GUIDE

Data testing with dbt: the practical guide

Discover the goals of testing with dbt, types of tests, principles & best practices for an effective testing strategy.

Table of Contents

What is data testing? Testing vs. observability 3

Data testing with dbt 4

Types of data tests for dbt 4

- Assertion tests 5
- Using live vs. mock data for assertions 6
- Tests based on live data (dbt tests) 7
- Mock (unit) tests for dbt 12
- Data Diff 14
- A comparison between types of dbt testing 19

Principles for effective data testing with dbt 20

What is data testing? Testing vs. observability

You're here to learn more about using tests in dbt. But what is data testing in the first place, and how is it different from data observability?

Data testing is the process of ensuring data quality by validating the code that processes data ***before its deployed to production***. The goal of testing is to prevent data quality regressions when data-processing code (e.g., dbt SQL) is written or modified. Data testing is about proactively identifying issues before they even happen.

Data testing can help detect and prevent issues like:

1. A change in a SQL column definition causing a major change in a critical business metric
2. Code refactoring causing an unexpected change in a KPI downstream
3. Column renaming breaking a downstream dashboard or reverse-ETL sync to Salesforce

You've probably also heard of **data observability**, but its not the same thing as data testing. Observability is aboutcontinuously monitoring the state of data thats already **in production** (e.g., what tables are there and what kind of data is in them) and identifying anomalies.

Observability helps identify live data quality issues in real-time, like:

1. A column has an unusual % of NULL values since yesterday
2. An analytics event stopped sending
3. A rollup of a revenue column is half of the expected value

But this all happens after your code is merged and your changes are live. The big limitation of data observability is that it applies to data in production and is thus reactive: it detects problems that have already occurred in production. We believe

It's critically important to address data quality proactively with testing as part of a well-formed data quality strategy.

Data testing with dbt

Before we get into the specifics of different types of dbt tests, it's worth stepping back and taking a look at how dbt tests work more broadly.

The concept of a dbt test is really simple: you're making sure the data in your dbt project (a specific model, or many) matches the format that you expect it to. You might be checking for NULL values, asserting that values fall within an expected range, or even just comparing what a table looks like before and after a major code change. Most of the kinds of tests you run will be based on a code change: you're making an update to a model and want to understand the impact of that change.

Logistically, you can define your native dbt tests in a few different places throughout your project, and then run `dbt test` manually or as part of your CI process. But we'll see that there are other, more powerful ways to ensure data quality in your dbt project using tools like data-diff.

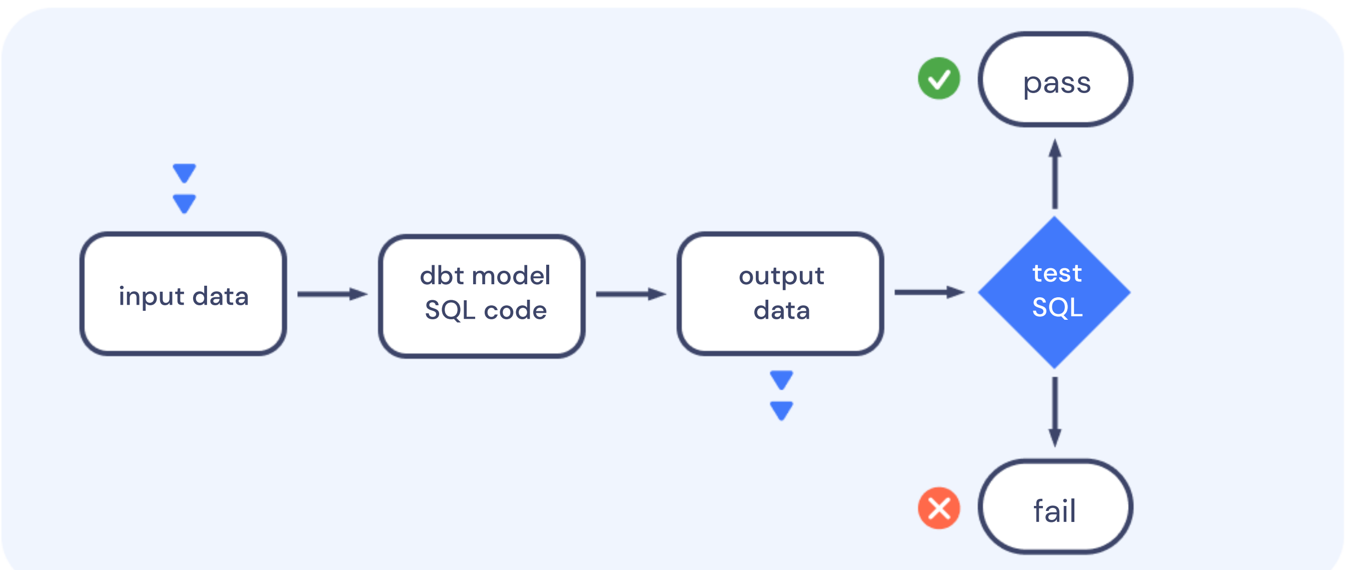
Types of data tests for dbt

There are three major types of testing techniques for dbt code, each with pros and cons. The first two are native to dbt, while the last is a separate open-source package called data-diff.

1. **Assertion tests on live data** – dbt's built-in testing framework for running assertions on your project's data
2. **Assertion tests on mock data** – more similar to unit tests in traditional software engineering
3. **Regression and integration tests** – analyzing the difference between your tables pre- and post-code changes

Assertion tests

Assertion tests, like assertions in software engineering, validate whether your dbt SQL code produces the expected results based on the input data. Here's how it works:



Simple enough, but there are two types of assertion tests you can run in dbt: tests on live data in your project or tests on mock data.

Using live vs. mock data for assertions

An essential distinction in assertion testing is where the input and output data for the test come from. The input/output data for the test can be:

Live (built-in dbt tests) – the dbt model SQL runs against the data in the development or production environment in your data warehouse. The input and output data can change over time.

Mock (unit tests) – the input and output data for testing a dbt model are predefined and stay constant over time (unless a developer makes changes).

The term "mock" comes from testing in software engineering. It means that the inputs and outputs are not real, but rather simulated (mocked), which allows for isolating the system components. In data engineering, mocking allows teams to separate changes in data from changes in code to reduce false positives and negatives.

We'll dive deeper into this distinction and when you should utilize each type of test. But before then, the chart below gives a higher-level overview of the use cases for each:

	Live assertion tests (dbt test)	Mock assertion tests (unit tests)
Test scope	Code and data	Code
Environment context	Development Deployment (CI/CD) Production Can be used to validate data in prod	Development Deployment (CI/CD)
Test input/output data	Variable Based on the data in the warehouse	Constant Mocked
Specificity	Medium Can be prone to noise due to changing data	High Tests precisely what's defined
dbt support status	Official Included in dbt Core and dbt Cloud	Community Exist as standalone packages
Effort to implement	Low Just write a SQL assertion	High Need to create mock input/output datasets
Scalability	Medium/Low Increase compute/runtime with data volume	High No dependencies on live data

Let's dive in.

Tests based on live data (dbt tests)

dbt tests run on live data (i.e., the data in your project) by default. The basic gist is as follows: you write SELECT statements in SQL queries to confirm that the data models, data sources,,

and other resources in a dbt project work as intended. These tests are designed to return rows or records that fail to meet the assertion criteria specified in the test. They can be as simple as asserting that a column has no NULL values or as complex as comparing the values of one column to an aggregation of another.

An important design principle of dbt tests is that these SELECT statements attempt to find failing records: records that would show a test to be incorrect. This is sort of the opposite framing from normal software testing, where certain conditions are asserted to be correct or true.

Within the spectrum of live tests, you can define two types of assertions:

1. **Singular tests** – custom tests meant to validate specific cases for a given model
2. **Generic tests (macros)** – tests that test for common scenarios to be used across multiple models, i.e., reusable tests

Singular tests

When a testing scenario is unique to a single case, and you can't envision applying it across multiple models, you'd write a singular test. Singular tests are focused, written as typical SQL statements, and stored in SQL files (or, in simpler cases, elsewhere) in your dbt project.

Imagine you're an analytics engineer at a SaaS company, and you're looking to add some tests to your dbt project. Your data model has an append-only events table called `stg_button_events` that records all types of button clicks in the web app. There's also a downstream table called `button_clicks_by_day` that aggregates the count of button clicks per day for each button in a column called `number_of_clicks`.

You can write a singular test to ensure the number of button clicks per day is always greater than or equal to zero, and is never negative. The test would be placed in a SQL file in your dbt project, something like `tests/assert_button_clicks_by_day_not_negative.sql`

```
select
  *
from {{ ref('button_clicks_by_day') }}
where number_of_clicks < 0
```

The test will fail if the SQL statement returns any rows, i.e. the number of clicks for a given button on a given day is less than zero.

Generic tests

When a testing scenario is generalizable to several cases or models, it should be implemented as a generic test. Generic tests are written and stored in YAML files, with parameterized queries that can be used across different dbt models. Back to our example: if the concept of testing for rows with a negative value is something you think might be relevant to multiple columns in your project, you could rewrite the above test as generic:

```
{% test column_is_not_negative(model, column_name) %}

select
    {{ column_name }} as should_be_non_negative
  from {{ model }}
where {{ column_name }} < 0

{% endtest %}
```

You'll note that we replaced the specific column we wanted to test in the previous example with`{{column_name}}`, and the table with `{{model}}`. Now, we can reuse this test in any model, for any column, using a quick bit of YAML:

```
models:
  - name: button_clicks_by_day
    columns:
      - name: number_of_clicks
        tests:
          - column_is_not_negative
```

Standard generic tests

Out of the box, dbt includes four generic tests already defined: unique, not_null, accepted_values, and relationships. You can use these in your model YAML without defining them up front in a separate file:

- **unique** – tests that all the values in a given column are unique
- **not_null** – tests that all the values in a given column are not null
- **accepted_values** – you pass an array of accepted values, and it tests that all the values in a given column intersect with that array
- **relationships** – tests for referential integrity with another table (i.e. a join will not miss any rows)

Here's a full example from your company's SaaS app using those tests on the `button_clicks_by_day` model:

```
models:  - name: button_clicks_by_day
    description: This table aggregates data from stg_button_events to
summarize the daily clicks per button.
    columns:
      - name: id
        tests:
          - unique
          - not_null
        description: Each row is associated with a unique combination
of button and date.
- name: button_id
        description: Foreign key to the buttons table.
        tests:
          - not_null
          - relationships:
              to: ref('stg_buttons')
              field: id
- name: button_type
        description: '{{ doc("button_type") }}'
        tests:
          - not_null
          - accepted_values:
              values: ['CTA', 'Administrative', 'External_Link', 'Other']
```

Though it’s only these 4 that are included natively in the core dbt module, the community has built a rich ecosystem of packages with pre-defined generic tests, like dbt-utils and dbt-expectations.

Here are a couple of examples of useful generic tests from the dbt-utils package:

- **equal_rowcount**
  The following test checks to confirm that two models, `date_spine` and `new_users_per_day`, have the same number of rows, i.e., there is a user count for each date in the calendar.

```
models:
  - name: new_users_per_day
    tests:
      - dbt_utils.equal_rowcount:
          compare_model: ref('date_spine')
```

- **unique_combination_of_columns**
  The following test checks to ensure every row in `button_clicks_by_day` represents a unique combination of date and `button_id`.

```
models:
  - name: button_clicks_by_day
  tests:
    - dbt_utils.unique_combination_of_columns:
        combination_of_columns:
          - date
          - button_id
```

And here are a few from the dbt_expectations package:

- **expect_table_aggregation_to_equal_other_table**
  The following test checks to ensure the number of unique date values in `button_clicks_by_day` matches the number of unique date values in the `date_spine` model.

```
models:
  - name: button_clicks_by_day
    tests:
      -
dbt_expectations.expect_table_aggregation_to_equal_other_table:
          expression: count(distinct date)
          compare_model: ref('date_spine')
          compare_expression: count(distinct date)
```

Check out Advanced Testing with dbt-expectations to learn more about using dbt-expectations for sophisticated tests.

Mock (unit) tests for dbt

The core concept of mock tests is that you execute dbt SQL code against a predefined, mocked dataset as opposed to live data in your data warehouse. This allows focusing the test on validating the **code logic**, isolating it from potential changes in the actual data in your warehouse. Because of that isolation, mock tests are fast and highly scalable.

As of today, there is no official support for mocking in dbt Core or dbt Cloud; the feature is in the discussion stage. Having said that, there are two community-developed dbt packages that simplify unit testing. Both packages add convenient methods for dealing with mock data management and differ primarily in how the mock data is curated.

datamocktool

datamocktool utilizes dbt seed functionality and relies on CSV as the main format for your mock data.

Using datamocktool, you can create mock CSV seeds for input and output test data. The input seed replaces the source/ref for a tested model during the test execution, and the output seed is used to validate the model output. Back to our SaaS example:

```
models:
  - name: button_clicks_by_day
    tests:
      - dbt_datamocktool.unit_test:
          input_mapping:
            source('saas_app', 'raw_button_events'): ref('dmt__raw_button_events')
          expected_output: ref('dmt__expected_button_events')
          depends_on:
            - ref('raw_button_events')
    columns: …
```

An advantage of using CSVs to define mock data is the ability to curate them in spreadsheets or easily generate them programmatically in Python / Pandas.

dbt-unit-testing

dbt-unit-testing is a SQL-first unit testing framework. Unlike datamocktool, it relies on SQL to define the inputs and outputs of the tests instead of CSVs. One advantage of defining mock data in SQL is the ability to use Jinja macros to autogenerate datasets. However, some may find this cumbersome and prefer CSV seeds.

```
{{ config(tags=['unit-test']) }}


{% call dbt_unit_testing.test('errors', 'should aggregate average
errors per button to calculate button_error_rate') %}

  {% call dbt_unit_testing.mock_ref ('stg_buttons') %}
    select 1 as id, '' as button_name, '' as button_description,
    select 2 as id, '' as button_name, '' as button_description
  {% endcall %}

  {% call dbt_unit_testing.mock_ref ('stg_button_events') %}
    select 1001 as id, 1 as button_id, true as error
    UNION ALL
    select 1002 as id, 1 as button_id, false as error
  {% endcall %}


  {% call dbt_unit_testing.expect() %}
    select 1 as button_id, 0.5 as error_rate
  {% endcall %}
{% endcall %}
```

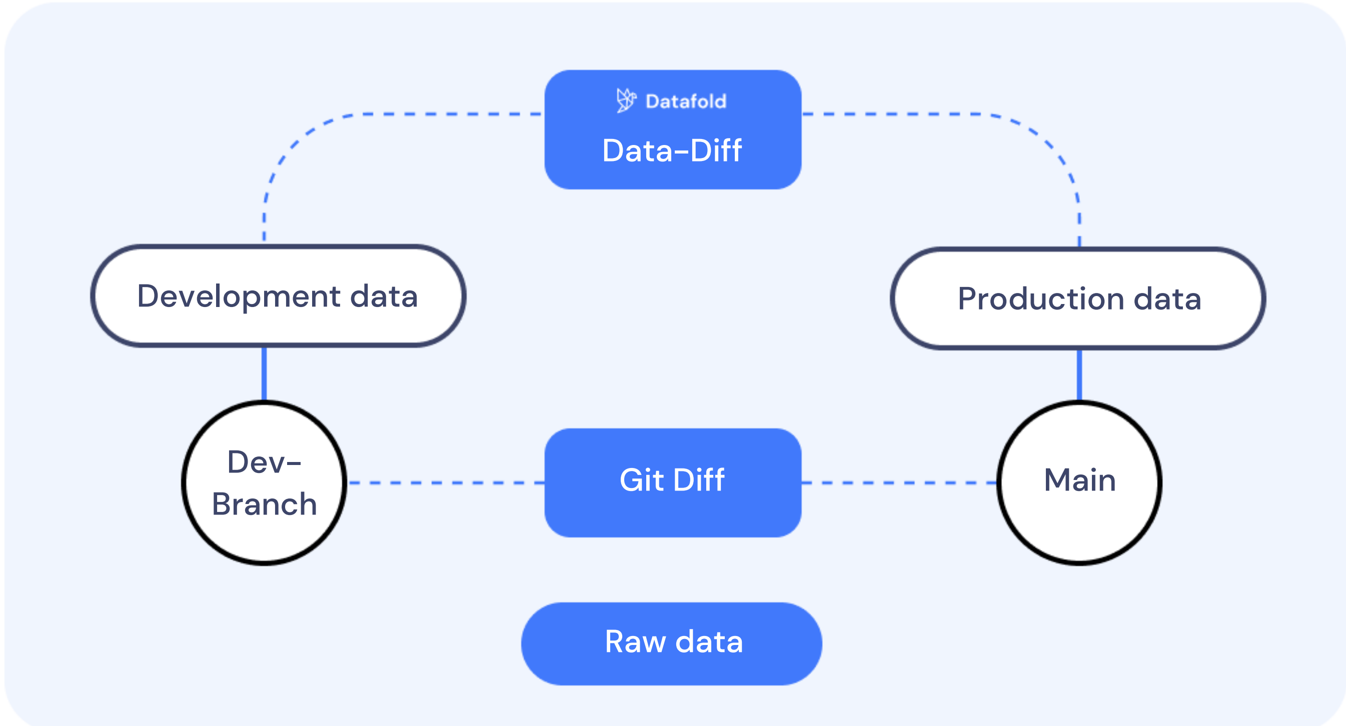
Further reading: learn how Shopify built a unit-testing framework internally.

Data Diff

The two methods of testing we've covered thus far – live and mock data assertions – both rely on the traditional assertion-based testing model. There's one major issue with assertions, though: **scalability**. Each test must be manually defined, implemented, and tuned, making it virtually impossible to cover all potential failure scenarios. Assertions also risk blind spots: we write tests to check for failure scenarios that either have occurred or that we anticipate may occur, but many failure modes are impossible to anticipate. In other words, we don't know what we don't know when it comes to testing.

This is where **data diff** – a tool for comparing datasets – can come in handy. Similar to how one runs dbt tests on the staging environment to validate changes to dbt SQL,

you can run data diff to compare staging data with production data to see ***how each code change will impact the data practically***. The power of data diff is in its ability to highlight all changes and therefore help you catch all potential issues without having to write tests beforehand.



data diff compares data in a similar way git diff compares code

Let's take a look at how you can use open-source data-diff to implement testing for your dbt project.

Using open-source data-diff with dbt in development

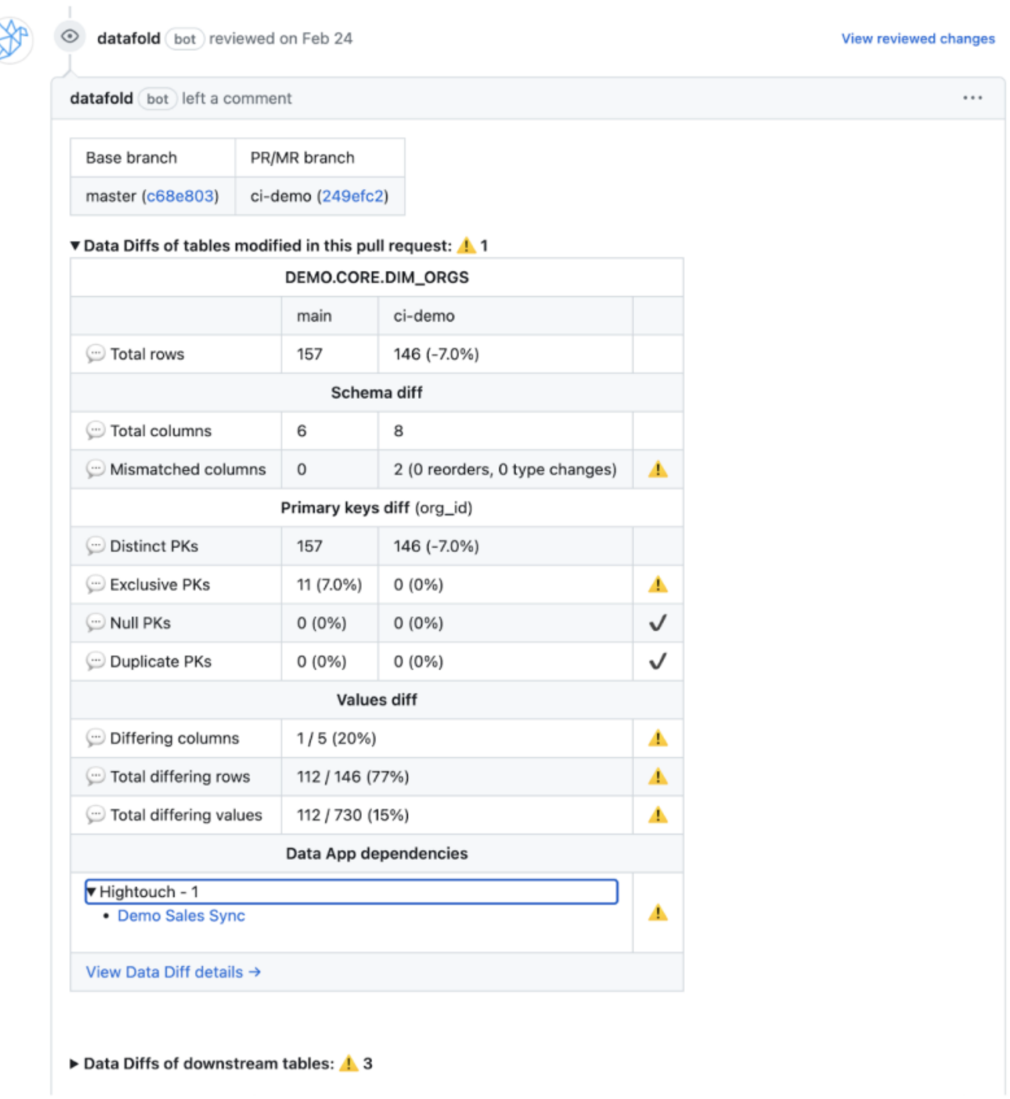
data-diff is open-source and is easy to integrate into your dbt project. After installation, you can run data-diff after executing the `dbt run` command locally to see how the most recent code change impacted your data. The diff shows you useful information like how many rows were added and removed, which values changed in which columns, and more.

```
(env) ➜  demo git:(duckdb_demo) ✗ data-diff --dbt
Running with data-diff=0.7.7


duckdb_demo.prod.dim_orgs <> duckdb_demo.dev.dim_orgs
Column(s) added: {'employee_range', 'org_name'}

  Rows Added    Rows Removed
------------  --------------
           0              11

Updated Rows: 112
Unchanged Rows: 34

Values Updated:
sub_price: 0
created_at: 112
sub_plan: 0
num_users: 0
sub_created_at: 0
```

data-diff supports all major cloud data warehouses and even lets you compare tables across databases.

▶ Getting data-diff into your CI process, and the cloud UI

Where data-diff gets really powerful is the ability to integrate it into your **CI process**. When you make a code change in dbt and open a Pull Request in GitHub, you can configure data-diff to run automatically and show results inline:



datafold bot reviewed on Feb 24 — View reviewed changes

datafold bot left a comment

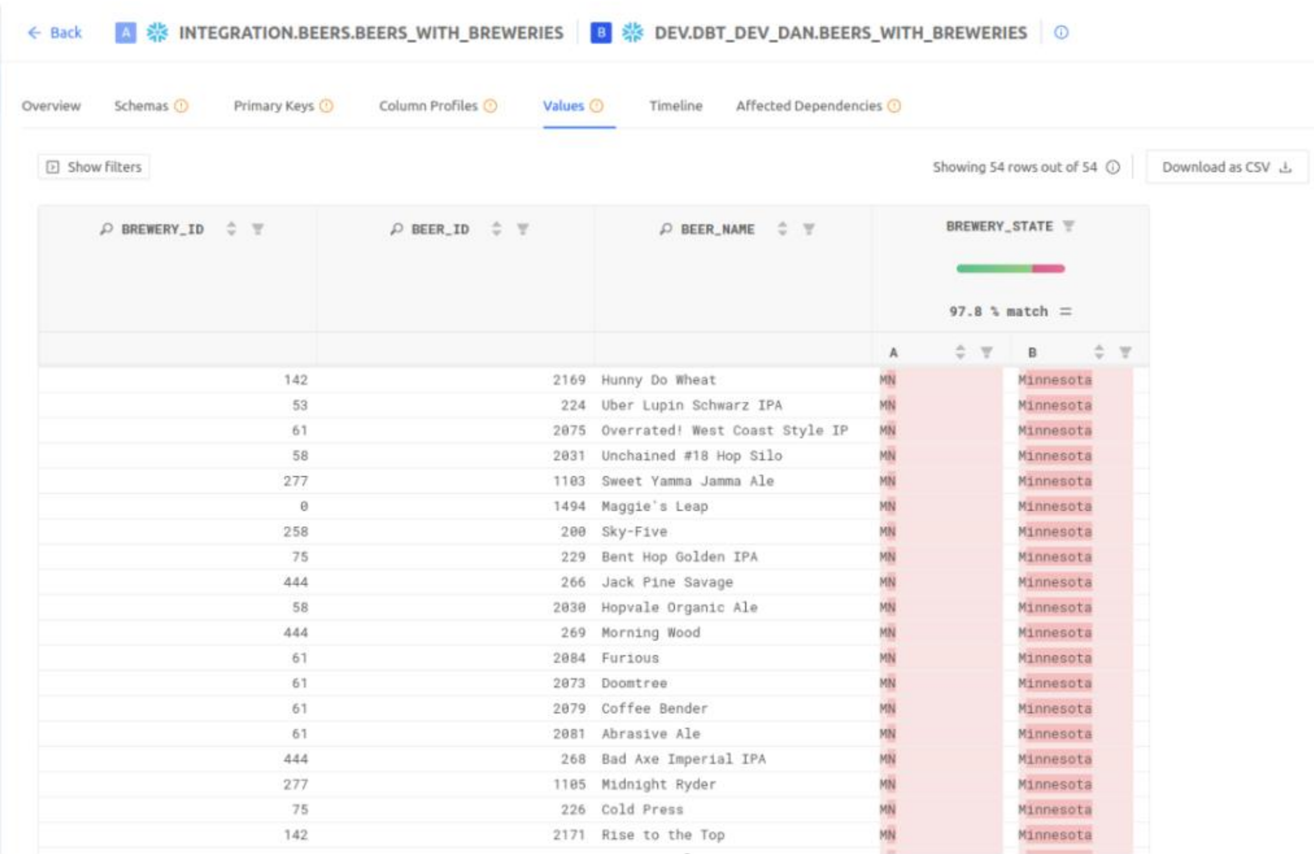
Base branch	PR/MR branch
master (c68e803)	ci-demo (249efc2)

▼ Data Diffs of tables modified in this pull request: ⚠ 1

DEMO.CORE.DIM_ORGS			
	main	ci-demo	
Total rows	157	146 (-7.0%)	
Schema diff			
Total columns	6	8	
Mismatched columns	0	2 (0 reorders, 0 type changes)	⚠
Primary keys diff (org_id)			
Distinct PKs	157	146 (-7.0%)	
Exclusive PKs	11 (7.0%)	0 (0%)	⚠
Null PKs	0 (0%)	0 (0%)	✓
Duplicate PKs	0 (0%)	0 (0%)	✓
Values diff			
Differing columns	1 / 5 (20%)		⚠
Total differing rows	112 / 146 (77%)		⚠
Total differing values	112 / 730 (15%)		⚠
Data App dependencies			
▼ Hightouch - 1 • Demo Sales Sync			⚠
View Data Diff details →			

▶ Data Diffs of downstream tables: ⚠ 3

You can also get access to a powerful UI that helps visualize your data diffs:



Like dbt, data-diff is open-core but also offers a cloud component. The CI integration and cloud UI are part of the [cloud version of data-diff](), and arent available in the simpler open-source package.

A comparison between types of dbt testing

We've covered three types of testing for your dbt project: two types of assertion-based testing and data diffing. A healthy data quality regimen will use several types of tests, and they're not one size fits all. Here's a quick comparison chart:

	dbt tests	mock tests	data diff
Effort to implement	Medium Requires test cases to be written	High Requires test cases and input/output dataset curation	Low Doesn't require manual test setup
Expected code coverage	Medium Relatively easy to add standard tests	Low Given the high effort to set up, should be applied to the most critical and complex logic only	High Automatically captures all changes to data
Specificity How clear the test results are	Medium Reacts to changes in code and data	High Focused on testing code given fixed data inputs	Medium Requires the user to interpret results
Optimal for	Testing general rules such as uniqueness, value ranges, and sets, referential integrity	Testing complex business logic	Understanding the impact of every code change on data and downstream models
Scalability with data volume	Poor Can slow down the project if the input data grows large	Excellent Independent of the data volume	Excellent Scales well using sampling and filtering

Principles for effective data testing with dbt

Now that we've covered the specifics of tests in dbt, the final section of this guide is going to run through how to use those tests effectively. Our team works with some of the most advanced organizations in the world when it comes to data quality, and we've gathered general principles for approaching data testing with dbt. These are designed to be helpful to both mature dbt deployments and teams just getting starting with dbt.

1) Implement basic assertion tests

The idea of writing an assertion test for every possible failure case probably seems daunting; just start with essential tests first. The **CUR framework** defines three types of easy tests:

1. **Completeness** (testing for NULL values)
2. **Uniqueness** (testing primary key uniqueness)
3. **Referential Integrity** (e.g. ensuring that joining related tables wont miss any records)

Each of these test types is included in the generic dbt test functionality, making it easy to add to every model.

2) Test during CI

While it's alright to start small, organizations serious about data quality should testing of every code change. To do that reliably, every code change that is proposed for a dbt project (pull/merge request) has to be tested during CI.

Setting up a CI pipeline for your dbt project is very straightforward if you're using dbt Cloud. For dbt Core users, you can implement similar functionality by using custom CI runners like Github Actions to build and test your data before deploying code to production.

3) Keep your tests healthy

As the number of tests in your dbt project grows, it's essential to keep good test hygiene. The #1 rule of test hygiene is **never to proceed to deploy with failing tests**. That means if certain tests fail, the developer needs to either fix their code, update the test, or remove the test completely. When the team starts to deploy with failing tests, the tests lose their relevance and value.

We've seen organizations define owners for each model, which can help make the process of updating tests easier. Owners are responsible for the entire model and the tests associated with that model.

4) Use tests as documentation

Besides validating code correctness, assertion tests are a valuable source of knowledge about the data, helping others on the team familiarize themselves with the model's contents and behavior of data without having to spend time extensively querying it.

For example, by including a simple `unique` test to a column, you communicate to others that this column is unique, and they wouldn't need to verify that or write excessive deduplication logic when querying that data.

5) Establish and enforce testing guidelines

Writing and maintaining tests is yet another activity that data teams "have" to do. To ensure great data quality while minimizing the burden on the team, it's best to establish project-

wide testing guidelines that define what types of tests should be written. For example, you may require that each dimension table should have at least one uniqueness test, and so on.

Enforcing testing guidelines is essential to maintaining a healthy codebase. And just like with the tests themselves, it's most effective when fully automated as part of the CI process.

dbt-coverage is an excellent, highly configurable package for calculating and reporting on the code coverage for your dbt project. You can plug it into CI to ensure that every code change or new model complies with the defined testing guidelines

```
$ cd jaffle_shop
$ dbt run  # Materialize models
$ dbt docs generate  # Generate catalog.json and manifest.json
$ dbt-coverage compute doc --cov-report coverage-doc.json  # Compute doc coverage, print it and wri

Coverage report
======================================================================
jaffle_shop.customers                                  6/7      85.7%
jaffle_shop.orders                                     9/9     100.0%
jaffle_shop.raw_customers                              0/3       0.0%
jaffle_shop.raw_orders                                 0/4       0.0%
jaffle_shop.raw_payments                               0/4       0.0%
jaffle_shop.stg_customers                              0/3       0.0%
jaffle_shop.stg_orders                                 0/4       0.0%
jaffle_shop.stg_payments                               0/4       0.0%
======================================================================
Total                                                 15/38     39.5%
```

6) Implement data-diff

Data diffing is a great place to start with testing. From the very first run, it adds visibility into the data and helps develop faster and with higher confidence. And unlike assertion tests, it doesn't require developers to create test cases in advance.

Open-source data-diff takes a few minutes to install and configure, and makes it easy to visualize the downstream impacts of your code changes in dbt. And for a richer experience, you can automate testing in CI (plus use Datafold's intuitive cloud UI) to make sure bad dbt code never makes it into production.

Data testing with dbt: the practical guide

Goals of testing with dbt, types of tests, principles & best practices for an effective testing strategy.

Datafold, 2023

support@datafold.com
datafold.com