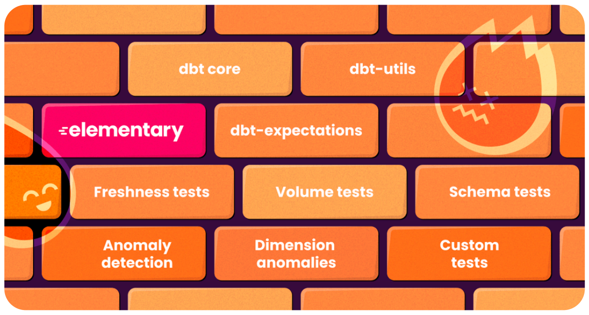


dbt tests: How to write fewer and better data tests?

By Ari Bajo Rouvinen

Share to

DBT

dbt has simplified the process of creating data models in SQL. Many data teams quickly grow their data warehouse to hundreds of tables. Then, inevitably, things start to break, and they follow up by adding hundreds of dbt tests. Consequently, with code changes and data updates, your data team may quickly experience alert fatigue with failing tests.

While data modeling helps you write fewer and better data models, this guide is here to help you write fewer and better dbt tests. But why would you want to write fewer tests?

There are two aspects to writing fewer dbt tests:

1. Develop fewer tests. Leverage existing dbt testing packages. Don't spend time writing code that someone else already wrote.
2. Add fewer tests. With each new test, there is a maintenance, performance, and alerts overhead. Don't add tests that you will ignore when they fail.

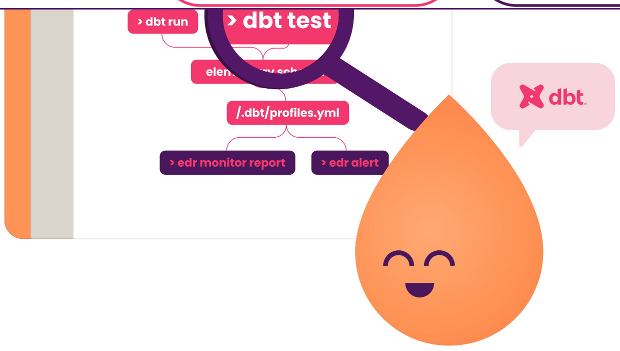
Fewer tests don't necessarily mean a lower test coverage of your data and code, as you can increase test coverage with better dbt tests. What do I mean by better dbt tests?

A better dbt test is a test that is easier to add, easier to understand, and easier to fix. This article gives examples of what I consider better tests than their counterparts. My experience with data testing comes from building internal testing libraries as a Data Engineer and, more recently, as a Developer Advocate at Airbyte and Datafold.

If you want to add dbt tests but don't know how to get started or your team is overwhelmed maintaining dbt tests, I hope you find this guide helpful. It covers popular dbt testing packages, schema tests, freshness tests, volume tests, column tests and custom dbt tests.

Search based on use-case, package, or what the test applies to.

Go to dbt test hub



dbt testing packages: testing data vs. testing code changes

Data teams add data tests to increase confidence in the quality of their data and code tests to streamline the development and code review process. On the one hand, dbt data tests are used to catch data quality issues even when you haven't updated your dbt code, after a dbt run. On the other hand, dbt code tests help you validate code changes before deploying to production.

Hundreds of generic dbt tests are available in dbt packages such as dbt-core, dbt-utils, dbt-expectations, and elementary. dbt-core provides a limited selection of built-in data tests. dbt-utils contains generic dbt tests and macros to simplify the process of writing custom dbt tests. dbt-expectations contains hundreds of individual tests that require hardcoding expectations. elementary anomaly detection tests compare new data with historical data to compute expectations dynamically.

Popular code testing packages include dbt-unit-testing, dbt-audit-helper, and data-diff. dbt unit tests help catch regressions when refactoring dbt macros, Jinja logic, and dbt models. dbt-audit-helper and data-diff allow you to compare development and production data, removing the need to write custom code testing logic.

This article covers popular dbt tests spread across multiple dbt packages and best practices over when to use each dbt test, as there can be quite an overlap.

Writing dbt schema tests: Detect schema changes

Getting started with dbt tests can feel overwhelming as you can add multiple tests for each model and column. Adding dbt schema tests to critical models to detect schema changes is a good starting point. What are critical models? The ones you will care the most about if they break. These include your dbt source models, final tables connected to applications, and models with a high volume of queries. For example, detecting schema changes on dbt source models can save you time debugging downstream errors.

When dbt creates a table in a data warehouse, the warehouse's SQL engine automatically assigns a datatype to every column. By default, no other constraints are applied to your table and columns, such as primary key constraints. Thus, many data teams resort to casting types when creating staging models and adding tests to check for the schema after the model is created. From dbt version 1.5, you can use a data contract definition data_type. While dbt tests validate your models after they are built, data contracts prevent models that don't fit the contract from getting built.

test foreign keys.

```
models:
  - name: orders
    columns:
      - name: order_id
        tests:
          - not_null
          - unique
      - name: status
        tests:
          - accepted_values:
              values: ['placed', 'shipped', 'completed', 'returned']
      - name: customer_id
        tests:
          - relationships:
              to: ref('customers')
              field: id
```

<filename>.yml hosted with ❤ by GitHub view raw

These are useful tests but limited. However, dbt packages can extend your project with many additional use cases. For example, to test types, you can use the dbt-expectations column test dbt_expectations.expect_column_values_to_be_of_type or Elementary model schema tests (more below).

```
models:
  - name: login_events
    columns:
      - name: loaded_at
        tests:
          - dbt_expectations.expect_column_values_to_be_of_type:
              column_type: timestamp
```

<filename>.yml hosted with ❤ by GitHub view raw

Manually adding column-type tests for all columns can be tedious work. The elementary.schema_changes_from_baseline test will check for schema changes against the baseline column schema defined. The configuration for this test can be auto-generated with a dbt macro elementary.generate_schema_baseline_test. You can reduce the configuration to monitor only the columns you care about.

```
models:
  - name: login_events
    columns:
      -name: loaded_at
        data_type: timestamp
      - name: event_name
        data_type: text
      - name: event_id
        data_type: integer
    tests:
      - elementary.schema_changes_from_baseline:
          tags: ["elementary"]
```

<filename>.yml hosted with ❤ by GitHub view raw

Elementary also provides a model test elementary.schema_changes to detect all schema changes. The dbt schema change test provided by Elementary alerts on deleted tables, deleted or added columns, or changes in the data type of a column. This test is more suited to get a feed of all schema changes.

tests:

```
models:
  - name: login_events
    columns:
      - name: raw_event_data
        tests:
          - elementary.json_schema:
              type: object
              properties:
                event_id:
                  type: integer
                event_name:
                  type: string
                event_args:
                  type: array
                  items:
                    type: string
              required:
                - event_id
                - event_name
```

<filename>.yml hosted with ❤ by GitHub view raw

Writing dbt freshness tests: Detect missing data

One of the most common data quality issues is having no new data in your dbt models or late arriving data. dbt freshness tests check for the latest time your dbt sources and models were updated. You may miss new data for many reasons, such as your data ingestion jobs failing or your upstream dbt models failing. dbt-core provides logic to check for dbt sources' freshness. To check for the freshness of downstream models, you can leverage dbt testing packages.

dbt source freshness tests are configured within a freshness block under your source definition. Your dbt source needs to have a column that can be converted into a timestamp (specified as the loaded_at_field) for dbt to detect the last time the model got new data. On top of that, you need to specify one or both of the warn_after and error_after blocks with a period count and a time period.

```
sources:
  - name: jaffle_shop
    database: raw

    freshness: # default freshness
      warn_after: {count: 12, period: hour}
      error_after: {count: 24, period: hour}

    loaded_at_field: _etl_loaded_at

    tables:
      - name: customers # this will use the freshness defined above

      - name: orders
        freshness: # make this model freshness more strict
          warn_after: {count: 6, period: hour}
          error_after: {count: 12, period: hour}
          # Apply a where clause in the freshness query
          filter: datediff('day', _etl_loaded_at, current_timestamp) < 2

      - name: product_skus
```

To test the freshness of other dbt models (not sources), you can leverage packages such as dbt-utils, dbt-expectations and elementary. dbt-utils provides a dbt_utils.recency model test with options datepart, field, and interval. dbt-expectations provides a column test dbt_expectations.expect_row_values_to_have_recent_data with options datepart and interval.

```yaml
models:
  - name: login_events
    columns:
      - name: loaded_at
        tests:
          - dbt_expectations.expect_row_values_to_have_recent_data:
              datepart: day
              interval: 1
```

<filename>.yml hosted with ❤️ by GitHub view raw

One of the biggest challenges with freshness tests is determining the expected data frequency. While you can do this by querying your data to see how often it gets updated or hardcode it based on your requirements, the elementary.freshness_anomalies test computes expected frequencies and only requires specifying a timestamp_column.

The test is highly configurable with options such as anomaly_sensitivity and time_bucket if you want to fine-tune the freshness expectations for specific models.

```yaml
models:
  - name: login_events
    tests:
      - elementary.freshness_anomalies:
          timestamp_column: "loaded_at"
          anomaly_sensitivity: 2
```

<filename>.yml hosted with ❤️ by GitHub view raw

Writing dbt volume tests: Detect volume anomalies

Checking only for new data with freshness tests may not be enough to detect when your data is incomplete. For example, if you get events from several platforms and only data from one platform is missing, the data would be up to date but lacking.

To test dbt row counts, you can use the dbt-expectations model test dbt_expectations.expect_row_values_to_have_data_for_every_n_datepart or the more flexible elementary.volume_anomalies test. Elementary provides a smaller collection of tests than dbt-expectations but with more flexibility than hardcoding strict expectations. This makes it easier to start adding dbt tests, relying on Elementary to learn for you what to expect based on your historical data, while still being able to fine-tune the anomaly detection algorithm with arguments.

For example, the elementary.volume_anomalies test monitors the row count of a table per time bucket and compares it to previous time buckets. The anomaly_sensitivity argument (default to 3) checks that values fall within an expected range of 3 standard deviations from the average in the training set. The training set can be controlled with the options such as days_back, min_training_set_size, time_bucket, and seasonality.

```yaml
models:
  - name: login_events
    tests:
      - elementary.volume_anomalies:
          timestamp_column: “loaded_at”
```

```yaml
     days_back: int
     backfill_days: int
     min_training_set_size: int
     time_bucket:
       period: [hour | day | week | month]
       count: int
     seasonality: day_of_week
```

<filename>.yml hosted with ❤️ by GitHub view raw

A limitation with volume tests is that sometimes you can ingest a similar number of rows for a table but experience a significant variation within one dimension value (ex: events within one country). The dimension_anomalies test monitors the frequency of values in the configured dimension over time and alerts on unexpected changes in the distribution.

```yaml
models:
 - name: login_events
  config:
   elementary:
    timestamp_column: "loaded_at"
  tests:
   - elementary.dimension_anomalies:
     dimensions:
      - event_type
      - country_name
     time_bucket:
      period: hour
      count: 4
```

<filename>.yml hosted with ❤️ by GitHub view raw

Writing dbt column tests: Detect value anomalies

Once your row volume counts are covered, you may monitor values and value properties across columns. Elementary provides a column test column_anomalies, which computes a dozen metrics (null_percent, average_length, variance, …) for each column to detect anomalies.

```yaml
models:
 - name: login_events
  config:
   elementary:
    timestamp_column: 'loaded_at'
  columns:
   - name: user_name
    tests:
     - elementary.column_anomalies:
       column_anomalies:
        - missing_count
        - min_length
       where_expression: "event_type in ('event_1', 'event_2') and country_name != 'unwanted country'
       time_bucket:
        period: day
        count: 1
```

<filename>.yml hosted with ❤️ by GitHub view raw

Elementary makes it easier to monitor column values with the all_column_anomalies test than adding a single test for each column and column property with dbt-expectations.

column level monitors included as part of elementary.column_anomalies.

Writing dbt custom tests: Check business rules

When it comes to detecting data quality issues, there is likely an existing test in a dbt testing package that you can leverage. Most dbt tests can be configured with arguments, such as the where clause in dbt built-in tests. This is more useful than you would expect. For example, validating that every paying user has an 'activated_at' timestamp and a valid payment method sounds like a custom business logic, but it can be as simple as:

```yaml
columns:
  - name: activated_at
    tests:
      - not_null:
          where: "user_status = 'activated'"
  - name: payment_method
    tests:
      - accepted_values:
          values: ['credit_card', 'gift_card']
          where: "user_status = 'activated'"
```

<filename>.yml hosted with ❤ by GitHub view raw

If after checking existing dbt tests, you still miss a test that fits your need, consider writing a custom dbt test as a singular test or generic test. For example, you may have specific business rules to test or KPIs to monitor. Besides detecting data that need to be flagged or fixed, tests can also be used to monitor operations, such as an unusual number of business orders. Even if the numbers are legitime, you still want to be alerted of operational issues.

A singular dbt test is written in a SQL file with a query that returns records that fail the test. A generic dbt test is defined in a SQL file with a test block and a macro definition. Like in a singular test, a dbt test macro should contain a select statement that returns records that don't pass the test. Additionally, the macro takes a model and column_name to be injected with Jinja templates, and extra arguments passed when configuring the test.

You can find dbt test examples by inspecting the code of popular open-source dbt packages. For example, this is how the built-in relationship test code looks on dbt-core.

```sql
{% macro default__test_relationships(model, column_name, to, field) %}

with child as (
    select {{ column_name }} as from_field
    from {{ model }}
    where {{ column_name }} is not null
),

parent as (
    select {{ field }} as to_field
    from {{ to }}
)

select
    from_field

from child
left join parent
    on child.from_field = parent.to_field
```

```
{% endmacro %}
```

relationships.sql hosted with ❤️ by GitHub view raw

When writing custom tests, you don't always need to start from scratch. You can leverage dbt-utils introspective macros such as dbt_utils.get_column_values or turn a custom SQL expression into a test with dbt_utils.expression_is_true. For example, this can be useful to test in financial models that subtotal columns equal a total.

```
models:
  - name: payments
    tests:
      - dbt_utils.expression_is_true:
          expression: "inbound_amount = payout_amount + refund_amount"
      - dbt_utils.expression_is_true:
          expression: "inbound_amount = inbound_cash_amount + inbound_credit_amount"
```

<filename>.yml hosted with ❤️ by GitHub view raw

Planning for test failures: Add description, tag, severity, and owner

The best time to document what to do when the test fails is when you add it. You can use the dbt test description field to explain which data quality issues the tests catch, the dbt test owner field to notify owners, the dbt test tag field to group tests together, and the dbt test severity field to trigger a response.

Elementary also makes it easy to generate and customize Slack alerts for dbt tests with extra fields in the configuration: custom channel, suppression interval, alert fields, and more.

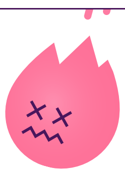
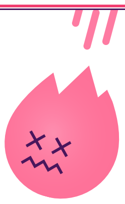
```
columns:
  - name: payment_method
    tests:
      - not_null:
          where: user_status = 'activated'
          meta:
            description: "Validate that each activated user has a payment method. If fails, there is an issu
            owner: '@Jonh Smith'
            channel: 'application_team_data_alerts'
```

<filename>.yml hosted with ❤️ by GitHub view raw

Conclusion

This guide covered popular data tests spread over dbt packages, such as dbt-core, dbt-utils, dbt-expectations, and elementary. Each package has a different approach to testing. dbt-core provides only basic schema and source freshness tests and allows you to write custom tests. dbt-utils contains useful dbt macros that can be used as standalone tests or to streamline the process of writing custom tests. While dbt-expectations contains an extensive collection of individual tests that require hardcoding expectations, Elementary takes a different approach to dbt testing, where expectations are learned from your past data.

Adding tests is the first challenge to increasing confidence in your data and code; you should also run the tests as part of your workflows and consume test results. I like to think that data tests can be an efficient method to have a recurrent conversation with your data and occasionally won't replace

Stop firefighting.
It's Elementary

Start a free trial

Schedule a call

Y Combinator

Product

Live demo

Book a demo

Pricing

Elementary Cloud

Elementary Open Source

Anomaly detection

dbt package

Resources

Documentation

Blog

Slack community

Product updates

Company

Handbook

Contact us

Careers

Star 1,770