**TEST COVERAGE:**

1. **Test on One Database Object**

What/Why

* Assert something about the data that you think is true
* Contents of the data
* Constraints of the table
* -The grain of the table

**Example Tests**

**Generic Testing:**

* Unique
* not null
* accepted values

**Other package's tests like:**

* dht expectations.expect colum proportion of unique values\_to\_\_\_be between

**How it is performed or written:**

* Generic Tests are defined in YML file format in any folder (models or raw)
* Kind: Not null, unique, relationship, accepted values
* Tests acts on: Columns in the raw data or in models *(many at once too)*
* To run test: dbt test – dbt build (all or specific models)
* When: Can be done anytime while development or during production time plus can be added into the production jobs to run in an orchestration/schedule.
* Example: test\_order\_payments.yml

1. **Test How One Database Object Refers to Another Database Object:**

**What/Why**

* Compare values in one model to a source of truth in another model
* Ensure data has neither been erroneously added or removed

**Example Tests**

* relationships

**Other package's tests like:**

1. dat utils equality
2. dbt expectations.expect\_table row count to equal other table
3. **Test Something Unique About Your Data**

**What/Why**

* Tests usually involve some business logic, edge case, or rare event
* **Example Tests**

**Singular Testing** *(Write your sql script according to the conditions you want and test it against the data. It is also called as Custom Testing)*

* Orders should have payments >= 0
* Billing Total should equal the sum of all parts: anbrotal + tax credits - Total

**How it is performed or written:**

* Singular/Single/Custom Tests are defined in .sql file format in a tests folder which references the specific model against which it is written.
* Kind: A custom sql script as a test against specific models.
* Tests acts on: Any models or fields which are specifically referenced.
* To run test: dbt test – dbt build (all or specific models)
* When: Can be done anytime while development or during production time plus can be added into the production jobs to run in an orchestration/schedule.
* Example: tests/test\_order\_payments.sql

1. **Test the Freshness of Your Raw Source Data**

**What/Why**

* See if your loading tool has added raw data to your source table in the last **X** hours.
* Get notified if your underlying raw source data is not up to date Consider as the first step in your job to prevent models from running if data is Delayed.
* Search for the fresh data added into the source data if found then testing against them are to be proceeded.
* Freshness means when was the last time you updated your data

**How it is performed or written:**

* It only works on raw data and finds whether the new data is added into it or not
* Kind: YAML file for checking the newly arrived data and new table is created with a column added loaded\_at which refers the time at which new data is loaded.
* Tests acts on: Any of the source file
* To run test: dbt source freshness
* When: Can be done during production time
* Example: raw.raw\_schema.chicagocrimedata.sql -- source
* raw.raw\_schema.chicagocrimedata\_updated.sql – new table with a column of time

**Example Tests**

* Freshness tests
* -run by dbt source Freshness *command*

1. **Temporary Testing While Refactoring**

**What/Why**

* Create confidence
* Efficiently refactoring
* Auditing your changes while in development

**Example Tests**

* audit\_helper package to compare your new refactored model to your existing legacy model

**Test on the project as a whole:**

**How it is performed or written:**

* Test on the project as a whole: The configuration is set in the dbt\_project.yml which implies it all over the project and globally.
* Kind:
* Tests acts on: All of the yml files stored in the folders/models are run and tested along with the test configurations set in the dbt\_project.yml
* To run test: dbt run-operation, github action
* When: development adhoc , during continuous integration checks
* Example: all of the yml files and tests in the dbt\_peoject\_yml

**Run Tests:**

**Run all tests**

Dbt test – run all tests

**Run test for specific model**

Dbt test –select one\_specific\_model

Dbt test –select chicagocrimearrest

**Run tests for all models in a sub folder (example/staging)**

Dbt test –select example.staging.\*

**Run tests for all the models in package**

Dbt test –select some\_package.\*

**Run only tests defined singularly**

Dbt test –select test\_type:singular

**Run only tests defined generically**

Dbt test –select test\_type:generic

**Run singular tests limited to one\_specific\_model**

Dbt test –select one\_specific\_model, test\_type:singular

Dbt test –select chicagocrimelocation, test\_type:singular

**Run Generic tests limited to one\_specific\_model**

Dbt test –select one\_specific\_model, test\_type:generic

Dbt test –select chicagocrimelocation, test\_type: generic

**Run only source tests**

Dbt test –select source:\*

**Run only tests on your dbt models** (it will exclude all the sources and select everything except source as it is explicitly excluded from it.)

Dbt test –exclude source:\*

**Run only source tests for a particular source and all its tables** (it might be a folder or anything that contains all of its tables)

Dbt test –select source:jaffle\_shop

**Run only source tests for a particular source and only one tables**

Dbt test –select source:jaffle\_shop.orders

**Store failures of your tests for easier debugging** *(by putting the name of specific table having some error in testing, this command will show us the exact point where the issue is)*

dbt test –store-failures

**Testing Commands:**

1. Good:

**Dbt run**: first run all the models and then,

**Dbt test:** test all the models

1. Better:

**Dbt test –s source:\*** -- first test all the sources

**Dbt run** – now, run the models and sources   
**dbt test –exclude source:\*** -- in the end test all the models or anything except the one which was tested earlier i.e., source

1. Best:

**Dbt build –fail-fast:** The build command will first run and test the sources and models as soon as they become available plus it will do seeds and snapshots as they come in their way along other resources. It starts from testing and running sources to models and snaps and seeds coming into its way are processed. It does it from start to end automatically.

*There are several ways to run tests against your data in dbt. In development, it likely will make more sense to write a model and materialize it with `dbt run` and then test your models with `dbt test`.  However, once you are materializing and testing multiple models at once, we highly recommend running `dbt build` to materialize and test all nodes in DAG order.*

**Singular Tests:**

Singular Tests are built when you want to test an assumption about only one particular model. The test should return any rows that fail the assumption. All you need is a SELECT statement in a new .sql file stored in the /tests/folder

*It is a custom test for only one table/model and cannot be applied to other if the nature of other model/table is different.*

Example:

This singular test tests the assumption that the amount column of the orders model is always greater than 5

SELECT AMOUNT FROM {{REF (‘ORDER’)}} WHERE AMOUNT <=5

**Custom Generic Tests:**

Custom generic tests are built when you want to test the same assumption on multiple models. We can build a macro-like-test, with input parameters, and apply it to any relevant models

We will have to use jinja test tag In which test will have to be written, test name, parameters, and your SELECT statement in a new .sql file stored in the /tests/generic folder

Like we want every model to be applied same test according to our own conditions, so here comes custom generic tests

It will convert singular test into generic test and applied all over models.

**Singular test:**

select \* from {{ ref("dim\_chicagocrimedata") }} where arrest = 'TRUE'

**Custom generic test:**

{% test arrestments\_true(model, column\_name) %}

select {{ columm\_name }}

from {{ model }}

where {{ columm\_name }} = 'TRUE'

{% endtest %}

These kind of parameters are used to make it usable generally and that’s why it is not specified to any column or model because this job is already done by singular tests (specified table and model)

This kind of code is written in tests/generic/ arrestments\_true.sql – and then in the schema.yml where all of the generic tests are performed we have to put the name of this test in there.

All we have to do is to give the conditoin

- name: dim\_chicagocrimedata

      columns:

        - name: id

          tests:

            - unique

            - not\_null

        - name: arrest

            - arrestments\_true

And run the test against that folder. **dbt test --select dim\_chicagocrimedata**

**Overwriting native tests**

We can overwrite native tests by creating a generic test with the exact same name as the native test (unique, not\_null, relationships, acccepted\_values). You can put your new version in the tests/generic/subdirectory, and rebuild the test block with your own SQL. dbt will then use your specified test rather than the native version. Note: Before you run off and customize the native tests, be sure to learn about test configurations later in the course!

https://docs.getdbt.com/guides/best-practices/writing-custom-generic-tests

DBT Audit helper:

This package is used to compare the old and new tables inorder to check if they match or not and many other operations

It is generally used to check the values of the new model if it matches to the older one (correct one) if changes are made in the new table.

particularly useful when you want to check that a refactored model, or a model that you are moving over from a legacy system, match up.

* compare\_relations
* compare\_queries
* compare\_column\_values
* compare\_relation\_columns
* compare\_all\_columns
* compare\_column\_values\_verbose

Test Configuration:

Severity Test:

1. **For generic tests**.

It mainly applies on generic tests when in schema.yml file.

- name: chicagocrime\_arrestments

      description: "configuration for severity for generic testing can done by this way - it will show error when there are non unique values grater than 50"

      columns:

        - name: ARRESTMENT\_ID

          tests:

            - unique:

                config:

                  severity: error

                  error\_if: '>10'

this will check if there is any error and if condition will be executed. The condition becomes true then this will see that errors must be greater than 10 if they are, error will pop up and it wont do anything if the error is below 10 which will pop up if this condition is not applied.

models:

    - name: chicagocrime\_details

      description: "Generic testing - severity check is used to give us only warning when null values are found more than 10 - else it would have stopped testing and thrown error"

      columns:

          - name: id

            description: "The primary key for this table"

            tests:

                - unique

                - not\_null:

                    config:

                      severity: warn

                      warn\_if: ">10"

the same as above but it will only give warning and the process will not stop as it does when any test is failed. It will move forward.

1. **For Singular tests:**

{{

    config(

        severity = 'warn'

    )

}}

select arrest

from {{ ref('dim\_chicagocrimedata') }}

where arrest = 'FALSE'

we only have to put this configuration in the singular testing file where there are singular tests to be carried out

Where Test:

1. **For generic tests**.

    - name: dim\_chicagocrimedata

      description: "Check the unique values in arrest column when the date starts from 2010-12-30"

      columns:

        - name: ARREST

          tests:

            - unique:

                config:

                  where: "date > '2010-12-30'"

#            - arrestments\_true

# This is the Singular converted into generic test and arrestments\_true is used as a macro and its roots lies in test/generic

Where defines the condition that I want to test the uniqueness of the column after this date.

store failures for config test in database:

1. **For generic tests**.

    - name: dim\_chicagocrimedata

      description: "Check the unique values in arrest column when the date starts from 2010-12-30"

      columns:

        - name: ARREST

          tests:

            - unique:

                config:

                  where: "date > '2010-12-30'"

                  store\_failures: true

                  schema: "DBT\_PRACTISE"

Now this will tell us the exact line of code with which we can reach out to the data that is causing failures and the data that is causing failure will be stored in database. We can specify the schema where the table will be created of that data.

with crime\_count\_description AS

(

    select crime\_description, count(crime\_description) as crime\_count

    from {{ ref('dim\_chicagocrimedata') }} group by crime\_description

)

select crime\_description, crime\_count

from crime\_count\_description

where crime\_count > 1

{% test c\_generic\_crimelocation\_for\_multiplecrimes(

    model, columm\_name, group\_by\_column

) %}

    select {{ column\_name }}, (count{{ column\_name }}) as crime\_count

    from {{ model }}

    group by 1

    having crime\_count > 1

{% endtest %}