BI 2.0
Actionable Knowledge Intelligent Decisions

TechCEED
Exceed Yourself Everyday

dbt

Data Build Tool Training

Exceed yourself everyday

Govind.K

Govind.K

HCL

Data Build Tool Training

Here are some tips to help you follow the training guidelines effectively:

1. **Plan Ahead**: Make sure your laptop or desktop is ready, and you have a stable internet connection. Test your equipment before the session starts.
2. **Be Punctual**: Set reminders to log in 15 minutes early. This will give you time to settle in and address any technical issues.
3. **Stay Focused**: Minimize distractions by finding a quiet place to attend the training. Inform those around you that you'll be unavailable during the session.
4. **Engage Actively**: Participate in discussions and hands-on activities. This will help you retain information better and make the session more enjoyable.
5. **Use Quality Equipment**: Invest in good headphones to ensure you can hear clearly and avoid background noise.
6. **Follow Protocols**: Keep your microphone on mute unless you're speaking to avoid disrupting the session. Turn on your video when requested to make the session more interactive.
7. **Stay Organized**: Keep a notebook or digital document handy to jot down important points and questions you might have.

Data Build Tool Training

- Daily Quiz
  - MS Form Quiz will be delivered every day before the session
- Everyday Hands-on Check
  - Experience with Demo
- Capstone project and Demo
  - 2 addition days will be given to complete after the SME sessions
  - Project SPOC will be invited to evaluate
- Dreyfus Final Rating
  - Based on

    Interim Assessments(35%),

    Capstone Project (25%),

    Final Assessment MCQ based(30%),

    SME Feedback (10%)

Data Build Tool Training

- Self-Introduction - SME and Trainees..
- Update dbt skill in SPS after the training.
  - Click here to access SPS: For HCLT/INFRA: https://wf4.myhcl.com/SPS/Default.aspx
- Bidirectional Candid Feedback
- Environments details and Access
  - DBT access will be provided. Follow the mail which you received from dbt.
- Questions
- Enjoy Learning….

Data Build Tool Training

General…

- About Data & Database
- About Data warehouse (DWH)
- Difference between ETL & ELT
- About dbt
- Dbt core and dbt cloud
- dbt tool Look and feel
- register to dbt portal

Data Build Tool Training

- dbt Fundamentals
- Key Features of dbt
- dbt Languages
- Jinja, Macros, and Packages (loops and custom macros)
- Advanced Materializations
- Incremental models and snapshots
- Hooks
- Testing, Advanced Testing (custom tests)
- Advanced Deployment (CI/CD)
- Model Governance (Model Group, Model Version, Model Contracts and Access Modifiers)
- Advance Node Selection (Graph and Set Operators)
- Exposures(Dashboards, Reports, etc)
- Source Freshness(Validating Data from Upstream)
- Project Evaluator(DBT packages and more)

Database

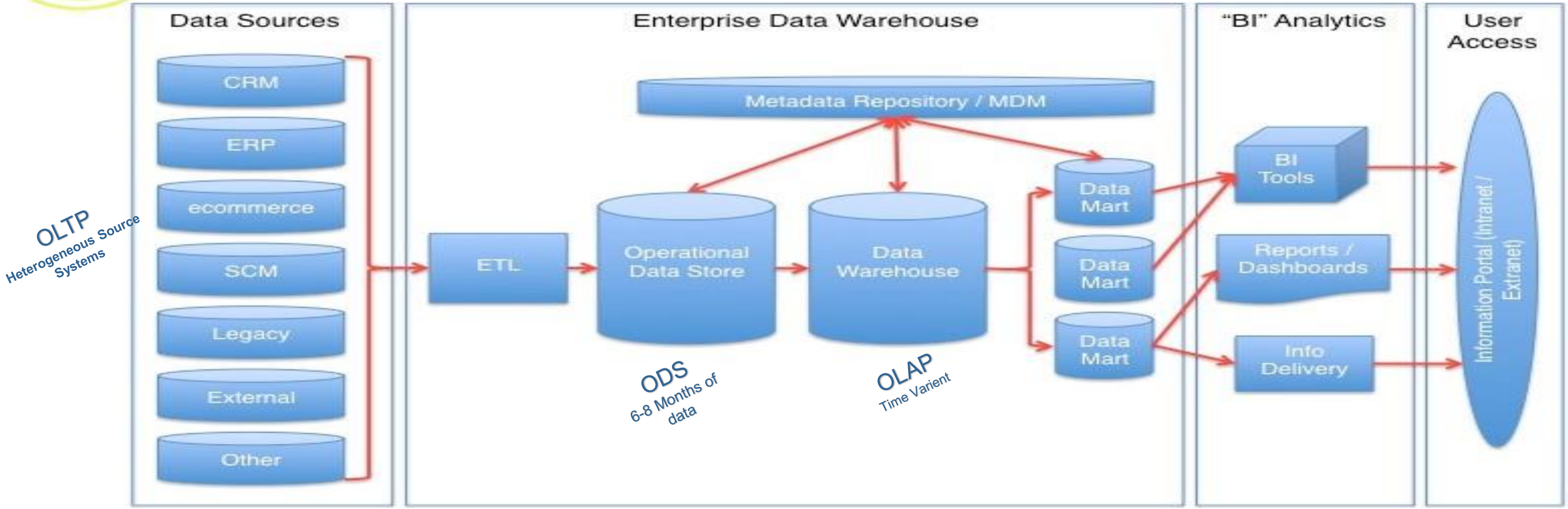
What is data?

- In simple words, data can be facts related to any object in consideration.
- For example, your name, age, height, weight, etc. are some data related to you.
- A picture, image, file, pdf, etc. can also be considered data.

What is Database?

- A database is a systematic collection of data. They support electronic storage and manipulation of data.
- A structured set of data held in a computer, especially one that is accessible in various ways.
- A database is usually controlled by a database management system (DBMS)

Data WareHouse (DWH)



What is data warehouse?

- A large store of data accumulated from a wide range of sources within a company and used to guide management decisions.
- A data warehouse (DW or DWH), also known as an enterprise data warehouse (EDW), is a system used for reporting and data analysis and is considered a core component of business intelligence. DWs are central repositories of integrated data from one or more disparate sources. They store current and historical data in one single place that are used for creating analytical reports for workers throughout the enterprise.

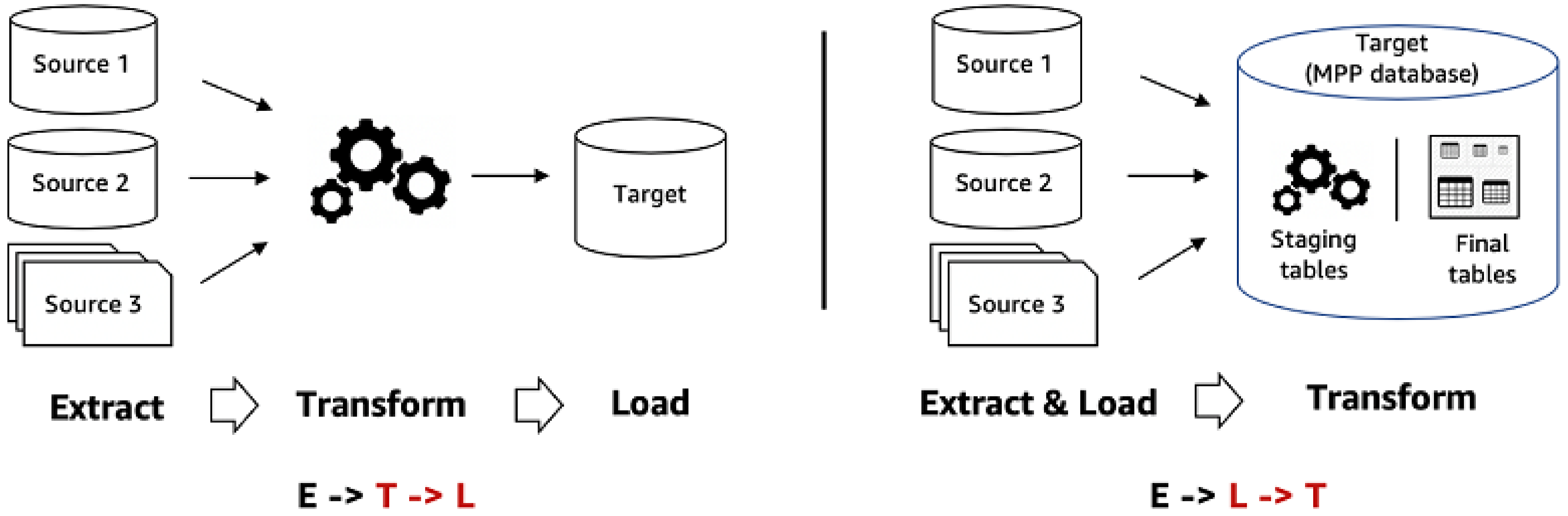
Data Warehouse Definition

The Data Warehouse is

- Subject Oriented
- Integrated
- Time variant
- Non-volatile

Collection of data in support of management decision processes

ETL vs. ELT


Source 1
Source 2
Source 3
Target
Extract
Transform
Load
E -> T -> L
Source 1
Source 2
Source 3
Target
(MPP database)
Staging
tables
Final
tables
Extract & Load
Transform
E -> L -> T

Differences: ETL vs. ELT

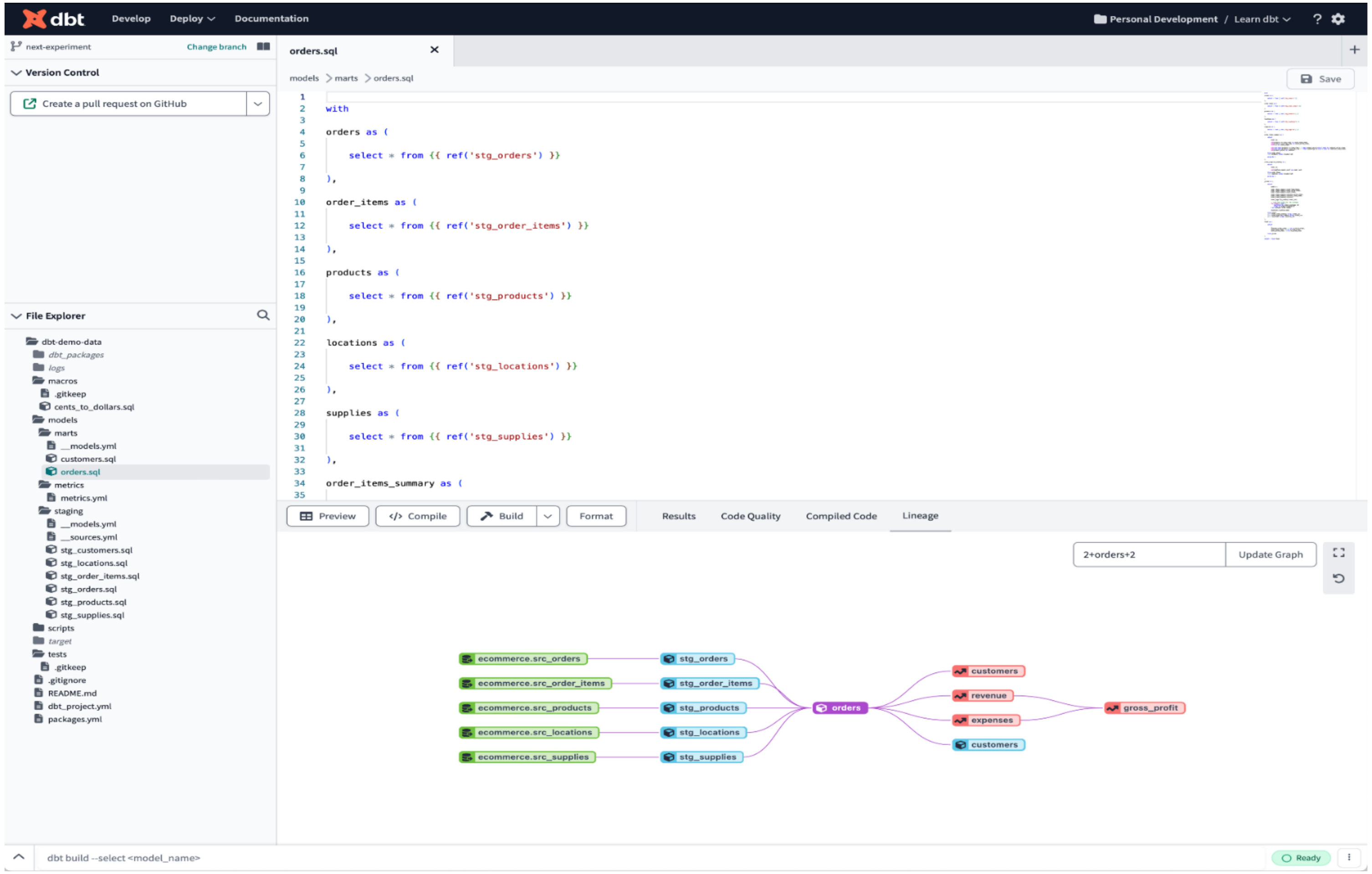
Category	ETL	ELT
Definition	Data is extracted from a source system, transformed on a secondary processing server, and loaded into a destination system.	Data is extracted from a source system, loaded into a destination system, and transformed inside the destination system.
Extract	Raw data is extracted using API connectors.	Raw data is extracted using API connectors.
Transform	Raw data is transformed on a processing server.	Raw data is transformed inside the target system.
Load	Transformed data is loaded into a destination system.	Raw data is loaded directly into the target system.
Speed	ETL is a time-intensive process; data is transformed before loading into a destination system.	ELT is faster by comparison; data is loaded directly into a destination system, and transformed in-parallel.
Code-Based Transformations	Performed on secondary server. Best for compute-intensive transformations & pre-cleansing.	Transformations performed in-database; simultaneous load & transform; speed & efficiency.
Maturity	Modern ETL has existed for 20+ years; its practices & protocols are well-known and documented.	ELT is a newer form of data integration; less documentation & experience.
Privacy	Pre-load transformation	Direct loading of data requires more privacy safeguards.
Maintenance	Secondary processing server adds to the maintenance burden.	With fewer systems, the maintenance burden is reduced.
Costs	Separate servers can create cost issues.	Simplified data stack costs less.
Requeries	Data is transformed before entering destination system; therefore raw data cannot be requeried.	Raw data is loaded directly into destination system and can be requeried endlessly.
Data Lake Compatibility	No, ETL does not have data lake compatibility.	Yes, ELT does have data lake compatibility.
Data Output	Structured (typically).	Structured, semi-structured, unstructured.
Data Volume	Ideal for small data sets with complicated transformation requirements.	Ideal for large datasets that require speed & efficiency.
Security	May require building custom applications to meet data protection requirements.	You can use built-in features of the target database to manage data protection.

What is dbt?

dbt is a SQL-first transformation workflow that lets teams quickly and collaboratively deploy analytics code following software engineering best practices like modularity, portability, CI/CD, and documentation.

Anyone on the data team can safely contribute to production-grade data pipelines.

1. **DBT allows data engineers and analysts to perform data transformations by writing SQL SELECT statements.**
2. **Internally, DBT translates these statements into tables and views, facilitating the creation of transformations on the data available in the data warehouse.**
3. **It focuses on the 'T' in ETL (Extract, Transform, Load) and integrates seamlessly with modern cloud-based data platforms.**



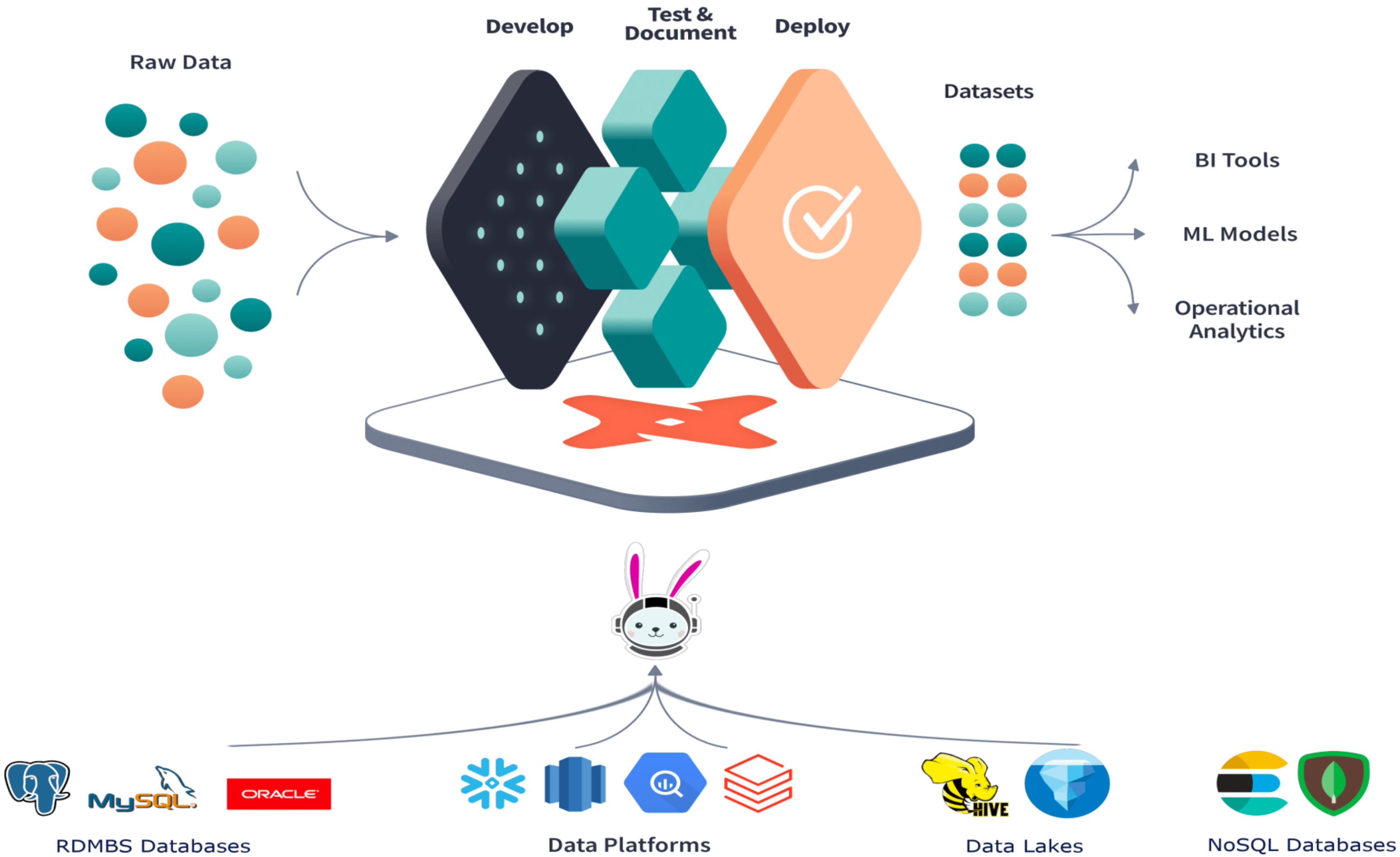
Origin of Data Build Tool

dbt started at RJMetrics in 2016 as a solution to add basic transformation capabilities to Stitch (acquired by Talend in 2018). The earliest versions of dbt allowed analysts to contribute to the data transformation process following the best practices of software engineering.

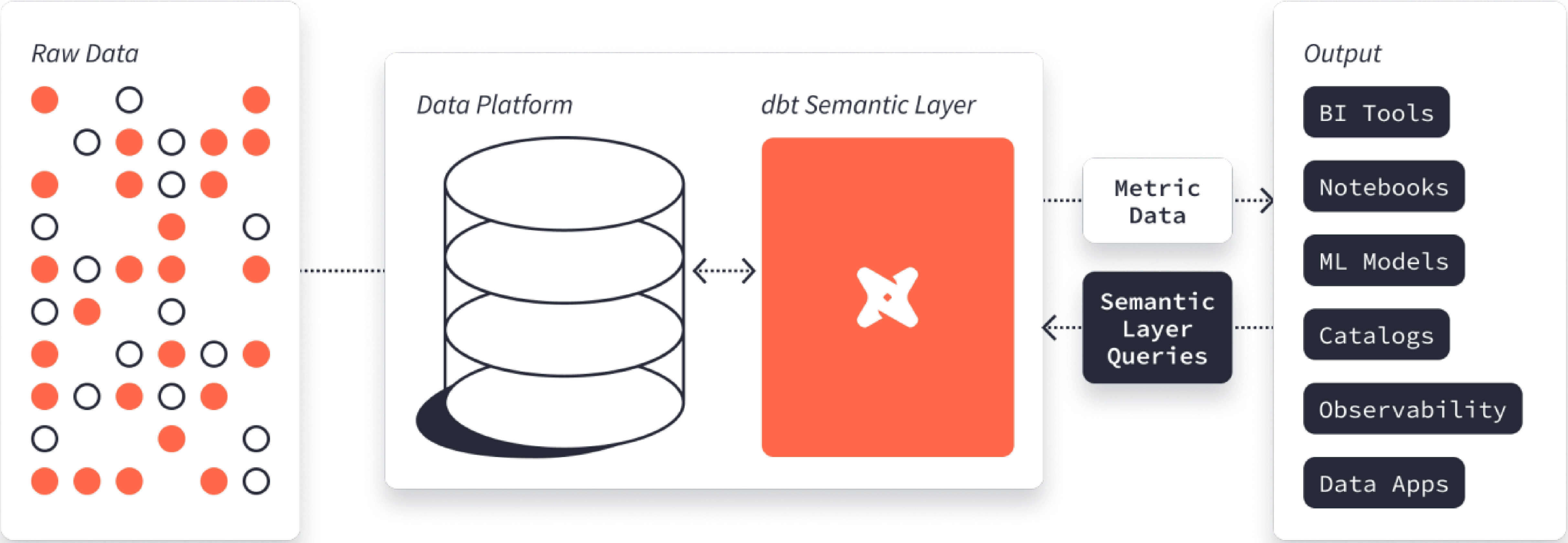
From the beginning, dbt was open source. In 2018, the dbt Labs team (then called Fishtown Analytics) released a commercial product on top of dbt Core.

Tristan Handy - Founder & CEO

DBT


Raw Data
Develop
Test & Document
Deploy
Datasets
BI Tools
ML Models
Operational Analytics
RDMBS Databases
Data Platforms
Data Lakes
NoSQL Databases

DBT


Raw Data
Data Platform
dbt Semantic Layer
Metric Data
Semantic Layer Queries
Output
BI Tools
Notebooks
ML Models
Catalogs
Observability
Data Apps

Build. Ship. Improve. Repeat.

Cloud data platform

dbt Cloud

Design **dbt Mesh**	Build **Cloud CLI & IDE**	Deploy **Scheduler & CI**
Discover **dbt Explorer**	Align **dbt Semantic Layer**	Observe **Tests & alerts**

Central active metadata framework

Develop – Test – Document

Key Features of DBT

Main Features of DBT

Develop	Test and Document	Version Control and CI/CD
Collaboration	Early Bug Detection	Accelerated Release Cycle
History Tracking	Faster Feedback Loop	Reliable and Consistent Deployments
Rollback and Recovery	Improved Code Quality	Rapid Feedback and Validation
Branching and Merging	Reduced Integration Risks	Continuous Improvement

Key Features of DBT

1. **Transformation-Centric:** DBT emphasizes data transformation, making it efficient for building data models.
2. **SQL-Driven:** Developers write SQL code to define transformations.
3. **Version Control:** DBT projects can be version-controlled using Git.
4. **Testing and Documentation:** Built-in features for testing and documenting data models.
5. **Collaborative Workflow:** Facilitates collaboration among data teams.
6. **Integration with Data Cloud Platforms:** Connects to platforms like Snowflake, Databricks, and BigQuery.

How Does DBT Work?

1. Connect to your data cloud platform (e.g., Snowflake, BigQuery) using DBT Cloud.
2. Create high-quality data models using SQL.
3. Reuse existing datasets to save time and increase productivity.
4. Benefit from collaborative workflows, self-serve analytics, and guardrails for safe production

In summary, DBT simplifies data transformations, promotes collaboration, and helps build reliable data products faster.

DBT

Who can use dbt tool?

- SQL Developer
- Data Analyst
- Data Engineers
- Business Users

How does dbt works?

- Version control
- CI/CD
- Test & Documentation

Why ELT is better?

- Scalability
- Cost effective
- Simplified Architecture

Dbt Core and dbt Cloud

The primary difference between dbt Core and dbt CLI (Command Line Interface) is how you approach working with each. In the simplest terms, dbt Core is a command-line interface (CLI), and dbt Cloud is an integrated development environment (IDE).

dbt Core is an open-source project that enables you to transform data in your data warehouse. It is a task runner and framework that helps you write, test, and deploy data transformations. dbt Core is installed locally on your computer and you can use it to run dbt commands on your local data warehouse.

dbt CLI is a command-line tool that you can use to run dbt commands on dbt Cloud. It is designed specifically for dbt Cloud's infrastructure and provides a streamlined interface for dbt Core users. dbt CLI is installed locally on your computer, and you can use it to run dbt commands on your dbt Cloud project.

Here is a table that summarizes the key differences between dbt Core and dbt CLI:

Feature	dbt Core	dbt Cloud
Type	Open-source project, Command-line tool CLI	Command-line tool, GUI
Installation	Locally on your computer	No need to install, but need license.
Usage	Run dbt commands on your local data warehouse	Run dbt commands on dbt Cloud
Features	Task runner and framework for writing, testing, and deploying data transformations	Streamlined interface for dbt Core users

Ultimately, the best choice for you will depend on your specific needs and preferences. If you are looking for a more flexible and customizable solution, then dbt Core may be a better choice for you. If you are looking for a more streamlined and integrated solution, then dbt Cloud may be a better choice for you.

Transform

Transform

- ✓ Cleaning the Data
- ✓ Transform the data
- ✓ Prepare the data
- ✓ Convert the data
- ✓ Merge the data
- ✓ Separate the data

Transformations

Conversions

- Data type (e.g. Char to Date)
- Bring data to common units
  - (Currency,Measuring Units)

Classifications

- Changing continuous values to discrete
  - ranges (e.g. Temperatures to
  - Temperature Ranges)

Splitting of fields

Merging of fields

Aggregations (e.g. Sum, Avg., Count)

Derivations (Percentages, Ratios,

Indicators)

Additive

- Average
- Aggregate

Format transformation

- Data Type Conversions
- Splitting
- Simple conversions (Money, Pounds,etc…)

Classification

Data Consistency Transformations (Sex : M/F)

Reconciliation of Duplicated data (Multiple Addresses)

DBT Languages

3 Main Languages in DBT

1. SQL – to create models and tests (.sql)
2. YAML – for configuration (.yml)
3. Jinja – to make SQL and YAML dynamic

Configuration in .sql file

E.g. `select 'Govind' as name, 'SME' as ROLE from dual`

my_first_dbt_model.sql

```
------------------
/*
    Welcome to your first dbt model!
    Did you know that you can also configure models directly
within SQL files?
    This will override configurations stated in dbt_project.yml

    Try changing "table" to "view" below
*/

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


with source_data as (

    select 1 as id
    union all
    select null as id
)

select *
from source_data

/*
    Uncomment the line below to remove records with null `id`
values
*/
-- where id is not null
```

my_second_dbt_model.sql

```
----------------------
-- Use the `ref` function to select from other
models

select *
from {{ ref('my_first_dbt_model') }}
where id = 1
```

Type __ double underscore, you will get lot of options to choose Jinja code templates…

__config

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

__ref

```
{{ ref('model_name') }}
```

Importance of YAML file

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.

- dbt uses YAML(Yet Another Markup Language) in a few different places. If you're new to YAML, it would be worth learning how arrays, dictionaries, and strings are represented.

- By default, dbt looks for the 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.

- Specify your dbt Cloud project ID in the dbt_project.yml file using project-id under the dbt-cloud config. Find your project ID in your dbt Cloud project URL: For example, in https://cloud.getdbt.com/11/projects/123456, the project ID is 123456.

- Note, you can't set up a "property" in the dbt_project.yml file if it's not a config (an example is macros). This applies to all types of resources. Refer to Configs and properties for more detail.

dbt_project.yml

```yaml
# Name your project! Project names should contain only lowercase
characters and underscores.
# A good package name should reflect your organization's
# name or the intended use of these models
name: 'g_project'
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"]
```

```yaml
target-path: "target"  # directory which will store compiled SQL
files
clean-targets:         # directories to be removed by `dbt clean`
  - "target"
  - "dbt_packages"

# In dbt, the default materialization for a model is a view. This
means, when you run dbt run or dbt build, all of your models will
be built as a view in your data platform.
# The configuration below will override this setting for models
in the example folder to instead be materialized as tables.
# Any models you add to the root of the models folder will
continue to be built as views. These settings can be overridden
in the individual model files using the `{{ config(...) }}`
macro.

models:
  g_project:
    # Applies to all files under models/RAW/
    RAW:
      +materialized: table
    STAGING:
      +materialized: table
    REPORTING:
      +materialized: view
    example:
      +materialized: view
```

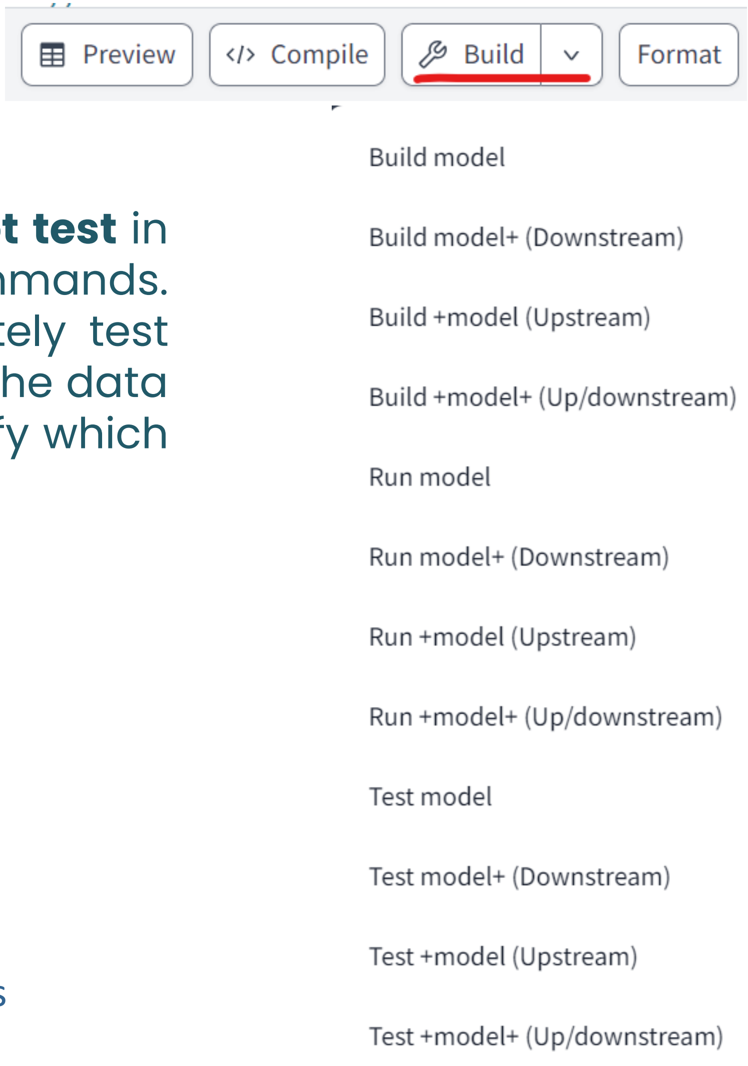
dbt build vs dbt run

dbt build vs dbt run

dbt run = execute your models.

dbt build = execute your models and test them.

dbt build is a composite command that runs both **dbt run** and **dbt test** in a single step. It's essentially a shortcut for executing both commands. Running **dbt build** will first run your models and then immediately test them. This ensures that the transformations are correct and meet the data quality checks you've defined. Just like with **dbt run**, you can specify which models to build or exclude.



RUN models

dbt run : it will run all the models

dbt run --select STG_ORDERS : it will run only selected model

dbt run -s STG_ORDERS : shortcut

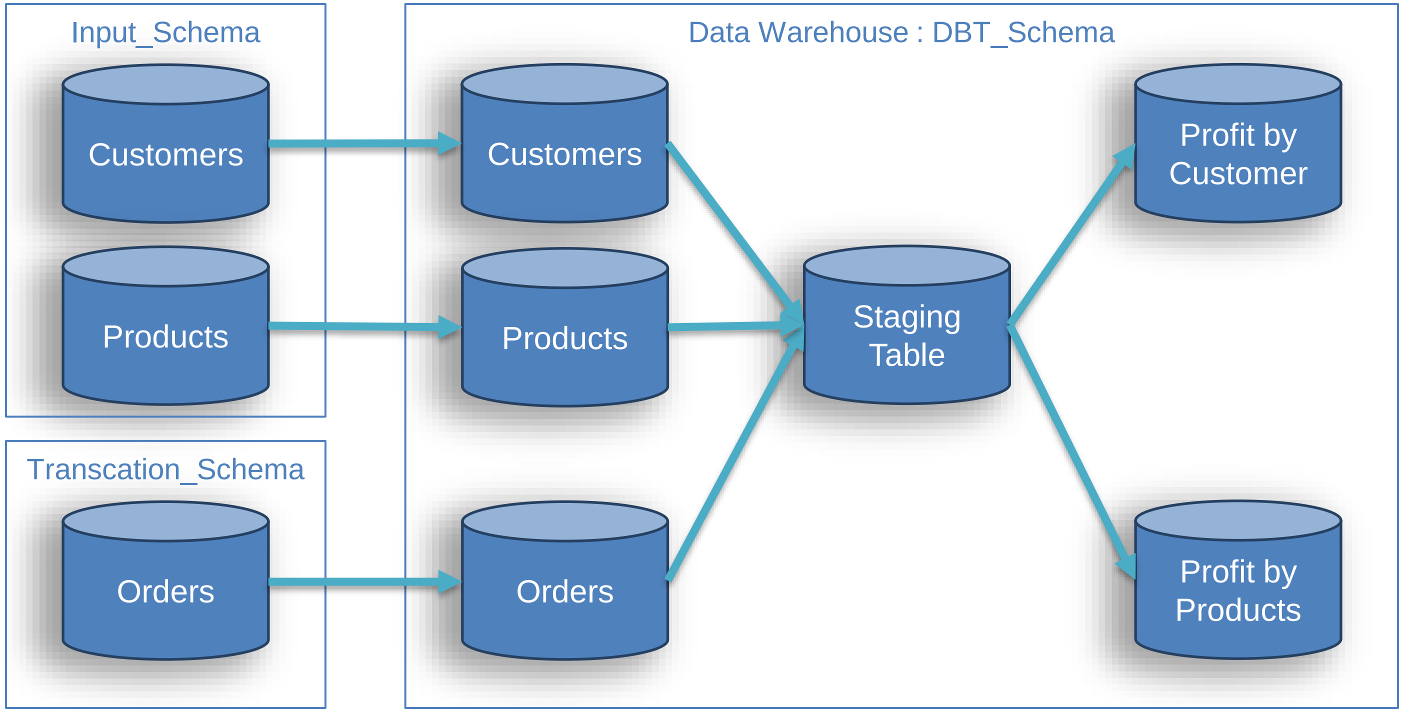
dbt run -s +STG_ORDERS : it will run only selected model with Up stream models

dbt run -s STG_ORDERS+ : it will run only selected model with down stream models

dbt run -s +STG_ORDERS+ : it will run only selected model with up/down stream models

….. You can also execute ‘build’ and ‘test’ in same way.

Small Exercise thru DBT



1. Create 2 source schemas in your Snowflake db.
2. Create 3 models in dbt to extract and load Customers and Products, and Orders data to your DWH. Create staging model with a transformation to find the order profit and create as table. Use reference to point the source tables.
3. Create Source file and pass the parameter to pick your stage table.
4. Create 2 models to create views as profit by customer and profit by product.

Sample .sql code for staging and reporting models

STG_ORDERS.SQL

```
SELECT
-- from ORDER
 O.ORDERID
,O.ORDERDATE
,O.SHIPDATE
,O.SHIPMODE
,O.CUSTOMERID AS CUST
,O.PRODUCTID AS PRODID
,O.ORDERCOSTPRICE
,O.ORDERSELLINGPRICE
--from customer
,C.CUSTOMERID
,C.CUSTOMERNAME
,C.SEGMENT
,C.COUNTRY
,C.STATE   … … …
```

```
--from PRODUCT
,P.CATEGORY
,P.PRODUCTID
,P.PRODUCTNAME
,P.SUBCATEGORY
,(O.ORDERSELLINGPRICE-O.ORDERCOSTPRICE)
AS ORDER_PROFITS
from
{{ ref('RAW_ORDERS') }} AS O
LEFT JOIN
{{ ref('RAW_PRODUCTS')}} AS P
ON O.PRODUCTID = P.PRODUCTID
LEFT JOIN
{{ ref('RAW_CUSTOMERS')}} AS C
ON O.CUSTOMERID = C.CUSTOMERID
```

report_profit_by_customer.sql

```
SELECT
CUSTOMERID, CUSTOMERNAME, SEGMENT,COUNTRY,
SUM(ORDER_PROFITS) AS PROFIT
FROM
{{ ref('STG_ORDERS') }}
  -- reporting
GROUP BY CUSTOMERID,CUSTOMERNAME,
SEGMENT,COUNTRY
```

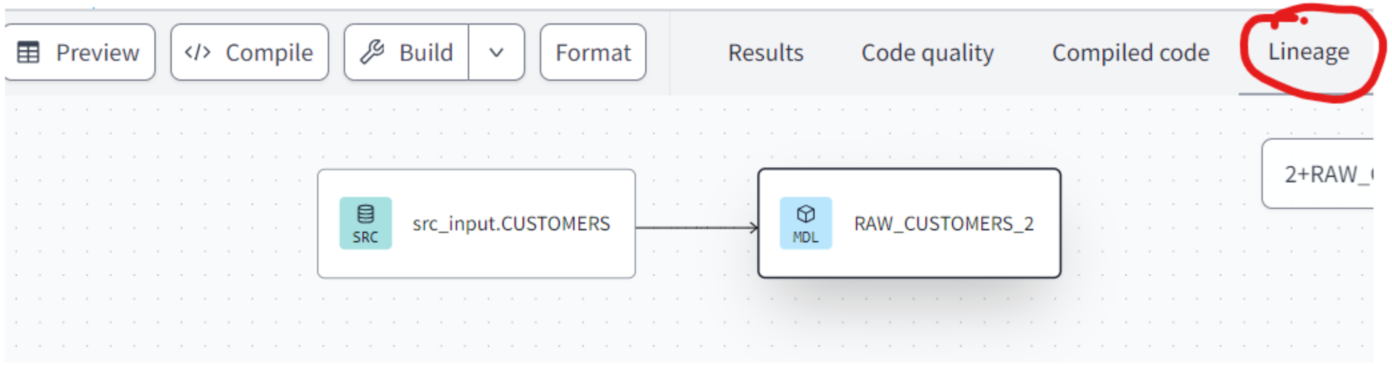
report_profit_by_product.sql

```
SELECT
PRODUCTID,PRODUCTNAME, CATEGORY,
SUM(ORDER_PROFITS) AS PROFIT
FROM
--{{ ref('STG_ORDERS') }}
{{ source('globalmart', 'STG_ORDERS') }}
GROUP BY PRODUCTID,PRODUCTNAME, CATEGORY
```

Yml examples – 2 DB config

in staples_src.yml

```
version: 2
sources:
  - name: RAW
    database: DB_STAPLES
    schema: RAW
    tables:
      - name: CUSTOMERS
      - name: PRODUCTS
      - name: ORDERS
  - name: STG
    database: DB_STAPLES
    schema: STG
    tables:
      - name: profit
```



RAW_CUSTOMERS.sql

```
SELECT * FROM
{{ source(‘RAW', 'CUSTOMERS') }}
--{{ ref('model_name') }}
--DEMO_DB.G_INPUT_SCHEMA.CUSTOMERS
```

Source - Parameterization

models › RAW › globalmart.yml

```yaml
version:  2

sources:
  - name: globalmart
    database: DEMO_DB
    schema: G_DBT_SCHEMA
    tables:
      Generate model
      - name: RAW_CUSTOMERS
        columns:
          - name: CUSTOMERID
            tests:
            - unique
      Generate model
      - name: RAW_PRODUCTS
      Generate model
      - name: RAW_ORDERS
      Generate model
      - name: STG_ORDERS
```

Call from….. __source

RAW_ORDERS_COUNT.SQL ….

```sql
SELECT COUNT(*) FROM {{
source('globalmart', 'RAW_ORDERS') }}
```

report_profit_by_product.sql ….

```sql
SELECT
PRODUCTID,PRODUCTNAME, CATEGORY,
SUM(ORDER_PROFITS) AS PROFIT
FROM
--{{ ref('STG_ORDERS') }}
{{ source('globalmart', 'STG_ORDERS') }}
GROUP BY PRODUCTID,PRODUCTNAME, CATEGORY
```

DBT Testing

What is dbt testing

Data testing is the process of ensuring data quality by validating the code that processes data **before it is deployed to production**, and dbt testing aims to prevent data quality regressions when data-processing code (dbt SQL) is written or modified.

Data testing is all about proactively identifying issues before they even happen.

DBT Testing

Approaches of testing

1. Testing in YML file under the any folder(.yml).
2. Testing under the Test folder(.sql)
3. Testing in Source file(source.yml)

The most common generic tests include:

- **Uniqueness**, which asserts that a column has no repeating values
- **Not null**, which asserts that there are no null values in a column
- **Referential integrity**, which tests that the column values have an existing relationship with a parent reference table
- **Source freshness,** which tests data against a freshness SLA based on a pre-defined timestamp
- **Accepted values**, which checks if a field always contains values from a defined list

BI 2.0 Actionable Knowledge Intelligent Decisions

TechCEED Exceed Yourself Everyday

DBT Testing - Syntex

```
version: 2

models:
  - name: <model_name>
    tests:
      - <test_name>:
          <argument_name>: <argument_value>
          config:
            <test_config>: <config-value>

    columns:
      - name: <column_name>
        tests:
          - <test_name>
          - <test_name>:
              <argument_name>: <argument_value>
              config:
                <test_config>: <config-value>
```

Some data tests require multiple columns, so it doesn't make sense to nest them under the columns: key. In this case, you can apply the data test to the model (or source, seed, or snapshot) instead:

```
version: 2

models:
- name: orders
tests:
- unique:
column_name: "country_code || '-' ||
order_id"
```


HCL

DBT Testing - predefined

Unique Testing – Not Null Testing – Accepted value testing

RAW_test.yml

```yaml
version:  2

models:
  - name: RAW_CUSTOMERS
    columns:
      - name: CUSTOMERID
        tests:
          - unique
          - not_null
      - name: CUSTOMERNAME
        tests:
          - not_null
… … …
```

```yaml
  … … …
- name: RAW_PRODUCTS
    columns:
      - name: CATEGORY
        tests:
          - accepted_values:
              values: ['Furniture', 'Office','Technology']
          - not_null
```

To run : dbt test –s RAW_PRODUCTS
dbt test –s RAW_CUSTOMERS

Test – Referential test

```
version:  2

sources:
  - name: globalmart
    database: DEMO_DB
    schema: A2_STG
     …….. … .. …
      - name: RAW_ORDERS
        columns:
          - name: CUSTOMERID
            test:
              - relationships:
                  to: ref('RAW_CUSTOMERS')
                  field: CUSTOMERID
          - name: PRODUCTID
            test:
              - relationships:
                  to: ref('RAW_PRODUCTS')
                  field: PRODUCTID
```

DBT Testing - using source file

globalmart.yml

```yaml
version: 2

sources:
  - name: globalmart
    database: DEMO_DB
    schema: A2_STG
    tables:
      - name: RAW_CUSTOMERS
        columns:
          - name: CUSTOMERID
            tests:
              - unique
      - name: RAW_PRODUCTS
      - name: RAW_ORDERS
      - name: STG_ORDERS
```

To run : dbt test -s source:globalmart

DBT Testing - Singular

Singular Testing

manual_test_unique_constraint.sql

```
WITH TMP
AS
(
    SELECT * FROM {{
ref('RAW_CUSTOMERS') }}
)

select
     CUSTOMERID, COUNT(*)
from TMP
group by CUSTOMERID
having COUNT(*) > 1
```

To run : dbt test –s RAW_CUSTOMERS

Business rule validation

```
-- tests/order_amt_test.sql
SELECT *
FROM {{ ref('RAW_ORDERS') }}
WHERE ORDERSELLINGPRICE < 0
```

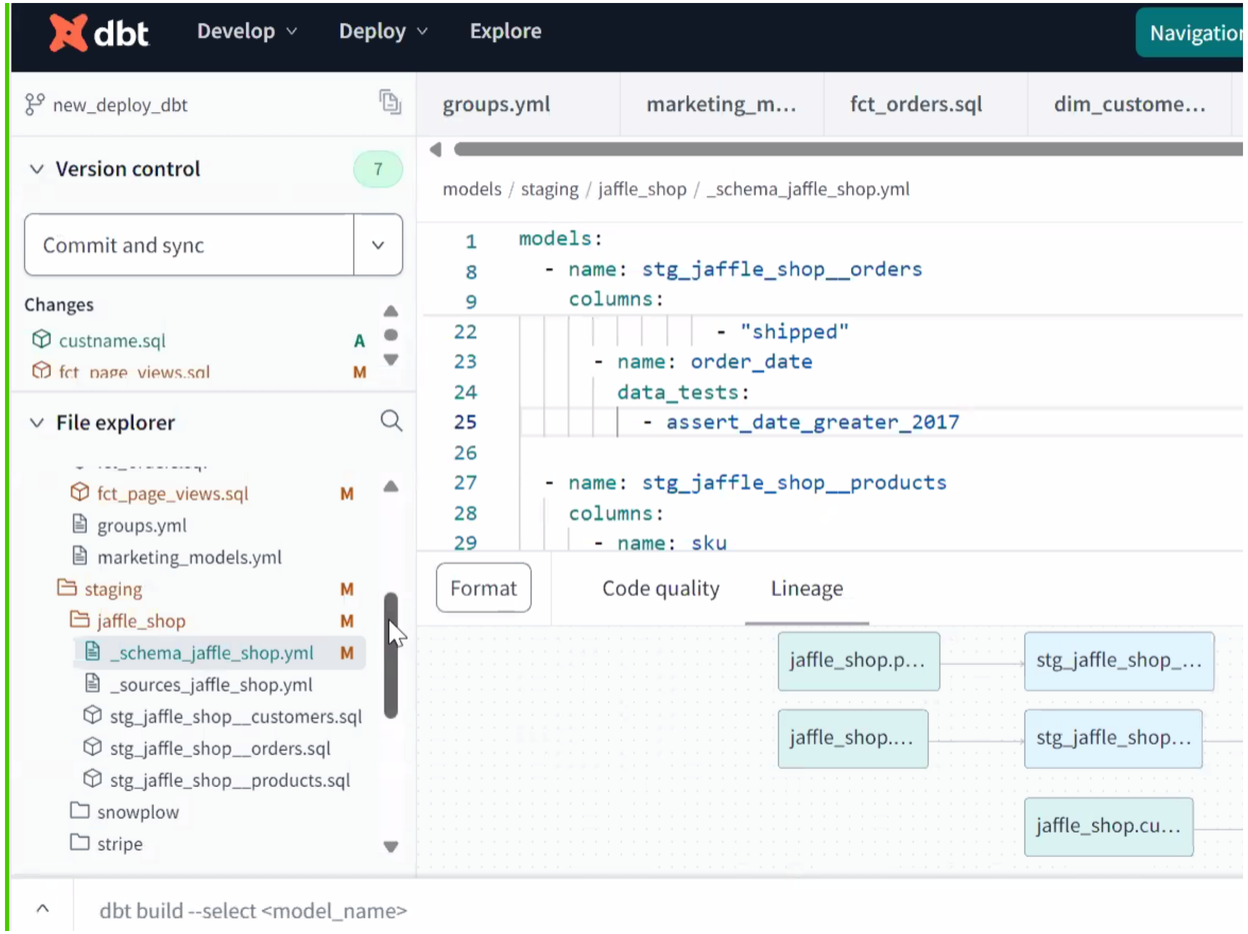
Run : dbt test -s order_amt_test.sql
Or dbt test –s RAW_ORDERS

DBT Testing - Generic

Generic Testing

Tests/**generic**/assert_date_greater_2017.sql
Must create under generic.

```
{% test not_null(model, column_name) %}
   select *
   from {{ model }}
   where {{ column_name }} is null
{% endtest %}
```



Documentation with DBT

Good documentation for your dbt models will help downstream consumers discover and understand the datasets which you curate for them.
dbt provides a way to generate documentation for your dbt project and render it as a website.

The documentation for your project includes:
•**Information about your project**: including model code, a DAG of your project, any tests you've added to a column, and more.
•**Information about your data warehouse**: including column data types, and table sizes. This information is generated by running queries against the information schema.

- You can generate a documentation site for your project (with or without descriptions) using the CLI.
- First, run **dbt docs generate** — this command tells dbt to compile relevant information about your dbt project and warehouse into manifest.json and catalog.json files respectively. To see the documentation for all columns and not just columns described in your project, ensure that you have created the models with dbt run beforehand.
- Then, run **dbt docs serve** to use these .json files to populate a local website.

Documentation with DBT

stg_orders.md

```
{% docs shipmode %}
```

Govind markdown file : `One of the following values:`

```
| SHIPMODE       | Definition                                          |
| -------------- | --------------------------------------------------- |
| First Class    | Orders are shipped via first class with Courier     |
| Second Class   | Orders are shipped via second class with Courier    |
| Standard Class | Orders are shipped via standard class with Courier  |
| Same day       | Orders are personally shipped via globalmart team   |
```

This is my markdown file to give Definition to Shipment mode

```
{% enddocs %}
```

stg_orders_doc.yml or source yml file

```yaml
version: 2

models:
  - name: STG_ORDERS
        description: it is stg_orders table. descrtiption in globalmart_src yaml file. It has all 3 tables columns
        columns:
          - name: orderid
            description: this is order id primary key and it is from orders table
          - name: shipmode
            description: '{{ doc("shipmode")}}'
```

Jinja code

Jinja code

{% %}	Control Statements	{% set variable_a = 10 %} {% if name == "Bard": %} ..... {% endif %} {% for i in range(variable_a) %} ..... {% endfor %}
{{ }}	Expressions	{{ 10 + 20 }} {{ variable_a }}
	Texts	SELECT UNION
{# #}	Comments	{# This is a comment #}

Jinja code samples

Variables in Jinja .sql

```
Ex 1.
{%- set a = 'Welcome' -%}
--'{{a}}',
select '{{a}} '|| customername as name from
{{ref('RAW_CUSTOMERS') }}

To run  :  preview or run the model
----------
```

Case when example .sql

```
Ex 2.
{% set a,x,y = 'Welcome To DBT', 'Texas',10  %}

select
    '{{a}} ' || customername || {{y}}  as "Cust
name",
    case
        when state = '{{x}}'
        then    'My Texas'
        else state
    end as statename
from {{ ref("RAW_CUSTOMERS") }}
```

```
Ex 3.
```

IF Else example .sql

```
Ex 4.
{%- set myscore=67-%}
{%- set passingscore=60-%}

{% if myscore>passingscore%}
you have passed the exam
{%-else-%}
you have failed
{%endif%}

To check the output : compile the code.
```

Jinja code samples

Ex 4. For loop with list

```
{%- set
country=['usa','uk','india','netherlands'] -%}
{%- for i in country -%}
{{ i | capitalize}}
{% endfor %}
```

To check the output : compile the code.

Ex 5. For Numbers

```
{%- for i in range(1, 11) -%}

    No.:  {{ i }}

{% endfor %}
```

Example of a dbt model that leverages Jinja

Case When in select statement order_payment_method_amounts.sql

Ex 5.

```
{% set shipmodes = ["Second Class", "Standard Class", "First Class", "Same Day", "Unknown"] %}
select productid,
    {% for shipmode in shipmodes %}
    sum(case when shipmode = '{{shipmode}}' then ORDERCOSTPRICE end) as "{{shipmode}}_amount",
    {% endfor %}
    sum(ORDERCOSTPRICE) as total_amount
from db_staples.raw.orders
group by 1
```

This query will get compiled to:

```
select productid,
    sum(case when shipmode = 'Second Class' then ORDERCOSTPRICE end) as "Second Class_amount",
    sum(case when shipmode = 'Standard Class' then ORDERCOSTPRICE end) as "Standard Class_amount",
    sum(case when shipmode = 'First Class' then ORDERCOSTPRICE end) as "First Class_amount",
    sum(case when shipmode = 'Same Day' then ORDERCOSTPRICE end) as "Same Day_amount",
    sum(case when shipmode = 'Unknown' then ORDERCOSTPRICE end) as "Unknown_amount",
    sum(ORDERCOSTPRICE) as total_amount
from db_staples.raw.orders
group by 1
```

Macro

BI 2.0 Actionable Knowledge Intelligent Decisions

TechCEED Exceed Yourself Everyday

DRY : Don’t Repeat Yourself

```
--macro / profit_calc.sql
{% macro profit_calc() %}

( ORDERSELLINGPRICE - ORDERCOSTPRICE  )

{% endmacro %}

-- Calling macros…
select
    CUSTOMERID,
    PRODUCTID,
    ORDERSELLINGPRICE,
    ORDERCOSTPRICE,
    {{profit_calc()}} as profit
from {{ ref('RAW_ORDERS') }}
```

```
--macro with arguments
{% macro profit_calc(a,b) %}

( {{a}} - {{b}})

{% endmacro %}


select
    CUSTOMERID,
    PRODUCTID,
    ORDERSELLINGPRICE,
    ORDERCOSTPRICE,
    {{profit_calc('ORDERSELLINGPRICE’,
        'ORDERCOSTPRICE')}} as profit
from {{ ref('RAW_ORDERS') }}
```

Packages

Hub.getdbt.com

Dbt_utils

Installation

dbt version required: `>=1.3.0, <2.0.0`

Include the following in your `packages.yml` file:

```
packages: -
package: dbt-labs/dbt_utils version: 1.2.0
```

Run **dbt deps** to install the package (Dependencies)

FYI…

MD5, or Message Digest Algorithm 5, is a cryptographic hash function that can be used to create surrogate keys in database tables

```
Select 'Govind', MD5('Govind') as MD5_Code
```

```
SELECT
{{
dbt_utils.generate_surrogate_key(['O.ORDERID','C.
CUSTOMERID','P.PRODUCTID'])}} as surrogatekey
,O.ORDERID
,C.CUSTOMERID
,P.PRODUCTID
,(O.ORDERSELLINGPRICE-O.ORDERCOSTPRICE) AS
ORDER_PROFITS
, {{ dbt_utils.safe_divide('O.ORDERSELLINGPRICE',
'O.ORDERCOSTPRICE') }} as safedivide
--, O.ORDERSELLINGPRICE / 0  AS safedivideby0
from
{{ ref('raw_orders') }} AS O
LEFT JOIN
{{ ref('raw_products')}} AS P
ON O.PRODUCTID = P.PRODUCTID
LEFT JOIN
{{ ref('raw_customers')}} AS C
ON O.CUSTOMERID = C.CUSTOMERID
```

Packages ...

generate_series (source)
This macro implements a cross-database mechanism to generate an arbitrarily long list of numbers. Specify the maximum number you'd like in your list and it will create a 1-indexed SQL result set.

Usage:

```
{{ dbt_utils.generate_series(upper_bound=1000) }}
```

safe_divide (source)
This macro performs division but returns null if the denominator is 0.

Args:

numerator (required): The number or SQL expression you want to divide.
denominator (required): The number or SQL expression you want to divide by.
Usage:

```
{{ dbt_utils.safe_divide('numerator', 'denominator') }}
```

Seeds

dbt seeds are files that contain static data you load into your data warehouse. These files are typically CSVs, so they're easy to create, edit, and version control. They're in a simple format so you can manage your static data with the same tools and processes you use for your code.

Seeds >> Delivery_team.csv

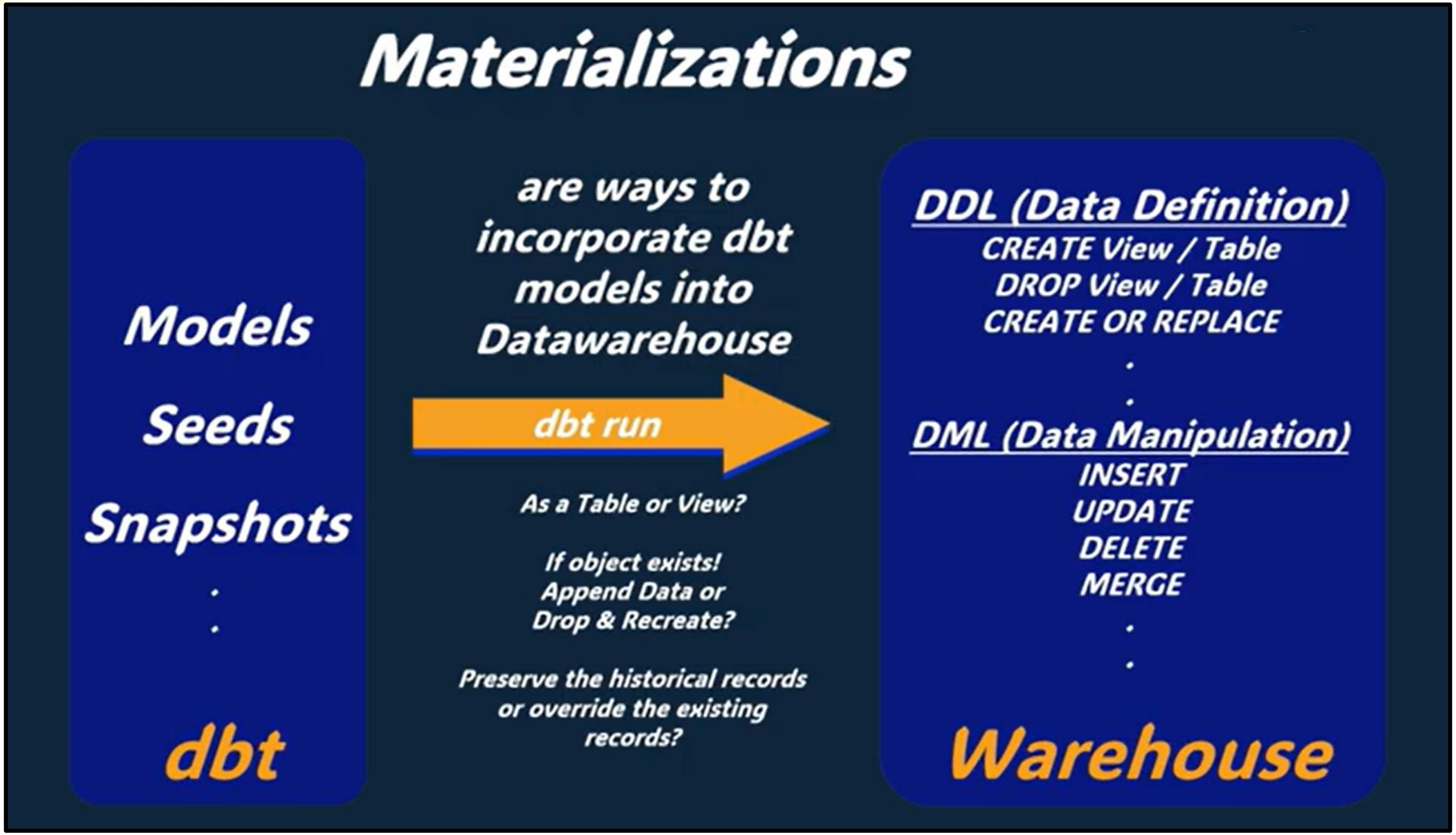
shipmode,delivery_team
First Class, RHL_couriers
Second Class, RHL_couriers
Standard Class, RHL_couriers
Same day, Globalmart Team

Run : dbt seed to insert the data in to your schema directly.

To use the data you can use __ref function…

```
select * from {{ ref('delivery_team') }}
Or
select * from db_staples.stg.delivery_team
```

Materializations



Materializations

Materialization types

1. materialized='view“
2. materialized='table'
3. materialized='ephemeral'
4. materialized='incremental’

5. {% snapshot snapshot_name %}

   ....

   {% endsnapshot %}

Materializations

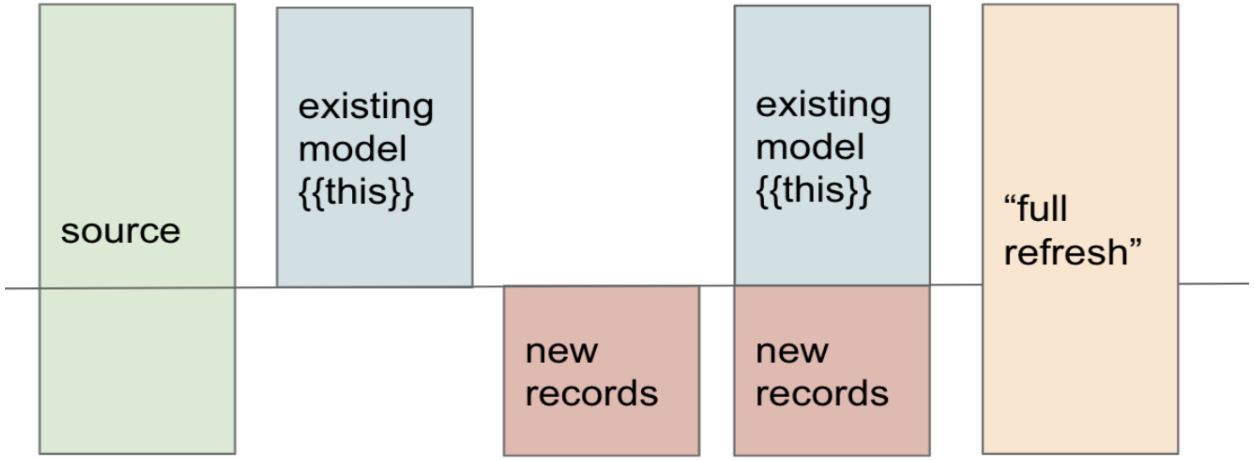
Table Vs View

	Table	View
Heavy Transformation Logics	✓	
Improve Performance	✓	
Lightweight Transformation logics		✓
Reduce Storage Cost		✓

Materialization : Incremental

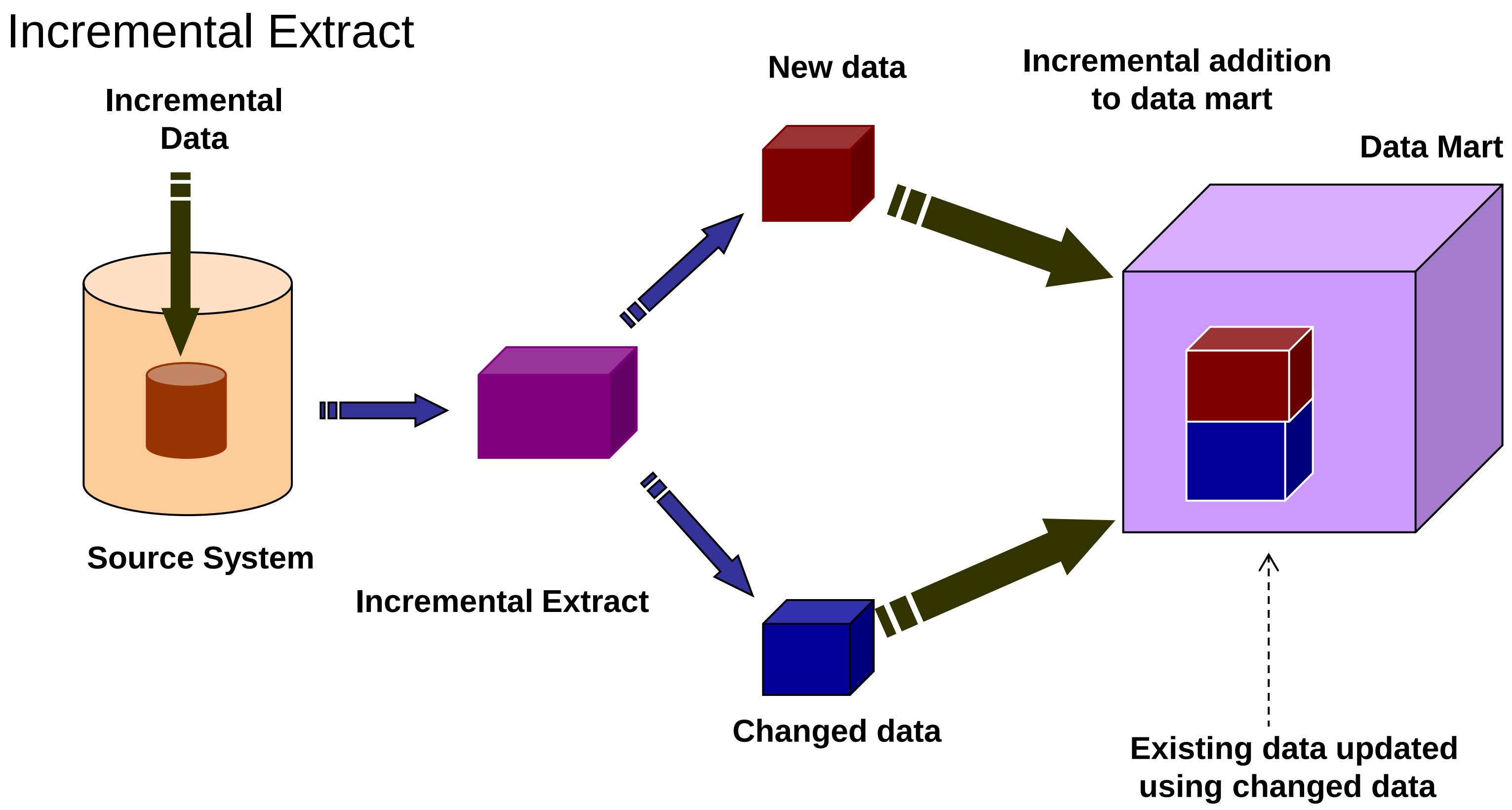
The **first time** a model is run, the table is built by transforming **all rows** of source data. On subsequent runs, dbt transforms only the rows in your source data that you tell dbt to filter for, **inserting** them into the target table which is the **table** that has already been built.

Incremental models in dbt is a materialization strategy designed to efficiently update your data warehouse tables by only transforming and loading new or changed data since the last run. Instead of processing your entire dataset every time, incremental models append or update only the new rows, significantly reducing the time and resources required for your data transformations.



- Significantly reduce the build time by just transforming new records.
- Useful for large datasets, where the cost of processing the entire dataset is high.
- Merge statement is used to insert new records and update (Delete then Insert) existing records.

Materialization : Incremental



Materialization : Incremental

Understand the is_incremental() macro
The is_incremental() macro powers incremental materializations. It will return True if *all* of the following conditions are met:

- The model must already exist in the database
- The destination table already exists in the database
- The full-refresh flag *is not* passed
- The running model is configured with materialized='incremental'

Note that the SQL in your model needs to be valid whether is_incremental() evaluates to True or False.

dbt makes it easy to query your target table by using the "[{{ this }}]" variable.

Incremental models > incr > merge_unique_key

```
{{ config(
    materialized = 'incremental',
    incremental_strategy = 'merge’,
    full_refresh = false,
    unique_key = 'id',
) }}

{% if not is_incremental() %}

select cast(1 as bigint) as id, 'hello' as msg
union all
select cast(2 as bigint) as id, 'goodbye' as msg

{% else %}

select cast(2 as bigint) as id, 'yo' as msg
union all
select cast(3 as bigint) as id, 'anyway' as msg

{% endif %}
```

Materialization : Incremental

Incremental sample

models > incr > sal_process.sql

```
{{
    config(
      materialized='incremental’,
      incremental_strategy = 'merge'
      unique_key = 'empid'
    )
}}

SELECT
 * from demo_db.a_sep24_schema.sal

{% if is_incremental() %}

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

{% endif %}
```

Run : dbt run -s sal_process.sql

Sample data SQL scripts for Incremental and snapshot

```
drop table demo_db.a_sep24_schema.sal;

create or replace TABLE demo_db.a_sep24_schema.sal (empid number(5),empname VARCHAR(30),sal number(8,2),modified_date TIMESTAMPTZ
DEFAULT CURRENT_TIMESTAMP() )


insert into demo_db.a_sep24_schema.sal values (100,'Govind',10000, CURRENT_TIMESTAMP());
insert into demo_db.a_sep24_schema.sal values (200,'Raja',20000,CURRENT_TIMESTAMP());
insert into demo_db.a_sep24_schema.sal values (300,'Vijay',30000,CURRENT_TIMESTAMP());
insert into demo_db.a_sep24_schema.sal values (400,'x',40000,CURRENT_TIMESTAMP());
-----
insert into demo_db.a_sep24_schema.sal values (500,'Bala',50000,CURRENT_TIMESTAMP());
insert into demo_db.a_sep24_schema.sal values (700,'y',70000,CURRENT_TIMESTAMP());
---------
insert into demo_db.a_sep24_schema.sal values (300,'Vijay',50000,CURRENT_TIMESTAMP());
insert into demo_db.a_sep24_schema.sal values (400,'x',60000,CURRENT_TIMESTAMP());


SELECT  * from demo_db.a_sep24_schema.sal_process order by modified_date;

select max(modified_date) from demo_db.a_sep24_schema.sal_process;
SELECT  * from demo_db.a_sep24_schema.sal;
SELECT  * from demo_db.a_sep24_schema.sal where modified_date >= (select max(modified_date) from
demo_db.a_sep24_schema.sal_process);

SELECT  * from DEMO_DB.G_SNAPSHOTS.snaps_sal_process order by modified_date;

delete from demo_db.a_sep24_schema.sal_process;
delete from demo_db.a_sep24_schema.sal where empname = 'x';
delete from DEMO_DB.G_SNAPSHOTS.snaps_sal_process;
```

Snapshot

snapshots

Analysts often need to "look back in time" at previous data states in their mutable tables. While some source data systems are built in a way that makes accessing historical data possible, this is not always the case. dbt provides a mechanism, snapshots, which records changes to a mutable table over time.

Snapshots implement type-2 Slowly Changing Dimensions over mutable source tables. These Slowly Changing Dimensions (or SCDs) identify how a row in a table changes over time.

Snapshot example

```
--snapshot > snaps_sal_process.sql

{% snapshot snaps_sal_process %}

{{
    config(
        target_schema = 'G_SNAPSHOTS',
        unique_key='empid',
        strategy='timestamp',
        updated_at='modified_date',
        invalidate_hard_deletes=True
    )
}}

select * from demo_db.a_sep24_schema.sal

{% endsnapshot %}
```

Run : dbt snapshot

Append:

1. Select the records filtered by where clause
2. Insert selected records

Destination

order	item	date
1	1	2022-01-01
2	4	2022-01-02
3	9	2022-01-03
4	6	2022-01-04
5	4	2022-02-01
6	5	2022-02-01

5	4	2022-02-01
6	5	2022-02-01
7	7	2022-02-02

2

Destination (inserted)

order	item	date
1	1	2022-01-01
2	4	2022-01-02
3	9	2022-01-03
4	6	2022-01-04
5	4	2022-02-01
6	5	2022-02-01
5	4	2022-02-01
6	5	2022-02-01
7	7	2022-02-02

Source

order	item	date
1	1	2022-01-01
2	4	2022-01-02
3	9	2022-01-03
4	6	2022-01-04
5	4	2022-02-01
6	5	2022-02-01
7	7	2022-02-02

1

5	4	2022-02-01
6	5	2022-02-01
7	7	2022-02-02

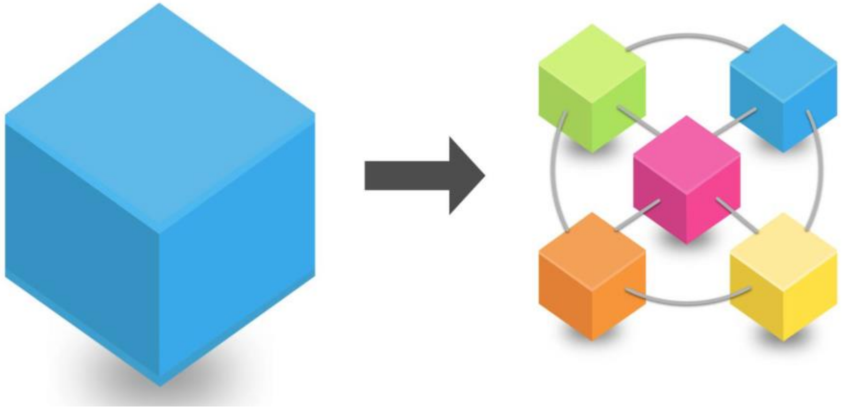
Transformation

When you have 5k+ models, it's hard to…

- Discover. Find the right models to reference or troubleshoot
- Collaborate. Create clear owners, handoffs, and interfaces between contributors
- Maintain quality. Avoid accidentally breaking downstream models

Good news! This problem has been solved before

- Software engineering pattern: Microservice architecture.
- Individual service components act as building blocks
- Each service has a well-defined output
- Building blocks are treated like products: specified owners, access, and support expectations



What is a part of Model Governance?

- Model Groups & Access Modifiers
- Model Contracts
- Model Versioning

Model Groups & Access Modifiers Defined

- **Model Groups:** Models that are related to one another and owned by a specific team - a way for a business to organize models based on ownership (e.g. finance, marketing, employee data, etc.) rather than stage of development (e.g. staging, intermediate, etc.)

- **Model Access Modifiers:** Which **models** can access (reference) a specific model

Creating Model Groups

- **Model Groups** will be designated within a yaml file nested under a **groups:** key
- Under the **groups:** key, you will be able to name the group and add owner information
  - At least name or email is required – additional properties are allowed (slack, github, etc.)

```
--   _groups.yml file

groups:
  - name: marketing
    owner:
      name: Person's Name
      email: marketing_team@jaffle.com
  - name: finance
    owner:
      name: finance's group
```

demo

Assigning Model Groups

- To assign **Model Groups** to a model, add the **group:** property to the model yaml file
- Each model can *only* belong to one group

```
models:
- name: stg_jaffle_shop__customers
group: finance
- name: stg_jaffle_shop__orders
group: finance
models:
- name: dim_customers
group: marketing
```

Access Modifiers

- Which models can access (reference) a specific model
- There are 3 different levels of model access that can be granted
  - **public** - can be accessed by models in any group, package, or project
  - **protected** - can be accessed by models in same project or group
  - **private** - can only be accessed by models in same group
- All models default to "**protected**"

So, which models should have which access modifiers?

● If everyone's in the same project, there's functionally not really a difference between protected and public models.
● A **public** model or a **protected** models should be one that is guaranteed to be **ready for final use.**
● A **private** model should be one that you don't really want other people pulling from. For whatever reason, the data is a private implementation detail reserved for use only by the team represented by the group, and **there are probably downstream models better suited to be pulled by others**

Apply Access Modifiers

● To assign **Access Modifiers** to a model, add the **access:** property to the model yaml file

● Indicate the level of access you want to assign to this model

○ private | protected | public

```yaml
models:
- name: stg_jaffle_shop__customers
group: finance
access: protected
- name: stg_jaffle_shop__orders
group: finance
access: private
models:
- name: dim_customers
group: marketing
access: public
```

Model Governance

Hands-on (7 min)

● Create a yml file in your marts folder. Create a group called marketing.
● Create another yml file in the marts folder and add a group and access modifier to dim_customers.sql
○ It should belong to the marketing group and be a public model.

```
#example
models:
- name: stg_jaffle_shop__customers
group: finance
access: protected
```

Model Contracts Defined

● Allow you to *guarantee* the shape of your model
○ The columns & names that exist in the model
○ The data type of each column
○ If your model is materialized as **table** or **incremental** (depending on the platform)
● If the model doesn't have those exact columns with those exact data types, you'll get an error message when you try to do a dbt run.

Specify additional constraints on columns:

■ not_null
■ Check - evaluates a boolean expression (e.g. ≥ 0)
■ others (to come)
- Primary key
- Foreign key
- Custom
- Expression

Docs:
https://docs.getdbt.com/reference/resource-properties/constraints

*Constraints Disclaimer

● Traditional transactional databases (like Postgres) can enforce many more types of constraints
○ not_null, unique, primary_key, foreign_key
● Sometimes analytical data platforms support defining these constraints, but only for metadata purposes - they aren't actually enforced. Check docs to see which platforms enforce which constraints.

Isn't this sort of like testing?

● **Model contracts** check up on the *shape of a dataset*, while **tests** check up on the *content of a data set*.
● **Tests are a more flexible way to validate the content of a model**
○ As long as you can write the query, you can run the test.
○ Tests are also more configurable in terms of severity, and custom thresholds are easier to debug after finding failures.
○ But:
■ constraints are a pre-flight check; if a particular constraint is enforced by your platform, the "bad data" won't even be able to get into your model
■ dbt data tests are post-hoc checks; bad data can get into your model and the tests lets you know about it
● You'll probably want to use model contracts to verify the shape/data types of a dataset, and you'll want to use tests to validate every other type of content about a model.

Creating Model Contracts

- To create a **Model Contract**, add a **contract:** configuration to the model yaml
- List each expected column, along with its data type
- Add a constraints property to create an even stricter contract*

```
models:
- name: dim_customers
group: marketing
access: public
config:
contract:
enforced: true
columns:
- name: customer_id
data_type: number
constraints:
- type: not_null
...
```

Enter: Model Versioning

Versioning Defined

● Versioning allows you to have different stages of a specific model
● This allows us to stage changes to our model and allow downstream
models to be updated ahead of the shipped change

Using Model Versions

● Adding versions to your model will require a few different steps

Step 1: Add a **latest_version:** property to your models yaml

*This will tell dbt which version of the model you're using

Step 2: Add a **versions:** key to your models yaml

*this will allow us to list out old and current versions of the model

Step 3: Add a **- v:** property for eachversion under your **versions**: key

*Notice we have 2 listed - one for the new version 2, and one for the original version 1

Step 4: Add a **defined_in:** property under the old version (this tells us where the file has moved to), as well as an **alias**: config (should match the latest model's name)

Step 5: Add a **columns**: property, where you will define the following:

● **include:**

○ Which columns from v1 are in v2 of my model

● **exclude:**

○ Which v1 columns changed

● **old column configs**

○ What was the original config of the changed column

```
models:
- name: dim_customers
latest_version: 2
columns:
- name: customer_id
data_type: number
- name: number_of_orders
data_type: number
versions:
- v: 2
- v: 1
defined_in: dim_customers_v1
config:
alias: dim_customers
columns:
- include: *
exclude: number_of_orders
- name: number_of_orders
data_type: text
```

Ref-ing & running a model version

● Now that you’ve created versions of your model, you’ll want to ensure the downstream models are using the appropriate version

● **Ref a version:** select * from {{ ref ('dim_customers', v=2 or version=2) }}

● **Run a version:** dbt run -s dim_customers.v2

● If you don’t identify a version in your ref or run, it will default to the latest version

Model Governance

Hands-on (skip)

1. Implement versioning on a stg_jaffle_shop__orders
2. Update a downstream ref to reference the updated version
3. Execute a dbt run on the new version of your model

```
#example
models:
- name: dim_customers
latest_version: 2
columns:
- name: customer_id
data_type: number
versions:
- v: 2
- v: 1
defined_in: marts
config:
alias: dim_customers
columns:
- include: *
exclude: number_of_orders
- name: number_of_orders
data_type: text
```

Model Governance

Resources

- Model Contracts
- Constraints
- Model Versions
- Model Access
- Model Groups

dbt_project_evaluator

Governance Enable Collaboration at Scale

Public models without contracts, undocumented public models

dbt_project_evaluator enforces best practices for .....

Models Improve development velocity

Hard-coded references, unused sources, direct join to source, model fanout, etc;

Testing Ensure data quality & build trust

Test coverage %, missing primary key tests

Structure Adhere to Enterprise standards

Naming conventions & Folder structure

Performance Improve execution speed to meet SLAs

Exposure parent materialization, chained views

Documentation Spread organizational knowledge

Undocumented models, sources, doc coverage %

dbt_project_evaluator

Why align with best practices?

Aligning with best practices makes your dbt projects….

- usable - data outputs are more reliable
- scalable - Duplicated code is eliminated
- organized - Developers find code easy to read & understand

Hands-on Practice: Run project evaluator

1. Create a packages.yml in your root directory.
2. Install the package by adding the following in packages.yml
3. Run the models in the package evaluator using the below command

dbt build -s package:dbt_project_evaluator

4. Check the warnings from the execution

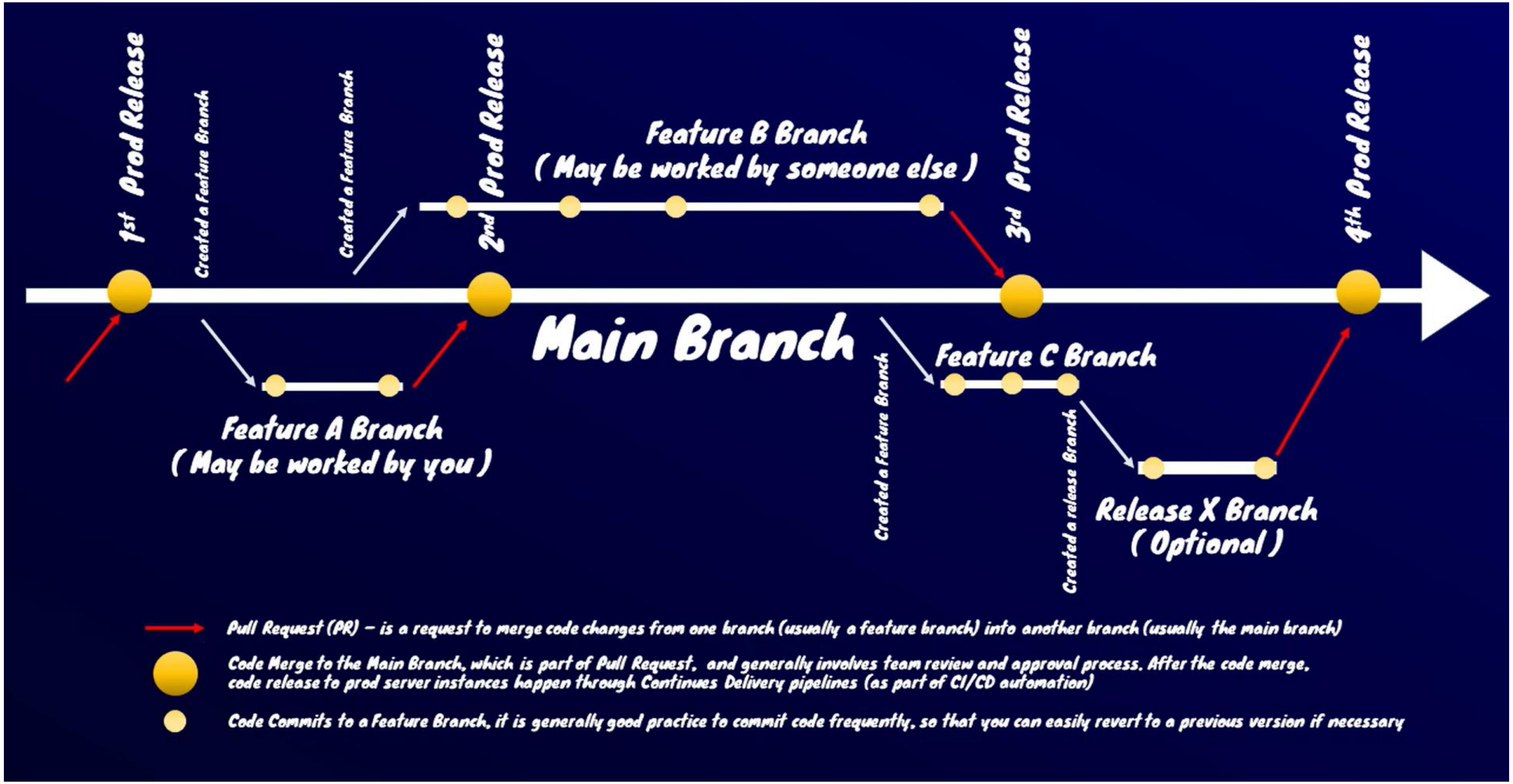
Hands-on Practice: Let's fix the warnings

Identify the model name

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

Fix the issues or add exceptions as required

Branch concept in DBT



Thank you!

http://wf13.myhcl.com/sites/techceed/index.html