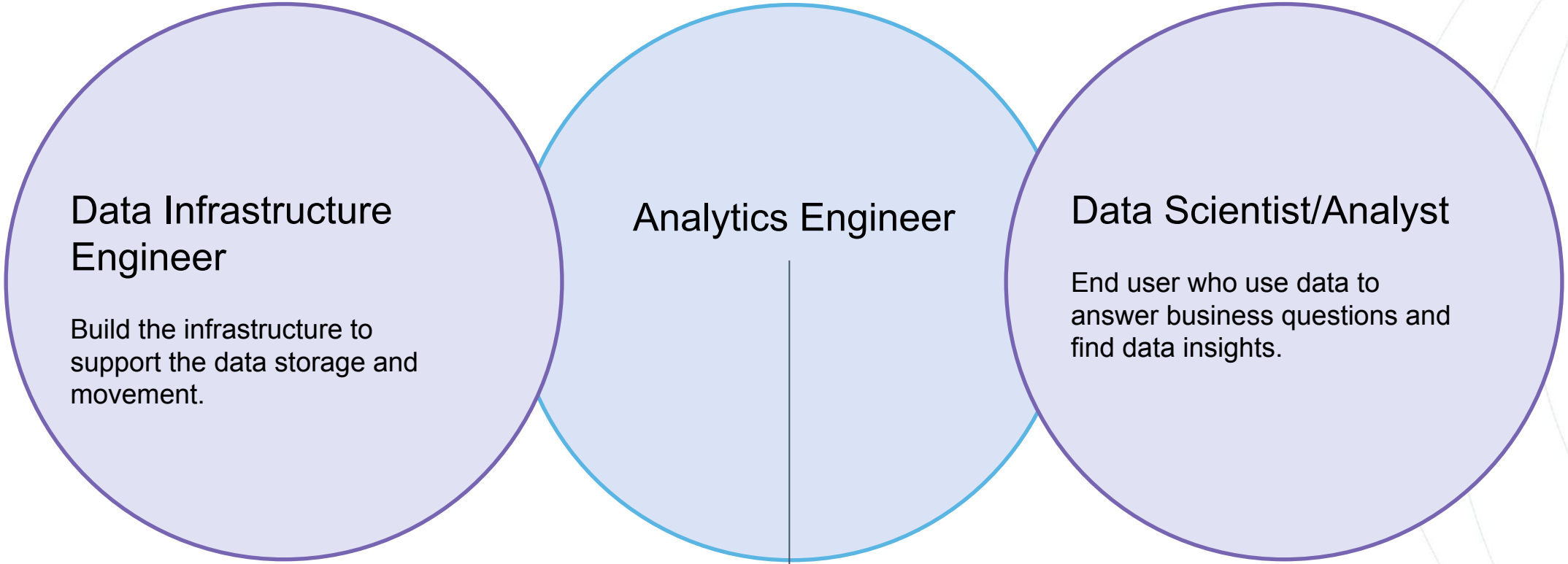
Introduction to DBT

A Powerful Data Transformation and Analytics Tool

Data Engineering Diploma

Analytics Engineering and Modern Data Stack

Data Roles

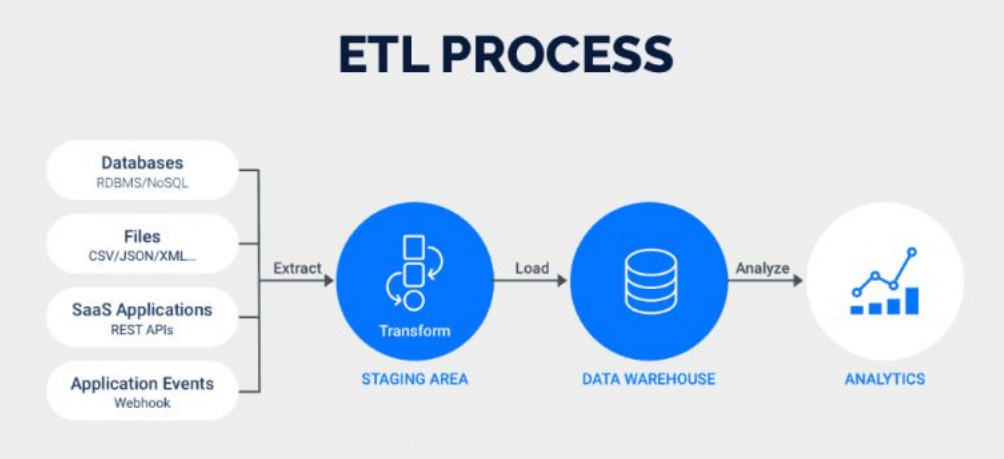


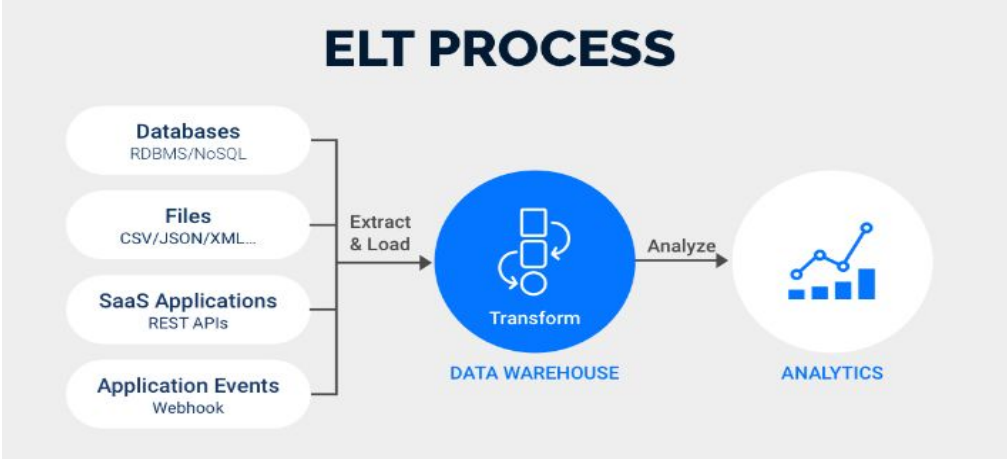
Primary Tools

- SQL, Python
- dbt
- modern data stack tools (Snowflake, Airbyte, Prefect, Fivetran, Matillion)

The Modern Cloud Data Stack

ETL vs ELT

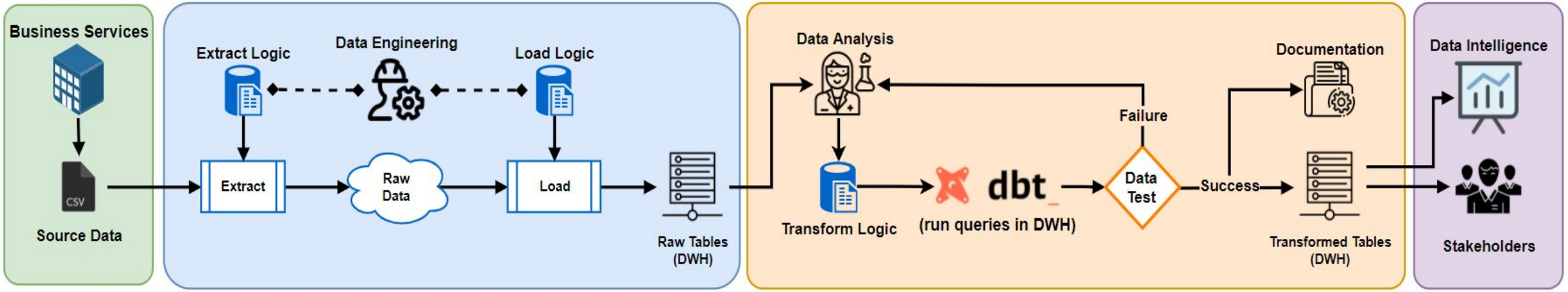




	ETL	ELT
Flexibility	Less flexible	More flexible
Scalability	Less scale	More scale
Cost	More cost (storage and compute)	Less cost
Maintenance	More maintenance on secondary transform server	Fewer system to manage

Introduction to dbt

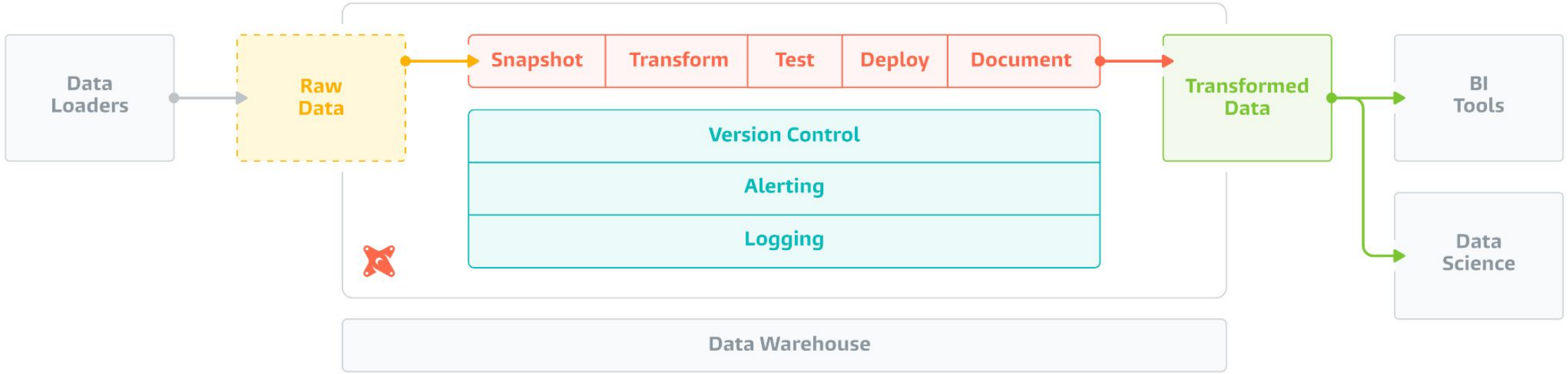
Where is DBT?



PNG Credits: https://ilegra.com/wp-content/uploads/2021/09/imagem4.png

What is DBT?

dbt is an open-source **data build** (transformation) **tool** that use SQL statement with software engineering best practice like **modularity**, **CI/CD**, and **documentation** to deliver reliable data to downstream analytics and reporting.



Why do we need dbt?

The Shortage in traditional data warehouse project:

- Modeling changes are not easy to follow and revert
- Hard to explicit and explore dependencies between models
- Hard to do data quality testing
- Error reporting is not standard
- Hard to track the history of dimension tables
- Bad documentation experience

Why do we need dbt?

After using dbt:

- No longer copy and paste SQL, which can lead to errors when logic changes. Instead, build reusable data models that get pulled into subsequent models and analysis. Change a model once and that change will propagate to all its dependencies.

- Publish the canonical version of a particular data model, encapsulating all complex business logic. All analysis on top of this model will incorporate the same business logic without needing to reimplement it.

- Use mature source control processes like branching, pull requests, and code reviews.

- Write data quality tests quickly and easily on the underlying data. Many analytic errors are caused by edge cases in the data: testing helps analysts find and handle those edge cases.

What are the advantages of DBT?

- Model with SQL SELECT statement, no DDL or DML. (Quick, Simple, Flexible)
- Build reusable and modular code structure for easier maintenance and scalability
- Increased productivity and collaboration between analysts and data engineers
- Flexibility and power of SQL for data transformations
- Version control integration for tracking changes and managing different versions
- Improved data quality and accuracy through testing and documentation

The Structure of the dbt Project

How to structure dbt projects

Structure Overview

Staging – prepare atomic building blocks (Views) 🧱

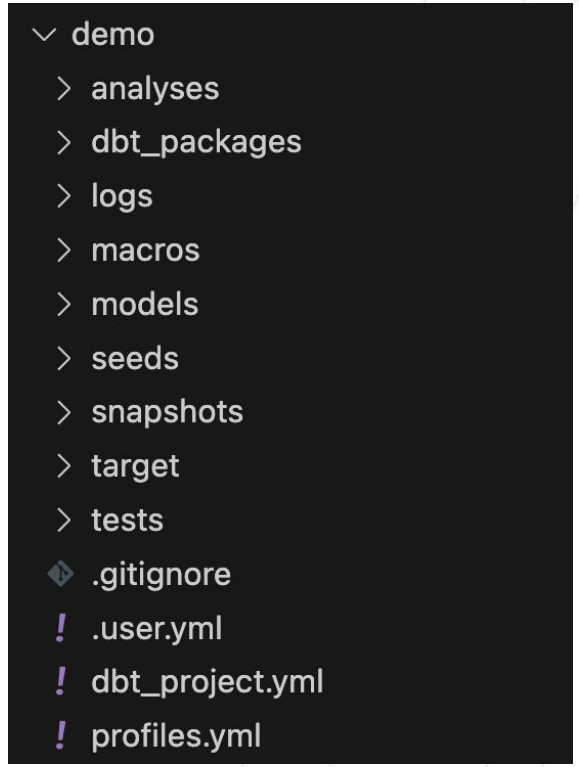
- Creating initial modular building blocks from source data

Intermediate – transformation steps with purpose (Ephemeral) 

- Stacking layers of logic with clear and specific purposes to join the staging tables into entities we want

Marts – business-defined entities (Tables/Incremental models) 🏠

- Bringing together all modular pieces into a business-defined entities your organization cares about



Staging

Folder Structure

- ❑ Subdirectories based on the source system.
- ❑ sources.yml file
  - Testing, documentation related to **Sources** Only
- ❑ models.yml file
  - Testing, documentation, configuration related to **Models** Only

```
models/staging
├── jaffle_shop
│   ├── _jaffle_shop__docs.md
│   ├── _jaffle_shop__models.yml
│   ├── _jaffle_shop__sources.yml
│   ├── base
│   │   ├── base_jaffle_shop__customers.sql
│   │   └── base_jaffle_shop__deleted_customers.sql
│   ├── stg_jaffle_shop__customers.sql
│   └── stg_jaffle_shop__orders.sql
└── stripe
    ├── _stripe__models.yml
    ├── _stripe__sources.yml
    └── stg_stripe__payments.sql
```

Staging Models

- ❑ Materialized as **Views**
  - Any downstream models referencing the staging models will always get the freshest data
  - Avoid wasting storage in warehouse that are not constantly queried by end users
- ❑ Most standard types of staging transformation
  - ✅ Renaming
  - ✅ Casting
  - ✅ Simple calculations (C to F, etc)
  - ✅ Categorizing
  - 🚫 Complex Joins
  - 🚫 Aggregations (group by/ rank)

Intermediate

Folder Structure

- ❑ Subdirectories based on the business groupings/interests.
- ❑ models.yml file
  - Testing, documentation, configuration related to **Models** Only

```
models/intermediate
└── finance
    ├── _int_finance__models.yml
    └── int_payments_pivoted_to_orders.sql
```

Intermediate Models

- ❑ Materialized ephemerally
- ❑ **Not exposed** to end users
- ❑ Some common use cases:
  - ✅ Structural simplification to
    - increase readability, flexibility and testing.
  - ✅ Re-graining
    - collapse models to the right composite grain
  - ✅ Isolating complex operations

Marts

Folder Structure

- ❑ Group by department or area of interest
- ❑ Name by entity
- ❑ models.yml file
  - Testing, documentation, configuration related to **Models** Only

```
models/marts
├── finance
│   ├── _finance__models.yml
│   ├── orders.sql
│   └── payments.sql
└── marketing
    ├── _marketing__models.yml
    └── customers.sql
```

Mart Models

- ❑ Materialized as **Tables/Incremental models**
  - Give end user much faster performance to query
- ❑ Wide and denormalized
- ❑ Build on separate marts thoughtfully
- ❑ Avoid too many joins in one mart
  - In that case, you might want to move some complex logic to Intermediate models

All Together!

```
models
├── intermediate
│   └── finance
│       ├── _int_finance__models.yml
│       └── int_payments_pivoted_to_orders.sql
├── marts
│   ├── finance
│   │   ├── _finance__models.yml
│   │   ├── orders.sql
│   │   └── payments.sql
│   └── marketing
│       ├── _marketing__models.yml
│       └── customers.sql
├── staging
│   ├── jaffle_shop
│   │   ├── _jaffle_shop__docs.md
│   │   ├── _jaffle_shop__models.yml
│   │   ├── _jaffle_shop__sources.yml
│   │   ├── base
│   │   │   ├── base_jaffle_shop__customers.sql
│   │   │   └── base_jaffle_shop__deleted_customers.sql
│   │   ├── stg_jaffle_shop__customers.sql
│   │   └── stg_jaffle_shop__orders.sql
│   └── stripe
│       ├── _stripe__models.yml
│       ├── _stripe__sources.yml
│       └── stg_stripe__payments.sql
└── utilities
    └── all_dates.sql
```

✅ Config per folder (models.yml and/or sources.yml)

❌ Don’t config per project (putting all configuration in one file)

✅ Use dbt_project.yml to set global default configurations

```yaml
-- dbt_project.yml

models:
  jaffle_shop:
    staging:
      +materialized: view
    intermediate:
      +materialized: ephemeral
    marts:
      +materialized: table
      finance:
        +schema: finance
      marketing:
        +schema: marketing
```

A dbt Demo

Demo Overview

Demo Overview

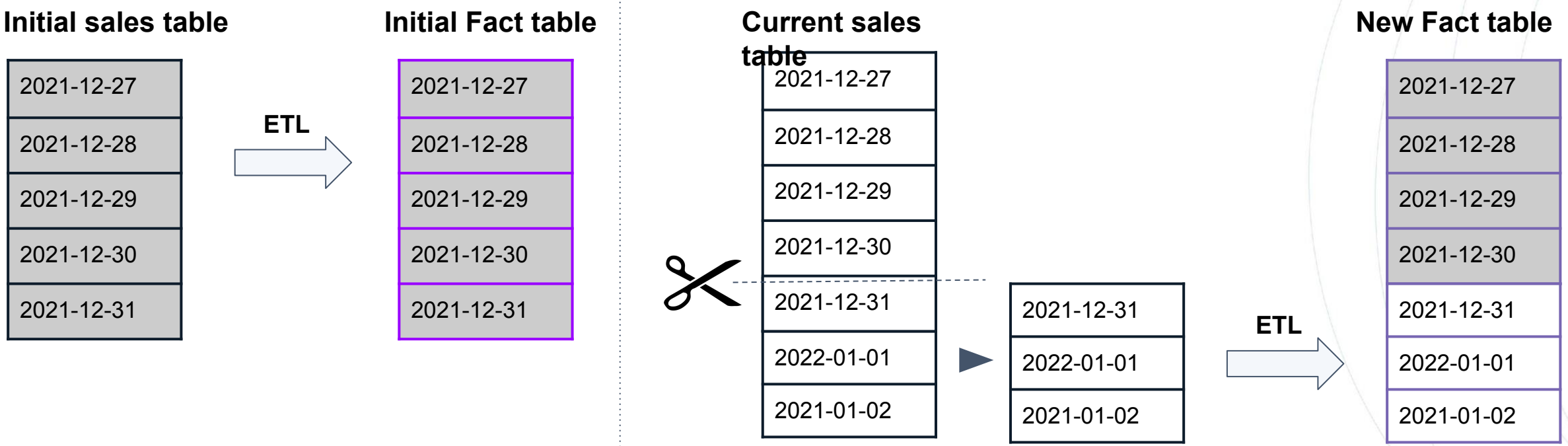
Demo Introduction

We are going to use a demo to walk you through the important parts of the dbt.

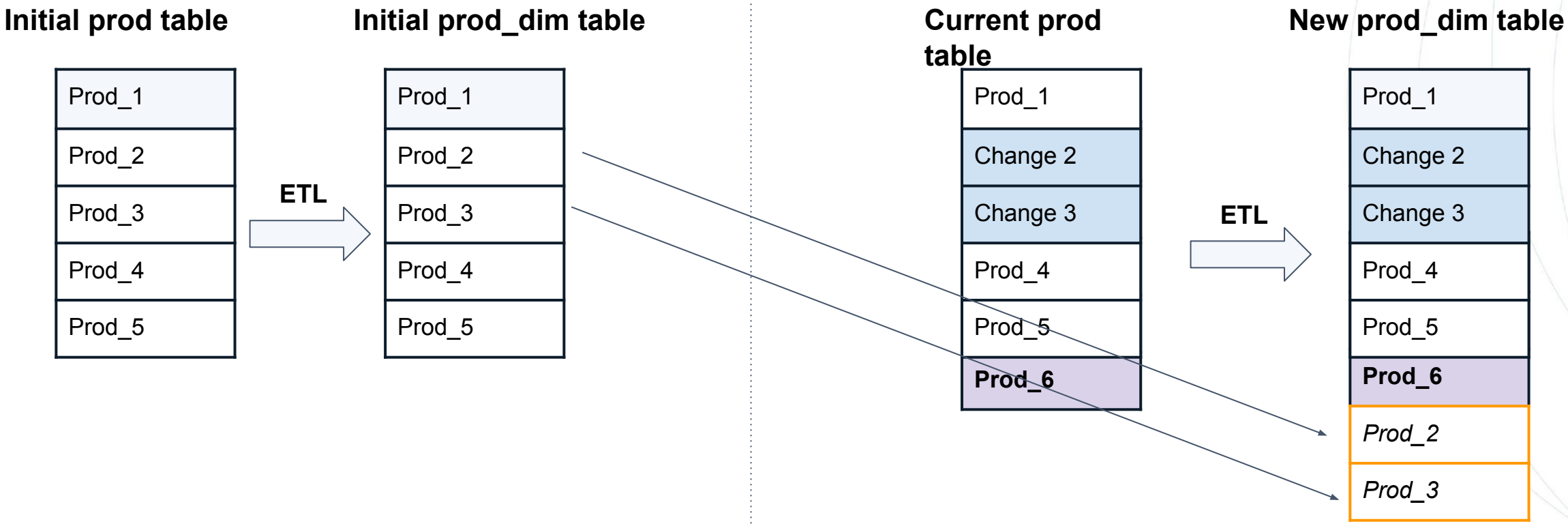
This demo we will use dbt to finish the previous data loading demo.

Let’s recall the previous demo steps, and the traditional SQL way:

Fact Table incremental



Type 2 DIM Table incremental



Previous SQL Scripts steps

1. Use this script to Initialize the demo environment.
   a. Load Sales and Product data into the LAND schema tables.
   b. Load Calendar data into ENTP schema Calendar_dim table.
2. Use this script to check the current status.
3. Use this script to load dimension data from LAND TO ENTP for the first time.
4. Use this script to load fact data from LAND TO ENTP for the first time.
5. Use this script to run delta loading for both dim and fact tables in LAND.
6. Use this script to load dimension data from LAND TO ENTP for the next time.
7. Use this script to load fact data from LAND TO ENTP for the next time.
8. Use this script to check the current status.

A dbt Demo

Project preparation

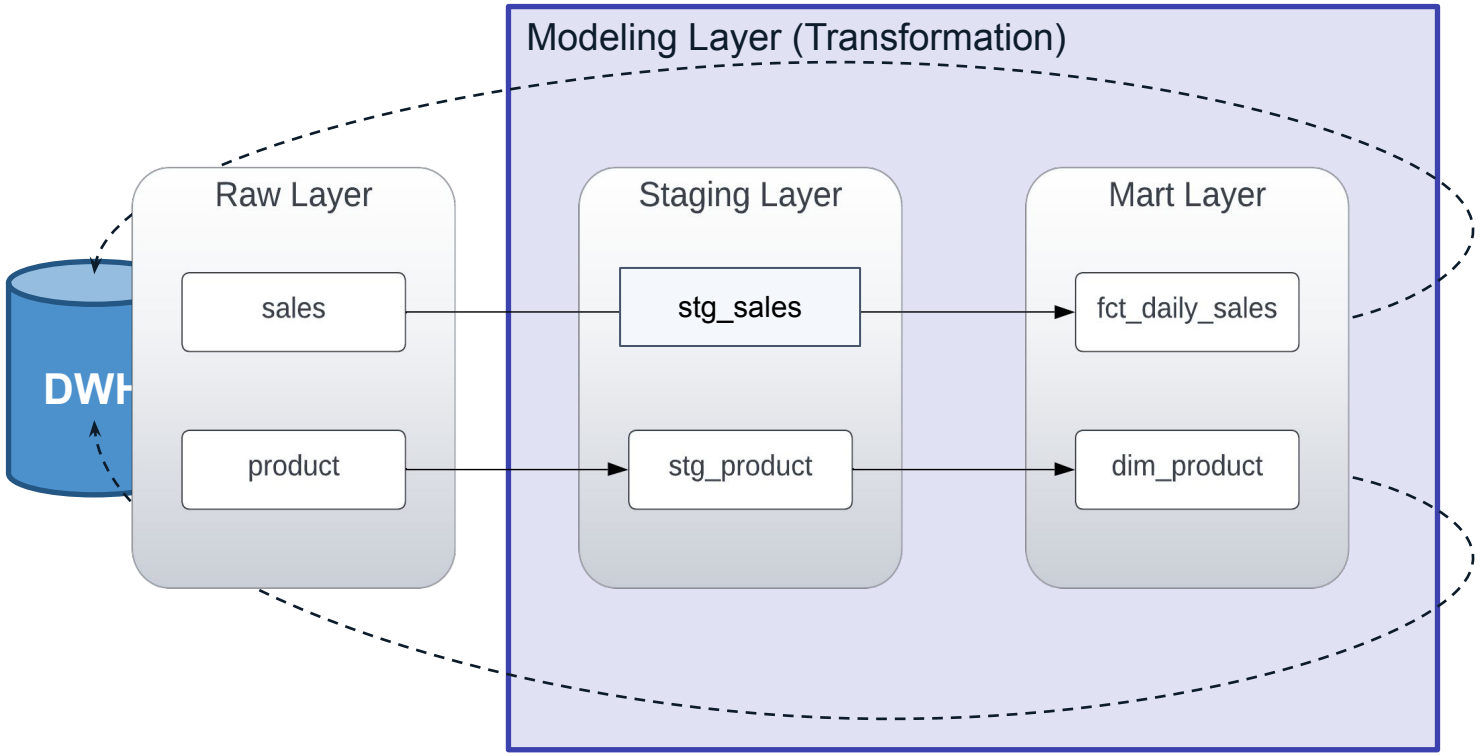
Demo Overview

Tech Stack

Setup

- Snowflake registration (30-day free trial)
- Dataset import
- DBT installation on EC2
- Configure DBT Snowflake Connection

Data Flow Overview



A dbt Demo

dbt installation and setup

DBT Setup

Installing DBT on an Ubuntu Virtual Machine

- Ensure EC2 Instance is running
  - Confirm that your EC2 instance is operational and running.
- Set Up Visual Studio Code
  - Install Visual Studio Code.
  - Add the “Remote - SSH” Extension.
- Configure SSH Config File
  - Create or edit *~/.ssh/config* on your local machine.
  - Include an entry with the instance alias, the EC2 instance’s public IP address, and the file path to your .pem key file.

    ```
    Host your-instance-alias
      HostName your-ec2-public-ip
      User ubuntu
      IdentityFile /path/to/your/keyfile.pem
    ```
- Connect via Visual Studio Code
- Check Python installation (Prerequisite)
  - Verify if you have Python or Python3 installed by checking the current version.
- Update python and pip libs (Optional)

  ```
  python3 -m pip install --upgrade pip
  ```

DBT Setup

Installing DBT on an Ubuntu Virtual Machine

- Install dbt-core and the dbt-snowflake adapter
  - Install dbt-core only
    - If you are building a tool that integrates with dbt Core, you may want to install the core library alone, without a database adapter. Note that you won't be able to use dbt as a CLI tool.
      pip install dbt-core
  - Install dbt-snowflake adapter
    - Installing dbt-snowflake will also install dbt-core and any other dependencies.
      pip install dbt-snowflake
- Verify if dbt is installed correctly
  dbt --version

DBT Setup

Initiating Your First dbt Project

- Initiate a dbt project named “demo” in the path. (Use a descriptive project name that reflects the purpose of the project. ***Project name can only contain letters, digits, and underscores.*** Avoid using spaces, special characters, or starting the project name with a number.)

  dbt init demo

- Choose snowflake as adapter to connect to your data warehouse.

DBT Setup

Advanced: Customizing a profile directory (Optional)

- After you initiate your dbt project, .dbt directory will be created at your $HOME path by default. The profiles.yml will contain project-specific connectivity information of the data warehouse.
  cat ~/.dbt/profiles.yml
  vi ~/.dbt/profiles.yml *or* *vim ~/.dbt/profiles.yml* *or* *nano ~/.dbt/profiles.yml*
- You may want to have your profiles.yml file stored in a different directory than ~/.dbt/
  - for example, if you are using environment variables to load your credentials, you might choose to include this file in the root directory of your dbt project.
    pwd
    mv ~/.dbt/profiles.yml path/to/directory → *i.e., move profiles.yml to path/to/directory*
    export DBT_PROFILES_DIR=path/to/directory → *i.e., replace path/to/directory with your pwd*
    dbt debug

✻ the file always needs to be called ***profiles.yml***, regardless of which directory it is in.

Reference: https://docs.getdbt.com/docs/core/connect-data-platform/connection-profiles

DBT Setup

Advanced: Customizing a profile directory (Optional) (cont.)

- Snowflake can be configured using basic user/password authentication as shown below.

~/.dbt/profiles.yml

```yaml
my-snowflake-db:
  target: dev
  outputs:
    dev:
      type: snowflake
      account: [account id]

      # User/password auth
      user: [username]
      password: [password]

      role: [user role]
      database: [database name]
      warehouse: [warehouse name]
      schema: [dbt schema]
      threads: [1 or more]
      client_session_keep_alive: False
      query_tag: [anything]

      # optional
      connect_retries: 0 # default 0
      connect_timeout: 10 # default: 10
      retry_on_database_errors: False # default: false
      retry_all: False  # default: false
      reuse_connections: False # default: false (available v1.4+)
```

Reference: https://docs.getdbt.com/docs/core/connect-data-platform/snowflake-setup

DBT Setup

Test the connectivity of dbt environment with your Snowflake DWH

- You should be getting a similar output as shown below.

  dbt debug

- Once all checks passed, you can execute dbt run command to run your dbt model.

A dbt Demo

Project Configuration

Define project (dbt_project.yml)

dbt_project.yml

Every dbt project needs a **dbt_project.yml** file — this is how dbt knows a directory is a dbt project. It also contains important information that tells dbt how to operate your project.

By default, dbt will look for **dbt_project.yml** in your current working directory and its parents, but you can set a different directory using the `--project-dir` flag or the `DBT_PROJECT_DIR` environment variable.

your project name

your profile name

Define your dbt model here

```yaml
name: string

config-version: 2
version: version

profile: profilename

models:
  <model-configs>

seeds:
  <seed-configs>
```

https://docs.getdbt.com/reference/dbt_project.yml

Define project (Example)

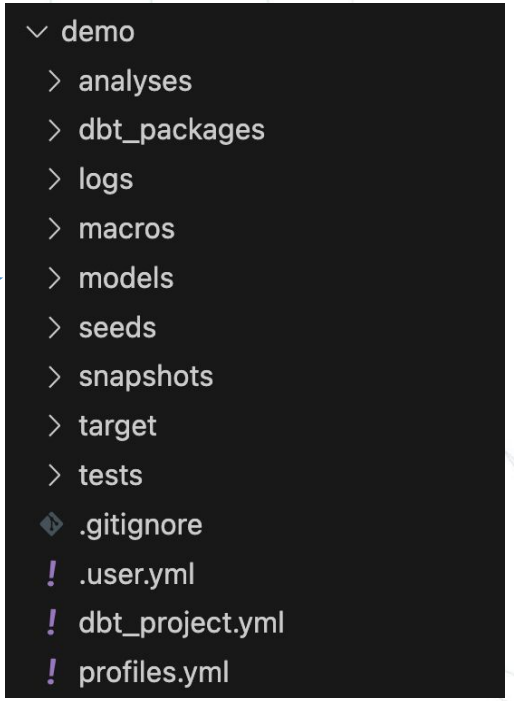
dbt_project.yml

```yaml
name: 'demo'
version: '1.0.0'
config-version: 2

# This setting configures which "profile" dbt uses for this project.
profile: 'default'

# These configurations specify where dbt should look for different types of files.
# The `model-paths` config, for example, states that models in this project can be
# found in the "models/" directory. You probably won't need to change these!
model-paths: ["models"]
analysis-paths: ["analyses"]
test-paths: ["tests"]
seed-paths: ["seeds"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]
packages-install-path: dbt_packages
log-path: logs
target-path: target

clean-targets:         # directories to be removed by `dbt clean`
  - "target"
  - "dbt_packages"


# In this example config, we tell dbt to build all models in the staging/
# directory as views. These settings can be overridden in the individual model
# files using the `{{ config(...) }}` macro.
models:
  demo:
```

This is the profile file name, lead to the profile.yml file

```yaml
default:
  target: dev
  outputs:
    dev:
      account:
      database: dbt_demo
      schema: entp
      password:
      role: accountadmin
      threads: 1
      type: snowflake
      user:
      warehouse: compute_wh
```

We defile the model here



Define project (profiles.yml)

profiles.yml

profiles.yml file contains the connection details for your data platform. When you run dbt from the command line, it reads your **dbt_project.yml** file to find the profile name, and then looks for a profile with the same name in your **profiles.yml** file. This profile contains all the information dbt needs to connect to your data platform.

```
<profile-name>:
  target: <target-name> # this is the default target
  outputs:
    <target-name>:
      type: <bigquery | postgres | redshift | snowflake | other>
      schema: <schema_identifier>
      threads: <natural_number>
```

Define project (Model)

Model

A model is an essential building block of the DAG that lives in a single file and contains logic that transforms data. This logic can be expressed as a SQL select statement or a Python dataframe operation.

Models can be configured in one of three ways:

1. Using a `config()` Jinja macro within a model
2. Using a config resource property in a .yml file
3. From the `dbt_project.yml` file, under the “models:” key.

Models can be materialized in the warehouse in different ways — most of these materializations require models to be built in the warehouse.

Model-specific Configurations

Model can be configured in one of the three ways:

1. Using a config() Jinja macro within a model (highest priority)

<model_name>.sql ~/[redacted]/models

```
{{ config(
    schema='<custom_schema_name>'
    , materialized='<materialized_name: view/table/incremental/ephemeral>'
) }}
select
    ...
```

2. Using a config resource property in a .yml file

properties.yml ~/[redacted]/models

```
version: 2

models:
  - name: [<model_name>]
    config:
      schema: '<custom_schema_name>'
      materialized: <materialized_name:view/table/incremental/ephemeral>
```

3. From the dbt_project.yml file, under the models: key

dbt_project.yml ~/[redacted]

```
models:
  <project_name>:
    +schema: <custom_schema_name>
    +materialized: <materialized_name:view/table/incremental/ephemeral>
    <sub_folder_name>:
      +schema: <custom_schema_name>
      +materialized: <materialized_name:view/table/incremental/ephemeral>
```

Reference: https://docs.getdbt.com/reference/model-configs

Model Configuration

How does dbt compile your model? → *dbt_model (.sql file)*

model-specific configurations

```
{{
        config(
                materialization='table',
                schema='entp'
        )
}}

select
        *,
        lag(start_date, 1) over(partition by prod_key order by start_date desc) as deactivate_date,
        iff(deactivate_date is null, true, false) as active_status
from {{ ref('stg_product')}}
```

dbt runs **compiled SQL statement in the data warehouse**

dbt compiled code

```
create or replace transient table dbt_demo.entp.product_dim
        as
        (select
        *,
        lag(start_date, 1) over(partition by prod_key order by start_date desc) as deactivate_date,
        iff(deactivate_date is null, true, false) as active_status
from dbt_demo.stage.stg_product
        );
```

A dbt Demo

Materializations

Materializations

Materializations are strategies for persisting dbt models in a warehouses.

There are five types of materialization built into dbt. Models can be configured with a different materialization by supplying the **materialized** configuration parameter.

- table
- view (default)
- incremental
- ephemeral
- materialized view (self- read)

	View (default)	Table	Incremental	Ephemeral
Some Use Cases	• Apply consistent **naming conventions** (e.x., renaming, recasting columns) • Abstract complex joins, filters, or transformation	• **Precompute** aggregated results for faster reporting and analytics • Materialize transformed or derived datasets for downstream consumption	• Use incremental models when your dbt runs are becoming too slow • Materialize incremental updates for datasets that **frequently change**, reducing processing time	• Very lightweight transformations that are early on in your DAG • Only used in one or two downstream models • Don't need to be queried directly

https://docs.getdbt.com/docs/build/materializations

Materializations

	View (default)	Table	Incremental	Ephemeral
Descr.	Model is rebuilt as a view on each run, via a ***create view as*** statement	Model is rebuilt as a table on each run, via a ***create table as*** statement	Models allow dbt to ***insert or update*** records into a table since the last time that dbt was run	Models are not persisted in the database
Pros	• No additional data is stored, views on top of source data will always have the latest records in them	• **Fast to query** as they store precomputed results	• Can significantly **reduce the build time** by allowing dbt to ***insert or update*** records into a table since the last time that dbt was run	• Reusable logic for multiple models • Can help keep your data warehouse clean by reducing clutter
Cons	• Views that perform a significant transformation, or are stacked on top of other views, are ***slow to query***	• Take a ***long time*** to rebuild, especially for complex transformations • New records in underlying source data are ***not automatically added*** to the table • Consume additional disk space	• Requires ***extra configuration*** and are an advanced usage of dbt • Incorrectly implemented incremental materialization can lead to data inconsistencies if not properly synchronized	• ***Cannot select directly*** from this model as models are not directly built into the database • Operations (e.x., dbt run-operation cannot reference ephemeral nodes) • Overuse of ephemeral can make queries ***harder to debug***

https://docs.getdbt.com/docs/build/materializations

Materializations

Comparisons and Tips

Rule of thumb regarding materialization

- Start with a **view** (as it takes up essentially no storage and always gives you up-to-date results), once that view takes too long to practically query.
- Build it into a **table**, and finally once that table takes too long to build and is slowing down your runs.
- Configure it as an **incremental model**.

	view	table	incremental
🛠️⏳ **build time**	💚 fastest — only stores logic	❤️ slowest — linear to size of data	💛 medium — builds flexible portion
🛠️💸 **build costs**	💚 lowest — no data processed	❤️ highest — all data processed	💛 medium — some data processed
📊💸 **query costs**	❤️ higher — reprocess every query	💚 lower — data in warehouse	💚 lower — data in warehouse
🍎🌱 **freshness**	💚 best — up-to-the-minute of query	💛 moderate — up to most recent build	💛 moderate — up to most recent build
🧠🤔 **complexity**	💚 simple - maps to warehouse object	💚 simple - map to warehouse concept	💛 moderate - adds logical complexity

Materializations

By default, dbt models are materialized as **"views"**. Models can be configured with a different materialization by supplying the materialized configuration parameter as shown below. Materialization configuration default file is dbt_project.yml. Alternatively, materializations can be configured directly inside of the model sql files.

dbt_project.yml

```yaml
models:
  my_project:
    events:
      # materialize all models in models/events as tables
      +materialized: table
    csvs:
      # this is redundant, and does not need to be set
      +materialized: view
```

model sql files

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

select *
from ...
```

https://docs.getdbt.com/docs/build/materializations

A dbt Demo

Materializations-Incremental

Incremental

Since in the previous lecture demo, we are using SQL script to produce the **Incremental** data loading scenario. In this demo, we will also use Incremental in dbt to make the data loading.

In addition, **Incremental** is the most important data loading method.

Mechanism:

The first time a model is run, the table is built by transforming **all rows** of source data. On **subsequent runs**, dbt transforms only **the rows in your source data that you tell dbt to filter for**, inserting them into the target table which is the table that has already been built.

Often, the rows you filter for on an incremental run will be the rows in your source data that have been created or updated since the last time dbt ran. As such, on each dbt run, your model gets built incrementally.

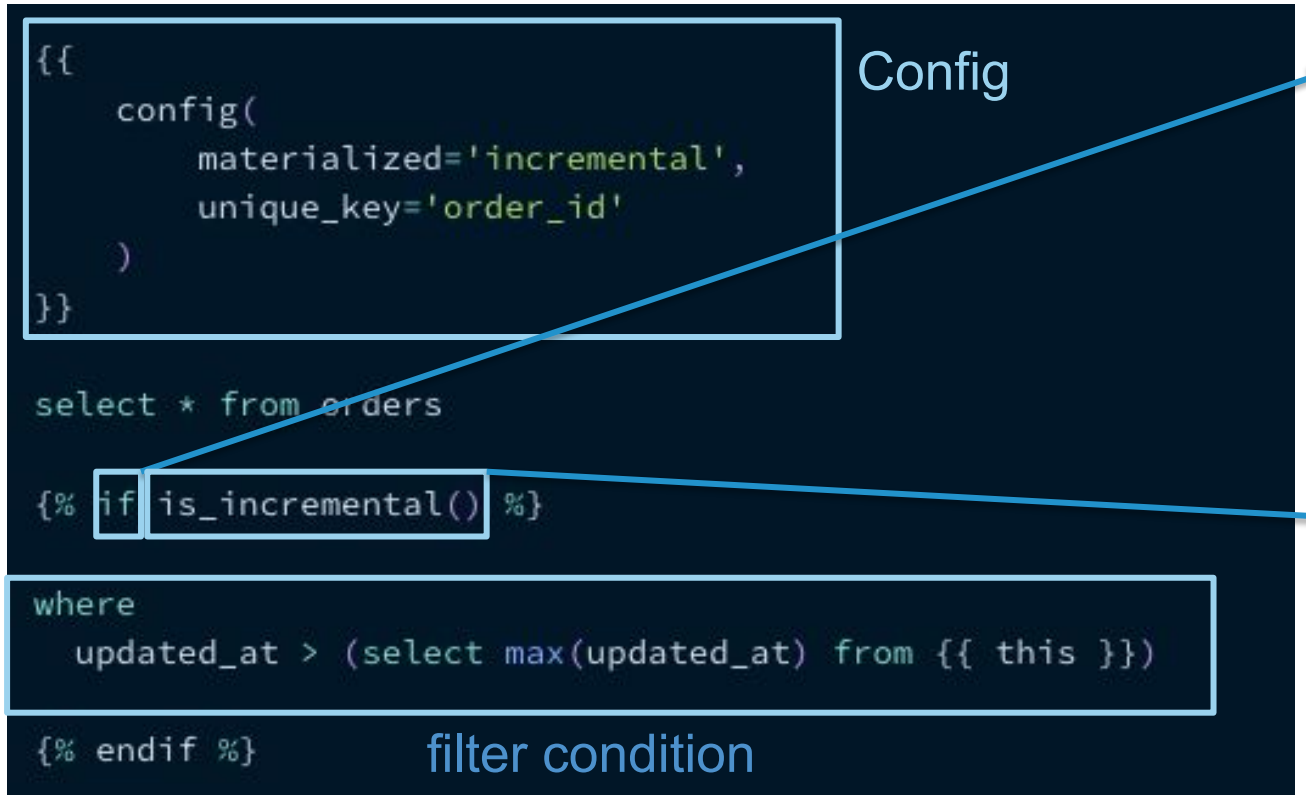
Benefits:

Using an incremental model limits the amount of data that needs to be transformed, vastly reducing the runtime of your transformations. This improves warehouse performance and reduces compute costs.

https://docs.getdbt.com/docs/build/materializations

Incremental

- Incremental models generate tables.
- Only apply transformations on the new/most recent updated data since the previous run.
- Three important elements
  - *a cutoff **filter** to select just the new or updated records*
  - *a **conditional block** that define the filter*
  - ***configuration** that define the model to be incremental and helps apply the filter when needed*



The **if** statement only works when the following are true

- materialization in config is **incremental**
- there is an **existing table** for this model to point to
- **--full-refresh** flags was not passed

- To tell dbt which rows it should transform on an incremental run, wrap valid SQL that filters for these rows in the is_incremental() macro.

Incremental

```
{{
    config(
        materialized='incremental',
        unique_key='order_id'
    )
}}

select * from orders

{% if is_incremental() %}

where
  updated_at > (select max(updated_at) from {{ this }})

{% endif %}
```

Often, you'll want to filter for "new" rows, as in, rows that have been created since the last time dbt ran this model. The best way to find the timestamp of the most recent run of this model is by checking **the most recent timestamp** in your target table. dbt makes it easy to query your target table by using the "**{{ this }}**" variable.

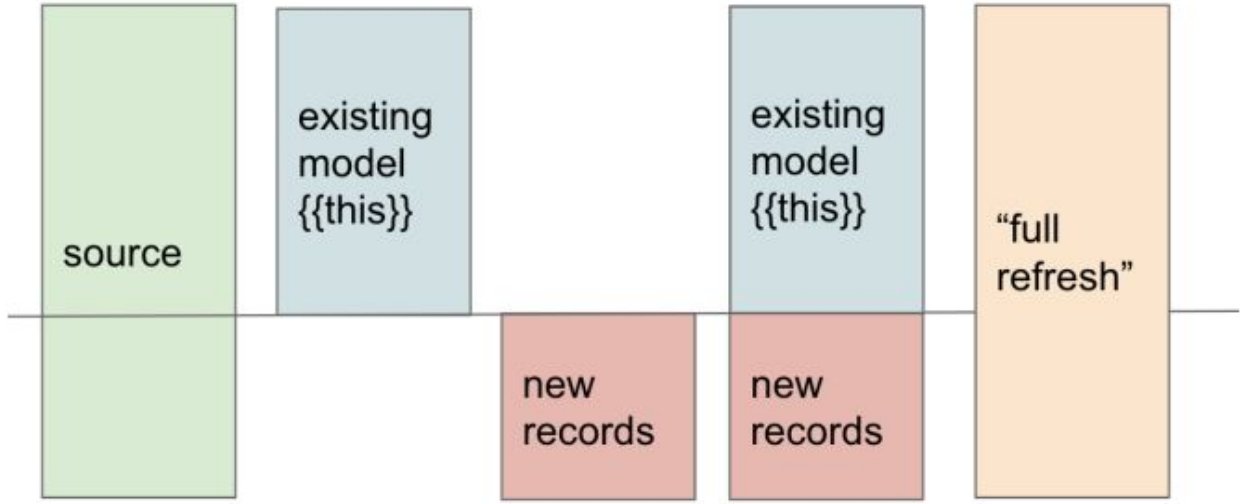
The function will make sure you find the most recent timestamp

“this” indicate you are going to filter in the target table, in the case, it is **orders** table.

Incremental

What you need to build an incremental model that only process new and updated data

- a timestamp indicating when a record was last updated
- the most recent timestamp from this table – *{{ this }}*



Incremental

Incremental Strategy(Snowflake)

The incremental_strategy config controls how dbt builds incremental models. By default, dbt will use a **merge** statement on Snowflake(for other data warehouse, please refer to this link) to refresh incremental tables.

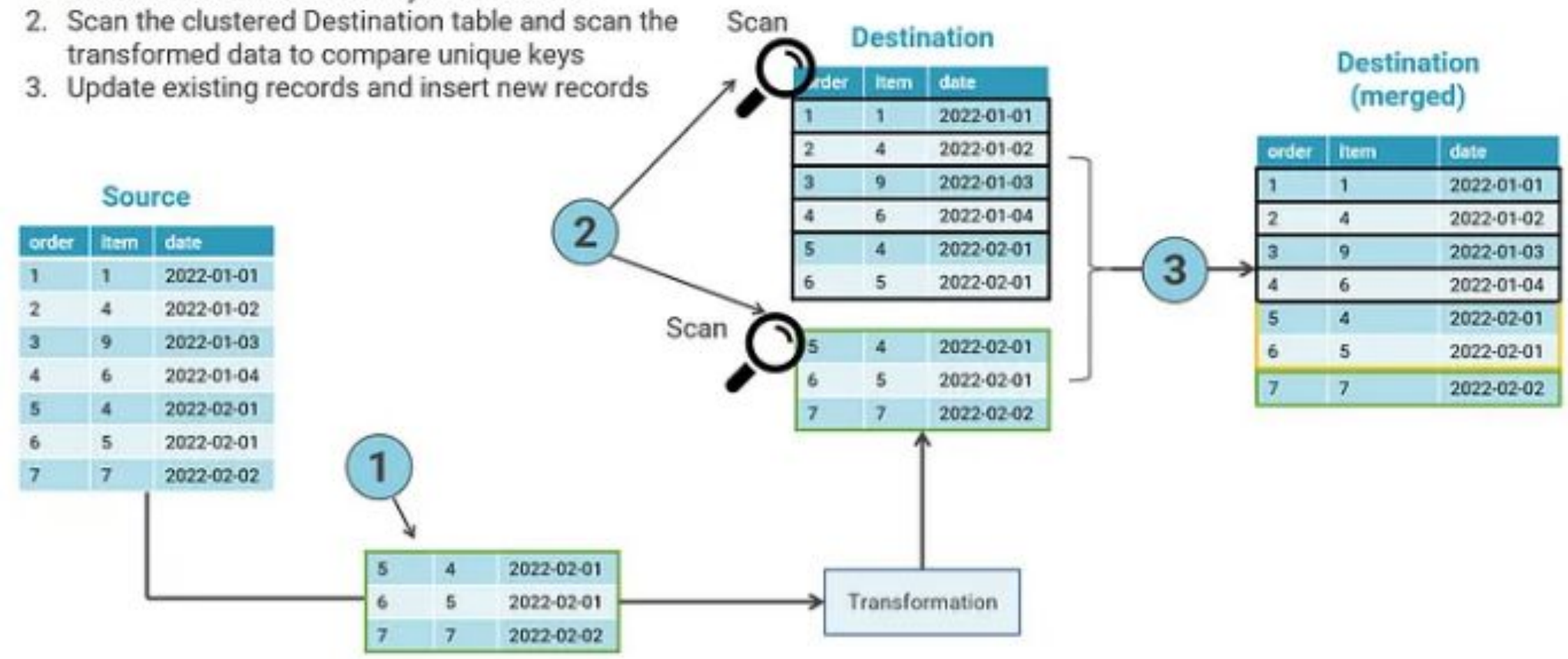
If you encounter this error when you are using **merge**, you can instruct dbt to use a two-step incremental approach by setting the incremental_strategy config for your model to **delete+insert**.

Reference: https://medium.com/indiciumtech/understanding-dbt-incremental-strategies-part-1-2-22bd97c7eeb5

Incremental

Merge:

1. Select the records filtered by where clause
2. Scan the clustered Destination table and scan the transformed data to compare unique keys
3. Update existing records and insert new records

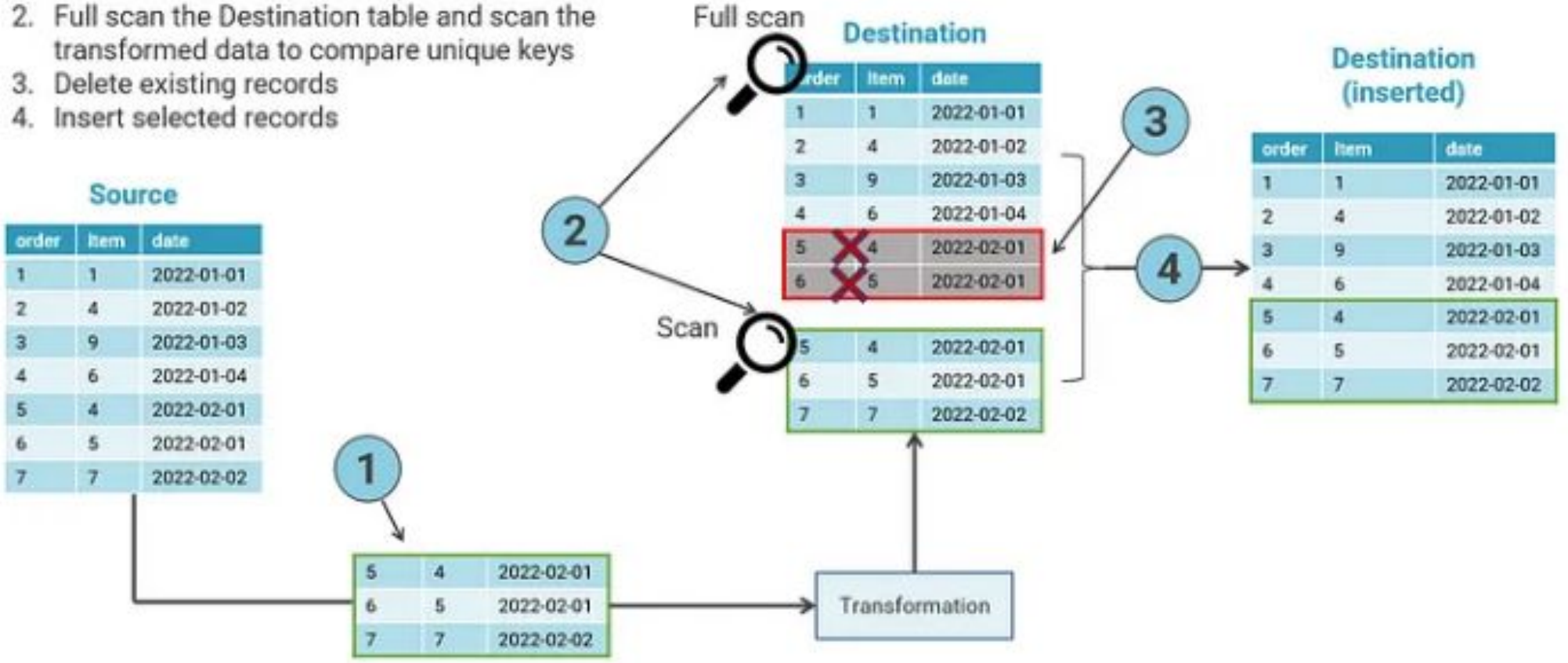


Reference: https://medium.com/indiciumtech/understanding-dbt-incremental-strategies-part-1-2-22bd97c7eeb5

Incremental

Delete+insert:

1. Select the records filtered by where clause
2. Full scan the Destination table and scan the transformed data to compare unique keys
3. Delete existing records
4. Insert selected records



Reference: https://medium.com/indiciumtech/understanding-dbt-incremental-strategies-part-1-2-22bd97c7eeb5

A dbt Demo

Source and reference

Source

Sources – {{ source('<source name>', '<table name>') }}

Using sources: Sources make it possible to name and describe the data loaded into your warehouse.

Declaring a source: Sources are defined in **.yml** files nested under a **sources:** key.

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
```

By default, schema will be the same as name. Add schema only if you want to use a source name that differs from the existing schema.

Source

Sources – {{ source('<source name>', '<table name>') }}

Selecting from a source: Once a source has been defined, it can be referenced from a model using the {{ source()}} function.

models/orders.sql

```
select
  ...

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

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

dbt will compile this to the full table name:

target/compiled/jaffle_shop/models/my_model.sql

```
select
  ...

from raw.jaffle_shop.orders

left join raw.jaffle_shop.customers using (customer_id)
```

Reference

Reference – {{ ref('<underlying model/table_name>') }}

- Reference the upstream models – tables, views, ephemerals
- Can also reference seeds – run dbt seed command before reference
- Dependency will be built automatically

upstream model

dbt-project > tpc > snapshots > customer_snapshot.sql

```
{% snapshot customer_snapshot %}

    {{
        config(
          target_schema='snapshots',
          strategy='check',

        )
    }}
```

dbt model (.sql file)

```
from {{ ref('customer_snapshot') }}
```

dbt compiled code (with depenendency)

```
from tpcds.snapshots.customer_snapshot
```

A dbt Demo

Macro and Jinja

Macros and Jinja

Jinja

In dbt, you can combine SQL with Jinja.

Jinja is a templating language. It turns dbt project into a programming environment for SQL, giving more possibility to SQL. This is the advantage of dbt comparing the traditional SQL scripts. Jinja:

- Use control structures (e.g. `if` statements and `for` loops) in SQL
- Use environment variables in your dbt project for production deployments
- Operate on the results of one query to generate another query, for example:
  - *Return a list of payment methods, in order to create a subtotal column per payment method (pivot)*
  - *Return a list of columns in two relations, and select them in the same order to make it easier to union them together*
- Abstract snippets of SQL into reusable macros — these are analogous to functions in most programming languages.

In fact, if you've used the `{{ ref() }}` function, you're already using Jinja!

Macros and Jinja

Jinja

Here's an example of a dbt model that leverages Jinja:

/models/order_payment_method_amounts.sql

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

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

The query will get compiled to:

```
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
```

Macros and Jinja

Jinja

You can recognize Jinja based on the delimiters the language uses, which we refer to as "curlies":

- **Expressions** `{{ ... }}` : Expressions are used when you want to output a string. You can use expressions to reference variables and call macros.
- **Statements** `{% ... %}` : Statements are used for control flow, for example, to set up `for` loops and `if` statements, or to define macros.
- **Comments** `{# ... #}` : Jinja comments are used to prevent the text within the comment from compiling.

Macros and Jinja

Macros

Macros in Jinja are pieces of code that can be reused multiple times – they are analogous to "functions" in other programming languages, and are extremely useful if you find yourself repeating code across multiple models.

Macros are defined in **.sql** files, typically in your **macros** directory (docs).

Macros similar like functions in Python.

```
def cents_to_dollars(column_name, scale=2)
```

Macro files can contain one or more macros.

https://docs.getdbt.com/docs/build/jinja-macros

Macros and Jinja

Macros

Macro files can contain one or more macros — This is a macro(.sql file):

macros/cents_to_dollars.sql

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

A model which uses this macro might look like:

models/stg_payments.sql

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

https://docs.getdbt.com/docs/build/jinja-macros

Macros and Jinja

Macros

This model would be compiled to::

target/compiled/models/stg_payments.sql

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

https://docs.getdbt.com/docs/build/jinja-macros

Macros

Custom Schemas

Understanding custom schemas

When first using custom schemas, it's common to assume that a model will be built in a schema that matches the schema configuration exactly, for example, a model that has the configuration **schema: marketing**, would be built in the **marketing** schema. However, dbt instead creates it in a schema like **<target_schema>_marketing** by default – there's a good reason for this!

In a typical setup of dbt, each dbt user will use a separate target schema (see Managing Environments). If dbt created models in a schema that matches a model's custom schema exactly, every dbt user would create models in the same schema.

Further, the schema that your development models are built in would be the same schema that your production models are built in! Instead, concatenating the custom schema to the target schema helps create distinct schema names, reducing naming conflicts.

Macros

Custom Schemas

How does dbt generate a model's schema name?

dbt uses a default macro called generate_schema_name to determine the name of the schema that a model should be built in.

The following code represents the default macro's logic:

```
{% macro generate_schema_name(custom_schema_name, node) -%}

    {%- set default_schema = target.schema -%}
    {%- if custom_schema_name is none -%}

        {{ default_schema }}

    {%- else -%}

        {{ default_schema }}_{{ custom_schema_name | trim }}

    {%- endif -%}

{%- endmacro %}
```

https://docs.getdbt.com/docs/build/custom-schemas

Macros

Custom Schemas

How do I use custom schemas?

Use the schema configuration key to specify a custom schema for a model. As with any configuration, you can either:

- apply this configuration to a specific model by using a config block within a model

orders.sql

```
{{ config(schema='marketing') }}

select ...
```

- Or, apply it to a subdirectory of models by specifying it in your dbt_project.yml file

dbt_project.yml

```
# models in `models/marketing/ will be rendered to the "*_marketing" schema
models:
  my_project:
    marketing:
      +schema: marketing
```

Macros

Custom Schema Configuration

You can customize schema name generation in dbt depending on your needs, such as creating a custom macro named generate_custom_schema in your project.

- The .sql file name doesn't need to align with the macro's name.

```
{% macro generate_schema_name(custom_schema_name, node) -%}

    {%- set default_schema = target.schema -%}
    {%- if custom_schema_name is none -%}

        {{ default_schema }}

    {%- else -%}

        {{ default_schema }}_{{ custom_schema_name | trim }}

    {%- endif -%}

{%- endmacro %}
```

References: https://docs.getdbt.com/docs/build/jinja-macros, https://docs.getdbt.com/docs/build/custom-schemas

A dbt Demo

Snapshot

Snapshots

Type-2 Slowly Changing Dimensions (SCDs)

SCDs: identify how *a row in a table changes over time*.

- Track order lifecycle. payment □ prepare shipping □ shipped (□ returned)
- Customer information update – address, email etc.

Snapshot: a mechanism that records changes to a mutable table over time in dbt.

- ***Timestamp strategy*** (recommended)
  - need a reliable column with timestamp/date type to track the change of the record.
  - set up **updated_at** in config (A column which represents when the source row was last updated)

- ***Check strategy***
  - only use when there is no reliable timestamp column in the table
  - set up **check_cols** in config (A list of columns to check for changes, or all to check all columns)

Snapshots

Steps to set up snapshots

1. Create a .sql file in the snapshots directory and use the following snapshot bloc to define the start and end of a snapshot.

snapshots/orders_snapshot.sql

```
{% snapshot orders_snapshot %}

{% endsnapshot %}
```

2. Write a select statement within the snapshot block. The select statement defines which records you want to track over time.

```
{% snapshot orders_snapshot %}

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

{% endsnapshot %}
```

3. Check if the source table/model includes a reliable timestamp column that indicates when a record was recent updated.
    - If yes, use timestamp strategy.
    - If no, use check strategy.

Snapshots

Steps to set up snapshots

4. Add configurations to your snapshot using config block.

timestamp strategy config

```
{% 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 %}
```

check strategy config

```
{% snapshot orders_snapshot_check %}

    {{
        config(
          target_schema='snapshots',
          strategy='check',
          unique_key='id',
          check_cols=['status', 'is_cancelled'],
        )
    }}

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

{% endsnapshot %}
```

5. Run dbt snapshot command.

6. Check the table created by snapshot in your warehouse.

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`

7. Use snapshot in your downstream models using the ref function.

SCDs: Snapshot vs. Incremental

Task:

- Set up slow changing dimension for product table using **snapshot** method
- Set up slow changing dimension for product table using **incremental** method

A dbt Demo

Test and Packages

Tests and Packages

Tests and Packages Overview

DBT Tests

- DBT allows writing tests to validate data quality and correctness.
- Powerful tool for maintaining data integrity throughout the pipeline.

DBT Packages

- DBT Packages are pre-built and shareable modules.
- Encapsulate SQL code, macros, and models for specific use cases.
- Accelerate development by reusing tested and optimized components.
- Leverage the DBT Hub to explore and integrate community-contributed packages.

Tests and Packages

Packages

1. Add a packages.yml file to your dbt project and it should be at the same level as your dbt_project.yml
2. Specify the packages needed for the project.

packages.yml ~/de-dbt/demo

```yaml
packages:
  - package: dbt-labs/dbt_utils
    version: 1.1.1

  - package: calogica/dbt_expectations
    version: [">=0.9.0", "<0.10.0"]
```

3. Run *dbt deps* command line to install the packages.

Tests and Packages

Tests – Generic Tests in dbt

- There are four generic tests in dbt : *unique*, *not_null*, *accepted_values* and *relationships*
- Generic tests are applied on column level

models.yml

```yaml
version: 2

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

Tests and Packages

Test – dbt-expectations

- Apply test on model level

! Schema.yml ~/de-dbt/demo/models

```yaml
version: 2

models:
  - name: dim_product_incr
    columns:
      - name: prod_key
        tests:
          - unique
          - not_null

  - name: fct_daily_sales
    tests:
      - dbt_expectations.expect_grouped_row_values_to_have_recent_data:
          group_by: [store_key]
          timestamp_column: cal_dt
          datepart: day
          interval: 1

      - dbt_expectations.expect_grouped_row_values_to_have_recent_data:
          group_by: [store_key]
          timestamp_column: update_time
          datepart: day
          interval: 2
```

A dbt Demo

dbt running result explanation

Demo - Model: product table

Tasks

- Create two subfolders named “staging” and “mart” within the “models” directory.
- Implement the **merge** incremental materialization strategy for inserting and updating the records.
  - Generate a .sql file named *stg_product_incr* within the “staging” subfolder. Utilize the merge incremental materialization strategy to capture the changes of *raw.product* table and add a “start_date” attribute and output the resulting table to the “stage” schema.
  - Develop a .sql file named *dim_product_incr* within the "mart" subfolder. Reference the "stg_product_incr" model, use a window function to include a "deactivate_date", create a flag named "active_status", and output the resulting table to the "entp" schema.

Demo - Model: product table

Tasks

- Apply the dbt **snapshot** technique for inserting and updating the records.
  - Generate a .sql file named *stg_product_snapshot* under the "snapshots" directory to capture changes of *raw.product* table and output the resulting table to the ”stage" schema.
  - Develop a .sql file named *dim_product_snapshot* within the "mart" subfolder. Reference the "stg_product_snapshot", use "dbt_valid_from" and "dbt_valid_to" to add "start_date", "deactivate_date", and a flag named "active_status", and output the resulting table to the "entp" schema.

Demo - Model: Sales table

Tasks

- Implement the **delete+insert** incremental materialization strategy for inserting and updating the records.
- Create a .sql file named *fct_daily_sales* inside the "mart" subfolder. Employ the delete+insert incremental materialization strategy to capture changes from the *raw.sales* table, aggregate the sales data to achieve daily granularity, and add an "update time" attribute for each record.

Incremental: delete+insert

dbt behind the scene

1st run

	PROD_KEY	PROD_NAME	START_DATE
1	657,768	Product-657768	2023-09-14 03:58:23.366
2	293,693	Product-293693	2023-09-14 03:58:23.366
3	484,597	Product-484597	2023-09-14 03:58:23.366
4	939,925	Product-939925	2023-09-14 03:58:23.366
5	234,470	Product-234470	2023-09-14 03:58:23.366

dbt compile

```
create or replace transient table dbt_demo.stage.stg_product_incr as
(   select
        prod_key,
        prod_name,
        sysdate() as start_date
    from
        dbt_demo.raw.product
)
```

delete

2nd run

	PROD_KEY	PROD_NAME	START_DATE
1	657,768	Product-657768	2023-09-14 03:58:23.366
2	293,693	Product-293693	2023-09-14 03:58:23.366
3	657,768	change-657768	2023-09-14 04:02:12.672
4	293,693	change-293693	2023-09-14 04:02:12.672
5	484,597	Product-484597	2023-09-14 04:02:12.672
6	939,925	Product-939925	2023-09-14 04:02:12.672
7	234,470	Product-234470	2023-09-14 04:02:12.672

insert

Incremental: delete+insert

Incremental: delete+insert

Step 1: dbt create temporary table from source table

	PROD_KEY	PROD_NAME	START_DATE
1	657,768	change-657768	2023-09-14 04:02:12.672
2	293,693	change-293693	2023-09-14 04:02:12.672
3	484,597	Product-484597	2023-09-14 04:02:12.672
4	939,925	Product-939925	2023-09-14 04:02:12.672
5	234,470	Product-234470	2023-09-14 04:02:12.672

Step 2: dbt delete rows from stg_product_incr that has the same unique key as the dbt temporary table

	PROD_KEY	PROD_NAME ···	START_DATE
1	657,768	Product-657768	2023-09-14 03:58:23.366
2	293,693	Product-293693	2023-09-14 03:58:23.366

Step 3: dbt insert all records from temporary table to stg_product_incr

	PROD_KEY	PROD_NAME	START_DATE
1	657,768	Product-657768	2023-09-14 03:58:23.366
2	293,693	Product-293693	2023-09-14 03:58:23.366
3	657,768	change-657768	2023-09-14 04:02:12.672
4	293,693	change-293693	2023-09-14 04:02:12.672
5	484,597	Product-484597	2023-09-14 04:02:12.672
6	939,925	Product-939925	2023-09-14 04:02:12.672
7	234,470	Product-234470	2023-09-14 04:02:12.672

Incremental: merge

dbt behind the scene

1st run

	PROD_KEY	PROD_NAME	START_DATE
1	657,768	Product-657768	2023-09-14 03:58:23.366
2	293,693	Product-293693	2023-09-14 03:58:23.366
3	484,597	Product-484597	2023-09-14 03:58:23.366
4	939,925	Product-939925	2023-09-14 03:58:23.366
5	234,470	Product-234470	2023-09-14 03:58:23.366

dbt compile

```
create or replace transient table dbt_demo.stage.stg_product_incr as
(   select
        prod_key,
        prod_name,
        sysdate() as start_date
    from
        dbt_demo.raw.product
)
```

update

2nd run

	PROD_KEY	PROD_NAME	START_DATE
1	657,768	Product-657768	2023-09-14 03:58:23.366
2	293,693	Product-293693	2023-09-14 03:58:23.366
3	657,768	change-657768	2023-09-14 04:02:12.672
4	293,693	change-293693	2023-09-14 04:02:12.672
5	484,597	Product-484597	2023-09-14 04:02:12.672
6	939,925	Product-939925	2023-09-14 04:02:12.672
7	234,470	Product-234470	2023-09-14 04:02:12.672

insert

Snapshot

behind the scene

1st run

	PROD_KEY	PROD_NAME	DBT_UPDATED_AT	DBT_VALID_FROM	DBT_VALID_TO
1	234,470	Product-234470	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	null
2	293,693	Product-293693	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	null
3	484,597	Product-484597	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	null
4	657,768	Product-657768	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	null
5	939,925	Product-939925	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	null

```
select *,
        md5(coalesce(cast(prod_key as varchar ), '')
         || '|' || coalesce(cast(to_timestamp_ntz(convert_timezone('UTC', current_timestamp())) as varchar ), '')
        ) as dbt_scd_id,
        to_timestamp_ntz(convert_timezone('UTC', current_timestamp())) as dbt_updated_at,
        to_timestamp_ntz(convert_timezone('UTC', current_timestamp())) as dbt_valid_from,
        nullif(to_timestamp_ntz(convert_timezone('UTC', current_timestamp())), to_timestamp_ntz(convert_timezone('UTC', current_timestamp()))) as dbt_valid_to
    from (



    select
        *
    from
        dbt_demo.raw.product
    order by
        prod_key

    ) sbq
```

Snapshot

behind the scene

2nd run

	PROD_KEY	PROD_NAME	DBT_UPDATED_AT	DBT_VALID_FROM	DBT_VALID_TO
1	939,925	Product-939925	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	null
2	657,768	Product-657768	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	2023-09-14 06:58:58.612
3	484,597	Product-484597	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	null
4	293,693	Product-293693	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	2023-09-14 06:58:58.612
5	234,470	Product-234470	2023-09-14 06:52:09.188	2023-09-14 06:52:09.188	null
6	657,768	change-657768	2023-09-14 06:58:58.612	2023-09-14 06:58:58.612	null
7	293,693	change-293693	2023-09-14 06:58:58.612	2023-09-14 06:58:58.612	null

Snapshot

behind the scene

2nd run

Step 1: dbt create a temporary snapshot for the source product data (raw.product)

	PROD_KEY	PROD_NAME	DBT_UPDATED_AT	DBT_VALID_FROM	··· DBT_VALID_TO	DBT_SCD_ID
1	939,925	Product-939925	2023-09-14 19:10:45.882	2023-09-14 19:10:45.882	null	2642bf5fe093e767fbe203b6acd3ca6e
2	484,597	Product-484597	2023-09-14 19:10:45.882	2023-09-14 19:10:45.882	null	c659ecad28554f500b913a51b3c7ea55
3	234,470	Product-234470	2023-09-14 19:10:45.882	2023-09-14 19:10:45.882	null	d3af2653a9c9eef6a90ea3a65150fdc4
4	657,768	change-657768	2023-09-14 19:10:45.882	2023-09-14 19:10:45.882	null	ace08cf7b1efae1ff2407cc0bd627758
5	293,693	change-293693	2023-09-14 19:10:45.882	2023-09-14 19:10:45.882	null	cb462065c75f9ff01f133b6924ade286

Step 2: dbt perform left outer join on temporary source product snapshot and the previous snapshot data stage.stg_product_snapshot to determine which data to insert

	DBT_CHANGE_TYPE	PROD_KEY	PROD_NAME	DBT_VALID_FROM	DBT_VALID_TO	PROD_KEY_2	PROD_NAME_2 ···	DBT_VALID_FROM_2	DBT_VALID_TO_2
1	insert	939,925	Product-939925	2023-09-14 19:43:44.871	null	939,925	Product-939925	2023-09-14 18:54:47.596	null
2	insert	484,597	Product-484597	2023-09-14 19:43:44.871	null	484,597	Product-484597	2023-09-14 18:54:47.596	null
3	insert	234,470	Product-234470	2023-09-14 19:43:44.871	null	234,470	Product-234470	2023-09-14 18:54:47.596	null
4	insert	657,768	change-657768	2023-09-14 19:43:44.871	null	657,768	Product-657768	2023-09-14 18:54:47.596	null
5	insert	293,693	change-293693	2023-09-14 19:43:44.871	null	293,693	Product-293693	2023-09-14 18:54:47.596	null

temp snapshot of raw.product

stage.stg_product_snapshot

Snapshot

behind the scene

```
insertions as (

    select
        'insert' as dbt_change_type,
        source_data.*

    from insertions_source_data as source_data
    left outer join snapshotted_data on snapshotted_data.dbt_unique_key = source_data.dbt_unique_key
    where snapshotted_data.dbt_unique_key is null
       or (
            snapshotted_data.dbt_unique_key is not null
        and (
            (snapshotted_data."PROD_KEY" != source_data."PROD_KEY"
    or
    (
        ((snapshotted_data."PROD_KEY" is null) and not (source_data."PROD_KEY" is null))
        or
        ((not snapshotted_data."PROD_KEY" is null) and (source_data."PROD_KEY" is null))
    ) or snapshotted_data."PROD_NAME" != source_data."PROD_NAME"
    or
    (
        ((snapshotted_data."PROD_NAME" is null) and not (source_data."PROD_NAME" is null))
        or
        ((not snapshotted_data."PROD_NAME" is null) and (source_data."PROD_NAME" is null))
    ))
        )
    )

)
```

	DBT_CHANGE_TYPE	PROD_KEY	PROD_NAME	DBT_VALID_FROM	··· DBT_VALID_TO
1	insert	293,693	change-293693	2023-09-14 20:14:23.578	null
2	insert	657,768	change-657768	2023-09-14 20:14:23.578	null

Snapshot

behind the scene

Step 3: dbt perform inner join on temporary updated source product snapshot and the previous snapshot data stage.stg_product_snapshot to determine which data to update

```
updates as (

    select
        'update' as dbt_change_type,
        source_data.*,
        snapshotted_data.dbt_scd_id

    from updates_source_data as source_data
    join snapshotted_data on snapshotted_data.dbt_unique_key = source_data.dbt_unique_key
    where (
        (snapshotted_data."PROD_KEY" != source_data."PROD_KEY"
    or
    (
        ((snapshotted_data."PROD_KEY" is null) and not (source_data."PROD_KEY" is null))
        or
        ((not snapshotted_data."PROD_KEY" is null) and (source_data."PROD_KEY" is null))
    ) or snapshotted_data."PROD_NAME" != source_data."PROD_NAME"
    or
    (
        ((snapshotted_data."PROD_NAME" is null) and not (source_data."PROD_NAME" is null))
        or
        ((not snapshotted_data."PROD_NAME" is null) and (source_data."PROD_NAME" is null))
    ))
    )
)
```

	DBT_CHANGE_TYPE ↑	PROD_KEY	PROD_NAME	DBT_VALID_FROM	DBT_VALID_TO
1	update	293,693	change-293693	2023-09-14 20:47:36.242	2023-09-14 20:47:36.242
2	update	657,768	change-657768	2023-09-14 20:47:36.242	2023-09-14 20:47:36.242

Snapshot

behind the scene

Step 4: dbt perform merge to stg_product_snapshot from the temporary dbt snapshot

```
merge into "DBT_DEMO"."STAGE"."STG_PRODUCT_SNAPSHOT" as DBT_INTERNAL_DEST
    using "DBT_DEMO"."STAGE"."STG_PRODUCT_SNAPSHOT__dbt_tmp" as DBT_INTERNAL_SOURCE
    on DBT_INTERNAL_SOURCE.dbt_scd_id = DBT_INTERNAL_DEST.dbt_scd_id

    when matched
     and DBT_INTERNAL_DEST.dbt_valid_to is null
     and DBT_INTERNAL_SOURCE.dbt_change_type in ('update', 'delete')
        then update
        set dbt_valid_to = DBT_INTERNAL_SOURCE.dbt_valid_to

    when not matched
     and DBT_INTERNAL_SOURCE.dbt_change_type = 'insert'
        then insert ("PROD_KEY", "PROD_NAME", "VOL", "WGT", "BRAND_NAME", "STATUS_CODE", "STATUS_CODE_NAME", "CATEGORY_KEY", "CATEG
"DBT_UPDATED_AT", "DBT_VALID_FROM", "DBT_VALID_TO", "DBT_SCD_ID")
        values ("PROD_KEY", "PROD_NAME", "VOL", "WGT", "BRAND_NAME", "STATUS_CODE", "STATUS_CODE_NAME", "CATEGORY_KEY", "CATEGORY_N
"DBT_VALID_FROM", "DBT_VALID_TO", "DBT_SCD_ID")

;
```

	DBT_CHANGE_TYPE	PROD_KEY	PROD_NAME	DBT_VALID_FROM	DBT_VALID_TO
1	update	293,693	change-293693	2023-09-14 20:47:36.242	2023-09-14 20:47:36.242
2	update	657,768	change-657768	2023-09-14 20:47:36.242	2023-09-14 20:47:36.242

	DBT_CHANGE_TYPE	PROD_KEY	PROD_NAME	DBT_VALID_FROM	DBT_VALID_TO
1	insert	293,693	change-293693	2023-09-14 20:14:23.578	null
2	insert	657,768	change-657768	2023-09-14 20:14:23.578	null

Summary

We have utilized both the merge incremental materialization strategy and dbt snapshot technique to enhance the efficiency of inserting and updating new records into the product table.

What distinguishes these two approaches?

- Merge Incremental Materialization Strategy:
  - Uses "merge" technique for updates.
  - Efficient for large datasets and incremental updates.
- dbt Snapshot Technique:
  - Captures data state at a specific time.
  - Ideal for historical tracking and point-in-time analysis.

✻ Choose the approach based on data volume, update frequency, and analytical requirements.

A dbt Demo

dbt schedule

dbt Schedule Options

After finishing the design of the dbt model, we need to schedule the dbt job to make sure the dbt can run for our daily ETL process. There are several ways to schedule dbt:

- Linux Cron job
  - It is the most basic schedule tool we can use. Since we run dbt on Linux, we can use cron job to schedule the dbt task. It is pretty simple but the drawback is that the cron job is not very flexible. In our project, we will use Cron job to schedule the dbt task.
- Airflow:
  - Airflow has more flexibility to schedule the jobs, it use DAG to set different tasks. Airflow can control the entire pipeline to make sure the entire pipeline is orchestrated in the same rhythm. We will learn how to use Airflow in the phase 3.
- Other schedulers:
  - Other schedulers like Control-M, Dagster, Prefect are options. Also, if your company use dbt cloud (we are using dbt core), dbt cloud has schedule functionality for you.

Self-Study

Tests and Packages (Study Materials)

Self-study Reference Materials

- Add tests to your DAG
- About tests property
- Packages
- dbt - Package hub

Documentation (self-study)

Learning Objectives

- Understand how to document models
- Learn how to generate project documentation and server
- Declare a docs block

DBT documentation Overview

- Documentations can be defined in two ways:
  - In .yml files
  - In standalone markdown files

Self-study Reference Materials

- About Documentation

Analyses, Hooks, and Exposures (self-study)

Learning Objectives

- Work with DBT hooks to manage table permissions
- Create a DBT exposure to document the dashboard

Hooks

- Hooks are predefined SQL scripts that get executed at specific times during the data build process. They can be configured at the project, subfolder, or model level.
- Hook Types:
  - on_run_start: executed at the start of dbt {run, seed, snapshot}
  - on_run_end: executed at the end of dbt {run, seed, snapshot}
  - pre-hook: executed before a model/seed/snapshot is built
  - post-hook: executed after a model/seed/snapshot is built