**Traditional Data Teams**

* Data engineers are responsible for maintaining data infrastructure and the ETL process for creating tables and views.
* Data analysts focus on querying tables and views to drive business insights for stakeholders.

**ETL and ELT**

* ETL (extract transform load) is the process of creating new database objects by extracting data from multiple data sources, transforming it on a local or third party machine, and loading the transformed data into a data warehouse.
* ELT (extract load transform) is a more recent process of creating new database objects by first extracting and loading raw data into a data warehouse and then transforming that data directly in the warehouse.
* The new ELT process is made possible by the introduction of cloud-based data warehouse technologies.

**Analytics Engineering**

* Analytics engineers focus on the transformation of raw data into transformed data that is ready for analysis. This new role on the data team changes the responsibilities of data engineers and data analysts.
* Data engineers can focus on larger data architecture and the EL in ELT.
* Data analysts can focus on insight and dashboard work using the transformed data.
* Note: At a small company, a data team of one may own all three of these roles and responsibilities. As your team grows, the lines between these roles will remain blurry.

**dbt**

* dbt empowers data teams to leverage software engineering principles for transforming data.
* The focus of this course is to build your analytics engineering mindset and dbt skills to give you more leverage in your work.

## Data platform

dbt is designed to handle the transformation layer of the ‘extract-load-transform’ framework for data platforms. dbt creates a connection to a data platform and runs SQL code against the warehouse to transform data.

In our demos, we will be using Snowflake as our data warehouse. You will need access to your own data platform to complete the practice exercises

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time.

## Version control

dbt also enables developers to leverage a version control system to manage their code base. A popular version control system is git. If you are unfamiliar with git, don’t worry, dbt Cloud provides a UI that makes it simple to use a git workflow.

All in all, dbt is going to be the transformation interface between the code we write (stored and managed in a git repository) and the sample data we have to work with (stored and transformed in your data platform).

QUIZ

* **What is required to materialize a model in your data platform with dbt? Select the best answer.**

A dbt project with .sql and/or .py files saved in a models directory

A connection to a data warehouse

dbt\_project.yml file

A & C only

A, B, & C

**When working in dbt, which one of the following should always be unique for each dbt developer when working in development?**

Data platform

Git repository

Target schema

dbt project name

**Which one of the following is a true statement about dbt connections?**

Data is stored in the data platform and code is stored in dbt

Data is stored in dbt and code is stored in the git repository

Data is stored in the data platform and code is stored in the git repository

Data and code are both stored in dbt

**Which of the following is true about version control with git in dbt?**

When making changes to a development branch, you will impact models running in production

Only one developer can work on a single development branch at a time

Separate branches allow dbt developers to simultaneously work on the same code base without impacting production

You have to update the code base to match the environment you are in

# Review

## Development vs. Deployment

* Development in dbt is the process of building, refactoring, and organizing different files in your dbt project. This is done in a development environment using a development schema (dbt\_jsmith) and typically on a non-default branch (i.e. feature/customers-model, fix/date-spine-issue). After making the appropriate changes, the development branch is merged to main/master so that those changes can be used in deployment.
* Deployment in dbt (or running dbt in production) is the process of running dbt on a schedule in a deployment environment. The deployment environment will typically run from the default branch (i.e., main, master) and use a dedicated deployment schema (e.g., dbt\_prod). The models built in deployment are then used to power dashboards, reporting, and other key business decision-making processes.
* The use of development environments and branches makes it possible to continue to build your dbt project without affecting the models, tests, and documentation that are running in production.

## Creating your Deployment Environment

* A deployment environment can be configured in dbt Cloud on the Environments page.
* **General Settings:**You can configure which dbt version you want to use and you have the option to specify a branch other than the default branch.
* **Data Warehouse Connection:** You can set data warehouse specific configurations here. For example, you may choose to use a dedicated warehouse for your production runs in Snowflake.
* **Deployment Credentials:** Here is where you enter the credentials dbt will use to access your data warehouse:
  + IMPORTANT: When deploying a real dbt Project, you should set up a **separate data warehouse account** for this run. This should not be the same account that you personally use in development.
  + IMPORTANT: The schema used in production should be **different** from anyone's development schema.

## Scheduling a job in dbt Cloud

* Scheduling of future jobs can be configured in dbt Cloud on the Jobs page.
* You can select the deployment environment that you created before or a different environment if needed.
* **Commands:** A single job can run multiple dbt commands. For example, you can run dbt run and dbt test back to back on a schedule. You don't need to configure these as separate jobs.
* **Triggers:**This section is where the schedule can be set for the particular job.
* After a job has been created, you can manually start the job by selecting Run Now

## Reviewing Cloud Jobs

* The results of a particular job run can be reviewed as the job completes and over time.
* The logs for each command can be reviewed.
* If documentation was generated, this can be viewed.
* If dbt source freshness was run, the results can also be viewed at the end of a job.

#### **Try it yourself: Creating a webhook in dbt Cloud**

#### Set up an endpoint:

* Travel to <https://beeceptor.com/>. This website will allow you to create an endpoint for free without signing up for anything.
* In the white box that says Endpoint Name, enter the name of your endpoint. Name it whatever you'd like.
* Copy your endpoint. It should be in the format **https://your\_endpoint.free.beeceptor.com**

Create the webhook:

Follow the previous video called **Webhooks in dbt Cloud:**

* Make sure you have a dbt Cloud project with a job already made.
  + (If there is not already a job made for your project, [make one](https://courses.getdbt.com/courses/take/fundamentals/lessons/27962337-setting-up-a-dbt-cloud-job).)
* Go to your account settings, select webhooks, and select create a new webhook.
* Name the webhook and give it a description. (Something like test\_webhook will do for both.)
* Select 'Run Started' under 'Events'.
* Select a job you'd like to run from the Jobs dropdown menu.
* Paste your beeceptor endpoint into the endpoint section.
* Save your webhook and then test your endpoint.

After testing your endpoint, you should get a notification at your beeceptor console address. (This address has a URL in the format **https://beeceptor.com/console/your\_endpoint**).

If you run the job you listed, you should also get a notification at your beeceptor console address.

**Note: You can only use beeceptor to preview incoming webhooks - if you ever wanted to build a webhooks integration with your dbt Project, you would need to use another tool.**

4Dbt test –select chicagocrime\_arrestments

## Jinja

Jinja a templating language written in the python programming language. Jinja is used in dbt to write functional SQL. For example, we can write a dynamic pivot model using Jinja.

## Jinja Basics

The best place to learn about leveraging Jinja is the [Jinja Template Designer documentation](https://jinja.palletsprojects.com/page/templates/).

There are three Jinja delimiters to be aware of in Jinja.

* {% … %} is used for statements. These perform any function programming such as setting a variable or starting a for loop.
* {{ … }} is used for expressions. These will print text to the rendered file. In most cases in dbt, this will compile your Jinja to pure SQL.
* {# … #} is used for comments. This allows us to document our code inline. This will not be rendered in the pure SQL that you create when you run dbt compile or dbt run.

A few helpful features of Jinja include dictionaries, lists, if/else statements, for loops, and macros.

**Dictionaries** are data structures composed of key-value pairs.

{% set person = {

‘name’: ‘me’,

‘number’: 3

} %}

{{ person.name }}

me

{{ person[‘number’] }}

3

**Lists** are data structures that are ordered and indexed by integers.

{% set self = [‘me’, ‘myself’] %}

{{ self[0] }}

me

**If/else statements** are control statements that make it possible to provide instructions for a computer to make decisions based on clear criteria.

{% set temperature = 80.0 %}

On a day like this, I especially like

{% if temperature > 70.0 %}

a refreshing mango sorbet.

{% else %}

A decadent chocolate ice cream.

{% endif %}

On a day like this, I especially like

a refreshing mango sorbet

**For loops** make it possible to repeat a code block while passing different values for each iteration through the loop.

{% set flavors = [‘chocolate’, ‘vanilla’, ‘strawberry’] %}

{% for flavor in flavors %}

Today I want {{ flavor }} ice cream!

{% endfor %}

Today I want chocolate ice cream!

Today I want vanilla ice cream!

Today I want strawberry ice cream!

**Macros** are a way of writing functions in Jinja. This allows us to write a set of statements once and then reference those statements throughout your code base.

{% macro hoyquiero(flavor, dessert = ‘ice cream’) %}

Today I want {{ flavor }} {{ dessert }}!

{% endmacro %}

{{ hoyquiero(flavor = ‘chocolate’) }}

Today I want chocolate ice cream!

{{ hoyquiero(mango, sorbet) }}

Today I want mango sorbet!

## Whitespace Control

We can control for whitespace by adding a single dash on either side of the Jinja delimiter. This will trim the whitespace between the Jinja delimiter on that side of the expression.

## Bringing it all Together!

We saw that we could refactor the following pivot model in pure SQL using Jinja to make it more dynamic to pivot on a list of payment methods.

**Original SQL:**

with payments as (

select \* from {{ ref('stg\_payments') }}

),

final as (

select

order\_id,

sum(case when payment\_method = 'bank\_transfer' then amount else 0 end) as bank\_transfer\_amount,

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 = 'gift\_card' then amount else 0 end) as gift\_card\_amount

   from payments

group by 1

)

select \* from final

**Refactored Jinja + SQL:**

{%- set payment\_methods = ['bank\_transfer','credit\_card','coupon','gift\_card'] -%}

with payments as (

select \* from {{ ref('stg\_payments') }}

),

final as (

select

order\_id,

{% for payment\_method in payment\_methods -%}

sum(case when payment\_method = '{{ payment\_method }}' then amount else 0 end)

as {{ payment\_method }}\_amount

{%- if not loop.last -%}

,

{% endif -%}

{%- endfor %}

   from payments

group by 1

)

select \* from final

# Review

## Tables

* Built as tables in the database
* Data is stored on disk
* Slower to build
* Faster to query
* Configure in dbt\_project.yml or with the following config block

{{ config(

    materialized='table'

)}}

## Views

* Built as views in the database
* Query is stored on disk
* Faster to build
* Slower to query
* Configure in dbt\_project.yml or with the following config block

{{ config(

    materialized='view'

)}}

## Ephemeral Models

* Does not exist in the database
* Imported as CTE into downstream models
* Increases build time of downstream models
* Cannot query directly
* [Ephemeral Documentation](https://docs.getdbt.com/docs/building-a-dbt-project/building-models/materializations#ephemeral)
* Configure in dbt\_project.yml or with the following config block

{{ config(

    materialized='ephemeral'

)}}

## Incremental Models

* Built as table in the database
* On the first run, builds entire table
* On subsequent runs, only appends new records\*
* Faster to build because you are only adding new records
* Does not capture 100% of the data all the time
* [Incremental Documentation](https://docs.getdbt.com/docs/building-a-dbt-project/building-models/materializations#incremental)
* [Discourse post on Incrementality](https://discourse.getdbt.com/t/on-the-limits-of-incrementality/303)
* Configuration is more advanced in this case. Consult the dbt documentation for building your first incremental model.

## Snapshots

* Built as a table in the database, usually in a dedicated schema.
* On the first run, builds entire table and adds four columns: dbt\_scd\_id, dbt\_updated\_at, dbt\_valid\_from, and dbt\_valid\_to
* In future runs, dbt will scan the underlying data and append new records based on the configuration that is made.
* This allows you to capture historical data
* [Snapshots Documentation](https://docs.getdbt.com/docs/building-a-dbt-project/snapshots)
* Configuration is more advanced in this case. Consult the dbt documentation for writing your first snapshot.

1. **Test on One Database Object**

What/Why

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

**Example Tests**

* Unique
* not null
* accepted values

**Other package's tests like:**

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

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

**What/Why**

* Compare values in one model to asource 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**
* Orders should have payments >= 0
* Billing Total should equal the sum of all parts: anbrotal + tax credits - Total

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

**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

**Dim\_chicagocrimedata.sql**

{{ config(materialized='table') }}

with chicagocrime\_details as

(

    select id,case\_number,date from raw.raw\_schema.CHICAGOCRIMEDATA

),

CHICAGOCRIMEDATA\_DESCRIPTION AS

(

    select \* from {{ ref ('chicagocrime\_description') }}

),

CHICAGOCRIMEDATA\_ARRESTMENTS AS

(

    select \* from {{ ref('chicagocrime\_arrestments')}}

),

CHICAGOCRIMEDATA\_location AS

(

    select \* from {{ ref ('chicagocrime\_location')}}

),

final\_data as

(

    select

    chicagocrime\_details.id,

    chicagocrime\_details.case\_number,

    chicagocrime\_details.date,

    CHICAGOCRIMEDATA\_DESCRIPTION.crime\_description,

    CHICAGOCRIMEDATA\_location.location,

    CHICAGOCRIMEDATA\_location.location\_description,

    CHICAGOCRIMEDATA\_ARRESTMENTS.arrest

    from (((chicagocrime\_details

    inner join CHICAGOCRIMEDATA\_DESCRIPTION on chicagocrime\_details.id = CHICAGOCRIMEDATA\_DESCRIPTION.description\_id)

    inner join CHICAGOCRIMEDATA\_ARRESTMENTS on chicagocrime\_details.id = CHICAGOCRIMEDATA\_ARRESTMENTS.arrestment\_id)

    inner join CHICAGOCRIMEDATA\_location on chicagocrime\_details.id = CHICAGOCRIMEDATA\_location.location\_id)

)

select \* from final\_data

**CHICAGOCRIME\_ARRESTMENTS**

{{ config(materialized='table') }}

with

CHICAGOCRIME\_ARRESTMENTS AS

(

    select id as arrestment\_id,arrest from raw.raw\_schema.CHICAGOCRIMEDATA

)

select \* from CHICAGOCRIME\_ARRESTMENTS

**CHICAGOCRIME\_DESCRIPTION**

{{ config(materialized='table') }}

with

CHICAGOCRIME\_DESCRIPTION AS

(

    select id  as description\_id,description as crime\_description from raw.raw\_schema.CHICAGOCRIMEDATA

)

select \* from CHICAGOCRIME\_DESCRIPTION

{{ config(materialized="table") }}

    -- first we have to define source and link it with the source yml file we have

    -- created. and then we will link our staging table with the source file directly.

    -- source {{source('source name which is defined in yml file' , 'name of the

    -- table')}}

    -- both the inf should be accurate and same as given in the yml file.

with

chicagocrimedata\_location as (

        select id as location\_id, location\_description, location, longitude, latitude

        from raw.raw\_schema.CHICAGOCRIMEDATA

)

-- select id as location\_id,location\_description,location,longitude,latitude from

-- raw.raw\_schema.CHICAGOCRIMEDATA

-- this command is working perfectly fine but to introduce sources in it we will have

-- to remove the database name but schema and table name will be there

select \*

from chicagocrimedata\_location

select \* from raw.raw\_schema.CHICAGOCRIMEDATA

version: 2

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

                      error\_if: ">10"

    - name: chicagocrime\_description

      description: "Great Expectations testing"

      columns:

        - name: description\_id

          tests:

            - dbt\_expectations.expect\_column\_to\_be\_unique

        - name: crime\_description

          tests:

            - dbt\_expectations.expect\_column\_values\_to\_not\_be\_null

    - name: chicagocrime\_location

      columns:

        - name: location\_id

          tests:

            - dbt\_utils.at\_least\_one

        - name: location\_description

          tests:

            - dbt\_utils.non\_null\_values

        - name: location

          tests:

            - dbt\_utils.non\_empty\_strings

    - name: dim\_chicagocrimedata

      columns:

        - name: ARREST

          tests:

            - arrestments\_true

{{ config(materialized="table") }}

with fct\_id as

(

    select \* from {{ ref('fct\_id') }}

),

dim\_date as

(

    select \* from {{ ref('dim\_date') }}

),

dim\_diagnosis as

(

    select \* from {{ ref('dim\_diagnosis') }}

),

dim\_location as

(

    select \* from {{ ref('dim\_location') }}

),

dim\_patient as

(

    select \* from {{ ref('dim\_patient') }}

),

dim\_payer as

(

    select \* from {{ ref('dim\_payer') }}

),

dim\_physician as

(

    select \* from {{ ref('dim\_physician') }}

),

dim\_transaction as

(

    select \* from {{ ref('dim\_transaction') }}

),

dim\_cptcode as

(

    select \* from {{ ref('dim\_cptcode') }}

),

fact\_table as

(

    select

    fct\_id.fact\_id,

    dim\_cptcode.cptcode\_id,

    dim\_date.datepost\_id,

    dim\_diagnosis.diagnosiscode\_id,

    dim\_location.location\_id,

    dim\_patient.patient\_id,

    dim\_payer.payer\_id,

    dim\_physician.physician\_id,

    dim\_transaction.transaction\_id

    from

    fct\_id

    left join dim\_cptcode on dim\_cptcode.cptcode\_id = fct\_id.fact\_id

    left join dim\_date on dim\_date.datepost\_id = fct\_id.fact\_id

    full join dim\_diagnosis on dim\_diagnosis.diagnosiscode\_id = fct\_id.fact\_id

    full join dim\_location on dim\_location.location\_id = fct\_id.fact\_id

    full join dim\_payer on dim\_payer.payer\_id = fct\_id.fact\_id

    full join dim\_physician on dim\_physician.physician\_id = fct\_id.fact\_id

    full join dim\_transaction on dim\_transaction.transaction\_id = fct\_id.fact\_id

    full join dim\_patient on dim\_patient.patient\_id = fct\_id.fact\_id

)

select \* from fact\_table

**MODELING:**

* Modeling is the shaping of the data from raw data through to your final transformed data
* Typically engineer are responsible for building tables that represent your source data and then on top of that building tables, views that transform that data step by step and by that final table is created which is then used as final table and for visualization purposes.
* In dbt, models are just SQL *select* statements and they are used as modular piece of logic that will take your raw data and build it into the final transformed data that we need.