workspace overview

projects

WHAT IS A PROJECT?

Projects are the highest-level directory for the files and code in dbt. There are different approaches to thinking about project structure - some insist on having one project for all models, and others advocate for having different projects for different layers of your transformation.

WHAT'S IN IT?

At a minimum, a project only needs the `dbt_project.yml` configuration file.

WHAT'S IN DBT_PROJECT.YML?

This **configuration** file defines the directory of the project and optional project configurations such as:

- project name, project version, dbt version required, database credentials, directories for models, seeds, tests, analyses, macros, snapshots, docs, and project-level variables.

dbt_project.yml

```
name: string

config-version: 2
version: version

profile: profilename

model-paths: [directorypath]
seed-paths: [directorypath]
test-paths: [directorypath]
analysis-paths: [directorypath]
macro-paths: [directorypath]
snapshot-paths: [directorypath]
docs-paths: [directorypath]
asset-paths: [directorypath]

target-path: directorypath
log-path: directorypath
packages-install-path: directorypath

clean-targets: [directorypath]

query-comment: string

require-dbt-version: version-range | [version-range]

quoting:
  database: true | false
  schema: true | false
  identifier: true | false

models:
  <model-configs>

seeds:
  <seed-configs>

snapshots:
  <snapshot-configs>

sources:
  <source-configs>

tests:
  <test-configs>

vars:
  <variables>

on-run-start: sql-statement | [sql-statement]
on-run-end: sql-statement | [sql-statement]

dispatch:
  - macro_namespace: packagename
    search_order: [packagename]
```

Reference: About dbt projects, DBT documentation

tags

WHAT IS A TAG?

You can apply a tag, or list of tags, to any resource. These tags help you organize your project and they can be used as part of the resource-selection syntax.

HOW SHOULD I USE TAGS?

There are many ways you can use tags, but the most common ways are:

- use tags to signal the schedule for a model (e.g. hourly, daily)
- use tags for the status of a model (e.g. published)
- use tags for areas of the business (e.g. finance, marketing)
- use tags to help identify tables and columns with sensitive data (e.g. contains_pii)

WHERE DO I ADD TAGS?

You can configure tags in the dbt_project.yml file, or in a config block.

dbt_project.yml

```
models:
  <resource-path>:
    +tags: <string> | [<string>]

snapshots:
  <resource-path>:
    +tags: <string> | [<string>]

seeds:
  <resource-path>:
    +tags: <string> | [<string>]
```

models/staging/stg_payments.sql

```
{{ config(
    tags=["finance"]
) }}

select ...
```

tags best practice

If you are using many tags, it can be a good idea to have a Valid Tags CSV to document which tags are used, and to validate against all merge requests to make sure no new tags are being used.

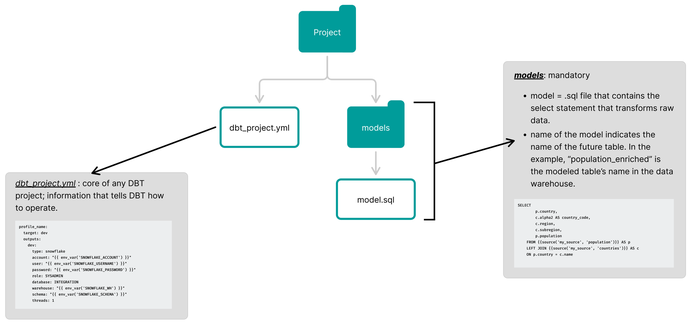
Adding a tag for any models that are used in analytics/reporting (e.g. TDF - Trusted Data Framework) is a good way to ensure the proper tests and validation are done on key tables.

Reference: GitLab Data Team Handbook

project hierarchy & contents

EXAMPLE PROJECT 1: A SIMPLE PROJECT

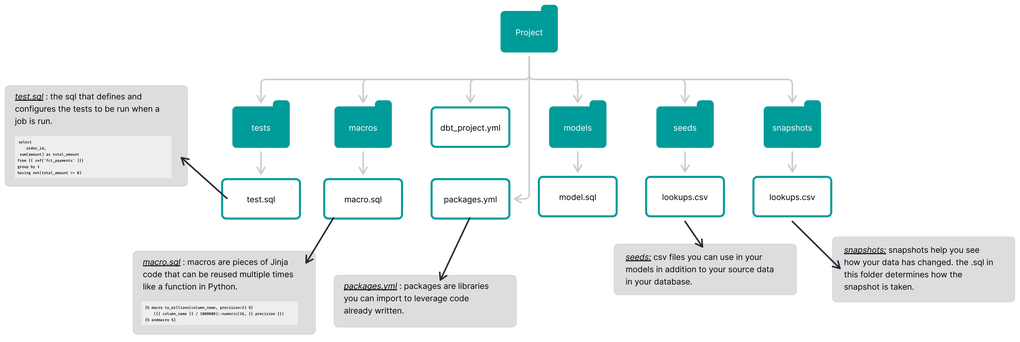
The simplest project you can have is this:



Reference: DBT 101 by Maria Eugenio

EXAMPLE PROJECT 2: A SLIGHTLY MORE COMPLICATED PROJECT

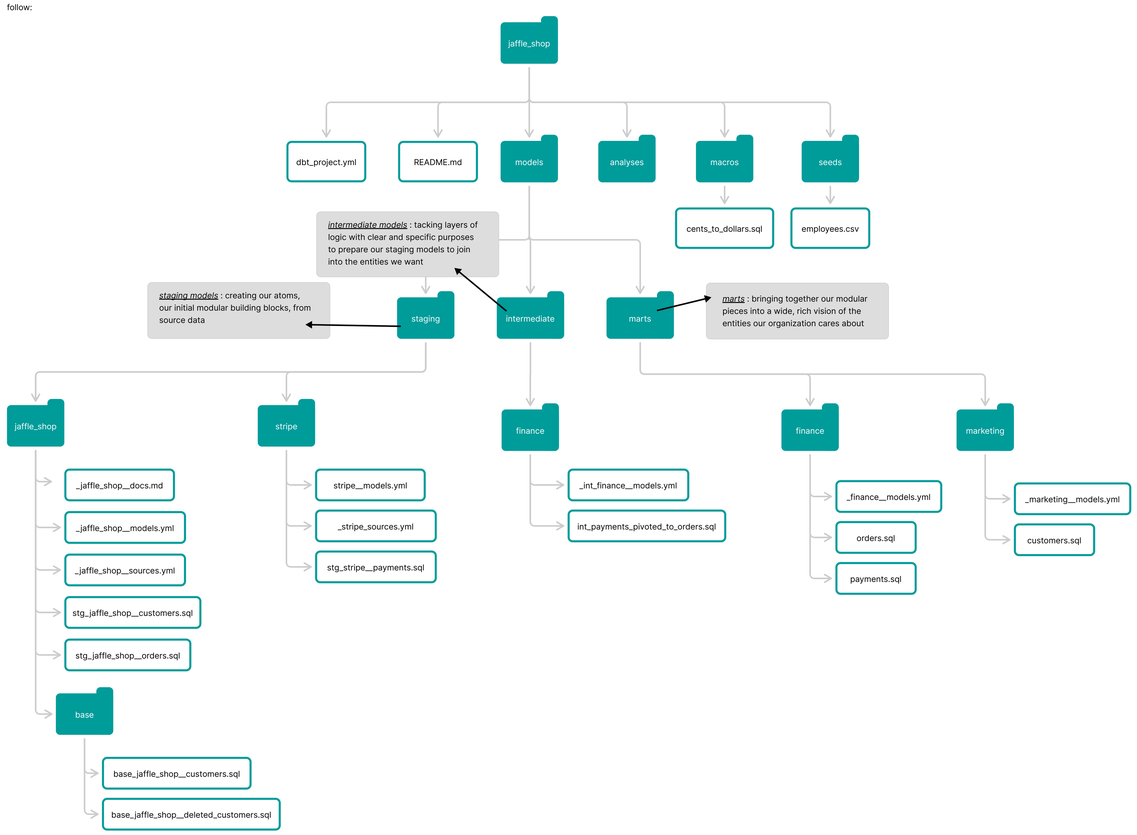
When you want to add more data sources (seeds), snapshots to better track changes to your source data, and macros and packages to re-use code, your project might look like this:



Reference: DBT 101 by Maria Eugenio

EXAMPLE PROJECT 3: BEST PRACTICE ORGANIZATION

According to dbt best practices, this is a good project hierarchy to follow:



Reference: How we structure our dbt projects, dbt docs

project, naming & organization best practice

[1] Have three (primary) types of models: staging, intermediate, and marts

1. **Staging** — creating our atoms, our initial modular building blocks, from source data
2. **Intermediate** — stacking layers of logic with clear and specific purposes to prepare our staging models to join into the entities we want
3. **Marts** — bringing together our modular pieces into a wide, rich vision of the entities our organization cares about

[2] Create a consistent pattern of file naming.

File names must be unique and correspond to the name of the model when selected and created in the warehouse. We recommend putting as much clear information into the name as possible, including a prefix for the layer the model exists in, important grouping information, and specific information about the entity or transformation in the model.

- ✓ Always go plural (`stg_stripe__invoices.sql`)
- ✓ All models should use the naming convention: `<ctype/dag_stage>_<source/topic>__<additional_context>`
- ✓ Schema, table and column names should be in `snake_case`
- ✓ Limit abbreviations that are related to domain knowledge (e.g. `current_order_status >> current_os`)
- ✓ Use business terminology (not source terminology)

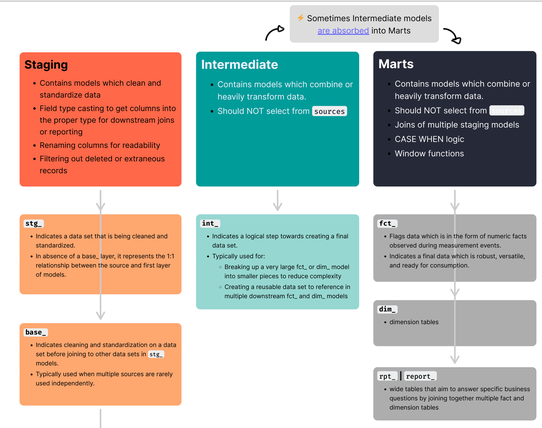
[3] Create a consistent pattern of column naming.

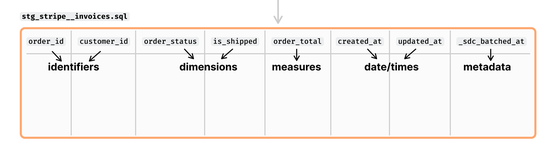
- ✓ Each model should have a primary key that can identify the unique row, and should be named `<object>_id`
- ✓ If a surrogate key is created, it should be named `<object>_sk`
- ✓ For base and staging models, columns should be ordered in categories: identifiers -> dimensions -> measures ->date/time fields -> metadata.
- ✓ Columns shuould be named according to these conventions:
  - Timestamps: `<event>_at` Format: UTC Example: `created_at`
  - Dates: `<event>_date` Format: Date Example: `created_date`
  - Booleans should be prefixed with `is_` or `has_`
  - Price/revenue fields should be in decimal currency (e.g. `19.99` for $19.99; many app databases store prices as integers in cents). If non-decimal currency is used, indicate this with suffix, e.g. `price_in_cents`.
  - Avoid using reserved words (such as these for Snowflake) as column names.
  - Consistency is key! Use the same field names across models where possible.

[4] Use Common Table Expressions

A Common Table Expression (CTE) is a temporary result that can be used in a SQL query. You can use CTEs to break up complex queries into simpler blocks of code that can connect and build on each other.

- ✓ CTEs should perform a single, logical unit of work
- ✓ CTE names should be as verbose as needed to convey what they do
- ✓ CTEs with confusing or notebale logic should be commented w/ SQL comments as you would any complex fns and should be located above the CTE
- ✓ CTEs that are duplicated across models should be pulled out and created as their own models
- ✓ CTEs fall into two categories:
  - import: used to bring data into a model. Kept relatively simple and refain from complex operations such as joins and column transformations
  - logical: used to perform a logical step w/ the data that is brought into the model toward the end results
- ✓ Always end w/ a simple select statement






```
-- Jaffle shop went international!
with

-- Import CTEs
regions as (
    select * from {{ ref('stg_jaffle_shop__regions') }}
),

nations as (
    select * from {{ ref('stg_jaffle_shop__nations') }}
),

suppliers as (
    select * from {{ ref('stg_jaffle_shop__suppliers') }}
),

-- Logical CTEs
locations as (
    select
        {{ dbt_utils.generate_surrogate_key([
            'regions.region_id',
            'nations.nation_id'
        ]) }} as location_sk,
        regions.region_id,
        regions.region,
        regions.region_comment,
        nations.nation_id,
        nations.nation,
        nations.nation_comment
    from regions
    left join nations
        on regions.region_id = nations.region_id
),

final as (
    select
        suppliers.supplier_id,
        suppliers.location_id,
        locations.region_id,
        locations.nation_id,
        suppliers.supplier_name,
        suppliers.supplier_address,
        suppliers.phone_number,
        locations.region,
        locations.region_comment,
        locations.nation,
        locations.nation_comment,
        suppliers.account_balance
    from suppliers
    inner join locations
        on suppliers.location_id = locations.location_sk
)

-- Simple select statement
select * from final
```

with statement

import CTEs

logical CTEs

simple select statement

References: Data modeling techniques for more modularity, How we structure our dbt projects, dbt docs, and dbt Style Guide

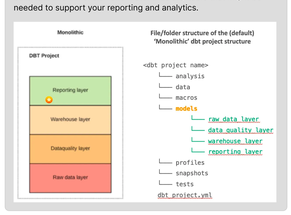
one project or many?

HOW TO MANAGE YOUR ORG?

This is a nuanced question, and really depends on your org and your specific situation. The following guide follows the advice from Stefano Solimito. There will be more support for multi-project management in future releases of dbt.

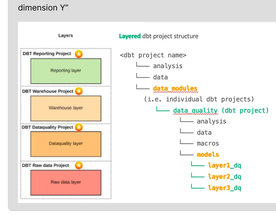
Monolithic

A single dbt project for your entire platform. It will clean your data, build fact and dimension and materialize any table that might be needed to support your reporting and analytics.



"Micro-services"

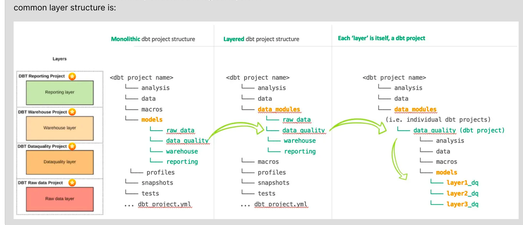
Multiple small projects, each with a single purpose. For instance "clean data source X and store it in data quality layer" or "Build dimension Y"



For more info on the repository configuration question - check out this discussion from the dbt discourse channel

Layers

Your data platform will be normally structured in logical layers. A common layer structure is:



Layers and verticals

This approach is similar to Layers approach but after dataquality one dbt project is built with all facts and dimensions relevant for the entire organisation and some vertical-specific projects are created to serve specific areas of the organisation.

Reference: Practical tips to get the best out of Data Build Tool (dbt) - Part 1 by Stefano Solimito, and How to Structure your DBT Project by Paul Fry

data & sources

sources

WHAT IS A SOURCE?

As the name suggests, sources in dbt are references to the data loaded into your warehouse that you want to use in dbt.

HOW TO CREATE A SOURCE?

Sources are defined in the model .yml file under a sources key.

EXAMPLE

models/<filename>.yml

```yaml
version: 2

sources:
  - name: jaffle_shop
    database: raw
    schema: jaffle_shop
    tables:
      - name: orders
      - name: customers

  - name: stripe
    tables:
      - name: payments
```

SELECTING FROM A SOURCE

When you want to reference a source in your model, you can use the `{{source()}}` jinja function:

models/orders.sql

```
select
  ...

from {{ source('jaffle_shop', 'orders') }}

left join {{ source('jaffle_shop', 'customers') }} using (customer_id)
```

schema table

jaffle_shop.orders

Reference: Add sources to your DAG, DBT documentation

sources best practices

If you're using Snowflake, you'll want to make note of the `quoting` property, which will automatically quote your sources when compiling the SQL

You can configure your source freshness, and test that freshness when compiling and running a dbt model. This is a helpful way to make sure your source is up to date when running a model

seeds

WHAT IS A SEED?

Seeds are CSV files in your dbt project (typically in your `seeds` directory), that dbt can load into your data warehouse using the `dbt seed` command.

HOW CAN THEY BE USED?

Seeds can be referenced in downstream models the same way you do with models (`ref` function).

models/orders.sql

```
-- This refers to the table created from seeds/country_codes.csv
select * from {{ ref('country_codes') }}
```

WHEN SHOULD THEY BE USED?

Seeds act as great lookup tables, like mapping country codes to country names, or excluding a list of user ids from a model. They are not recommeended for any kind of production or raw data, especially anything with PII (Personal Identifiable Information) data.

SEED CONFIGURATION ↗

You can configure your seeds in your `dbt_project.yml` file. From here you can specify things such as the schema of your file, the column types.

dbt_project.yml

```yaml
seeds:
  <resource-path>:
    +enabled: true | false
    +tags: <string> | [<string>]
    +pre-hook: <sql-statement> | [<sql-statement>]
    +post-hook: <sql-statement> | [<sql-statement>]
    +database: <string>
    +schema: <string>
    +alias: <string>
    +persist_docs: <dict>
    +full_refresh: <boolean>
    +meta: {<dictionary>}
    +grants: {<dictionary>}
```

Reference: Add Seeds to your DAG, DBT documentation

snapshots

WHAT IS A SNAPSHOT?

Snapshots help you see how your source data has changed over time. When a model is run, a snapshot is taken and a table is created in the specified `target_schema`. They are always defined by `SELECT` statements in a `.sql` file within the `snapshots` directory.

CONFIGURING A SNAPSHOT

There are many options when configuring your snapshot, including:

- `target_database`: where dbt will render the snapshot
- `target_schema`: the target schema of the snapshot table
- `strategy`: the detection strategy to use, either `timestamp` or `check`
- `unique_que`: a primary key column or expression for the record
- `check_cols` (req'd for strategy = `check` only): the columns to check
- `updated_at` (req'd for strategy = `timestamp` only): the timestamp column to compare
- `invalidate_hard_deletes`: find hard deleted records in source and set dbt_valie_to current time if no longer exists

You can see the full list of configurations available here.

snapshots/orders_snapshot.sql

```
{% snapshot orders_snapshot %}

{{
    config(
      target_database='analytics',
      target_schema='snapshots',
      unique_key='id',

      strategy='timestamp',
      updated_at='updated_at',
    )
}}

select * from {{ source('jaffle_shop', 'orders') }}

{% endsnapshot %}
```

SNAPSHOT STRATEGIES

There are two strategies available when configuring a snapshot: timestamp and check. dbt recommends using `timestamp` over `check`.

Timestamp strategy uses a timestamp column (e.g. updated_at) and checks if that column has a new value for each specified primary key. If it does, then dbt will invalidate the old row, but record the time range in which it was valid.

id	status	updated_at	dbt_valid_from	dbt_valid_to
1	pending	2019-01-01	2019-01-01	2019-01-02
1	shipped	2019-01-02	2019-01-02	`null`

Check strategy works when you don't have a reliable timestamp column like updated_at. It works by comparing a list of columns between their current and historical values. If any of these columns have changed, then dbt will invalidate the old record and record the new one.

SNAPSHOT METAFIELDS

Snapshot tables will be saved as a clone, plus the following additional meta-fields:

- dbt_valid_from: timestamp when this row was first inserted
- dbt_valid_to: timestamp when this row became invalidated
- dbt_scd_id: unique key generated for each snapshot record
- dbt_updated_at: updated_at timestamp of the source record when the row was inserted

snapshot best practices

Snapshot source data

Use the `source` function in your query

Include as many columns as possible

Avoid joins in your snapshot query

Limit the amount of transformation in your query

GitLab's Data Team Handbook has some great resources on Snapshots (and many other things).

The snapshot material covers:

- GDPR concerns w/ historical data
- building models over snapshot data
- recreating historical builds w/ snapshots

Anonymous

models

models

WHAT IS A MODEL?

This is the core of dbt. This is where your transformation logic lives. They are primarily written in SELECT statements saved in .sql files. Complexity will vary sigificantly. You can also write models using Python.

KEY THINGS TO KNOW ABOUT MODELS:

1. Each SQL model is only a single SELECT statement (but may have many CTEs)
2. You can reference other models, sources, or macros in a model
3. You can combine your SQL with Jinja to reference other entities and to do things you cannot normally do in SQL
4. When you run your model, your code will be compiled into database SQL
5. Depending on your materalization, your model output can be saved as a view, table, or just used as ephemeral logic for use in other models.
6. If saved as a view or table, dbt will save the model in the schema you specify
7. Your model name will be the name of the view/table

models/customers.sql

```
with customer_orders as (
    select
        customer_id,
        min(order_date) as first_order_date,
        max(order_date) as most_recent_order_date,
        count(order_id) as number_of_orders

    from jaffle_shop.orders

    group by 1
)

select
    customers.customer_id,
    customers.first_name,
    customers.last_name,
    customer_orders.first_order_date,
    customer_orders.most_recent_order_date,
    coalesce(customer_orders.number_of_orders, 0) as number_of_orders

from jaffle_shop.customers

left join customer_orders using (customer_id)
```

CONFIGURING MODELS

Model settings can be set in your `dbt_project.yml` file and in your model file using a `config` block.

You can configure a model's materialization (e.g. view or table), the output schema, and to apply tags. You can see the full list of config options here.

dbt_project.yml

```
name: jaffle_shop
config-version: 2
...

models:
  jaffle_shop: # this matches the `name:` config
    +materialized: view # this applies to all models in the current project
    marts:
      +materialized: table # this applies to all models in the `marts/` directory
      marketing:
        +schema: marketing # this applies to all models in the `marts/marketing/` directory
```

models/customers.sql

```
{{ config(
    materialized="view",
    schema="marketing"
) }}

with customer_orders as ...
```

There are also database-specific configuration options available like partioning and clustering for BigQuery. See the full list here.

REFERENCE OTHER MODELS

You can use the ref function to reference other models.

models/customers.sql

```
with customers as (

    select * from {{ ref('stg_customers') }}

),

orders as (

    select * from {{ ref('stg_orders') }}

),

...
```

materalizations

WHAT IS A MATERALIZATION?

This is how you want to save your model in a warehouse. The options are:

view

- the default option
- your model is rebuilt as a vew on each run
- 👍 no add'l data is stored
- 👎 can be slow, esp if a view depends on other views
- 🟢 best for models w/out significant transformation

table

- your model rebilds a table each time its run
- 👍 fast to query
- 👎 can be slow to create
- 🟢 best for data that is to be access in a BI tool, or intermediate models used in many downstream models

incremental

- your model inserts or updates records into a table
- 👍 faster build time by dealing only w/ new records
- 👎 extra config required
- 🟢 best for event-based data

Learn more about incremental models here.

ephemeral

- models not directly built into the database, rather just as a CTE
- 👍 reduce db clutter and storage
- 👎 cannot select directly from model, hard to debug
- 🟢 very light transformations early in the DAG that don't need to be queried directly

HOW TO SPECIFY A MATERALIZATION

You can specify a materalization option in the `materalization` parameter in the `dbt_project.yml` file, or directly in your model sql files.

models/events/stg_event_log.sql

```
{{ config(materialized='table', sort='timestamp', dist='user_id') }}

select *
from ...
```

jinja & macros

WHAT IS A JINJA?

Jinja is a way to template SQL. It can help achieve functionality not commonly found in SQL, and can help modularize and re-use code. There are certain expressions that are specific to dbt.

HOW DOES IT WORK?

At runtime, dbt will compile your Jinja syntax into usable SQL and run against the database.

/models/order_payment_method_amounts.sql

```
{% set payment_methods = ["bank_transfer", "credit_card", "gift_card"] %}   statement

select
    order_id,
    {% for payment_method in payment_methods %}
    sum(case when payment_method = '{{payment_method}}' then amount end) as {{payment_method}}_amount,
    {% endfor %}
    sum(amount) as total_amount
from app_data.payments
group by 1
```

expression

/models/order_payment_method_amounts.sql

compiled SQL

```
select
    order_id,
    sum(case when payment_method = 'bank_transfer' then amount end) as bank_transfer_amount,
    sum(case when payment_method = 'credit_card' then amount end) as credit_card_amount,
    sum(case when payment_method = 'gift_card' then amount end) as gift_card_amount,
    sum(amount) as total_amount
from app_data.payments
group by 1
```

WHAT IS A MACRO?

Macros are pieces of Jinja code that can be reused multiple times. They are like functions in Python (and other languages). They are specified in `.sql` files in your `macros` directory. You can also use macros from projects.

HOW TO DEFINE A MACRO?

macros/cents_to_dollars.sql

```
{% macro cents_to_dollars(column_name, precision=2) %}
    ({{ column_name }} / 100)::numeric(16, {{ precision }})
{% endmacro %}
```

HOW TO CALL A MACRO?

models/stg_payments.sql

```
select
  id as payment_id,
  {{ cents_to_dollars('amount') }} as amount_usd,
  ...
from app_data.payments
```

```
select
  field_1,
  field_2,
  field_3,
  field_4,
  field_5,
  count(*)
from my_table
{{ dbt_utils.group_by(5) }}
```

call from a package

HOW DOES IT WORK?

target/compiled/models/stg_payments.sql

```
select
  id as payment_id,
  (amount / 100)::numeric(16, 2) as amount_usd,
  ...
from app_data.payments
```

compiled SQL

packages

WHAT ARE PACKAGES?

Packages are modularized code that can be imported and used by anyone. Packages can be created and shared by anyone. dbt *packages* are in fact standalone dbt projects, with models and macros that tackle a specific problem area.

WHEN ARE THEY USEFUL?

Packages are useful in many scenarios, but in particular in helping you save time by not solving a problem someone else has already solved, e.g. transforming data from a consistently structured SaaS dataset.

HOW DO I ADD A PACKAGE?

You can add a package via your `package.yml` file in your dbt project. Just spcify the package you wish to add using one of the supported syntaxes.

packages.yml

```
packages:
  - package: dbt-labs/snowplow
    version: 0.7.0

  - git: "https://github.com/dbt-labs/dbt-utils.git"
    revision: 0.9.2

  - local: /opt/dbt/redshift
```

WHERE CAN I FIND PACKAGES?

- dbt Hub is a registry for dbt packages.
- many packages are stored on a Git server (e.g. github)
- you can also create your own private and local packages

HOW DO I MAKE MY OWN PACKAGE?

Check out this step-by-step guide: How to Create a dbt Package by Marie Truong

deployment

running your models

HOW TO RUN A MODEL?

Once we have our model (with at least one select statement) and our dbt_projects.yml file completed, we can run our model.

For anyone using dbt core, you can use your terminal to run your code. Use the command `dbt run`.

If you're using dbt Cloud, you can enter `dbt run` in the command prompt at the bottom of the screen to run the model.

code management & version control

WHY DO I NEED TO MANAGE MY CODE?

Version control is a fundamental compontent to using dbt. A version control system allows multiple people to work on the same code without conficts. Furthermore, it stores versions of historical code so you can roll back changes if needed.

CODE MANAGEMENT IN DBT

dbt does not come with its own version control system. For **dbt Cloud** users, you can leverage Git through GitHub or GitLab. If you are using the CLI, you will need to find your own code managment solution, but thost two are the most popular solutions.

environments

WHAT DO ENVIRONMENTS REFER TO?

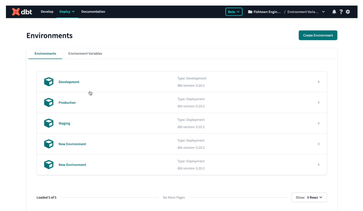
In software development, it's common to have different environments in which to develop and test code without impacting users. In analytics engineering, the idea is the same.

PRODUCTION VS DEVELEOPMENT

Most commonly, teams will have a production (prod) environment which has the data used in BI tools, live reports, etc. To test code before it goes to prod, teams often have a development (dev) environment.

HOW TO SPECIFY ENVIRONMENTS IN DBT?

If you're using dbt Cloud, there is an **Environments** tab in which all environments and their settings are stored. See the full instructions here ↗.



If you are using the dbt Core, you can specify your environments and variables within your profile. There will be a targed named dev, and by setting this to default all deployments will go to `dev` by default. And when you're happy with the results, you can deploy to prod.

It is recommended to use different schemas within the one database to separate your dev and prod environments.

deploying your jobs

WHAT DOES DEPLOYMENT MEAN?

Instead of manually executing commands whenever we want to execute one of our models, we can schedule these runs (known as jobs) to run at certain times.

WHAT IS A JOB?

A collection of run steps, settings, and a trigger to invoke dbt commands against a project in the user's cloud data platform.

HOW DO I DEPLOY MY JOBS?

There are many options here. If using dbt Cloud then you can use the in-app scheduler (and many other features) to run, manage, and get notifications about your jobs.

If you use the dbt CLI, you will need to make use of another tool such as Airflow, Prefect, Dagster, automation server, Cron, or ADF. You can choose to use these options to run jobs josted on dbt Cloud as well.

dbt Cloud scheduler

WHAT IS THE DBT CLOUD SCHEDULER?

The Scheduler is the backbone of running jobs in dbt Cloud, bringing power and simplicity to building data pipelines in both continuous integration and production contexts. The Scheduler frees teams from having to build and maintain their own infrastructure and ensures the timeliness and reliability of data transformations.

WHAT KIND OF SCHEDULES CAN I SET?

Jobs can be run based on a large number of critera, including:

- a set schedule (e.g. every 3 hours)
- triggered by a pull request submitted to the dbt repo
- triggered by another job
- triggered by API
- triggered manually by someone clicking "Run Now"

WHAT KINDS OF COMMANDS CAN I SCHEDULE?

You can do more than run a dbt model using the scheduler (or other solutions). To see the full list of commands available you can go to the docs here, or check out the section below.

You can also **chain multiple commands together** for a single job.

CAN MY JOB FAIL?

You better belive it! the common job failure reasons are:

- **Failure at `dbt run`**: executes compiled SQL model against target database. Failure due to an error in any of the built models.
- **Failure at `dbt test`**: A test can pass, fail, or warn depending on its deverity. Only an error will stop the next step.
- **Failure at `dbt build`**: If an upstream resource fails, all downstream resources are skipped

HOW DO I SCHEDULE MY JOB?

In the dbt Cloud scheduler, you'll see a Triggers tab. From there you can specify when and how you want your jobs to run:

orchestration

WHAT IS ORCHESTRATION?

Orchestration simply refers to how you manage all of your data transformations. When your dbt set-up is sufficently complex, you may need to invest more time in this.

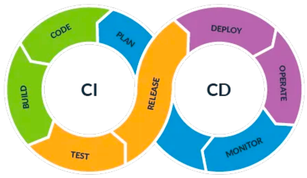
WHAT CAN I USE FOR DBT ORCHESTRATION?

dbtCloud has put in many features to help with orchestration including dbt Cloud scheduler, but in some cases you may want to use external tools like: Airflow, Prefect, Dagster, or Databricks workflows (many more options available).

CI/CD

WHAT DOES CI/CD MEAN?

CI/CD stands for Continuous Integration/ Continuous Deployment. This is a software engineering method of deploying applications that involves continuous monitoring and testing, and deployment via automation.



Reference: What is CI/CD, Kyle Farmer

data testing

WHAT ARE DBT TESTS?

Tests are the assertions you make about your models and other resources in your dbt project (e.g. sources, seeds and snapshots). When you run `dbt test`, dbt will tell you if each test in your project has passed or failed. They are a great way to:

- see if your code is working correcly
- identify problems in the source data
- prevent code regressions

WHAT KIND OF TESTS CAN I CREATE?

There are two ways of defining tests in dbt:

- a **singular test** is the simpliest. You can write a SQL query that returns failing rows, save the query in a `.sql` file in your `test` directory.
- a **generic test** is parameterized. These are defined in a special test block. Once defined you can reference the generic test in your `.yml` files. You can then define them on models, columns, sources, snapshots and seeds.

Singular tests

These are one-off SQL statements that **return failing records**. You can write the statement in your `tests` directory. They are easy to write but don't scale well.

tests/assert_total_payment_amount_is_positive.sql

```
-- Refunds have a negative amount, so the total amount should always be >= 0.
-- Therefore return records where this isn't true to make the test fail
select
    order_id,
    sum(amount) as total_amount
from {{ ref('fct_payments' )}}
group by 1
having not(total_amount >= 0)
```

Generic tests

Generic tests contain logic that can be reused over and over. They are defined in a `test` block similar to a `macro`. Once it is defined it can be added as a property on any existing model (or source, seed, or snapshot). These properties are added in the `.yml` files in the same directory as your resource.

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

    select *
    from {{ model }}
    where {{ column_name }} is null

{% endtest %}
```

BEST GENERIC TESTS AVAILABLE

Many generic tests are available to use either out-of-the-box in dbt or through packages. The following are popular generic tests to start using:

dbt out-of-the-box tests:

- `not_null`
- `unique`
- `accepted_values`
- `relationships`

dbt-utils tests (many more available):

- `equal_rowcount`
- `mutually_exclusive_ranges`
- `not_accepted_values`
- `recency`

dbt-expectations tests (many more available):

- `expect_column_to_exist`
- `expect_column_values_to_be_be tween`

CONFIGURING TEST SEVERITY

When you create a test, you an also configure what you want to happen when a test fails. The options are:

- If `severity: error`, dbt will check the `error_if` condition first. If the error condition is met, the test returns an error. If it's not met, dbt will then check the `warn_if` condition (defaulted to `!=0`). If it's not specified or the warn condition is met, the test warns; if it's not met, the test passes.
- If `severity: warn`, dbt will skip the `error_if` condition entirely and jump straight to the `warn_if` condition. If the warn condition is met, the test warns; if it's not met, the test passes.

models/<filename>.yml

```
version: 2

models:
  - name: large_table
    columns:
      - name: slightly_unreliable_column
        tests:
          - unique:
              config:
                severity: error
                error_if: ">1000"
                warn_if: ">10"
```

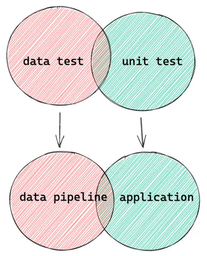
testing

WHAT DOES IT MEAN TO TEST MY MODEL??

Testing for dbt projects can be roughly bucketed into two categories:

- **Data testing**: Tests on top of dbt models built using production data, typically investigating data quality like uniqueness, nullability, and consistency.
- **Unit testing**: Tests evaluating model logic by running mock input data through the model and comparing it against expected output.

Both of these types of tests leverage dbt's built-in dbt testing framework, but only data testing is natively supported by dbt-core, for now.



Reference: How to do Unit Testing in dbt, Xiaoxu Gao

unit testing

WHAT IS UNIT TESTING?

Unit tests evaluate your logic, not your data. They are run by putting mock input data through your model and comparing against an expected output.

HOW DO I DO UNIT TESTING IN DBT?

Right now dbt doesn't support unit testing out of the box, but there are many packages out there that can help:

- josephmachado/unit_test_dbt
- EqualExperts/dbt-unit-testing
- dm03514/dbt-model-tests
- mjirv/dbt-datamocktool

CI/CD best practice & examples

Datafold's 7 dbt Testing Best Practices ↗

1. Start with generic tests
2. Leverage helpers and extensions
3. Model with testing in mind
4. Test data sources
5. Integrate alerting
6. Establish clear ownership of tests
7. Create testing documentation

How to unit test sql transforms in dbt ↗

Step-by-step tutorial on how to get started with unit testing in dbt.

A Beginner Guide to dbt Tests ↗

Excellent run through of the different kinds of tests with hands-on code and examples.

dbt at Super Part 2: Continuous Integration ↗

Great way to see how Super approached their entire CI/CD process, including balancing unit and data testing.

Test SQL Pipelines against Production Clones using DBT and Snowflake ↗

How Airtasker uses Snowflake cloning to test their models at runtime.

Environment-dependent Unit Testing in dbt ↗

Hifly Labs needed a unit testing solution that ran in a separate environment and had restricted test columns. Here's how they did it.

Writing Unit Tests for dbt - with TDD ↗

Part tutorial, part theory behind unit tests.

How do you test your data? ↗

Discussion in the dbt community around how different users tackle testing.

Continuous Integration and Automated Analytics Test Pipelines using dbt and dbtCloud ↗

Step-by-step guide on how and why to set-up CI-triggered jobs in dbt Cloud.

checking model performance

Retrieve Model Performance Statistics (see example here)

Check model Run Time

Improve model efficiency by reducing amount of data neeeded to be stored in memory.

- Reduce number of columns used
- Reduce number of rows used
- Check filter order
- Check join order
- Check for redundant joins

Check never full refresh viability

- Event stream data that does not change is a prime candidate for never full refreshing

Check for incremental viability

Not all models need to be incremental due to the increased design complexity.

- If the model loads new data regularly, but most of the data does not change it may be a good candidate for an incremental model.
- If the source data can be filtered in a way to return new or changed records, such as an `updated_date` field, it may be a good candidate for an incremental model.
- If the model represents the `current state` of data that changes it is `not` a good candidate for an incremental model.
- If the model is based on a snapshot model, it is `not` a good candidate for an incremental model.

Check Warehouse Size viability

Increasing the warehouse size will not always mean an increase in performance and an improvement in cost.

- If refactoring has not improved the Model Efficiency, an increase to the Warehouse Size may be warranted.
- If Model Efficiency is Good but the model is reaching the timeout limit for the warehouse then an increase to the Warehouse Size may be warranted.

References: GitBook Data Team Handbook & dbt observability 101: How to monitor dbt run and test results by Or Avidov

artifacts

WHAT ARE ARTIFACTS?

With every invocation, dbt generates and saves one or more *artifacts*. Several of these are JSON files (`manifest.json`, `catalog.json`, `run_results.json`, and `sources.json`) that are used to power:

- documentation
- state
- visualizing source freshness

They could also be used to:

- calculate project-level test coverage
- perform longitudinal analysis of run timing
- identify historical changes in table structure
- do much, much more

WHEN ARE THEY PRODUCED?

Most dbt commands produce artifacts.

WHAT KINDS OF ARTIFACS ARE PRODUCED?

The common artifacts produced are:

- manifest: produced by commands that read and understand your project
- run results: produced by commands that run, compile or catalog nodes in your DAG,
- catalog: produced by `docs generate`
- sources: produced by `source freshness`

WHAT ARE IN THE ARTIFACTS?

Manifest

This is a single file that contains a full representation of your dbt project's resources (models, tests, macros, etc). Even if you're only running some models or tests, all resources will appear in the manifest (unless they are disabled) with most of their properties. This is used to populate documentation and perform state comparison.

Run results

This file contains info about the completed job, including timing and status for each node (model, test) that was executed. In aggregate, these can be used to calculate average model runtime, test failure rate, number of record changes captured by snapshots, etc.

Catalog

This file contains info from your data warehouse on the tables and views produced and defined in the resources of your project. This is ued to populate metadata such as column types and table stats.

Sources

This file contains info on the sources with freshness checks.

documentation

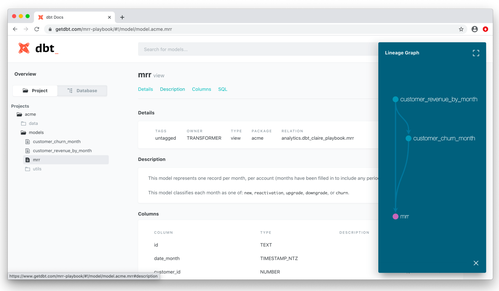
WHAT IS DBT DOCUMENTATION?

Documenting your models is a good way to make sure they are understood by consumers and other editors. dbt automatically generates documentation and makes it possible to render it as a website.

WHAT'S IN DBT DOCS?

In your dbt docs, you can see things like:

- model name
- model materalization
- tags,
- lineage graph (parent and child models)
- columns, types & descriptions,
- compiled SQL (if node was executed)



DEPOYING DOCS

If you're using dbtCloud, documentation will be hosted there. Otherwise, you can look at alternative hosting solutions like Amazon S3 or Netlify.

lineage & the DAG

WHAT IS A DAG?

A DAG is a **D**irected **A**cyclic **G**raph, a type of graph whose nodes are directionally related to each other and don't form a directional closed loop. In the practice of analytics engineering, DAGs are often used to visually represent the relationships between your data models.

WHAT IS A DAG GOOD FOR?

DAGs are an effective tool to help you understand relationships between your data models and areas of improvement for your overall data transformations.

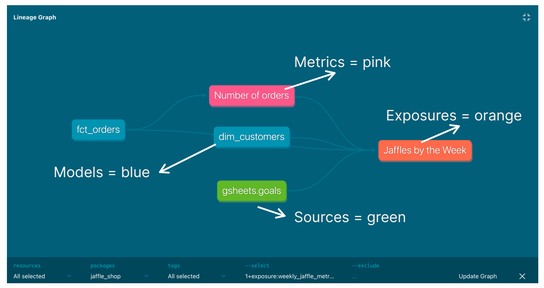
WHAT IS A DATA LINEAGE?

Data lineage provides a holistic view of how data moves through an organization, where it's transformed and consumed. It helps with:

- root cause analysis
- downstream impacts on upstream changes
- shared data literacy
- pipeline cleanliness

DBT DAGS

One of the key features of dbt is the lineage graph (or DAG). These are available in the docs in both dbt Cloud and dbt Core offerings.

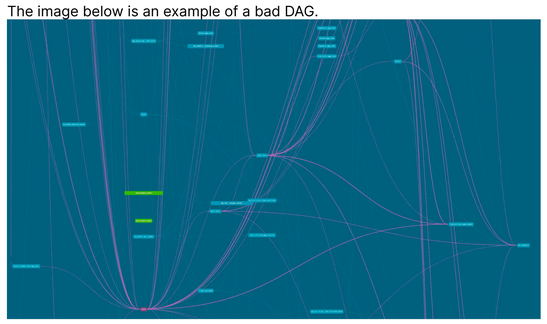


WHAT CAN A DAG TELL YOU?

Looking at your DAG can tell you quite a bit about your project, including:

- whether you should make a CTE into its own view
- perform a join earlier upstream
- where you have expensive joins
- extensive filtering or use of window function

The image below is an example of a bad DAG.



modular data modeling best practice

Leverage staging, intermediate, and mart layers to create layers of distinction between sources & transformed data

Abstract code that's been used across multiple models to its own model

Joining on surrogate keys versus on multiple values

Use the dbt project evaluator package to help audit your project

This video is a great walk-through of both why the DAG and a modular approach is handy, and how to use it to audit your own models

How to Audit Your Directed Acyclic Graph (DAG) and Cr...

YouTube

analyses

WHAT ARE ANALYSES?

Sometimes you want to run a query that doesn't quite fit into a model framework. By creating an analysis query you can use the compiled SQL in your BI tool after a dbt run.

analyses/running_total_by_account.sql

```
-- analyses/running_total_by_account.sql

with journal_entries as (

  select *
  from {{ ref('quickbooks_adjusted_journal_entries') }}

), accounts as (

  select *
  from {{ ref('quickbooks_accounts_transformed') }}

)

select
  txn_date,
  account_id,
  adjusted_amount,
  description,
  account_name,
  sum(adjusted_amount) over (partition by account_id order by id rows unbounded preceding)
from journal_entries
order by account_id, id
```

exposures

WHAT ARE EXPOSURES?

Exposures make it possible to define and describe a downstream use of your dbt project (e.g. a dashboard, data science pipeline). When you've defined an exposure, you can:

- run, test and list resources that feed into your exposure
- populate a dedicated page into the auto-generated docs with context relevant to data consumers

HOW TO DECLARE AN EXPOSURE?

To declare an exposure, use the `.yml` files under the `exposures:` key.

models/<filename>.yml

```
version: 2

exposures:

  - name: weekly_jaffle_metrics
    label: Jaffles by the Week
    type: dashboard
    maturity: high
    url: https://bi.tool/dashboards/1
    description: >
      Did someone say "exponential growth"?

    depends_on:
      - ref('fct_orders')
      - ref('dim_customers')
      - source('gsheets', 'goals')
      - metric('count_orders')

    owner:
      name: Callum McData
      email: data@jaffleshop.com
```

metrics

WHAT ARE METRICS?

Expect some changes in this area. Today, a metric is an aggregation over a table that supports zero or more dimensions (e.g. active users or monthly recurring revenue). When connected to analytical tools they allow end users to dynamically analyze these metrics in a consistent and well-defined way.

HOW TO DEFINE A METRIC?

You can define a metric in the .yml file under a metrics: key. It must:

- contain only letters, numbers, and underscores
- begin with a letter
- contain no more than 250 characters

models/<filename>.yml

```
# models/marts/product/schema.yml

version: 2

models:
 - name: dim_customers
   ...

metrics:
  - name: rolling_new_customers
    label: New Customers
    model: ref('dim_customers')
    description: "The 14 day rolling count of paying customers using the product"

    calculation_method: count_distinct
    expression: user_id

    timestamp: signup_date
    time_grains: [day, week, month, quarter, year]

    dimensions:
      - plan
      - country

    window:
      count: 14
      period: day

    filters:
      - field: is_paying
        operator: 'is'
        value: 'true'
      - field: lifetime_value
        operator: '>='
        value: '100'
      - field: company_name
        operator: '!='
        value: "'Acme, Inc'"
      - field: signup_date
        operator: '>='
        value: "'2020-01-01'"

    # general properties
    config:
      enabled: true | false
      treat_null_values_as_zero: true | false

    meta: {team: Finance}
```

CUSTOMIZING A METRIC

There are many ways to define and customize a metric. These include specifying properties (e.g. calculation methods, filters), and customizing the configuration.

Properties available: (bold = req'd)

- **name**
- **model**
- **label**
- description
- **calculation_method** (count, count_distinct, sum, average, min, max, median, derived)
- **expression**
- timestamp
- time_grains
- dimensions
- window
- filters
- config
- meta

HOW TO QUERY A METRIC?

You must have the dbt_metrics_package installed before querying metrics directly. Then you can use the command `metrics.calculate` along with the defined metrics to generate a SQL statement that runs the metric aggregation to return the correct metric dataset.

```
select *
from {{ metrics.calculate(
    metric('new_customers'),
    grain='week',
    dimensions=['plan', 'country']
) }}
```

Jaffle Shop Example

The following is the SQL & logic behind the example dbt project jaffle_shop. It can help to see all the logic laid out in one place.

This doesn't include the yaml files that customize exactly how all this is run and saved on your databsae. It is just meant to demonstrate how the logic goes from source tables to final outputs.

Sources (seeds in this case)

raw_customers

raw_customers.csv

id	first_name	last_name
1	Michael	P.
2	Shawn	M.
3	Kathleen	P.
4	Jimmy	C.
5	Katherine	R.
6	Sarah	R.
7	Martin	M.
8	Frank	R.
9	Jennifer	F.
10	Henry	W.
11	Fred	S.

DuckDB · 2 ms (Just now) · 3 columns · 100 rows

raw_orders

raw_orders.csv

id	user_id	order_date	status
2	3	2018-01-02	completed
3	94	2018-01-04	completed
4	50	2018-01-05	completed
5	64	2018-01-05	completed
6	54	2018-01-07	completed
7	88	2018-01-09	completed
9	53	2018-01-12	completed
11	99	2018-01-14	completed
12	59	2018-01-15	completed
13	84	2018-01-17	completed

DuckDB · 4 ms (Just now) · 4 columns · 99 rows

raw_payments

raw_payments.csv

id	order_id	payment_method	amount
1	1	credit_card	1000
2	2	credit_card	2000
3	3	coupon	100
4	4	coupon	2500
5	5	bank_transfer	1700
6	6	credit_card	600
7	7	credit_card	1600
8	8	credit_card	2300
9	9	gift_card	2300
10	9	bank_transfer	0
11	10	bank_transfer	2600

DuckDB · 3 ms (Just now) · 4 columns · 113 rows

Staging models

The SQL has been compiled into duckdbt syntax and run locally in this canvas

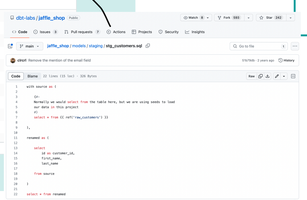
Anonymous

stg_customers

```
select
  id as customer_id,
  first_name,
  last_name
from
  raw_customers
```

customer_id	first_name	last_name
23	Mildred	A.
24	David	G.
25	Victor	H.
26	Aaron	R.
27	Benjamin	B.
28	Lisa	W.
29	Benjamin	K.
30	Christina	W.
31	Jane	G.
32	Thomas	O.
33	Katherine	M.

DuckDB · 3 ms (Just now) · 3 columns · 100 rows

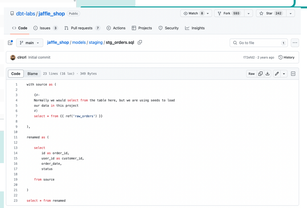


stg_orders

```
select
  id as order_id,
  user_id as customer_id,
  order_date,
  status
from
  raw_orders
```

order_id	customer_id	order_date	status
1	1	2018-01-01	returned
2	3	2018-01-02	completed
3	94	2018-01-04	completed
4	50	2018-01-05	completed
5	64	2018-01-05	completed
6	54	2018-01-07	completed
7	88	2018-01-09	completed
8	2	2018-01-11	returned
9	53	2018-01-12	completed
10	7	2018-01-14	completed
11	99	2018-01-14	completed

DuckDB · 2 ms (Just now) · 4 columns · 99 rows

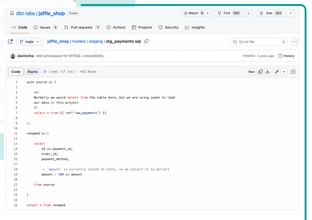


stg_payments

```
select
  id as payment_id,
  order_id,
  payment_method,
  --`amount` is currently stored in cents, so we convert it to dollars
  amount / 100 as amount
from
  raw_payments
```

payment_id	order_id	payment_method	amount
1	1	credit_card	10
2	2	credit_card	20
3	3	coupon	1
4	4	coupon	25
5	5	bank_transfer	17
6	6	credit_card	6
7	7	credit_card	16
8	8	credit_card	23
9	9	gift_card	23
10	9	bank_transfer	0
11	10	bank_transfer	26

DuckDB · 5 ms (Just now) · 4 columns · 113 rows



customers.sql

customer_payments

```
select
  stg_orders.customer_id,
  sum(amount) as total_amount
from
  stg_payments
  left join stg_orders on stg_payments.order_id = stg_orders.order_id
group by
  stg_orders.customer_id
```

customer_id	total_amount
1	33
2	23
3	65
6	8
7	26
8	45
9	30
11	3
12	4
13	26
16	12

DuckDB · 10 ms (Just now) · 2 columns · 62 rows

customer_orders

```
select
  customer_id,
  min(order_date) as first_order,
  max(order_date) as most_recent_order,
  count(order_id) as number_of_orders
from
  stg_orders
group by
  customer_id
```

customer_id	first_order	most_recent_order	number_of_orders
32	2018-01-28	2018-01-28	1
33	2018-02-13	2018-02-13	1
34	2018-02-26	2018-02-26	1
35	2018-02-21	2018-03-21	2
36	2018-03-10	2018-03-10	1
38	2018-02-06	2018-02-06	1
39	2018-01-18	2018-01-18	1
40	2018-01-17	2018-01-17	1
41	2018-04-07	2018-04-07	1
42	2018-02-04	2018-03-12	2
46	2018-03-24	2018-03-27	2

DuckDB · 3 ms (Just now) · 4 columns · 62 rows

Will be saved as a table called `customers`

Anonymous

customers

```
select
  stg_customers.customer_id,
  stg_customers.first_name,
  stg_customers.last_name,
  customer_orders.first_order,
  customer_orders.most_recent_order,
  customer_orders.number_of_orders,
  customer_payments.total_amount as customer_lifetime_value
from
  stg_customers
  left join customer_orders on stg_customers.customer_id = customer_orders.
customer_id
  left join customer_payments on stg_customers.customer_id = customer_payments.
customer_id
```

customer_id	first_name	last_name	first_order	most_recent_order	number_of_orders	custo
1	Michael	P.	2018-01-01	2018-02-10	2	33
2	Shawn	M.	2018-01-11	2018-01-11	1	23
3	Kathleen	P.	2018-01-02	2018-03-11	3	65
6	Sarah	R.	2018-02-19	2018-02-19	1	8
7	Martin	M.	2018-01-14	2018-01-14	1	26
8	Frank	R.	2018-01-29	2018-03-12	2	45
9	Jennifer	F.	2018-03-17	2018-03-17	1	30
11	Fred	S.	2018-03-23	2018-03-23	1	3
12	Amy	D.	2018-03-03	2018-03-03	1	4
13	Kathleen	M.	2018-03-07	2018-03-07	1	26
15	Amanda	H.	2018-02-02	2018-02-02	1	12

DuckDB · 12 ms (Just now) · 7 columns · 100 rows

orders.sql

order_payments

```
select
 order_id,
 sum(
   case
     when payment_method = 'credit_card' then amount
     else 0
   end
 ) as credit_card_amount,
 sum(
   case
     when payment_method = 'coupon' then amount
     else 0
   end
 ) as coupon_amount,
 sum(
   case
     when payment_method = 'bank_transfer' then amount
     else 0
   end
 ) as bank_transfer_amount,
 sum(
   case
     when payment_method = 'gift_card' then amount
     else 0
   end
 ) as gift_card_amount,
 sum(amount) as total_amount
from
 stg_payments
group by
 order_id
```

order_id	credit_card_amount	coupon_amount	bank_transfer_amount	gift_card_amount	total_amo
1	10	0	0	0	10
2	20	0	0	0	20
3	0	1	0	0	1
4	0	25	0	0	25
5	0	0	17	0	17
6	6	0	0	0	6
7	16	0	0	0	16
8	23	0	0	0	23
9	0	0	0	23	23
10	0	0	26	0	26
11	27	0	0	0	27

DuckDB · 5 ms (Just now) · 6 columns · 99 rows

Will be saved as a table called `orders`

Anonymous

orders

```
select
  stg_orders.order_id,
  stg_orders.customer_id,
  stg_orders.order_date,
  stg_orders.status,
  order_payments.credit_card_amount,
  order_payments.coupon_amount,
  order_payments.bank_transfer_amount,
  order_payments.gift_card_amount,
  order_payments.total_amount as amount
from
  stg_orders
  left join order_payments on stg_orders.order_id = order_payments.order_id
```

order_id	customer_id	order_date	status	credit_card_amount	coupon_amount	ban
1	1	2018-01-01	returned	10	0	0
2	3	2018-01-02	completed	20	0	0
3	94	2018-01-04	completed	0	1	0
4	50	2018-01-05	completed	0	25	0
5	64	2018-01-05	completed	0	0	17
6	54	2018-01-07	completed	6	0	0
7	88	2018-01-09	completed	16	0	0
8	2	2018-01-11	returned	23	0	0
9	53	2018-01-12	completed	0	0	0
10	7	2018-01-14	completed	0	0	26
11	99	2018-01-14	completed	27	0	0

DuckDB · 7 ms (Just now) · 9 columns · 99 rows