

KillTest

高品質 より良いサービス



練習問題

<https://www.killtest.jp>

1年間無料アップデート

Exam : Dbt Analytics Engineering

**Title : Dbt Analytics Engineering
Certification Exam**

Version : DEMO

1. Which two are true about version controlling code with Git? Choose 2 options.

- A. Git automatically creates versions of files with suffixes.
- B. All the code changes along the lifecycle of a project are tracked.
- C. When bugs are raised, email notifications are automatically sent by Git to repository users.
- D. Git prevents any sensitive fields from being saved in code.
- E. Code can be reverted to a previous state.

Answer: B, E

Explanation:

The correct answers are B: All the code changes along the lifecycle of a project are tracked, and E: Code can be reverted to a previous state.

Git is a distributed version control system designed to maintain a complete, chronological history of all code changes. Every commit records who made the change, when it occurred, and what the modification included. This ensures transparency, reproducibility, and accountability across the development lifecycle, which makes B correct. Git also allows users to revert code to any previous commit, branch, or tag, making E correct as well. This capability is critical for recovering from mistakes, undoing faulty deployments, and ensuring stable releases.

Option A is incorrect because Git does not create file versions with suffixes; instead, it stores changes as snapshots within a repository. File suffixing is not part of Git's functionality.

Option C is incorrect because Git does not automatically send email notifications. Notification mechanisms come from hosting platforms like GitHub, GitLab, or Bitbucket—not from Git itself.

Option D is incorrect because Git does not prevent committing sensitive information. Developers must manually ensure secrets are excluded via .gitignore, secret managers, or pre-commit hooks. Git will store whatever is committed unless prevented through tooling. Thus, only B and E correctly describe how Git supports version control in analytics engineering and dbt workflows.

2. Consider this DAG:

```
app_data.detail_categories -> stg_detail_categories -> skills_with_details app_data.details -> stg_details  
-> lessons_with_details
```

What will support making this DAG more modular? Choose 1 option.

- A. Union stg_detail_categories and stg_details in the staging layer to reduce the need for downstream joins.
- B. Join stg_detail_categories and stg_details in the staging layer to reduce the need for downstream joins.
- C. Consolidate the two staging models into one model and then use this downstream for both skills_with_details and curriculum_with_details.
- D. Combine lessons_with_details and skills_with_details into one wide table called curriculum_with_details.
- E. Examine the SQL of lessons_with_details and skills_with_details to see if there is a candidate for an intermediate model.

Answer: E

Explanation:

Modularity in dbt means breaking transformations into small, reusable steps that are easy to reason about, test, and reuse. In this DAG, two staging models (stg_detail_categories and stg_details) feed into two mart-level models (skills_with_details and lessons_with_details). If there is repeated logic in those

two mart models—such as common joins, filters, or derived fields—that logic is a good candidate to be extracted into an intermediate model.

Option E follows dbt best practices: inspect the SQL of `lessons_with_details` and `skills_with_details` and identify shared logic to move into a new intermediate model (for example, `int_details_with_categories`). Both downstream models can then reference this intermediate model using `ref()`, centralizing logic and improving maintainability.

Options A and B push joins/unions into the staging layer, which should focus on clean, 1:1 representations of sources—not combining multiple sources.

Option C collapses distinct staging models into one, reducing clarity and breaking the source-to-staging contract.

Option D creates an overly wide, less reusable final table instead of focusing on shared reusable logic. Therefore, the action that truly increases modularity is Option E: creating an intermediate model based on common SQL patterns.

3. Given this `dbt_project.yml`:

```
name: "jaffle_shop"
version: "1.0.0"
config-version: 2
profile: "snowflake"
model-paths: ["models"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]
target-path: "target"
clean-targets:
  - "logs"
  - "target"
  - "dbt_modules"
  - "dbt_packages"
models:
  jaffle_shop:
    orders:
      materialized: table
```

When executing a dbt run your models build as views instead of tables:

```
19:36:14 Found 1 model, 0 tests, 0 snapshots, 0 analyses, 179 macros, 0 operations, 0 seed files, 0 sources, 0 exposures, 0 metrics
```

```
19:36:16 Concurrency: 1 threads (target='default')
```

```
19:36:17 Finished running 1 view model in 3.35s.
```

```
19:36:17 Completed successfully
```

```
19:36:17 Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

Which could be a root cause of why the model was not materialized as a table?

The `target-path` is incorrectly configured.

A. Yes

B. No

Answer: A

Explanation:

The behavior described—dbt running the orders model as a view despite being explicitly configured as a table—indicates that dbt is not correctly detecting or applying the model-level configuration during compilation. dbt relies heavily on the target-path directory to write compiled SQL, manifest files, and run artifacts. If the target-path is misconfigured, pointing to a location that dbt does not handle correctly or that overlaps with another folder used internally, dbt may fail to load the correct configuration from the merged project settings. When dbt cannot locate the compiled configuration for a model, it defaults to its standard materialization type, which is view. This explains why the logs show: “Finished running 1 view model”

even though the dbt_project.yml clearly declares:

materialized: table.

Additionally, the logs indicate no warnings or parsing errors, meaning dbt ran successfully but with incorrect settings—another indicator of configuration metadata being overridden or misplaced due to an incorrect target-path.

By resolving the target-path issue, dbt will successfully load the model configuration and materialize the orders model as a table as intended.

4.A developer imports a package from a private repository called timeformat for use within their project. Which statement is correct? Choose 1 option.

A. “The package can be added with this configuration in the packages.yml file:” packages: - local: /opt/dbt/timeformat

B. “The package can be installed by running the command dbt build.”

C. “The package default schema can be overridden in the dbt_project.yml file as:” models: timeformat: +schema: timeseries

D. “Including the package version/revision in the packages.yml file, for private git packages will result in an error.”

Answer: C

Explanation:

The only correct statement is Option C, because dbt allows you to override the schema for a package’s models directly within your project’s dbt_project.yml file.

Packages behave just like subfolders within your own project, and therefore the standard configuration pattern applies:

models:

timeformat:

+schema: timeseries

This correctly overrides the package’s default schema, allowing all models within the timeformat package to be built into the timeseries schema.

Why the other options are incorrect

Option A

Local packages can be added using a path reference, but that is not what a private Git repository refers to. A private repo must be added using git: plus SSH/HTTPS URL, not local: unless the developer literally cloned the repo into a local directory. The question specifically describes a private repository, so this is not the correct statement.

Option B

Packages are installed using dbt deps, not dbt build.

dbt build only runs models, tests, and snapshots; it does not install dependencies.

Option D

Including a version or revision is allowed for private Git-based packages. You can specify branches, tags, or commit hashes safely in packages.yml.

5. You wrote this test against your fct_orders model to confirm status filters were properly applied by a parent model:

```
{{
config(
  enabled=true, severity='error'
)
}}
select *
from {{ ref('fct_orders') }}
where status_code = 13
```

Which statement about this test is true?

- A. You must attach the test to the model in orders.yml for it to run.
- B. This test will warn instead of erroring when you use the --warn-error flag.
- C. The file must be saved in the tests/generic directory.
- D. Records with status_code = 13 will cause this test to fail.

Answer: D

Explanation:

This test is a singular test, meaning it is defined as a standalone SQL file placed inside the /tests directory (not attached through YAML like generic tests). dbt executes singular tests by running the SQL query, and the test fails if the query returns any rows. In this case, the test selects all records from fct_orders where status_code = 13. That means if even one row in fct_orders has a status code of 13, the test will fail, because dbt interprets returned rows as evidence of a data-quality issue.

The configuration block at the top sets severity='error', meaning the test raises an error when it fails, rather than a warning. This setting overrides default severity behavior, so option B is incorrect. Singular tests do not need to be attached in YAML because they run automatically when placed under /tests making option A incorrect. Meanwhile, generic tests live in /tests/generic, but singular tests should live directly under /tests (option C incorrect).

Therefore, the only correct statement is that records with status_code = 13 will cause the test to fail, because the test query is explicitly written to identify such rows.