14. Advance DBT Concept

Custom Schema Generation

By default, DBT generates schema names based on your target. You can customize this behavior by overriding the `generate_schema_name` macro:

sql

```
{% macro generate_schema_name(custom_schema_name, node) %}
    {%- set default_schema = target.schema -%}

    {%- if custom_schema_name is none -%}
        {{ default_schema }}
    {%- elif target.name == 'prod' -%}
        {{ custom_schema_name }}
    {%- else -%}
        {{ default_schema }}_{{ custom_schema_name }}
    {%- endif -%}
{% endmacro %}
```

This macro will:

- Use the default schema if no custom schema is specified
- Use the custom schema as-is in production
- Prefix the custom schema with the default schema in development

Hooks

Hooks are SQL statements that run at specific points in the DBT workflow. They're useful for:

- Setting session parameters
- Managing permissions
- Logging operations
- Creating custom schema objects

Types of Hooks

1. **Run-Start and Run-End Hooks**: Execute at the beginning and end of `dbt run`
2. **Pre-Hook and Post-Hook**: Execute before and after a model is built
3. **On-Run-Start and On-Run-End**: Execute at the beginning and end of any DBT command

Configuring Hooks

Hooks can be configured at different levels:

1. **Project Level**:

   yaml

   ```yaml
   # dbt_project.yml
   on-run-start:
     - "CREATE SCHEMA IF NOT EXISTS {{ target.schema }}_audit"
     - "GRANT USAGE ON SCHEMA {{ target.schema }} TO ROLE reporter"

   on-run-end:
     - "GRANT SELECT ON ALL TABLES IN SCHEMA {{ target.schema }} TO ROLE reporter"
   ```

2. **Model Level**:

   yaml

   ```yaml
   # dbt_project.yml
   models:
     my_project:
       staging:
         +post-hook:
           - "GRANT SELECT ON {{ this }} TO ROLE reporter"
   ```

3. **Individual Model Level**:

   sql

   ```sql
   {{ config(
       post_hook=[
           "INSERT INTO audit.model_runs (model_name, run_at) VALUES ('{{
   ```

```
this.name }}', CURRENT_TIMESTAMP)",
        "GRANT SELECT ON {{ this }} TO ROLE reporter"
    ]
) }}

SELECT * FROM {{ ref('stg_customers') }}
```

Operational Tasks

DBT provides operations for running custom tasks:

Creating a Custom Operation

1. Define a macro for your operation:

sql

```sql
{% macro refresh_external_table(schema, table) %}
    {% set refresh_query %}
        ALTER EXTERNAL TABLE {{ schema }}.{{ table }} REFRESH
    {% endset %}

    {% do run_query(refresh_query) %}
    {% do log("Refreshed external table " ~ schema ~ "." ~ table,
info=True) %}
{% endmacro %}
```

2. Run the operation:

bash

```bash
dbt run-operation refresh_external_table --args '{schema: raw, table:
external_customers}'
```

Built-in Operations

DBT includes several built-in operations:

1. **generate_model_yaml**: Generates YAML files for your models

2. **generate_source_yaml**: Generates YAML files for your sources
3. **collect_freshness**: Checks the freshness of your sources

Best Practices and Optimization

Project Organization

Model Organization

A common way to organize models is the following structure:

1. **Staging Models**:
   - One model per source table
   - Minimal transformations
   - Clean and rename fields
   - One-to-one relationship with source tables
   - Example: `models/staging/stg_customers.sql`
2. **Intermediate Models**:
   - Join and transform staging models
   - Business logic applied
   - Reusable building blocks
   - Example: `models/intermediate/customer_orders.sql`
3. **Mart Models**:
   - Business-specific models
   - Organized by business area
   - Optimized for analytics
   - Example: `models/marts/marketing/customer_lifetime_value.sql`

Naming Conventions

Consistent naming helps maintain your project:

1. **Models**:
   - Staging: `stg_[source]_[entity]`
   - Intermediate: `int_[entity]_[verb]`
   - Marts: `[business_area]_[entity]_[verb]`

2. **Macros**:
   - Use snake_case
   - Prefix with purpose: `test_`, `util_`, `audit_`
3. **Tests**:
   - Generic: `test_[assertion].sql`
   - Singular: `[model]_[assertion].sql`

Performance Tuning

Query Optimization

1. **Use Incremental Models**: For large tables that change frequently
2. **Optimize Join Orders**: Join smaller tables first, then larger ones
3. **Use CTEs for Readability**: Break complex queries into manageable CTEs
4. **Leverage Database-Specific Features**: Use features like Snowflake clustering keys

Materialization Strategies

Choose the right materialization based on:

1. **Data Volume**:
   - Small data: Views
   - Large data: Tables or incremental models
2. **Update Frequency**:
   - Frequent updates: Views or incremental models
   - Infrequent updates: Tables
3. **Query Complexity**:
   - Simple transformations: Views
   - Complex transformations: Tables
4. **Query Patterns**:
   - Ad-hoc exploration: Views
   - Repeated reporting: Tables

Workflow Integration

Continuous Integration

Integrate DBT with CI/CD pipelines:

1. **Pull Request Checks**:
    - Run `dbt compile` to check syntax
    - Run `dbt test` to validate changes
    - Run `dbt docs generate` to update documentation
2. **Deployment**:
    - Run `dbt seed` to load reference data
    - Run `dbt run` to build models
    - Run `dbt test` to verify data quality

Orchestration

Schedule DBT jobs with orchestration tools:

1. **Airflow**:

    python

    ```python
    from airflow import DAG
    from airflow.operators.bash_operator import BashOperator

    with DAG('dbt_daily', schedule_interval='0 5 * * *') as dag:
        dbt_run = BashOperator(
            task_id='dbt_run',
            bash_command='cd /path/to/dbt && dbt run --target prod'
        )

        dbt_test = BashOperator(
            task_id='dbt_test',
            bash_command='cd /path/to/dbt && dbt test --target prod'
        )

        dbt_run >> dbt_test
    ```

2. **Prefect**:

    python

    ```python
    from prefect import task, Flow
    import subprocess
    ```

```python
@task
def dbt_run():
    subprocess.run(['dbt', 'run', '--target', 'prod'])

@task
def dbt_test():
    subprocess.run(['dbt', 'test', '--target', 'prod'])

with Flow("dbt_daily") as flow:
    run = dbt_run()
    test = dbt_test()
    test.set_upstream(run)
```