbt_alice`). In prod, models build in the configured production schema. The `ref()` macro always resolves to the current environment's schema — you never hardcode schema names. This means the same SQL runs correctly in both environments without changes.

Further Reading

- [dbt documentation](https://docs.getdbt.com/docs/introduction) (https://docs.getdbt.com/docs/introduction)
 - [How we structure our dbt projects](https://docs.getdbt.com/best-practices/how-we-structure/1-guide-overview) (https://docs.getdbt.com/best-practices/how-we-structure/1-guide-overview) — the standard staging/intermediate/marts layout
 - [Node selection syntax](https://docs.getdbt.com/reference/node-selection/syntax) (https://docs.getdbt.com/reference/node-selection/syntax)
 - [Incremental models in depth](https://docs.getdbt.com/best-practices/materializations/4-incremental-models) (https://docs.getdbt.com/best-practices/materializations/4-incremental-models)
 - [dbt_utils](https://hub.getdbt.com/dbt-labs/dbt_utils/latest/) (https://hub.getdbt.com/dbt-labs/dbt_utils/latest/) and [dbt_expectations](https://hub.getdbt.com/metaplane/dbt_expectations/latest/) (https://hub.getdbt.com/metaplane/dbt_expectations/latest/) packages
 - [SQLFluff](https://docs.sqlfluff.com/) (https://docs.sqlfluff.com/) — SQL linter with a dbt templater
 - [dbt Learn](https://learn.getdbt.com/) (https://learn.getdbt.com/) — free official courses
-

Previous: [Apache Hudi](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-hudi/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-hudi/) • **Next:** [Semantic Layer & Metrics](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/semantic-layer-metrics/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/semantic-layer-metrics/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

🕒 September 29, 2026