Advanced DBT

> Dbt (data build tool) is a popular open-source data transformation tool that allows developers and data analysts, data engineers to build and manage data transformation pipelines. Here are some of the ways you can leverage dbt to maximize its potential

1. Modularize your code: Use dbt to build a modularized data pipeline. Modularizing your code will make it easier to test, debug, and scale.
2. Automated testing: Dbt has a built-in testing framework that makes it easy to test data transformation pipelines. By automating testing, you can ensure that your data is accurate and reliable.
3. Use version control: Use git or another version control system to manage your dbt projects. Version control allows you to track changes to your code, collaborate with others, and rollback changes if necessary.
4. Document your code: Use dbt's built-in documentation features to document your code. Documenting your code will make it easier for others to understand how your data transformation pipeline works.
5. Use macros: Macros are reusable code snippets that can be used across multiple dbt models. Macros can help you reduce code duplication and improve code quality.
6. Use packages: Dbt packages are pre-built modules that can be used to simplify common data transformation tasks. By leveraging pre-built packages, you can speed up development time and improve code quality.
7. Monitor performance: Use dbt to monitor the performance of your data transformation pipelines. Dbt provides built-in metrics and dashboards that make it easy to track pipeline performance and identify performance issues

Example of modularized code in Dbt

Modularized code is a key feature of dbt. It allows you to separate out individual pieces of functionality into modules that can be reused across projects, improving code maintainability and making it easier to develop and test individual pieces of code.

Here is an example of modularized code in dbt:

1. Create a directory structure that separates your dbt project into modules:

```
my_dbt_project/
├── models/
│   ├── base/
│   │   ├── users.sql
│   │   ├── orders.sql
│   │   └── products.sql
│   ├── marketing/
│   │   ├── customers.sql
│   │   ├── campaigns.sql
│   │   └── leads.sql
│   ├── finance/
│   │   ├── expenses.sql
│   │   ├── revenue.sql
│   │   └── invoices.sql
│   └── ...
└── macros/
    ├── date_dimensions.sql
    └── aggregation.sql
```

2. In each module, define a set of models or macros that work together to provide a specific set of functionality. For example, the `marketing` module might contain models that provide insights into customer behavior and campaign effectiveness.
3. Define dependencies between modules by using the `refs` function in your models or macros. For example, a model in the `marketing` module might reference a model in the `base` module:

```sql
-- models/marketing/customers.sql s
{% set current_country = var("current_country") %}
{% set date_start = var("start_date") %}
{% set date_end = var("end_date") %}

{{ config(materialized='incremental', unique_key='audit_id', schema=get_schema_n
elect   ... from {{ ref('base.users') }}

```

4. Build and test each module independently, making sure that it works as expected before integrating it into the larger project.
5. Combine the individual modules into a larger dbt project by referencing them in your `dbt_project.yml` file:

```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: "my_dbt_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 file
# The `source-paths` config, for example, states that models in this project can
# found in the "models/" directory. You probably won't need to change these!
source-paths: ["models"]
analysis-paths: ["analysis"]
test-paths: ["tests"]
data-paths: ["data"]
macro-paths: ["macros"]
snapshot-paths: ["snapshots"]

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

vars:
  current_country: ""
  country_th: 3
  country_id: 2
  country_my: 49
  country_sg: 1
  from_date: ""
  start_date: ""
  end_date: ""

# Configuring models
# Full documentation: https://docs.getdbt.com/docs/configuring-models

# In this example config, we tell dbt to build all models in the example/ direct
# as tables. These settings can be overridden in the individual model files
# using the `{{ config(...) }}` macro.
# quoting:
#   database: true
# identifier: false
# schema: false

models:
```

```
my_dbt_project:
  base:
    +schema: my_dbt_project_base
  marketing:
      +schema: my_dbt_project_marketing
  finance:
      +schema: my_dbt_project_finance ...

```

6. Run the `dbt run` command to build the entire project, including all modules.

By breaking your dbt project into individual modules, you can make it easier to maintain, test, and reuse your code across projects.

Example of Automate testing in Dbt

Suppose you have built some Dbt packages and modules or wanted to update some more. Before pushing into Git repo, you will want to run local test Dbt. Here's an example of how to automate testing in Dbt:

1. Define your tests in a separate schema in your Dbt project. You can do this by creating a new schema called `tests` and adding test SQL files to that schema. For example, if you have a table called base_activities, you can create a test file called `test_base_activities.sql`
2. In the test file, define the test query that will be executed against the target data. For example, to test if there are any NULL values in the activities_id table, you can use the following query:

```
select count(*) as null_count
from {{ ref('base_activities') }}
where activities_id is null and date_activities_created_at is null;
```

3. Add the `config` block in your `dbt_project.yml` file to enable tests and specify the test schema. Here's an example:

```
config-version: 2

models:
  my_project:
    schema: my_schema
    materialized: view
```

```
7
8 tests:
9   my_project:
10     schema: tests
11     data-paths: ['data']
```

4. Run the tests using the command. This will execute all the test files in the `tests` schema and report any failures or errors.

```
1 dbt test
```

You can also set up automated testing using a CI/CD pipeline, such as CircleCI or Jenkins. In this case, you can add a test step to your pipeline that runs the `dbt test` command and reports the test results.

Here's an example of a CircleCI configuration file:

```
1  version: 2
2
3  jobs:
4    build:
5      docker:
6        - image: circleci/python:3.8.6
7      steps:
8        - checkout
9        - run:
10           name: Install Dbt
11           command: |
12             pip install dbt
13       - run:
14           name: Run Dbt Test
15           command: dbt test
```

This is just a basic example, and there are many more advanced features and options available for testing in Dbt. But hopefully this gives you a good starting point for automating your testing in Dbt.

Example of Use version control in Dbt

By using Git version control with Dbt, you can easily track changes to your project over time and collaborate with other members of your team.

Here's an example of how you can use version control with Dbt:

1. Initialize a Git repository and commit your Dbt project to it:

```
git init
git add .
git commit -m "Initial commit"
```

2. Create a new branch for your development work:

```
git checkout -b my-feature-branch
```

3. Make changes to your Dbt project, such as modifying a model or adding a new one.
4. Stage and commit your changes:

```
git add .
git commit -m "Add new country_id for countries"
```

5. Push your changes to a remote repository:

```
git push origin my-new-feature-branch
```

6. Create a pull request for your changes to be reviewed by others on your team.

7. Once your changes are approved, merge your changes into the main branch:

```
git checkout main
git merge my-new-feature-branch
```

Finally, deploy your changes to your production environment using a deployment tool like Jenkins or CircleCI.

DBT built-in documentation features

Dbt provides built-in documentation features to help document your data transformations and make them more understandable for your team. The documentation features include:

1. **Dbt Docs**: This is a web application that generates and hosts documentation for your dbt project. It provides a comprehensive view and data pipeline map / DAG_alike of tables in your models, tests, and macros.
2. **Inline Documentation**: You can add inline documentation to your dbt models, tests, and macros by including comments in your SQL code. For example:

```
select      -- Calculate total revenue
sum(price * quantity) as total_revenue
from sales
```

3. **Custom Metadata**: You can add custom metadata to your dbt models, tests, and macros to provide additional context or information. This metadata can be used in your documentation or for other purposes. For example:

```
{{
    config(
        metadata={
            "author": "John Doe",
            "description": "Model that calculates total revenue"
        }
    )
```

```
}}
select
    sum(price * quantity) as total_revenue
from sales
```

4. **Data Dictionary**: You can use the `generate_data_dictionary` command to generate a data dictionary for your dbt project. This creates a CSV file that lists all the columns in your models along with their descriptions, data types, and other information.

```
{% docs sales_report %}
## Sales Report

This model generates a report of sales by product category and month.

### Columns

| Column Name | Description |
|-------------|-------------|
| category    | Product category |
| month       | Month of sale |
| total_sales | Total sales for the category and month |
{% enddocs %}

{% macro sales_report() %}
select
    category,
    date_trunc('month', date) as month,
    sum(price * quantity) as total_sales
from sales
group by 1, 2
{% endmacro %}
```

In this example, the `docs` block provides documentation for the `sales_report` model, while the `macro` block contains the SQL code to generate the report.

About Macros Packages in DBT

Macros in Dbt are reusable blocks of code that can be shared across models, projects and repositories. Macros packages in Dbt provide a way to share and distribute macros as a set of modules, that can be easily installed and used in any dbt project.

Dbt_utils

One example of a macros package in Dbt is the `dbt-utils` package, this is a collection of macros that provide additional functionality to your dbt project, such as creating incremental models and testing for data anomalies. Here's an example of how to use the dbt-utils package to create an incremental model:

```
1
2  {% macro incremental(model_name, incremental_field) %}
3      {%- set table_name = ref(model_name).source.table_name -%}
4      {%- set last_loaded = ref('last_loaded_{}'.format(model_name)).last_loaded -
5      {%- if last_loaded is not none -%}
6  SELECT *
7  from {{ ref('base.users') }}
8  WHERE {{ ref('incremental_field') }} >= '{{ last_loaded }}'
9  {%- else -%}
10 SELECT *
11 from {{ ref('base.users') }}
12 {%- endif -%}
13 {% endmacro %}
14
```

```
1  {% macro country_name(country_id) %}
2  # this macro define country_name by country_id
3  CASE
4      WHEN {{ country_id }} = 1 THEN 'Singapore'
5      WHEN {{ country_id }} = 2 THEN 'Indonesia'
6      WHEN {{ country_id }} = 3 THEN 'Thailand'
7      WHEN {{ country_id }} = 49 THEN 'Malaysia'
8      WHEN {{ country_id }} = 84 THEN 'Taiwan'
9      ELSE NULL
10 END
11
12 {% endmacro %}
```

These macros can be called in your dbt models to simplify your code and reduce duplication (like can prevent from calling country_id many times) .

To install and use the `dbt-utils` package in a Dbt project, follow these steps:

1. Install the package by adding the following lines to your `packages.yml` file

```
packages:
- package: fishtown-analytics/dbt_utils
version: 0.8.0
```

2. Run `dbt deps` to install the package and its dependencies.
3. Import the macros in your Dbt project by adding the following line to your `macros` directory:

```
{% import "dbt_utils" as dbt_utils %}
```

1. Use the macros in your models or transformations by calling them using the `{{ }}` syntax. For example:

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

    {{ generate_schema_name_for_env(custom_schema_name, node) }}
    {{ dbt_utils.validate.non_empty('created_at', created_at) }}
{%- endmacro %}
```

This will validate that the `created_at` field is not empty, using the `validate.non_empty` macro from the `dbt-utils` package.

`custom_schema_name` will give the options to customize schema name that you can use later in building models, using the `generate_schema_name_for_env` macro.

```
{% set current_country = var("current_country") %}
{% set date_start = var("start_date") %}
{% set date_end = var("end_date") %}

{{ config(materialized='incremental', unique_key='activity_id', schema=get_schema_name_by_country_id(current_country), alias='activities') }}

SELECT
    activities.id AS activity_id,
    activities.user_id AS activity_user_id,
    activities.activitiable_type as activity_activitiable_type,
    activities.activitiable_id as activity_activitiable_id,
    activities.related_object_type AS activity_related_object_type,
```

Dbt_labs

1. The dbt-labs package: This package provides a set of macros and models that allow you to easily build data pipelines in dbt. One example of a macro in the dbt-labs package is the `merge` macro, which allows you to perform a merge operation on a target table based on a source table:

```
{% macro merge(target, source, merge_keys) %}
    {%- set merge_stmt = ['MERGE INTO', target, 'T USING', source, 'S ON ('] %}
    {%- for key in merge_keys -%}
        {%- if loop.index > 1 -%}
            {%- set merge_stmt = merge_stmt + [' AND '] %}
         {%- endif -%}
         {%- set merge_stmt = merge_stmt + ['T.', key, '=', 'S.', key] %}
    {%- endfor -%}
    {%- set merge_stmt = merge_stmt + [') WHEN MATCHED THEN UPDATE SET '] %}
    {%- set update_cols = [col for col in source.columns if col not in merge_key
    {%- for col in update_cols -%}
        {%- if loop.index > 1 -%}
            {%- set merge_stmt = merge_stmt + [', '] %}
        {%- endif -%}
        {%- set merge_stmt = merge_stmt + ['T.', col, '=', 'S.', col] %}
    {%- endfor -%}
    {%- set merge_stmt = merge_stmt + [' WHEN NOT MATCHED THEN INSERT ('] %}
    {%- set insert_cols = source.columns %}
    {%- for col in insert_cols -%}
        {%- if loop.index > 1 -%}
            {%- set merge_stmt = merge_stmt + [', '] %}
        {%- endif -%}
        {%- set merge_stmt = merge_stmt + [col] %}
    {%- endfor -%}
    {%- set merge_stmt = merge_stmt + [') VALUES ('] %}
    {%- for col in insert_cols -%}
        {%- if loop.index > 1 -%}
            {%- set merge_stmt = merge_stmt + [', '] %}
        {%- endif -%}
        {%- set merge_stmt = merge_stmt + ['S.', col] %}
    {%- endfor -%}
    {%- set merge_stmt = merge_stmt + [');'] %}
    {{ merge_stmt | join(' ') }}
{% endmacro %}

```

These macros can be called in your dbt models to simplify your code and reduce duplication.

How to monitor performance in DBT

Dbt provides several built-in features for monitoring and analyzing the performance of your data pipelines:

1. Logs: Dbt logs all the executed SQL statements and the time taken to execute them. You can use these logs to identify slow-running queries and optimize them.
2. Profiles: Dbt generates detailed profiles of each run that can be used to identify bottlenecks and optimize query performance. These profiles provide detailed information on query run time, resource utilization, and database performance.
3. Metrics: Dbt provides several built-in metrics, such as the number of executed queries, the time taken to execute them, and the number of rows affected by each query. You can use these metrics to track the performance of your data pipelines over time.
4. Notifications: Dbt can be configured to send notifications when specific events occur, such as when a query takes longer than a specified time to execute or when an error occurs.
5. Alerts: Dbt can be configured to raise alerts when specific conditions are met, such as when the number of failed queries exceeds a specified threshold.
6. Profiling tools: Dbt provides profiling tools that allow you to analyze the performance of your data pipelines at different stages of the pipeline. These tools can help you identify performance bottlenecks and optimize your data pipelines.
7. Tracing: Dbt provides tracing tools that allow you to trace the execution of your data pipelines across different stages of the pipeline. This can help you identify issues and optimize the performance of your data pipelines.

Overall, Dbt provides a comprehensive set of features for monitoring and analyzing the performance of your data pipelines, making it easy to identify and optimize performance bottlenecks.

Here are some popular packages in Dbt:

1. `dbt-utils` : A package that provides a set of macros to make it easier to work with dates, strings, and other common data types.
2. `snowplow` : A package that provides a set of macros to work with Snowplow data.
3. `jaffle_shop` : A sample e-commerce dataset and accompanying dbt project, which can be used as a starting point for building out your own dbt project.
4. `fivetran` : A package that provides a set of macros and models to work with Fivetran data.
5. `segment` : A package that provides a set of macros and models to work with Segment data.
6. `hubspot` : A package that provides a set of macros and models to work with HubSpot data.
7. `stripe` : A package that provides a set of macros and models to work with Stripe data.
8. `salesforce` : A package that provides a set of macros and models to work with Salesforce data.

9. `databook` : A package that provides a set of macros to automate the process of generating data dictionaries for your dbt project.
10. `dbt-labs/google-analytics` : A package that provides a set of macros and models to work with Google Analytics data.

These packages are all available on the dbt Hub, which is a community-driven package repository for dbt.