DUMPSBOSS.

dbt Analytics Engineering Certification Exam

dbt Labs dbt-Analytics-Engineering

Version Demo

Total Demo Questions: 10

Total Premium Questions: 100

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Analytics Engineering Workflow	20
Topic 2, Data Modeling	4
Topic 3, Building dbt Models	39
Topic 4, Testing and Documentation	25
Topic 5, Deployment	10
Topic 6, Mix Questions	2
Total	100

QUESTION NO: 1

Your model has a contract on it.

When renaming a field, you get this error:

This model has an enforced contract that failed.

Please ensure the name, data_type, and number of columns in your contract match

the columns in your model's definition.

| column_name | definition_type | contract_type | mismatch_reason |

|---|---|---|---|

| ORDER_ID | TEXT | TEXT | missing in definition |

| ORDER_KEY | TEXT | | missing in contract |

Which two will fix the error? Choose 2 options.

A. Remove order_id from the contract.

B. Remove order_key from the contract.

C. Remove order_id from the model SQL.

D. Add order_key to the contract.

E. Add order_key to the model SQL.

ANSWER: A D

Explanation:

Remove order_id from the contract and add order_key to the contract. An enforced dbt model contract compares the model's declared columns with the columns produced by its compiled SQL. The contract requires the column names, data types, and column set to match the model definition. Here, `ORDER_ID` is still declared in the contract but is reported as "missing in definition," which means that the model query no longer returns that name. Removing `order_id` from the YAML column declarations eliminates that obsolete contract requirement. At the same time, `ORDER_KEY` is returned by the model SQL but is reported as "missing in contract." Adding `order_key` to the model's contract declarations brings the contract's expected schema into alignment with the renamed output field. The declared data type should remain compatible with the produced type, shown as `TEXT` in the error output. Together, these changes update the contract to reflect the intended rename without changing the model's current SQL output. dbt documents that enforced contracts validate a model's name, data type, and constraints against its compiled query, and that columns must be declared for the contract to match successfully. See the dbt model contracts documentation and contract resource-property reference.

QUESTION NO: 2

Which command materializes my_model and **only its first-degree parent model(s)** in the data platform?

Choose 1 option.

A. dbt run --select +my_model

B. dbt compile --select +my_model

C. dbt run --select 1+my_model

D. dbt run --select +1 my_model

E. dbt compile --select +1 my_model

F. dbt run --select +1my_model

ANSWER: F

Explanation:

`dbt run --select +1my_model` is the command that materializes `my_model` together with only its direct upstream dependencies. In dbt's graph-selection syntax, a leading `+` selects parents (upstream nodes), and placing a number immediately after that operator limits the traversal depth. Therefore, `+1my_model` selects `my_model` and parents exactly one edge away in the dbt DAG. This is important when a model depends on intermediate models that must be built first, while more distant ancestors are intentionally excluded from the command's selection.

The numeric graph operator and the model selector form one selector expression, so there must not be a space between `+1` and `my_model`. Finally, `dbt run` executes the selected model SQL against the configured data platform and creates or updates their materialized relations. dbt documents graph operators, including depth-limited parent selection, in its graph operators reference. The behavior of `dbt run` is described in the dbt run command reference.

QUESTION NO: 3 - (SIMULATION)

Choose a correct command for each statement.

ANSWER: See the explanation for the answer

Explanation:

Correct matches:

Will always point to the latest version of the source schema → `defer`
Allows use of objects built in the target schema with any downstream tool → `dbt clone`
Allows safely modifying objects built in the target schema → `dbt clone`
Is a point-in-time operation → `dbt clone`

`--defer` resolves unbuilt upstream nodes to relations in the deferred/state environment. It does not create new relations, so queries use the current relation at that deferred location.

`dbt clone` creates relations in the current target schema based on an existing environment. Those cloned relations can be queried by downstream tools and changed without changing the original relations. A clone represents the source relation at the time the clone operation runs.

QUESTION NO: 4 - (SIMULATION)

Match the desired outcome to the dbt command or argument.

Match the desired outcome to the dbt command or argument.

Execute the last dbt command from the node point of failure.

Select a match:

- state
- defer
- clone
- retry
- continuous integration
- copy
- result

  - continuous integration
  - copy
  - result

Create a copy of an existing database object in a sandbox environment.

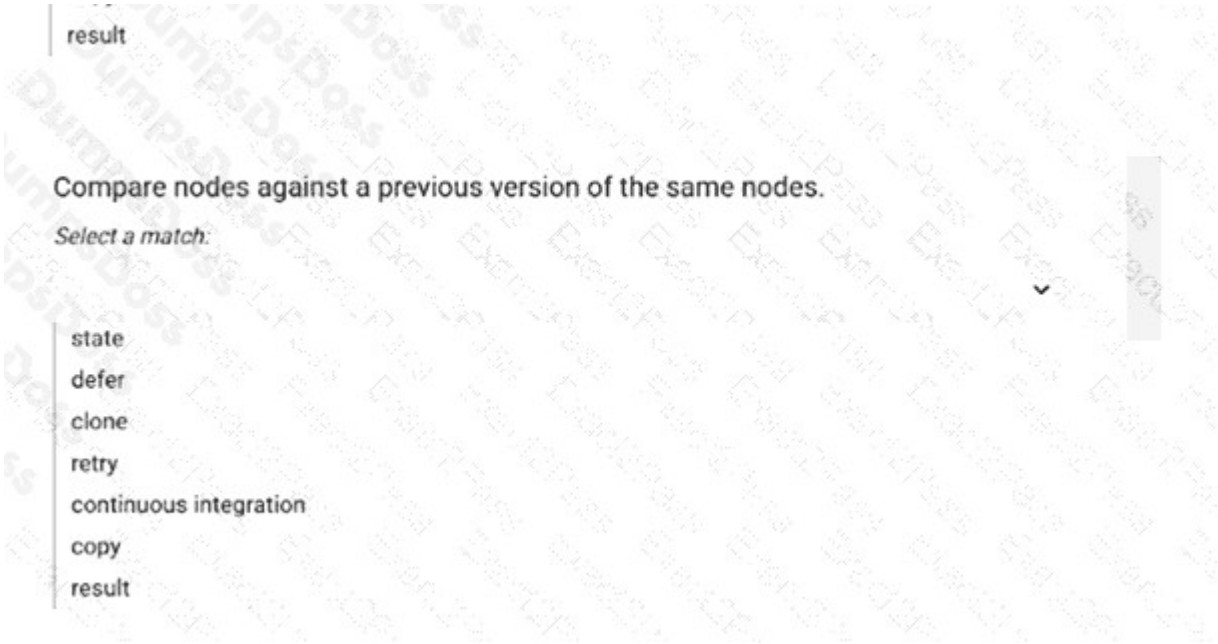
Select a match:

- state
- defer
- clone
- retry
- continuous integration
- copy
- result

Run a subset of models or tests in a sandbox environment without having to first build their upstream parents.

Select a match:

- state
- defer
- clone
- retry
- continuous integration
- copy
- result



ANSWER: See the explanation for the answer

Explanation:

Match the outcomes as follows:

Execute the last dbt command from the node point of failure: `retry`

Create a copy of an existing database object in a sandbox environment: `clone`

Run a subset of models or tests in a sandbox without first building upstream parents: `defer`

Compare nodes against a previous version of the same nodes: `state`

`dbt retry` resumes the prior invocation from its failure point. `dbt clone` creates cloned database objects. The `--defer` argument resolves unselected upstream references to objects in a state environment, and `--state` supplies prior artifacts for state comparison.

QUESTION NO: 5

You have a model named `fct_revenue` that is referenced by several downstream models using `{{ ref('fct_revenue') }}`.

You need the relation built in the data platform to be named `finance_revenue`, without changing any downstream `ref()` calls.

Which configuration should you add to the model? Choose 1 option.

A. `{{ config(alias='finance_revenue') }}`

B. `{{ config(schema='finance_revenue') }}`

C. Rename the model file to `finance_revenue.sql`.

D. Replace downstream references with `{{ ref('finance_revenue') }}`.

ANSWER: A

Explanation:

`{{ config(alias='finance_revenue') }}` is correct because `alias` changes the physical identifier dbt uses when creating the model relation. The dbt resource remains named `fct_revenue`, so downstream models should continue to reference it with `{{ ref('fct_revenue') }}`. At compilation time, dbt resolves that reference to the relation whose physical identifier is `finance_revenue`.

Renaming the SQL file or changing the argument passed to `ref()` would change the dbt resource name and require downstream updates. Setting `schema` changes the relation location rather than its identifier, while a database-level rename outside dbt would cause the project metadata and the warehouse relation to diverge.

QUESTION NO: 6

Examine this query:

select *

from {{ ref(' stg_orders ') }}

where amount_usd < 0

You want to make this a generic test across multiple models.

Which set of two standard arguments should be used to replace {{ ref(' stg_orders ') }} and amount_usd? Choose 1 option.

A. source and column

B. model and column_name

C. model_name and column_name

D. model and field

ANSWER: B

Explanation:

model and column_name is correct because these are the conventional arguments dbt supplies to a column-level generic test. A generic test is implemented as a macro and is invoked from a model's YAML properties, allowing the same validation logic to run against many relations and columns rather than being tied to one hard-coded model. Within the test SQL, `{{ model }}` resolves to the relation currently under test, so it replaces the explicit `ref('stg_orders')`. This lets dbt build the appropriate dependency and execute the test against each configured model.

For a test configured beneath a column in a model's YAML definition, dbt also passes `column_name`. Using `{{ column_name }}` in the predicate replaces the fixed `amount_usd` reference. The reusable test body can therefore select failing records with logic such as `from {{ model }} where {{ column_name }} < 0`, while each YAML test declaration determines which model and column receive that logic. dbt documents the generic-test macro pattern and its standard test context in its guide to writing custom generic tests, and describes configuring these tests on columns in the data tests property reference.

QUESTION NO: 7

Your project has a `packages.yml` file that declares a dependency on a utility package. A developer has changed the package version requirement and needs to install the resolved dependency set.

Which two statements are true about running `dbt deps`? Choose 2 options.

A. It resolves dependencies declared in `packages.yml`.

B. It creates or updates `package-lock.yml` with resolved package versions.

C. It installs declared packages only when `dbt run` is executed afterwards.

D. It requires package declarations to be placed in `dbt_project.yml`.

E. It makes direct edits in `dbt_packages` persistent across future installations.

ANSWER: A B

Explanation:

`dbt deps` reads the dependency declarations in `packages.yml` and installs the resolved packages for use by the project. By default, installed package files are placed in the project's `dbt_packages` directory, unless the project is configured differently.

The command also creates or updates `package-lock.yml`, which records the resolved package versions. This supports reproducible dependency installation for other developers and deployment environments. `dbt run` does not install packages automatically, and direct edits to installed package files are not durable because a subsequent dependency installation can replace them.

QUESTION NO: 8

In development, you want to avoid having to re-run all upstream models when refactoring part of your project.

What could you do to save time rebuilding models without spending warehouse credits in your next command?

A. Replace your {{ ref() }} functions with hard-coded references.

B. Refer to a manifest and utilize the --defer and --state flags.

C. Clone your upstream models from the production schema to the development schema.

D. Leverage artifacts from a prior invocation by passing only the --state flag.

ANSWER: B

Explanation:

Refer to a manifest and utilize the --defer and --state flags. is correct because dbt can resolve unselected upstream nodes against relations that were already built in another environment, typically production, rather than building those dependencies again in the current development schema. The `--state` flag supplies the path to a prior run's artifacts, including its `manifest.json`, which lets dbt compare the current project with that previous state. The `--defer` flag then tells dbt to use the referenced relations recorded in that state manifest when a required upstream node is not being built in the current invocation. For example, a developer can select and run only a changed model while its unchanged parents resolve to their existing production relations. This preserves dbt's dependency-aware `ref()` behavior while eliminating unnecessary warehouse work for the deferred upstream portion of the DAG. dbt documents deferral as a development and CI workflow for working efficiently with large projects, where rebuilding every ancestor would be slow and costly. The state artifacts must be available at a separate path from the target path used by the current invocation. See the dbt documentation on defer and state-based selection and comparison.

QUESTION NO: 9

Your team wants a description of a business calculation to be maintained once and reused on both a model and one of its columns.

Which two actions support this workflow? Choose 2 options.

A. Define the shared text in a Markdown `{% docs %}` block.

B. Reference the block in descriptions with `{{ doc('block_name') }}`.

C. Define the shared text with a `ref()` call in the model SQL.

D. Configure the text as a model contract.

E. Materialize the documentation block as a view.

ANSWER: A B

Explanation:

A documentation block can be defined in a Markdown file with a `{% docs %}` block. This allows longer documentation to be version controlled alongside the dbt project rather than repeated in multiple YAML files.

The block can then be referenced in a resource description with `{{ doc('block_name') }}`. dbt resolves that reference when generating documentation, allowing the same maintained text to appear for multiple resources or columns.

A documentation block does not create a database relation and is not defined with a `ref()` call. A model contract controls an output schema; it does not provide reusable descriptive text.

QUESTION NO: 10

Which two configurations can be applied to a dbt test?

Choose 2 options.

A. on_schema_change

B. tags

C. enabled

D. materialized

E. persist_docs

ANSWER: B C

Explanation:

`tags` and `enabled` can both be configured on dbt tests. Tags provide metadata for organizing tests and allow targeted execution or exclusion through dbt's selection syntax. For example, a team can tag a test as `critical` or `nightly`, then use selectors such as `dbt test --select tag:critical` to run the appropriate subset. Tags may be configured for generic data tests in YAML and can also be configured in a singular test's `config()` block.

`enabled` controls whether dbt includes a test in the project graph and executes it. Setting `enabled: false` is useful when a test is temporarily inappropriate for a target, environment, or deployment context. Like other resource configurations, it can be set directly where the test is defined or conditionally through project configuration and variables. dbt documents `enabled` as a resource configuration and explicitly lists configuration capabilities for data tests, including tags and test-specific configuration such as severity and failure thresholds. See the dbt enabled configuration reference and the dbt tags configuration reference.