dbt Labs

dbt Analytics Engineering Certification Exam Study Guide

Your comprehensive guide to preparing for the dbt Analytics Engineering Certification Exam updated for dbt Core 1.11

How to use this study guide

This is the official study guide for the **dbt Analytics Engineering Certification Exam** from the team at dbt Labs. While the guide suggests a sequence of courses and reading material, we recommend using it to supplement (rather than substitute) real-world use and experience with dbt.

The exam now supports dbt Core 1.11

What's inside

The exam overview provides high-level details on the format of the exam. Be mindful of the number of questions and time constraints.

The **topics outline** provides a clear list of the topics assessed on the exam. dbt subject matter experts used this outline to write and meticulously review all exam items.

The **sample questions** offer an overview of what to expect, though they do not represent all question formats.

The **learning path** walks you through a suggested series of courses, readings, and documentation, along with guidance for building real-world dbt experience.

Question formats you'll encounter

- Multiple-choice
- Fill-in-the-blank
- Matching
- Hotspot
- Build list
- Discrete Option Multiple Choice (DOMC)

Exam Overview

The **dbt Analytics Engineering Certification Exam** evaluates your ability to build, test, and maintain models to make data accessible to others, and to use dbt to apply engineering principles to analytics infrastructure.

We recommend at least SQL proficiency and 6+ months of dbt experience before attempting the exam.

Logistics

Duration	2 hours
Format	Online proctored
Length	65 questions
Version	dbt core 1.11
Passing Score	65% or higher. You will know your score immediately after completion of the exam.
Price	$200*
Language	English and Japanese
Expiration	2 years after date awarded
Supported Browsers	Caveon Web browsers

Scoring

The exam is scored on a point basis, with 1 point for each correct answer, and 0 points for incorrect answers. All questions are weighted equally.

An undisclosed number of unscored questions will be included on each exam. These are unmarked and indistinguishable from scored questions. Answers to these questions will be used for research purposes only, and will not count towards the score.

Retakes & Cancellations

If you do not pass the exam, you may schedule a retake. You will need to pay a registration fee for each retake. You can reschedule or cancel without penalty on Talview up to 24 hours before a scheduled exam. We will not issue refunds for no-shows.

Accommodations

For accommodation requests, please contact us via email at *accessibility@dbtlabs.com*

Topics Outline

The **dbt Analytics Engineering Certification Exam** has been designed to assess the following topics and sub-topics.

01 Developing and optimizing dbt models

- Identifying and verifying any raw object dependencies
- Understanding core dbt materializations
- Conceptualizing modularity and how to incorporate DRY principles
- Using commands such as build, run, test, docs, show, snapshot, and seed
- Creating a logical flow of models and building clean DAGs
- Defining configurations in dbt_project.yml
- Using dbt Packages
- Creating Python Models
- Providing access to users to models with the "grants" config
- Creating snapshots in YAML
- Selecting the optimal incremental strategy based on a dataset's characteristics
- Validating model logic and schema definitions in dry-runs using the --empty flag
- Running models in sample mode using the --sample flag
- Understanding advanced dbt materializations such as microbatch

02 Managing dbt models governance

- Adding contracts to models to ensure the shape of models
- Creating different versions of our models and deprecating the old ones
- Defining constraints in YAML to enforce data integrity at the platform level

03 Debugging data modeling errors

- Understanding logged error messages
- Troubleshooting using compiled code
- Troubleshooting .yml compilation errors
- Developing and implementing a fix and testing it prior to merging
- Managing dbt behavior with flags

04 Troubleshooting and optimizing dbt pipelines

- Troubleshooting and managing failure points in the DAG
- Using dbt clone

05 Implementing dbt tests

- Using generic, singular, custom, custom generic, and unit tests on a wide variety of models and sources
- Testing assumptions for dbt models and sources
- Implementing various testing steps in the workflow

06 Implementing and maintaining external dependencies

- Implementing dbt exposures
- Implementing source freshness

07 Leveraging the dbt state

- Understanding state and state selection
- Using dbt retry

Sample Exam Questions

Question 01

One of your models is incremental and contains a lot of data.
You want to test that some columns are not null but do not want to test all the data every time.

Which option is used to configure dbt to not test the entire table every time?

A. Use the test parameter `limit` to add a limit to the number of rows tested.

B. Use the test parameter `store_failures` to reuse previous test results.

C. Use the test parameter `error_if` to stop the test as soon as we see that a minimum number of rows failed.

D. Use the test parameter `where` to add a condition on the data tested.

E. Use the test parameter `where` to add a condition on the data tested.

Explanation: `dbt test` accepts many configuration parameters but not all of them would avoid testing an incremendbt test accepts many configuration parameters but not all of them would avoid testing an incremental model entirely after every run.

The correct answer here is that we could use:

- a `where` parameter on the test to potentially restrict data for a specific set of dates.

Question 02

Which statement regarding incremental materializations in dbt is true?

A. Incremental models are ideal for datasets with millions of rows where rows are only added and never updated.

B. Incremental models are always rebuilt upon execution.

C. Incremental models must be dropped and rebuilt after schema changes in the upstream model.

D. Incremental models never need to leverage the `is_incremental()` macro.

E. Incremental models are an appropriate choice for datasets where a large percentage of the dataset is updated each run.

Explanation: If a table consists of a large number of rows and only new data is being added at each run, setting the model as incremental allows dbt to run the transformation only on the new data, saving a lot of time and compute on the data warehouse. Therefore, the answer is that incremental models are ideal for datasets with millions of rows, where data is added but previous rows are not updated.

Question 03

You have been working on your feature branch while your fellow analytics engineers have been merging changes to the head branch. Which statement regarding keeping your separate branches synced is true?

A. You should never merge changes directly to the head branch.

B. Use `git fetch` to bring all remote branches to local.

C. Use `git pull` to reconcile with the head branch.

D. Merge conflicts must be resolved via the dbt Cloud IDE.

Explanation: The correct answer is to use `git pull`. This will actually run both – `git fetch`, downloading the code that has been pushed to the repository, and `git merge`, which will combine the code just fetched with the code in the current branch.

Question 04

Consider these models files materialized as tables:

stg_accounts.sql

```
SQL
select
...
from salesforce.accounts
```

stg_opportunities.sql

```
SQL
select
...
from salesforce.opportunities
```

accounts.sql

```
SQL
select
...
from dbt_user.stg_accounts
left join dbt_user.stg_opportunities on account_id
```

When running `dbt run` the models build in the incorrect order. What should you implement to ensure that the models build in the intended order?

A. A `ref` macro in both the from clauses of `stg_accounts.sql` and `stg_opportunities.sql`.

B. Update the from clauses of `stg_opportunities` and `stg_accounts` to use the source macro.

C. A ref macro in both the from and join clauses of accounts.sql

D. Change to ephemeral materialization strategy for `stg_opportunities` and `stg_accounts`.

Explanation: dbt recognizes dependencies between models when the `ref` macro is used. To ensure that `accounts.sql` model runs after the two other models, every hard coded table in `accounts.sql` must be replaced with the relevant `ref` statement. Therefore, the answer is that the `ref` macro needs to be set for both the from and join clauses of `accounts.sql`.

Question 05

You want to convert this legacy SQL to dbt models:

```
SQL
select
        customer_id,
        customer_name,
        cagg.*
from sample_data.public.customer c
left join (
        select
                customer_id,
                count(order_id) as total_orders,
                min(order_date) as first_order,
                max(order_date) as most_recent_order

        from sample_data.public.orders
        group by customer_id
) cagg
  on c.customer_id = cagg.customer_id
where customer_id in (

        select customer_id
        from sample_data.public.customer c
        left join sample_data.public.nation n
                on c.nation_id=n.nation_id
        left join sample_data.public.region r
                on n.region_id = r. region_id
        where r.region_name = 'EUROPE'
)
```

How many sources and tables will be defined in sources.yml?

A. 1 source with 5 tables

B. 1 source with 4 tables

C. 4 sources with 5 tables

D. 4 sources with 4 tables

Explanation: Every source in dbt maps to a database + schema combination. Because the database `sample_data` and schema `public` are used on all references, this is one source. Tables are configured within a source's tables configuration. There are four tables referenced: `customers`, `orders`, `nations`, and `regions`.

Question 06

Which command materializes my_model and only its first-degree parent model(s)?

A. `dbt run --select 1+my_model`

B. `dbt run --select +1my_model`

C. `dbt compile --select 1+my_model`

D. `dbt run --select +1 my_model`

Explanation: The `dbt run` command executes your model against your data warehouse, meaning it materializes the database object. By comparison, `dbt compile` will simply generate the sql, which makes those options not correct. `1+my_model` is the correct format for selecting first degree parents, as described in node selection syntax.

Question 07

You run `dbt build --empty` and a unique test passes despite source data having duplicates. Why?

A. Test ran against zero rows, so no duplicates found.

B. dbt skips all tests with `--empty` flag.

C. Test evaluates schema config rather than scanning data.

D. `--empty` restricts tests to only validate column data types.

Explanation: The `--empty` flag limits the refs and sources to zero rows, so you have essentially built empty tables. When you run a test on those tables, for example, a `unique` test, it looks for duplicates and does not find any, so the test passes. Option 2 is incorrect because dbt does not skip tests when running `dbt build --empty` . Option C and D are incorrect because `--empty` does not change how a test works fundamentally.

Question 08

You have added a `check` constraint to a model and enabled the contract. However, `dbt run` fails with a parsing error indicating an invalid configuration for the `created_at` column.

Which action resolves this error while adding the constraint?

A. Defining the `data_type` for the `created_at` column in the YAML.

B. Removing the `check` constraint from the `created_at` column.

C. Changing the materialization from table to view.

D. Adding a unique test to the `created_at` column.

Explanation: Within constraints, `check` is a flexible row-level constraint that evaluates to a boolean expression. While we do not know the current state of the YAML file, ensuring the YAML configuration is correct is the best course of action.

Option B is incorrect because removing the constraint does not solve the task.

Option C is incorrect because changing the materialization would not change the behavior of the constraint.

Option D is incorrect because adding a unique test wouldn't solve this problem, as the test would run after the parsing error, which we need to fix first.

Question 09

During a `dbt build`, you observe this sequence in the terminal log:

```
1 of 4 START test not_null_stg_orders_order_id ..... [RUN]
1 of 4 FAIL 23 not_null_stg_orders_order_id ..... [FAIL 23]
2 of 4 START sql table model marts.fct_orders ..... [RUN]
2 of 4 SKIP relation marts.fct_orders ..... [SKIP]
```

Why did the `fct_orders` table get skipped?

A. The `fct_orders` model was skipped because of a warehouse timeout; setting `job_execution_timeout_seconds` in `profiles.yml` would fix this

B. The `not_null` test failed, and dbt skipped the downstream `fct_orders` model by default because `dbt build` enforces test-before-model sequencing; adding `--fail-fast` to the run command would allow all models to execute

C. The `fct_orders` model was skipped because its test failure threshold was exceeded; setting `warn_if: ">0"` and `error_if: ">100"` in the test configuration would change the failure to a warning and allow downstream models to proceed

D. The `fct_orders` model was skipped because `dbt build` halts all execution after any test failure; the only way to proceed is to run `dbt run` and `dbt test` separately

Explanation: In `dbt build`, tests run before their downstream models, and a `FAIL` status causes dependent nodes to be skipped. However, using `warn_if` and `error_if` thresholds in the test config lets you control when a test escalates to a true failure vs. a warning. A warning does not block downstream execution, whereas a hard failure does. In option A, an execution timeout could cause an model to `FAIL`, but this model was skipped, so this is not the right issue. `--fail-fast` (option B) stops the entire run early – it does not unblock nodes that are skipped due to a failed upstream test. Option D is incorrect because all execution is not stopped after any test failure (that would be the behavior of `--fail-fast`).

Question 10

A dbt project references a raw source table using `{{ source('ecommerce', 'raw_orders') }}`.

After running dbt run, a senior analyst reports that the model is querying stale data and suspects the source table may have been moved to a different schema by the data engineering team.

Which combination of steps would most reliably identify and verify the raw object dependency?

A. Run `dbt compile` to render the SQL, then manually query the warehouse to confirm the table exists in the expected schema

B. Run `dbt source freshness` to check whether the source is up to date. Inspect `sources.yml` to confirm the database, schema, and table name properties match the actual warehouse object. Use schema to override the schema location if it differs from the source name.

C. Run `dbt test --select source:ecommerce` to validate the source, then update the `ref()` function in the model to point directly to the raw table

D. Check the `target/manifest.json` for the source node definition, then run `dbt run --full-refresh` to force dbt to re-resolve all dependencies from scratch

Explanation: dbt source freshness confirms whether data is arriving in the source table as expected, but the more important verification step is checking `sources.yml` — specifically the database, schema, and table name fields. The schema property is what dbt actually uses to resolve the physical schema name in the warehouse, and it defaults to the source name if not explicitly set. If the data engineering team moved the table to a new schema or renamed it, updating schema and schema in `sources.yml` is the correct fix.
Option A is incorrect because we already know the table exists as the run did run and return stale data.
Option C is incorrect because `ref()` is for models, not raw sources — using it to point to a raw table would break dbt's DAG lineage entirely.
Option D is incorrect because `--full-refresh` is for incremental models and the real problem lies in source configurations.

Learning Path

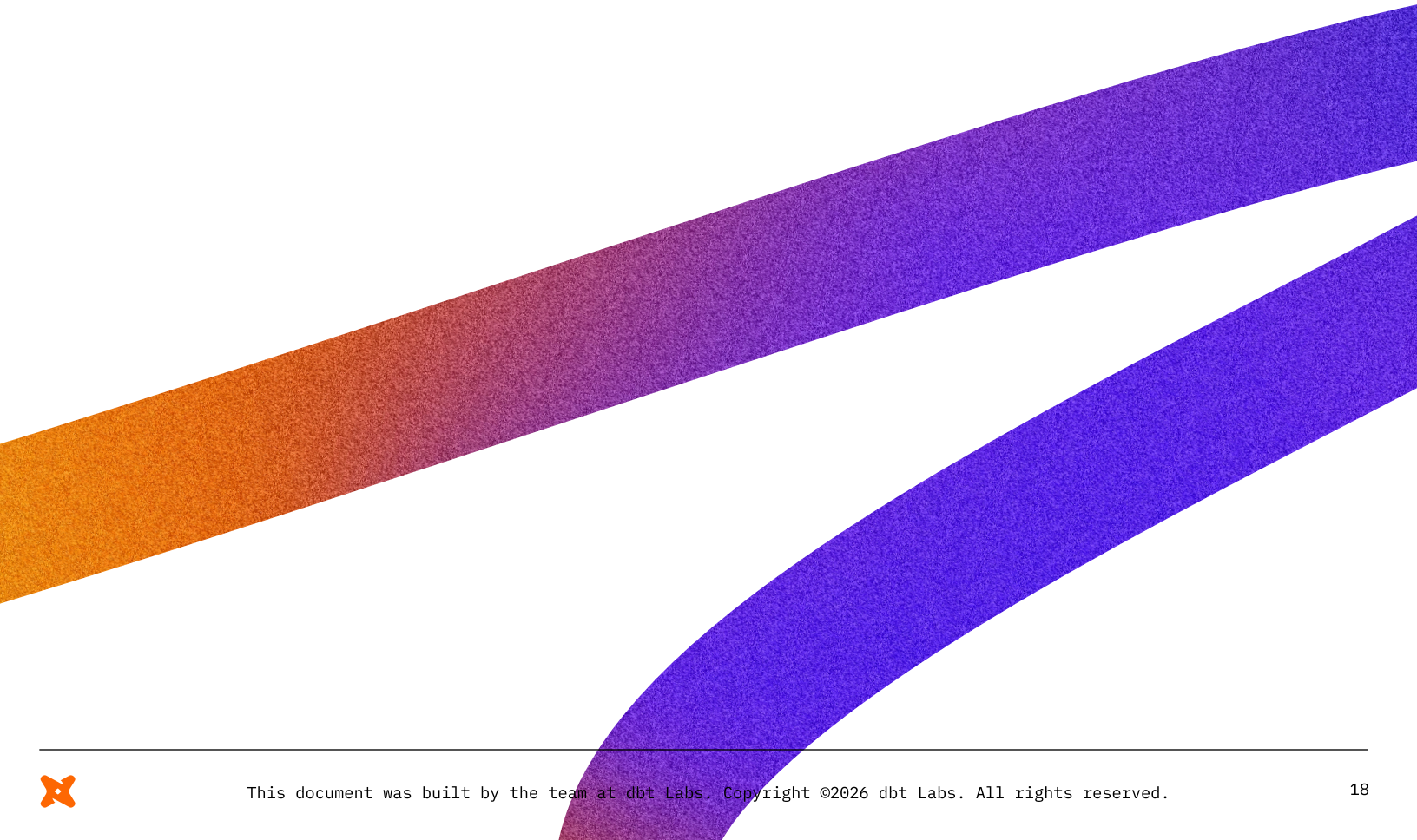
One recommended path for someone new to dbt. Each checkpoint provides courses, readings, documentation, and real-world experience. Reorganize based on your preferences.

Checkpoint 0

Prerequisites

dbt brings together several technical skills. Start after developing foundational git and SQL skills.

- SQL: Familiarity with joins, aggregations, CTEs, and window functions.
- Git: Familiarity with branching strategies, basic commands, and pull requests.



Learning Path

Checkpoint 1

Build a Foundation

Courses

- dbt Fundamentals (dbt Studio)
- Materialization Fundamentals
- Refactoring SQL for Modularity
- Analyses and Seeds
- Snapshots
- Incremental Models
- Jinja, Macros, and Packages

Readings

- dbt viewpoint
- How we structure our dbt projects
- Refactoring legacy SQL to dbt

Documentation

- Add sources to your DAG
- Source configurations
- Materializations
- Materialization best practices
- Best practice workflows
- About dbt build command
- About dbt run command
- About dbt test command
- About dbt docs command
- About dbt show command
- About dbt snapshot command
- About dbt seed command

Documentation (continued)

- dbt_project.yml
- dbt Packages
- Python models
- grants
- Add snapshots to your DAG
- About incremental models
- About incremental strategy
- About microbatch incremental models
- About the --empty flag
- About the --sample flag

Experience

- Creating a dbt pipeline from scratch
- Refactoring SQL for performance
- Implementing all core materializations
- Utilizing packages and macros

Commands

`dbt build` `dbt run` `dbt test`

`dbt show` `dbt snapshot`

`dbt seed` `dbt docs generate`

Checkpoint 2

Govern and Debug Your Models

Courses

- dbt Mesh — Model Governance module

Experience

- Being familiar with model access, contracts and versions
- Understanding data product (producer and consumer) management best practices
- Identifying, understanding and debugging errors in dbt logs

Documentation

- Model contracts
- Model versions
- Constraints
- Debugging errors
- dbt compile command
- Behavior changes
- About flags (global configs)

Readings

- Data product management: best practices

Checkpoint 3

Build Resilient Pipelines at Scale

Courses

- Advanced Testing
- Unit Testing
- Exposures

Readings

- How we structure our dbt projects
- Test smarter not harder
- Test smarter not harder: Where should tests go in your pipeline?
- Your Essential dbt Project Checklist
- To defer or to clone, that is the question

Documentation

- About dbt clone command
- Clone incremental models
- Add data tests to your DAG
- Testing and documenting sources
- Writing custom generic data tests
- Data test configurations
- Unit tests
- About dbt test command
- Add Exposures to your DAG
- Exposure properties
- dbt source freshness
- About state in dbt
- state method
- dbt retry

Commands

`dbt test` `dbt source freshness` `dbt clone` `dbt retry` `dbt state`

Additional Resources

dbt Slack Community

Join the conversation and connect with fellow analytics engineers preparing for the certification.

#dbt-certification #learn-on-demand #advice-dbt-help

#advice-dbt-for-power-users

Partner & Enterprise Benefits

If you are a dbt Labs partner or enterprise client, contact your partner manager or account team for additional benefits.

Get in Touch

Have questions about the certification?

certification@dbtlabs.com

Ready to get certified?

Register for the exam