ChatGPT

dbt Incremental Examples by Day

This companion document to the seven-day dbt plan provides concrete examples that build on each other day by day. The goal is to illustrate how the concepts in the plan translate into code and practical use.

Day 0 Example – Manual Script vs. dbt Model

Before dbt, teams often used ad-hoc SQL scripts and cron jobs to transform data. For example, a manual transformation of a **customers** table might look like:

```
-- manual_transformation.sql
SELECT
  id AS customer_id,
  UPPER(first_name || ' ' || last_name) AS customer_name,
  email,
  created_at::date AS signup_date
FROM raw.customers
WHERE status = 'active';
```

In dbt, you would define this as a **model** and reference the source table through `source()` in a Jinja-templated SQL file:

```
-- models/dim_customers.sql
{{ config(materialized='table') }}
SELECT
  id AS customer_id,
  UPPER(first_name || ' ' || last_name) AS customer_name,
  email,
  DATE(created_at) AS signup_date
FROM {{ source('raw', 'customers') }}
WHERE status = 'active';
```

You'd also define the raw table in a YAML file:

```
version: 2
sources:
  - name: raw
```

```
    tables:
      - name: customers
```

Running `dbt run` would create the table in your warehouse; running `dbt test` would execute any tests you've defined on this model; and `dbt docs generate` would update the documentation.

Day 1 Example – Initialize a dbt Project

On the first day you set up your dbt environment. After installing dbt and creating a project with `dbt init my_project`, try creating a very simple model in the `models/example/` directory:

```
-- models/example/my_first_dbt_model.sql
{{ config(materialized='view') }}
SELECT 1 AS id, 'hello dbt' AS message;
```

Then run your project:

```
dbt run
```

This will compile the model and create a view in your warehouse. You can verify the output by querying the view directly. Use `dbt docs generate` followed by `dbt docs serve` to explore the project documentation and DAG.

Day 2 Example – Staging and Transformation Models

Start by defining your **sources** in a YAML file. This tells dbt where the raw data lives:

```
# models/staging/schema.yml
version: 2
sources:
  - name: raw
    tables:
      - name: customers
      - name: orders

models:
  - name: stg_customers
    description: "Staging model that selects and cleans the raw customers table"
  - name: stg_orders
    description: "Staging model for orders"
```

Next, create staging models that lightly clean and cast the raw data:

```
-- models/staging/stg_customers.sql
{{ config(materialized='view') }}
SELECT
    id,
    name,
    email,
    created_at
FROM {{ source('raw', 'customers') }};

-- models/staging/stg_orders.sql
{{ config(materialized='view') }}
SELECT
    id AS order_id,
    customer_id,
    order_date,
    amount AS revenue
FROM {{ source('raw', 'orders') }};
```

Finally, create a transformation model that produces a customer dimension table. It uses `ref()` to depend on the staging model so that dbt understands the build order:

```
-- models/marts/dim_customers.sql
{{ config(materialized='table') }}
SELECT
    id AS customer_id,
    UPPER(name) AS customer_name,
    email,
    DATE(created_at) AS signup_date
FROM {{ ref('stg_customers') }};
```

Run `dbt run` to build both staging and mart models. Inspect the DAG in the docs site to see how `stg_customers` flows into `dim_customers`.

Day 3 Example – Seeds and Snapshots

Seeds are static CSV files that dbt can load into your warehouse. Create a file in the `data/` directory:

```
# data/regions.csv
region_code,region_name
US,United States
```

```
CA,Canada
EU,Europe
```

Then run:

```
dbt seed
```

This will create a table called `regions` in your warehouse. You can reference this seed like any other model:

```
-- models/dim_region.sql
SELECT region_code, region_name
FROM {{ ref('regions') }};
```

For **snapshots**, suppose you want to track changes in customer status over time. Create a snapshot file:

```
-- snapshots/customer_status.sql
{% snapshot customer_status %}
  {{ config(
       target_schema='snapshots',
       unique_key='id',
       strategy='timestamp',
       updated_at='updated_at'
  ) }}
  SELECT id, status, updated_at
  FROM {{ source('raw', 'customers') }}
{% endsnapshot %}
```

Running `dbt snapshot` will populate a table in the `snapshots` schema that records historical changes to each customer's status. This enables slowly changing dimension (SCD) type 2 behavior.

Day 4 Example – Testing and Documentation

dbt allows you to define **schema tests** in YAML files. For example, you may want to ensure that `customer_id` in your `stg_customers` model is unique and not null:

```
# models/staging/schema.yml (continued)
models:
  - name: stg_customers
    columns:
      - name: id
```

```
        description: "Primary key for customers"
        tests:
          - not_null
          - unique

  - name: stg_orders
    columns:
      - name: order_id
        tests:
          - not_null
          - unique
      - name: order_date
        tests:
          - not_null
```

You can also write **custom data tests**. For example, ensure that no orders have future dates:

```
-- tests/no_future_orders.sql
SELECT *
FROM {{ ref('stg_orders') }}
WHERE order_date > CURRENT_DATE;
```

Running `dbt test` will execute both schema and custom tests. Any failures will be reported in the output. To generate documentation, run:

```
dbt docs generate
dbt docs serve
```

This launches a local website where you can browse models, sources, tests and their descriptions.

Day 5 Example – Macros and Packages

You can extend dbt with **macros**—custom Jinja functions that generate SQL. Suppose you need to calculate a person's age from a birth date:

```
-- macros/calculate_age.sql
{% macro calculate_age(column_name) %}
  DATEDIFF('year', {{ column_name }}, CURRENT_DATE)
{% endmacro %}
```

Use the macro inside a model:

```sql
-- models/marts/customer_age.sql
{{ config(materialized='table') }}
SELECT
    id AS customer_id,
    {{ calculate_age('created_at') }} AS account_age
FROM {{ ref('stg_customers') }};
```

To call macros outside of models, you can use `dbt run-operation`:

```
dbt run-operation calculate_age --args '{"column_name": "created_at"}'
```

Finally, leverage community **packages** like `dbt-utils` by adding them to `packages.yml`:

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

Run `dbt deps` to install the package, then use macros such as `dbt_utils.surrogate_key` in your models.

Day 6 Example – Incremental Models in Practice

An **incremental model** appends only new or changed data to an existing table. Here's a typical example that builds an orders fact table:

```sql
-- models/fct_orders.sql
{{ config(materialized='incremental', unique_key='order_id') }}

SELECT *
FROM {{ ref('stg_orders') }}

{% if is_incremental() %}
  WHERE order_date > (SELECT MAX(order_date) FROM {{ this }})
{% endif %}
```

On the first run, dbt creates the table and loads all rows from `stg_orders`. On subsequent runs, only rows with a `order_date` greater than the maximum `order_date` already present in the table are inserted.

You can simulate this logic in pandas to see how incremental loads behave:

```python
import pandas as pd

# Existing data in the fact table after an initial run
existing = pd.DataFrame({
    'order_id': [1, 2],
    'order_date': pd.to_datetime(['2025-01-01', '2025-01-03']),
    'revenue': [100, 200]
})

# New batch of orders arriving
new_data = pd.DataFrame({
    'order_id': [3, 4],
    'order_date': pd.to_datetime(['2025-01-05', '2025-01-10']),
    'revenue': [150, 300]
})

# Incremental logic: keep only rows where order_date > max existing
incremental_rows = new_data[new_data['order_date'] >
existing['order_date'].max()]

result = pd.concat([existing, incremental_rows], ignore_index=True)
display(result)
```

Running this code will show that rows with order IDs 3 and 4 are appended because their dates are newer than the existing maximum. This mirrors how dbt's `is_incremental()` filter works during an incremental run.

Day 7 Example – Exposures and Metrics

dbt supports declaring **exposures** and **metrics** to describe how data models feed downstream assets and to centralize business logic.

Exposure example

Define a dashboard exposure in a YAML file:

```yaml
# models/marts/exposures.yml
version: 2
exposures:
  - name: revenue_dashboard
    type: dashboard
    maturity: medium
    url: https://bi.example.com/dashboards/revenue
    depends_on:
```

```
      - ref('fct_orders')
    owner:
      name: Data Team
      email: data@example.com
```

This tells dbt that a BI dashboard depends on the `fct_orders` model. When you run `dbt docs generate`, the exposure will appear in the lineage graph.

Metric example

In dbt v1.5+ and with the Semantic Layer enabled, you can define metrics:

```
# models/marts/metrics.yml
version: 2
metrics:
  - name: total_revenue
    label: Total Revenue
    model: ref('fct_orders')
    calculation_method: sum
    expression: revenue
    timestamp: order_date
    time_grains: [day, month, year]
```

This metric can then be queried via the Semantic Layer or your BI tool, ensuring consistent definitions across reports.

By following these examples in tandem with the seven-day plan, you can see how dbt models, seeds, snapshots, tests, macros, incremental builds, and exposures work together to create maintainable and auditable data pipelines.
