dbtLabs.dbt-Analytics-Engineering.v2026-05-12.q29

Exam Code:	dbt-Analytics-Engineering
Exam Name:	dbt Analytics Engineering Certification Exam
Certification Provider:	dbt Labs
Free Question Number:	29
Version:	v2026-05-12
# of views:	102
# of Questions views:	290
https://www.freecram.net/torrent/dbtLabs.dbt-Analytics-Engineering.v2026-05-12.q29.html	

NEW QUESTION: 1

Choose a correct command for each statement.

Will always point to the latest version of the source schema
Select a match
dbt clone
defer
Select a match

Allows to use objects built in the target schema with any downstream tool
dbt clone
defer

Allows to safely modify objects built in the target schema
Select a match
dbt clone
defer

Is a point-in-time operation
Select a match
dbt clone
defer

Allows to use objects built in the target schema with any downstream tool
Select a match
dbt clone
defer

Allows to safely modify objects built in the target schema
Select a match
dbt clone

freecram.net

Answer:

14. Choose a correct command for each statement.

Will always point to the latest version of the source schema

Select a match:

dbt clone

defer

Select a match:

Allows to use objects built in the target schema with any downstream tool

dbt clone

defer

Allows to safely modify objects built in the target schema

Select a match:

dbt clone

defer

Is a point-in-time operation

Select a match:

dbt clone

defer

Allows to use objects built in the target schema with any downstream tool

Select a match:

dbt clone

defer

Allows to safely modify objects built in the target schema

Select a match:

dbt clone

defer

freecram.net

Explanation:

Will always point to the latest version of the source schema

Correct Match: # defer

Allows to use objects built in the target schema with any downstream tool Correct Match: # dbt clone

3## Allows to safely modify objects built in the target schema

Correct Match: # defer

4## Is a point-in-time operation

Correct Match: # dbt clone

defer and dbt clone serve very different purposes in dbt, and understanding their behavior is essential for choosing the right command.

The --defer flag tells dbt to use already-built objects from a previous environment (often production) instead of rebuilding them. Because it always references the existing target schema's most recent objects, it "always points to the latest version of the source schema." Since no objects are overwritten when using defer, it also

"allows safely modifying objects built in the target schema"-your development environment uses production objects without altering them.

By contrast, dbt clone creates a point-in-time copy of existing relations. This cloned schema is static; it does not auto-update when source data changes. Therefore, clone is classified as a "point-in-time operation." Since clone copies physical tables/views into a new schema, downstream tools (BI dashboards, ML pipelines) can safely query the cloned environment without

affecting production, making "allows to use objects built in the target schema with any downstream tool" the correct match.

Thus, defer is used for logic substitution without copies, while clone is used for replicable, point-in-time snapshots.

NEW QUESTION: 2

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: (SHOW ANSWER)

The correct answer is B: Refer to a manifest and utilize the --defer and --state flags.

According to dbt's official documentation, the --defer flag enables developers to defer the resolution of model references (ref(), source()) to an existing manifest-commonly the one generated from a production run.

When this flag is used along with --state, dbt reads the prior project state and uses the already-built production models instead of rebuilding upstream dependencies. This is essential because production models are typically large and compute-intensive. By deferring to production artifacts, developers can test only the models they have modified, dramatically reducing warehouse costs and speeding up development cycles.

This approach is foundational to dbt's recommended development workflow, especially when working with CI

/CD or large DAGs. It preserves the integrity of the dependency graph while avoiding unnecessary recomputation.

Option A violates dbt best practices by removing dependency awareness. Option C is unnecessary operational overhead and does not integrate with dbt's state management. Option D is incomplete because --state alone does not prevent upstream model execution---defer is required to substitute those models with existing artifacts.

NEW QUESTION: 3

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

freecram.net

Compare nodes against a previous version of the same nodes.

Select a match:

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

Answer:

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.

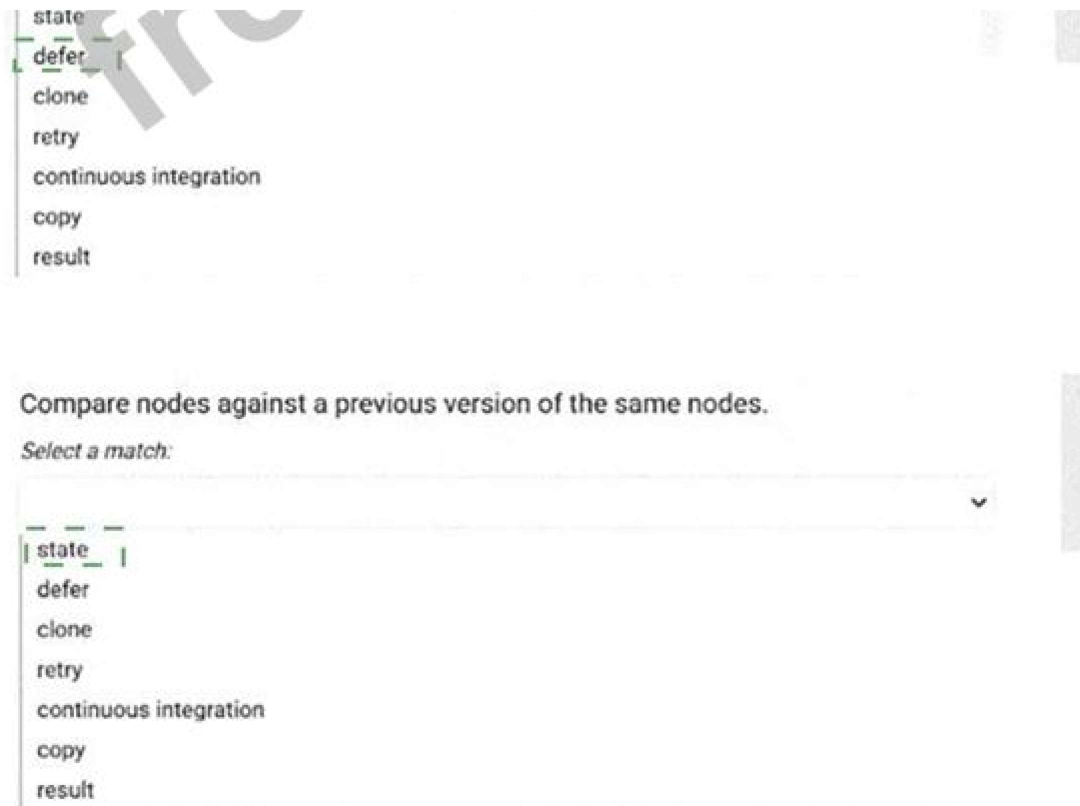
Select a match:

- 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:

freecram.net



Explanation:
Execute the last dbt command from the node point of failure.
The Answer:
retry
The --retry flag in dbt is specifically designed to re-execute a dbt command starting from the node where execution previously failed. In large DAGs, rebuilding all upstream models repeatedly is inefficient, especially when a downstream transformation is the only point of failure. dbt includes retry logic to support resiliency within orchestrated pipelines and CI/CD processes.
Using --retry allows dbt to leverage the metadata stored in the run_results.json to determine which model triggered the failure. Instead of re-running the entire DAG, dbt intelligently continues execution from the failed node, speeding up recovery and making iterative debugging far more efficient. This is particularly valuable in production pipelines where minimizing runtime is critical.
This behavior aligns with dbt's philosophy of incremental, test-driven, modular development: failures should be isolated, reproducible, and simple to resume. By re-running only the subset of affected models, dbt improves developer productivity and orchestration reliability.
**Create a copy of an existing database object in a sandbox environment.
The Answer:
clone
The clone capability in dbt refers to creating a zero-copy clone (Snowflake) or snapshot-like duplicated object using the data platform's native features. This is extremely useful when developers or analysts need a safe sandbox environment to test transformations, experiment, or validate logic without affecting production data.
A clone operation duplicates the table or view metadata instantly without physically copying data, which is both cost-efficient and fast. dbt leverages this through commands such as dbt clone (dbt

Cloud) or through adapters that support cloning.
Sandboxes created via cloning allow engineers to develop safely by ensuring that testing and prototyping has no impact on production tables. Because cloned objects share underlying data blocks, they are nearly storage- free unless mutated-keeping warehouse costs low.
This practice is aligned with modern analytics engineering best practices where separation of environments (dev/test/prod) is essential for governance, data quality, and reproducibility.
**Run a subset of models or tests in a sandbox environment without having to first build their upstream parents.
The Answer:
defer
The --defer flag allows dbt to reference already-built objects from another environment-commonly production-so that developers can run only the models they are modifying, without recreating all upstream dependencies.
When combined with --state, dbt compares the working branch's DAG with previously generated artifacts and determines which nodes should be rebuilt. Any upstream nodes not modified in the current branch are deferred
, meaning dbt will point to the existing production-built version instead of materializing them again.
This dramatically speeds up development because staging layers or intermediate models, which may take significant computation time, do not need to be rebuilt for every developer iteration.
The mechanism also prevents accidental overwrites of production objects because deferred references always point to a safe, isolated schema established by state comparison.
This concept is central to dbt's "safe development via deferral," enabling rapid experimentation while ensuring consistency between dev and prod environments.
**Compare nodes against a previous version of the same nodes.
The Answer:
state
The --state flag allows dbt to compare the current project against a previously generated manifest. This enables dbt to determine which models have changed, which tests must be re-run, and how the DAG differs between two versions.
State comparison is foundational to features such as Slim CI, deferred builds, and change-aware testing. dbt reads the state directory-typically containing artifacts from a production run-and examines differences in SQL, configurations, macros, or schema definitions.
This allows dbt to selectively build only modified nodes, increasing efficiency in CI/CD pipelines and reducing unnecessary warehouse costs. Additionally, state comparison is used for validating backward compatibility, testing versioned models, or enforcing governance rules such as model contracts.
By comparing nodes to their historical equivalents, teams can adopt modern engineering strategies such as impact analysis, regression detection, and selective deployment. This capability aligns with dbt's focus on modular analytics engineering and automated change management.

NEW QUESTION: 4

What must happen before you can build models in dbt?

Choose 1 option.

A. Sources must have been defined in your dbt project.

B. You must have created a service account in your data platform.

C. Underlying data must be accessible on your data platform.

D. Raw data must be cleaned.

Answer: (SHOW ANSWER)

The correct answer is C: Underlying data must be accessible on your data platform.

dbt does not perform data ingestion or data loading. Instead, dbt operates after raw data is already available in your warehouse. This means that before dbt can build any models-whether staging, intermediate, or mart- layer models-the underlying source data must already exist and be accessible in the connected data platform (Snowflake, BigQuery, Redshift, Databricks, etc.). dbt uses SQL to transform existing relations; therefore, if the data platform cannot access the underlying tables or external sources, model execution will fail.

Option A is incorrect because sources do not need to be defined before building models. Models can be built without using sources at all. Source definitions are optional metadata and lineage declarations, not prerequisites.

Option B is incorrect because service accounts are not required; dbt can connect through any credential mechanism supported by the warehouse (OAuth, user accounts, tokens, etc.).

Option D is incorrect because dbt itself performs transformations on raw data-cleaning raw data beforehand is not required; in fact, that is one of dbt's main responsibilities.

Thus, the only true prerequisite is that the warehouse must contain accessible underlying data.

NEW QUESTION: 5

Which two mechanisms allow dbt to write DRY code by reusing logic, preventing writing the same code multiple times?

Choose 2 options.

A. Copy/pasting folders containing multiple models

B. Writing and using dbt macros

C. Creating singular tests

D. Using dbt packages

E. Changing a model materialization from view to ephemeral

Answer: (SHOW ANSWER)

The correct answers are B: writing and using dbt macros and D: using dbt packages.

dbt strongly encourages DRY (Don't Repeat Yourself) principles, and two of the core mechanisms that support reusable logic are macros and packages. Macros allow you to write Jinja-powered reusable functions that can generate SQL statements dynamically, reducing duplication across models, tests, and project logic.

Macros can encapsulate filters, joins, auditing logic, timestamps, and more-allowing developers to centralize logic in one place while referencing it across many models.

Packages extend this concept even further by allowing entire sets of macros, models, tests, and utilities to be imported into a project. Packages like dbt-utils contain widely used generic macros that help standardize transformations and testing. Using packages ensures consistent logic across teams and eliminates the need to rewrite common transformations.
Option A contradicts DRY principles because copy/pasting increases maintenance burden.
Option C is not a mechanism for reusing logic; singular tests validate logic but do not reduce duplication. Option E simply changes a model's materialization and does not support code reuse.
Thus, macros and packages are the only correct dbt mechanisms that provide reusable, modular, DRY logic.

NEW QUESTION: 6

You are working with git to version control the dbt logic.

Order these steps to add or modify transformations to your dbt project.

Options
Move options from here to the answer list

Merge the updated code to the main git branch

Create a new branch in git and switch to it

Run some automated CI checks and/or manual review of the updated code

Update the dbt code according to the new logic

Create a Pull / Merge Request

Answer List
Move options here and sort them into a desired order

freecram.net

Answer:

Options
Move options from here to the answer list

Merge the updated code to the main git branch

Create a new branch in git and switch to it

Run some automated CI checks and/or manual review of the updated code

Update the dbt code according to the new logic

Create a Pull / Merge Request

Answer List
Move options here and sort them into a desired order

Create a new branch in git and switch to it

Update the dbt code according to the new logic

Create a Pull / Merge Request

Run some automated CI checks and/or manual review of the updated code

Merge the updated code to the main git branch

freecram.net

* Create a new branch in git and switch to itB. Update the dbt code according to the new logic

C. Create a Pull / Merge RequestD. Run some automated CI checks and/or manual review of the updated codeE. Merge the updated code to the main git branch
* Version control best practices in dbt follow the same engineering workflow used in modern software development. The first step is always to create a new git branch and switch into it, ensuring all work is isolated from production-ready code and allowing your team to develop safely without affecting others.
Once inside the feature branch, you update the dbt code according to the new logic, which may include modifying models, tests, macros, or documentation.
* Next, you create a Pull Request (PR), also known as a Merge Request, to propose integrating your changes into the main branch. This is important because dbt projects are collaborative, and PRs facilitate peer review, enforce project standards, and prevent regressions. Once the PR is created, automated CI pipelines-such as running dbt build, schema tests, data quality checks, and code-style checks-are executed. Reviewers may also manually inspect code for logic correctness, naming conventions, and modeling consistency.
* After all checks have passed and reviewers approve the PR, the final step is to merge the updated code into the main branch, making the new transformations part of the production dbt project. This workflow ensures governance, reliability, and auditable development practices, all of which are core principles in analytics engineering.

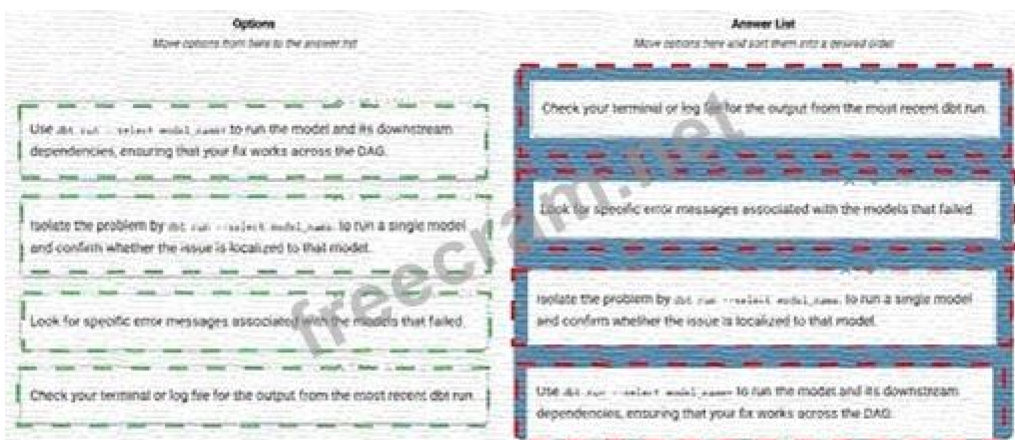
NEW QUESTION: 7
You are working on a complex dbt model with many Common Table Expressions (CTEs) and decide to move some of those CTEs into their own model to make your code more modular.
Is this a benefit of this approach?
The new model can be documented to explain its purpose and the logic it contains.
A. Yes
B. No
Answer: (SHOW ANSWER)
Yes, this is a benefit of breaking large CTE-heavy SQL models into modular dbt models.
According to dbt and Analytics Engineering best practices, modularity improves clarity, maintainability, and documentation quality. When CTEs remain embedded inside a single large SQL file, their purposes are often unclear, difficult to document, and hard for other developers to reuse. By extracting a logical CTE into its own model, dbt treats it as a first-class resource-meaning it can have its own description, tests, documentation, lineage, and metadata defined in YAML.
dbt's documentation system allows each model to include a description explaining what the transformation does, the assumptions being made, and the expected behavior of the data. This aligns with the Analytics Engineering principle of creating self-documenting pipelines, where transformations are transparent and easier for downstream users to understand.
Additionally, modular models improve lineage visualization in the DAG. Instead of a single model hiding multiple transformation layers, a modular structure reveals how data flows through each intermediate step, helping both debugging and governance. Modularization also enables

reusability-other models can reference the intermediate model rather than rebuilding the same logic through duplicated CTEs, supporting DRY (Don't Repeat Yourself) principles.
Therefore, moving CTEs into separate dbt models absolutely provides a documentation benefit and improves the overall engineering quality of the project.

NEW QUESTION: 8

A dbt run failed with an error message.
Order these steps to fix your pipeline.

Answer:



Explanation:
Check your terminal or log file for the output from the most recent dbt run.
Look for specific error messages associated with the models that failed.
3## Isolate the problem by dbt run --select model_name to run a single model and confirm whether the issue is localized to that model.
4## Use dbt run --select model_name+ to run the model and its downstream dependencies, ensuring that your fix works across the DAG.
Brief Explanation
* First, you always inspect the latest run output (step 1).
* Then, identify the exact failing models and error messages (step 2).
* Next, you reproduce the issue on the individual model to be sure the fix works locally (step 3).
* Finally, you re-run the model plus its downstream dependencies to validate the fix across the DAG (step 4).

NEW QUESTION: 9

Examine model stg_customers_sales that exists in the main branch:

```
select
id as customer_id,
name as customer_name
from {{ source('my_data','my_source') }}
```

A developer creates a branch feature_a from main and modifies the model as:

```
select
id as customer_id,
name as customer_name,
country as customer_country
from {{ source('my_data','my_source') }}
```

A second developer also creates a branch feature_b from main and modifies the model as:

```
select
id as customer_id,
name as customer_name,
address as customer_address
from {{ source('my_data','my_source') }}
```

The first developer creates a PR and merges feature_a into main.

Then the second developer creates a PR and attempts to merge feature_b into main.

How will git combine the code from feature_b and the code from main, which now contains the changes from feature_a as well?

Statement:

"As feature_a is already approved and merged to main, the code for the model stg_customers_sales will stay as-is and the changes from feature_b won't be added."

A. Yes

B. No

Answer: (SHOW ANSWER)

Git does not automatically reject or ignore changes from the second branch (feature_b). Instead, Git attempts to merge both sets of changes, and if they modify the same lines or nearby blocks of code, Git produces a merge conflict that must be manually resolved. In this scenario, both feature_a and feature_b introduce new columns into the same SELECT statement of the same model file, meaning Git must reconcile two different edits made in parallel on branches that diverged from the same commit.

Once feature_a is merged into main, the code in main contains the new column customer_country. When developer B then tries to merge feature_b, Git compares the modified file in feature_b with the updated file in main. Since both branches changed the same section of SQL, Git cannot automatically determine the correct combined output. It will not discard feature_b's changes; instead, Git requires the developer to manually merge both sets of additions, typically resulting in a combined SELECT clause with both customer_country and customer_address, unless the developer chooses otherwise.

This behavior is documented in Git fundamentals: when multiple developers modify the same file

region, manual conflict resolution is required. Therefore, the statement claiming feature_b's changes "won't be added" is incorrect.

NEW QUESTION: 10

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

Freecram.net

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: (SHOW ANSWER)

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.

NEW QUESTION: 11

59. When a dbt project is stored in a git repository, a developer wanting to add new models to the dbt project starts by creating a new [dropdown].

- pull request
- branch
- commit
- repository

Once created, the developer can then modify the code of the project and [dropdown] those changes so that they are saved in git.

- commit
- push
- checkout
- pull

Once all the required logic has been added, the developer can create a [dropdown] to have the code go through Continuous Integration and allow manual review.

- push request
- clone
- merge request
- remote
- pull request
- checkout

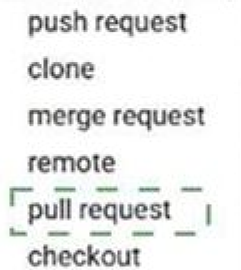
freecram.net

Answer:

59. When a dbt project is stored in a git repository, a developer wanting to add new models to the dbt project starts by creating a new [▾].

Once created, the developer can then modify the code of the project and [▾] those changes so that they are saved in git.

Once all the required logic has been added, the developer can create a [▾] to have the code go through Continuous Integration and allow manual review.



freecram.net

Explanation:
(branch)
(commit)
(pull request)
In dbt development workflows, version control using Git is essential for ensuring collaborative, safe, and trackable changes to analytics code. The correct first step when making updates-such as adding new models-is to create a new Git branch. This isolates development work from the production (main) branch, preventing incomplete or experimental logic from affecting deployed transformations. Branching supports dbt' s modular development approach and aligns with best practices for analytics engineering.
Once the branch is created, the developer modifies SQL models, tests, macros, or documentation as required.
To permanently record these modifications in Git, the developer must commit the changes. A commit serves as a snapshot of progress and creates an auditable history of transformations

made to the project, enabling rollbacks, diffs, and peer review.
After development is complete, the developer submits a pull request (PR). The pull request triggers CI checks-often including dbt build, schema tests, and contract validations-to ensure code quality and identify impacts on downstream models. PRs allow team members to review and comment before changes merge into the main branch, enforcing governance, consistency, and reliability. This workflow embodies the engineering rigor dbt encourages: modular development, testing, versioning, and peer review.

NEW QUESTION: 12

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: (SHOW ANSWER)

The correct answers are B: tags and C: enabled.
dbt tests-both generic and singular-support a limited but clearly defined set of configurations. Two of the supported configurations are tags, which allow grouping and selecting tests using dbt's selector syntax, and enabled, which can be set to true or false to control whether a test should run. These configurations are commonly used to manage large test suites, such as disabling slow tests in development or tagging tests for CI pipelines.
Option A (on_schema_change) is a model-only configuration and cannot be applied to tests.
Option D (materialized) applies only to models, not tests, since tests compile into queries that evaluate failures rather than physical objects.
Option E (persist_docs) applies to models and sources for documentation purposes. Tests do not support persisted documentation properties beyond what appears in test metadata.
dbt's documentation makes clear that tests inherit only a small subset of configurations compared to models- primarily enabled, tags, severity, and custom test arguments. Among the options provided, only tags and enabled are valid and supported.
Thus, the correct answers are B and C.

NEW QUESTION: 13

Which two are true for a dbt retry command?
Choose 2 options.

A. It reruns all nodes in your previous invocation statement.
B. It retries the previous command if it is not a syntax error in a model.
C. It picks up from the error without running all of the upstream dependencies.
D. It reuses selectors from the previous command.
E. It reads a manifest.json file to identify the models and tests that failed in the last run.

Answer: (SHOW ANSWER)
The correct answers are B and E.
The dbt retry command was introduced to help developers quickly re-run only what failed in the prior dbt invocation. According to dbt documentation, dbt retry examines metadata from the previous execution, including the run results and the manifest.json, to determine which nodes failed. This aligns with Option E, as dbt uses the manifest to identify the models, tests, and other nodes that errored or were not executed successfully.
Option B is also correct because dbt retry works only when the previous run failed due to execution errors (e.
g., a warehouse failure or a logic error in SQL). If the failure was due to a syntax error that prevented dbt from compiling, there will be no valid manifest produced, so retry cannot execute.
Thus, dbt retry can only re- run models when the previous failure occurred after successful compilation.
Options A and C are incorrect because retry does not run all nodes nor does it automatically pick up exactly where an execution stopped - instead, it runs failed and downstream unbuilt nodes as determined by the manifest. Option D is also incorrect; retry uses the execution metadata, not selectors, to determine what should run.

NEW QUESTION: 14
28. Consider this DAG:
* model_a # model_c # model_e
* model_b # model_d # model_f
(With model_c and model_d both feeding into the final layer.)
You execute:
dbt run --fail-fast
in production with 2 threads. During the run, model_b and model_c are running in parallel when model_b returns an error.
Assume there are no other errors in the model files, and model_c was still running when model_b failed.
Which model or models will successfully build as part of this dbt run? Choose 1 option.
A. model_a, model_c, model_d, model_e, model_f
B. model_a, model_c
C. model_a
D. model_a, model_c, model_e
Answer: (SHOW ANSWER)
The --fail-fast flag tells dbt to stop scheduling any new nodes as soon as one node fails.
Importantly, dbt does not kill models that are already running; in-flight nodes are allowed to finish.
Here's what happens step by step with 2 threads:
* Roots model_a and model_b start first.
* model_a finishes successfully. That makes model_c eligible to run.
* dbt now runs model_b and model_c in parallel.

* While they are running, model_b fails.
* Because --fail-fast is set, dbt immediately stops scheduling any additional models (like model_d, model_e, or model_f).
* model_c was already running when model_b failed, so it is allowed to complete successfully.
Downstream models of either branch (model_d, model_e, and model_f) never start, because fail-fast prevents any further nodes from being queued after the first failure.
So, the only models that successfully build during this run are:
* model_a (completed before model_b failed)
* model_c (already running at the time of failure and allowed to finish) Hence the correct choice is B: model_a, model_c.

NEW QUESTION: 15
Examine the code:
select
left(customers.id, 12) as customer_id,
customers.name as customer_name,
case when employees.employee_id is not null then true else false end as is_employee,
event_signups.event_name, event_signups.event_date, sum(case when
visit_accomodations.type = 'hotel' then 1 end)::boolean as booked_hotel, sum(case when
visit_accomodations.type = 'car' then 1 end)::boolean as booked_ride from customers
-- one customer can sign up for many events
left join event_signups
on left(customers.id, 12) = event_signups.customer_id
-- an event signup for a single customer can have many types of accommodations booked left join
visit_accomodations on event_signups.signup_id = visit_accomodations.signup_id and
left(customers.id, 12) = visit_accomodations.customer_id
-- an employee can be a customer
left join employees
on left(customers.id, 12) = employees.customer_id
group by 1, 2

Match the operations to the locations which will centralize the transformation steps for downstream use:

sum(case when visit_accomodations.type = 'hotel' then 1 end)::boolean as booked_hotel,
sum(case when visit_accomodations.type = 'car' then 1 end)::boolean as booked_ride

Select a match:

In the first layer of models
In a model between the first layer and final layer of models
In a model dedicated to pre-aggregate data for reporting

left(customers.id, 12) as customer_id

Select a match:

In the first layer of models
In a model between the first layer and final layer of models
In a model dedicated to pre-aggregate data for reporting

customers.name as customer_name

Select a match:

In the first layer of models
In a model between the first layer and final layer of models
In a model dedicated to pre-aggregate data for reporting

left join employees
on left(customers.id, 12) = employees.customer_id

Select a match:

In the first layer of models
In a model between the first layer and final layer of models
In a model dedicated to pre-aggregate data for reporting

freecram.net

Answer:

Match the operations to the locations which will centralize the transformation steps for downstream use:

sum(case when visit_accomodations.type = 'hotel' then 1 end)::boolean as booked_hotel,
sum(case when visit_accomodations.type = 'car' then 1 end)::boolean as booked_ride

Select a match:

In the first layer of models
In a model between the first layer and final layer of models
In a model dedicated to pre-aggregate data for reporting

left(customers.id, 12) as customer_id

Select a match:

In the first layer of models
In a model between the first layer and final layer of models
In a model dedicated to pre-aggregate data for reporting

customers.name as customer_name

Select a match:

In the first layer of models
In a model between the first layer and final layer of models
In a model dedicated to pre-aggregate data for reporting

left join employees
on left(customers.id, 12) = employees.customer_id

Select a match:

In the first layer of models
In a model between the first layer and final layer of models
In a model dedicated to pre-aggregate data for reporting

freecram.net

Explanation:

Operation
Correct Location
Aggregations (booked_hotel, booked_ride)
In a model dedicated to pre-aggregate data for reporting
Standardizing customer_id
In the first layer of models
Selecting customer_name
In the first layer of models
Joining employees
In a model between the first layer and final layer of models
In dbt's recommended modeling framework, transformations should be centralized according to their purpose and level of abstraction. Basic cleaning and column standardization-such as shortening customers.id using left(customers.id, 12)-belongs in the first layer of models, commonly known as staging models. This layer is responsible for producing clean, consistent, analytics-ready fields. Selecting raw descriptive fields like customers.name also belongs in this layer.
Joins that enrich a dataset by combining cleaned staging outputs across domains-such as joining customers with employees-belong in intermediate models. These models unify business logic that sits between the raw staging layer and the final aggregations used for reporting.
Finally, aggregations like counting types of accommodations (booked_hotel and booked_ride) belong in marts or reporting models, especially when they involve summarization that downstream tools (dashboards, BI reports) will consume. This keeps heavy business logic centralized and reusable.

NEW QUESTION: 16
You want to run and test the models, tests, and snapshots you have added or modified in development.
Which will you invoke? Choose 1 option.
Options:
A. dbt build --select state:modified --defer <path/to/artifacts>
B. dbt run --select state:modified --defer --state <path/to/artifacts>
dbt test --select state:modified --defer --state <path/to/artifacts>
C. dbt build --select state:modified --defer --state <path/to/artifacts>
D. dbt run --select state:modified --state <path/to/artifacts>
dbt test --select state:modified --state <path/to/artifacts>
E. dbt build --select state:modified --state <path/to/artifacts>
Answer: (SHOW ANSWER)
The goal in this scenario is:
* Select only models, tests, and snapshots that were modified in development, using:state:modified
* Use already-built production objects whenever possible, so development does not need to

rebuild everything. This is done with:--defer
* Compare development code against production artifacts, which requires:--state <path/to/artifacts>
* Run everything (models + tests + snapshots) in a single command. This is exactly what dbt build does.
* dbt build = run models + run tests + run snapshots + run seeds.
Putting it all together, the correct command is:
dbt build --select state:modified --defer --state <path/to/artifacts>
This single command satisfies all requirements:
* It runs only modified resources.
* It uses the production state artifacts for comparison.
* It defers to production-built models where possible.
* It runs all resource types automatically.
Why the other options are incorrect:
* A # Missing --state, so state:modified cannot function.
* B # Splits run + test separately; question asks which single command you will invoke.
* D # Missing --defer, which defeats the purpose of using production objects.
* E # Also missing --defer.
Thus, Option C is the only correct answer.

Valid dbt-Analytics-Engineering Dumps shared by EduDump.com for Helping Passing dbt-Analytics-Engineering Exam! EduDump.com now offer the **newest dbt-Analytics-Engineering exam dumps**, the EduDump.com dbt-Analytics-Engineering exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com dbt-Analytics-Engineering dumps with Test Engine here: https://www.edudump.com/exams/dbt-Labs/dbt-Analytics-Engineering/premium/ (**359** Q&As Dumps, **35%OFF Special Discount Code: freecram**)

NEW QUESTION: 17
You define a new generic test on model customers in a YAML file:
version: 2
models:
- name: customers
columns:
- name: customer_id
tests:
- unique
- not_null
The next time your project compiles you get this error:

Raw Error:
mapping values are not allowed in this context
in "<unicode string>", line 7, column 21
What is the cause of this error?
A. tests should be a dictionary key, not a list
B. tests should be wrapped in double quotes (")
C. unique and not_null should not be elements in a list
D. unique and not_null should be indented at the same level as tests
Answer: (SHOW ANSWER)
This error occurs because the YAML structure is incorrectly indented, causing dbt's parser (and YAML itself) to misinterpret the test definitions. In dbt, generic tests must be declared as a list under the tests: key, but YAML is extremely sensitive to indentation levels. In the faulty YAML, unique and not_null are indented incorrectly relative to the tests: key, which produces the error: "mapping values are not allowed in this context." According to the dbt Testing documentation, valid generic test syntax follows this exact pattern:
columns:
- name: id
tests:
- unique
- not_null
The indentation under tests: must be consistent and aligned so that YAML interprets the items as list elements, not as malformed mappings. When indentation is wrong, YAML attempts to parse list entries as key-value mappings, which leads to the error seen during compilation.
dbt does not require generic test names to be quoted, nor does it expect tests to be a dictionary. The test list format is correct-only the indentation is wrong. Therefore, the root cause is incorrect YAML indentation, making Option D correct.

NEW QUESTION: 18

15. The `dbt_project.yml` contains this configuration for `grants`:

```
models:
  +grants:
    +select: ['reporter', 'bi']
  finance:
    +grants:
      +select: ['finance']
```

The tables/views for the models not stored under the `finance` folder will be accessible in the data warehouse to the users [dropdown], and the tables/views for the models stored under the folder `finance` will be accessible in the data warehouse to the users [dropdown].

Dropdown 1 options:
- finance
- finance, reporter, and bi
- reporter and bi
- finance, report, bi, and public

Dropdown 2 options:
- finance, reporter, bi, and public
- reporter and bi
- finance
- finance, reporter, and bi

Previous page

Submit page

freecram.net

Answer:

15. The dbt_project.yml contains this configuration for grants:

```
models:
  +grants:
    +select: ['reporter', 'bi']
  finance:
    +grants:
      +select: ['finance']
```

The tables/views for the models not stored under the finance folder will be accessible in the data warehouse to the users [dropdown], and the tables/views for the models stored under the folder finance will be accessible in the data warehouse to the users [dropdown].

- finance
- finance, reporter, and bi
- reporter and bi
- finance, report, bi, and public

- finance, reporter, bi, and public
- reporter and bi
- finance
- finance, reporter, and bi

Previous page

Submit page

freecram.net

Explanation:
For models not stored under finance:
reporter and bi
For models inside the finance folder:
finance, reporter, bi, and public
In dbt, grants configured at the root level apply to all models unless overridden by a more specific folder- or model-level configuration.
The root-level grant is:
+grants:
+select: ['reporter', 'bi']
This means all models by default are selectable by:
* reporter
* bi
Now the finance folder contains its own override:
finance:
+grants:
+select: ['finance']

When dbt merges grants, overrides do not replace the entire list-they add onto inherited grants unless explicitly cleared. Therefore, models inside the finance folder inherit the parent grants and add the finance grant.
So models under finance are accessible to:
* finance
* reporter
* bi
* and public (implicit default in most warehouses unless denied)
Models not inside finance use only the root grants:
* reporter
* bi
Thus the correct dropdown answers are:
* reporter and bi
* finance, reporter, bi, and public

NEW QUESTION: 19
Which is true about writing generic tests?
Choose 1 option.
A. They should accept one or both of these arguments: model and column_name.
B. They must contain a {{ ref() }} snippet to a model.
C. They do not need to be specified under a model's YAML configurations.
D. They should always accept a column_name.
E. They are written using the {% macro %} wrapper.
Answer: (SHOW ANSWER)
The correct answer is E: They are written using the {% macro %} wrapper.
Generic tests in dbt are implemented as macros, which means they must be defined using the {% macro %} Jinja syntax. These macros return a SQL query that evaluates to rows representing test failures. dbt executes these queries and marks the test as failed if any rows are returned. This requirement is explicitly documented:
all generic tests are macros located in a test directory or within a package.
Option A is partially correct but not universal. Generic tests typically accept arguments such as model or column_name, but dbt does not require both or either. Custom tests can accept any parameters defined by the user. Option B is incorrect because tests do not require ref(); dbt injects the model relation for you when the test is executed. Option C is false because generic tests must be declared in YAML under a model or source for dbt to run them. Option D is incorrect because tests do not always need a column_name (e.g., relationship tests, row-count tests, table-level validations).
Therefore, the defining and universally required characteristic is that generic tests are written as macros, making E the correct answer.
NEW QUESTION: 20
Choose whether these scenarios describe a test or a contract:

Choose whether these scenarios describe a test or contract:

Can only be defined on SQL models

Select a match:

test
contract

Errors are returned before the model is built

test
contract

May be configured to customize severity

test
contract

May be run on ephemeral models

test
contract

freecram.net

Answer:

Choose whether these scenarios describe a test or contract:

Can only be defined on SQL models

Select a match:

test

contract

Errors are returned before the model is built

test

contract

May be configured to customize severity

test

contract

May be run on ephemeral models

test

contract

freecram.net

Explanation:

1. Can only be defined on SQL models

The Answer:

contract

2. Errors are returned before the model is built

The Answer:

contract

3. May be configured to customize severity

The Answer:

test

4. May be run on ephemeral models

The Answer:
test
dbt tests and contracts serve different purposes in ensuring data quality and model correctness. Tests evaluate data after it is produced, while contracts validate the structure of a model before dbt attempts to build it.
A model contract is schema-level enforcement that describes required columns, data types, and constraints.
Contracts can only be applied to SQL models, not Python or ephemeral models. Because contracts validate the model's schema before executing any SQL, dbt surfaces those errors before the model is built, preventing invalid schemas from being deployed.
In contrast, tests evaluate the data after dbt builds a model. Tests can be written generically (unique, not_null, relationships, accepted_values) or as custom SQL tests.
They run after the model is materialized. Tests also allow severity configuration, enabling warnings instead of failures for less critical issues-something contracts do not support.
Tests also run on ephemeral models, because dbt expands ephemeral SQL inline within downstream models, allowing tests to still execute logically against the resulting compiled SQL. Contracts, however, do not apply because ephemeral models never materialize into database objects.
Thus:
* "SQL-only" and "errors before build" # contract
* "custom severity" and "run on ephemeral models" # test
If you want the next question formatted the same way, send it!

NEW QUESTION: 21
Examine how freshness is defined at the database level:
- name: raw
database: raw
freshness: # default freshness
warn_after: {count: 12, period: hour}
error_after: {count: 24, period: hour}
loaded_at_field: _etl_loaded_at
How can one table from the source raw be excluded from source freshness?
A. Add freshness: null to the table configuration.
B. Since freshness is defined at the source level, all tables in the source must adhere to the freshness parameters.
C. Add loaded_at_field: null to the table configuration.
D. Add error_after: null to the table configuration.
Answer: (SHOW ANSWER)
In dbt, source freshness can be defined at the source level (affecting all tables) or overridden at the individual table level. When freshness is declared at the source root, dbt applies the freshness policy to every table within that source unless a specific table opts out. The official dbt

documentation states that setting:
freshness: null
at the table level disables freshness checks for that specific table, even when the source itself defines default freshness rules. This is the only documented way to exclude an individual table from a source freshness policy.
Option B is incorrect because tables can override and disable freshness even when the parent source defines it. dbt's configuration hierarchy supports overriding child-level settings on a per-resource basis.
Option C (loaded_at_field: null) does not disable freshness-it simply removes the loaded_at_field, which would cause dbt to error because a freshness check requires a timestamp field to operate.
Option D (error_after: null) only removes the error threshold but does not disable freshness. The table would still participate in freshness checks and could trigger warnings.
Thus, the correct and dbt-supported method to exclude a table from freshness is to explicitly set:
freshness: null
making A the correct answer.

NEW QUESTION: 22
Which two configuration items can be defined under models: in your dbt_project.yml file?
Choose 2 options.
A. schema
B. source
C. tags
D. test
E. target
Answer: (SHOW ANSWER)
The correct answers are A: schema and C: tags.
In dbt, the dbt_project.yml file is the central configuration file that defines model-level settings. Under the models: section, you can specify a wide range of model configurations such as schema, materialized, tags, alias, and custom meta fields. The schema configuration allows you to control which database schema a model should be built in, giving analytics engineers the flexibility to organize models by domain or environment. The tags configuration is also valid under models: and is widely used to group models logically for selection, documentation, or orchestration workflows.
Option B (source) is incorrect because sources are defined under YAML files in the sources: section, not under models: in dbt_project.yml. Option D (test) is incorrect because tests must be defined in model or source YAML files, not inside the project configuration. Option E (target) is not a configuration that applies to models; rather, it refers to dbt runtime environments and cannot be configured under the models: block.
dbt's project configuration system ensures that model-level behavior is managed centrally and consistently, and schema and tags are two of the officially supported configuration keys under models:.

NEW QUESTION: 23

13. An analyst on your team has informed you that the business logic creating the is_active column of your stg_users model is incorrect.

You update the column logic to:

case

when state = 'Active'

then true

else false

end as is_active

Which test can you add on the state column to support your expectations of the source data?

Choose 1 option.

A. - name: state

tests:

- accepted_values:

values: ['active', 'churned', 'trial']

- not_null

B. - name: is_active

tests:

- accepted_values:

values: ['active', 'churned', 'trial']

- not_null

C. - name: state

tests:

- not_null

- unique

D. - name: is_active

tests:

- not_null

- unique

Answer: (SHOW ANSWER)

The purpose of this question is to determine how to validate that the input values in the state column support the business logic that determines the is_active field. Since the logic checks whether state = 'Active', it is critical that the state column only contains values that the business process expects. In the example shown, acceptable states appear to be: 'active', 'churned', and 'trial'.

The correct way to enforce this expectation is to apply an accepted_values test on the state column. This ensures that any unexpected state (e.g., 'inactive', 'pending', 'deleted', or NULL) will cause the test to fail, alerting the team that the upstream system is producing unexpected or invalid values. Additionally, adding not_null ensures every user record contains a valid state.

Option A is the only configuration that applies the accepted values test to the correct column (state) and reflects the expected domain of values.

Options B and D incorrectly apply tests to the derived column is_active, not the source column that needs validation. Option C only checks nullability and uniqueness, which does not validate the range of allowed values and thus does not protect the business logic.
Therefore, Option A is the only correct answer.

NEW QUESTION: 24

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: (SHOW ANSWER)

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.

NEW QUESTION: 25

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: (SHOW ANSWER)
When converting a model-specific SQL query into a generic test, dbt expects the test macro to accept the two standard arguments used across all generic tests: model and column_name.
These arguments allow dbt to automatically pass the correct table and the correct column when the test is applied in YAML.
The argument model represents the actual relation (table/view) being tested. dbt compiles the reference itself-so instead of writing ref('stg_orders'), generic tests always use {{ model }} to represent the referenced relation.
The argument column_name represents the actual column being tested. In this case, the hard-coded column amount_usd becomes the dynamic argument {{ column_name }}, allowing you to apply the logic to any numeric or validated column across multiple models.
So the generic version of your SQL becomes:
select *
from {{ model }}
where {{ column_name }} < 0
The other options are incorrect:
* source is not the standard argument for generic tests.
* model_name is not automatically passed by dbt.
* field is not a recognized standard test argument.
Thus, the two correct standard arguments are model and column_name.

NEW QUESTION: 26
Consider this DAG for a dbt project. You have configured your environment to use one thread. When running dbt run, you determine that model_d fails to materialize.

How will changing the command from dbt run to dbt run --fail-fast impact the execution of dbt run when model_d fails to materialize? Choose 1 option.
Options:
A. dbt will attempt to materialize the rest of the models.

B. dbt will drop the existing version of model_d in the data platform.
C. dbt will attempt to materialize everything else except for model_f.
D. dbt will stop building any additional nodes in the DAG.
Answer: (SHOW ANSWER)
The --fail-fast flag changes dbt's execution behavior by immediately stopping the scheduling of any further models as soon as the first failure occurs. This is true regardless of the number of threads; in this case, the environment uses one thread, meaning models run strictly in sequence based on dependency order.
In the DAG provided, model_d is downstream of model_b, and model_f depends on model_d.
The execution sequence when using a single thread would be something like:
* model_a runs
* model_c runs
* model_e runs
* model_b runs
* model_d runs # fails
Once model_d fails:
* With normal dbt run, dbt would attempt to continue running any unrelated models (though in this DAG, there are no more independent nodes left anyway).
* With --fail-fast, dbt immediately stops scheduling any additional work, halting the run.
Therefore, no further models-including model_f or any other-will run. dbt does not drop models, retry them, or attempt related branches. It simply stops.

NEW QUESTION: 27
You are building an incremental model.
Identify the circumstances in which is_incremental() would evaluate to True or False.

You are building an incremental model.

Identify the circumstances in which `is_incremental()` would evaluate to True or False.

There is a config block at the top of the model with `materialized = 'table'`

Select a match:

True
False

The corresponding table already exists in the data warehouse

Select a match:

True
False

The incremental model was executed with the `--full-refresh` flag

Select a match:

True
False

The corresponding table does not already exist in the data warehouse

Select a match:

True
False

freecram.net

Answer:

You are building an incremental model.

Identify the circumstances in which `is_incremental()` would evaluate t
True or False.

There is a config block at the top of the model with `materialized = 'table'`

Select a match:

True
False (selected)

The corresponding table already exists in the data warehouse

Select a match:

True (selected)
False

The incremental model was executed with the `--full-refresh` flag

Select a match:

True
False (selected)

The corresponding table does not already exist in the data warehouse

Select a match:

True
False (selected)

freecram.net

Explanation:

1. There is a config block at the top of the model with materialized = 'table'
Answer: False
2. The corresponding table already exists in the data warehouse
Answer: True
3. The incremental model was executed with the --full-refresh flag
Answer: False
4. The corresponding table does not already exist in the data warehouse
Answer: False
The is_incremental() macro in dbt is used within an incremental model to determine whether the current invocation should perform an incremental update or a full rebuild. It evaluates to True only during an incremental run when the target table already exists in the data warehouse and dbt is not performing a full refresh.
For is_incremental() to return True, two conditions must be met:
* The model must be materialized as incremental, not table or view.Therefore, if the config specifies materialized = 'table', the model is not incremental, and the function evaluates to False.
* The table already exists in the target schema.In this case, dbt will run the model in incremental mode, making is_incremental() return **True`.
If the incremental model is run with --full-refresh, dbt intentionally rebuilds the entire table, meaning is_incremental() evaluates to False, even if the table exists.
Lastly, if the target table does not exist yet, dbt must create it from scratch, so the run is treated as a full rebuild, not an incremental update, causing is_incremental() to evaluate to False.
Thus, the only scenario where is_incremental() returns True is when the table exists and the model is actually incremental.

NEW QUESTION: 28

Match the information generated from the `dbt docs` command to where the information is retrieved from.

singular tests

Select a match:

data platform information schema
.yml configuration
.sql files

column data types

Select a match:

data platform information schema
.yml configuration
.sql files

generic tests

Select a match:

data platform information schema
.yml configuration
.sql files

SQL code

Select a match:

data platform information schema
.yml configuration
.sql files

column descriptions

Select a match:

data platform information schema
.yml configuration
.sql files

model dependencies

Select a match:

data platform information schema
.yml configuration
.sql files

Answer:

Match the information generated from the `dbt docs` command to where the information is retrieved from.

singular tests

Select a match:

data platform information schema
.yml configuration

freecram.net

.yml configuration

.sql files

column data types

Select a match:

data platform information schema

.yml configuration

.sql files

generic tests

Select a match:

data platform information schema

.yml configuration

.sql files

SQL code

Select a match:

data platform information schema

.yml configuration

.sql files

column descriptions

Select a match:

data platform information schema

.yml configuration

.sql files

model dependencies

Select a match:

data platform information schema

.yml configuration

.sql files

freecram.net

Explanation:

Information Type

Retrieved From

Singular tests

.sql files

Column data types

Data platform information schema

Generic tests
.yml configuration
SQL code
C. .sql files
Column descriptions
.yml configuration
Model dependencies
.sql files
The dbt docs command compiles metadata about your project by gathering information from three primary sources: your warehouse's information schema, your YAML configuration files, and your SQL model files. Understanding which metadata comes from which source is essential for debugging and for effective documentation practices.
Singular tests live inside .sql files within the /tests directory. Since dbt renders these tests directly from SQL files, their definitions appear in documentation sourced from that location.
Column data types come from the warehouse itself. dbt introspects the data platform information schema to retrieve actual types because dbt does not infer or define column types-only the warehouse does.
Generic tests (e.g., unique, not_null, accepted_values) are declared in .yml files. These YAML definitions contain test configurations, descriptions, and parameters, which dbt uses to document and execute these tests.
SQL code for models is naturally sourced from .sql files where the models are defined. This includes logic such as SELECT statements, CTEs, and transformations.
Column descriptions are written exclusively in .yml files. dbt never extracts descriptions from SQL comments-only from YAML.
Model dependencies come from the ref() and source() calls inside .sql model files, which dbt parses to build the DAG.

NEW QUESTION: 29
A developer has updated multiple models in their dbt project, materialized as tables and views. They want to run and test all models upstream and downstream from the modified models that are materialized as views.
What command will achieve this? Choose 1 option.
A. dbt build --select +state:modified, config.materialized:view+
B. dbt build --select +state:modified+
C. dbt build --select @state:modified+, @config.materialized:view+
D. dbt build --select +state:modified +materialized:view+
E. dbt build --select +state:modified, +config.materialized:view+
Answer: (SHOW ANSWER)
The requirement is:
* Select all models that have been modified # this uses the state selector:state:modified
* But only those modified models that are materialized as views # this uses the config selector:config.

materialized:view
* Then include upstream and downstream dependencies of those models # this requires adding + around the selectors.
To combine both selectors, dbt uses a comma-separated list within --select, meaning both selectors must match simultaneously.
The correct syntax is:
dbt build --select +state:modified, +config.materialized:view+
This command:
* Finds models whose state is modified.
* Filters them to only those materialized as views.
* Includes upstream (+) and downstream (+) dependencies.
* Runs and tests the selected nodes because dbt build includes running models, tests, seeds, and snapshots.
Why the other options are wrong:
* A and C use invalid selector syntax (config.materialized:view+ must be prefixed with +).
* B only selects modified models but does not filter by materialization type.
* D uses materialized:view, which is invalid without the config. prefix.
Thus, only Option E satisfies all conditions and uses correct dbt selector syntax.

Valid dbt-Analytics-Engineering Dumps shared by EduDump.com for Helping Passing dbt-Analytics-Engineering Exam! EduDump.com now offer the **newest dbt-Analytics-Engineering exam dumps**, the EduDump.com dbt-Analytics-Engineering exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com dbt-Analytics-Engineering dumps with Test Engine here: https://www.edudump.com/exams/dbt-Labs/dbt-Analytics-Engineering/premium/ (**359** Q&As Dumps, **35%OFF Special Discount Code: freecram**)