SNOWFLAKE Vs DBT

```
Yes, we can do transformations directly in Snowflake using SQL.
But we use DBT because it helps us manage transformations in a more structured and efficient way.
In DBT, each transformation is written as a model (SQL file).
It supports version control with Git, so we can collaborate better and track changes.
We can add data quality checks like tests for nulls or duplicates.
It generates documentation and generate data lineage graphs.
Easier to manage different environments (dev/prod) for transformations using DBT configs.
--->Snowflake = Where your data lives and gets queried
--->DBT = How you manage and transform that data using SQL (in Snowflake or other warehouses)
```

1)What is DBT?

Data Build Tool (DBT) is a popular open-source tool used in the data analytics and data engineering fields. DBT helps data professionals transform, model, and prepare data for analysis. If you're preparing for an interview related to DBT, it's important to be well-versed in its concepts and functionalities. To help you prepare, here's a list of common interview questions and answers about DBT.

1. What is DBT?

Answer: DBT, short for Data Build Tool, is an open-source data transformation and modeling tool. It helps analysts and data engineers manage the transformation and preparation of data for analytics and reporting.

2) What are the primary use cases of DBT?

Answer:DBT is primarily used for data transformation, modeling, and preparing data for analysis and reporting. It is commonly used in data warehouses to create and maintain data pipelines.

3) How does DBT differ from traditional ETL tools?

Answer: Unlike traditional ETL tools, DBT focuses on transforming and modeling data within the data warehouse itself, making it more suitable for ELT (Extract, Load, Transform) workflows. DBT leverages the power and scalability of modern data warehouses and allows for version control and testing of data models.

4) What is a DBT model?

Answer: A DBT model is a SQL file that defines a transformation or a table within the data warehouse. Models can be simple SQL queries or complex transformations that create derived datasets.

5) Explain the difference between source and model in DBT.

Answer: A source in DBT refers to the raw or untransformed data that is ingested into the data warehouse. Models are the transformed and structured datasets created using DBT to support analytics.

6) What is a DBT project?

Answer: A DBT project is a directory containing all the files and configurations necessary to define data models, tests, and documentation. It is the primary unit of organization for DBT.

7) What is a DAG in the context of DBT?

Answer: DAG stands for Directed Acyclic Graph, and in the context of DBT, it represents the dependencies between models. DBT uses a DAG to determine the order in which models are built.

8) How do you write a DBT model to transform data?

Answer: To write a DBT model, you create a `.sql` file in the appropriate project directory, defining the SQL transformation necessary to generate the target dataset.

9) What are DBT macros, and how are they useful in transformations?

Answer: DBT macros are reusable SQL code snippets that can simplify and standardize common operations in your DBT models, such as filtering, aggregating, or renaming columns.

Here's how dbt macros work:

1. **Definition**: A macro is defined in a separate file with a `.sql` extension. It contains SQL code that can take parameters, making it flexible and reusable.

```
-- my_macro.sql
{% macro my_macro(parameter1, parameter2) %}
SELECT
  column1,
  column2
FROM
  my_table
WHERE
  condition1 = {{ parameter1 }}
  AND condition2 = {{ parameter2 }}
{% endmacro %}
```

2. **Invocation**: You can then use the macro in your dbt models by referencing it.

```
-- my_model.sql
{{ my_macro(parameter1=1, parameter2='value') }}
```

When you run the dbt project, dbt replaces the macro invocation with the actual SQL code defined in the macro.

3. **Parameters:** Macros can accept parameters, making them dynamic and reusable for different scenarios. In the example above, `parameter1` and `parameter2` are parameters that can be supplied when invoking the macro.

4. **Code Organization**: Macros help in organizing and modularizing your SQL code. They are particularly useful when you have common patterns or calculations that need to be repeated across multiple models.

10) How can you perform testing and validation of DBT models?

Answer: You can perform testing in DBT by writing custom SQL tests to validate your data models. These tests can check for data quality, consistency, and other criteria to ensure your models are correct.

11) Explain the process of deploying DBT models to production?

Answer: Deploying DBT models to production typically involves using DBT Cloud, CI/CD pipelines, or other orchestration tools. You'll need to compile and build the models and then deploy them to your data warehouse environment.

12) How does DBT support version control and collaboration?

Answer: DBT integrates with version control systems like Git, allowing teams to collaborate on DBT projects and track changes to models over time. It provides a clear history of changes and enables collaboration in a multi-user environment.

13) What are some common performance optimization techniques for DBT models?

Answer: Performance optimization in DBT can be achieved by using techniques like materialized views, optimizing SQL queries, and using caching to reduce query execution times.

14) How do you monitor and troubleshoot issues in DBT?

Answer: DBT provides logs and diagnostics to help monitor and troubleshoot issues. You can also use data warehouse-specific monitoring tools to identify and address performance problems.

15) Can DBT work with different data sources and data warehouses?

Answer: Yes, DBT supports integration with a variety of data sources and data warehouses, including Snowflake, BigQuery, Redshift, and more. It's adaptable to different cloud and on-premises environments.

16) How does DBT handle incremental loading of data from source systems?

By using the `is_incremental()` function, DBT loads only new or changed data instead of refreshing the entire table, improving performance and resource usage.

Answer: DBT can handle incremental loading by using source freshness checks and managing data updates from source systems. It can be configured to only transform new or changed data.

17) What security measures does DBT support for data access and transformation?

Answer: DBT supports the security features provided by your data warehouse, such as row-level security and access control policies. It's important to implement proper access controls at the database level.

18) How can you manage sensitive data in DBT models?

Answer: Sensitive data in DBT models should be handled according to your organization's data security policies. This can involve encryption, tokenization, or other data protection measures.

19) Types of Materialization?

Materialization in DBT defines how a model's SQL logic is executed and stored in the database. Common types include **table** (stored physically), **view** (virtual), **incremental** (updates existing data), and **ephemeral** (not stored at all). I choose the materialization type based on **performance, storage, and data freshness** requirements.

Answer: DBT supports several types of materialization are as follows:

1)View (Default):

Purpose: Views are virtual tables that are not materialized. They are essentially saved queries that are executed at runtime.
Use Case: Useful for simple transformations or when you want to reference a SQL query in multiple models.

```
{{ config(
  materialized='view'
) }}
SELECT
  ...
FROM ...
```

2)Table:

Purpose: Materializes the result of a SQL query as a physical table in your data warehouse.
Use Case: Suitable for intermediate or final tables that you want to persist in your data warehouse.

```
{{ config(
  materialized='table'
) }}
SELECT
  ...
INTO {{ ref('my_table') }}
FROM ...
```

3)Incremental:

Purpose: Materializes the result of a SQL query as a physical table, but is designed to be updated incrementally. It's typically used for incremental data loads.
Use Case: Ideal for situations where you want to update your table with only the new or changed data since the last run.

```
{{ config(
  materialized='incremental'
) }}
SELECT
  ...
FROM ...
```

4)Table + Unique Key:

Purpose: Similar to the incremental materialization, but specifies a unique key that dbt can use to identify new or updated rows.

Use Case: Useful when dbt needs a way to identify changes in the data.

```
{{ config(
  materialized='table',
  unique_key='id'
) }}
SELECT
  ...
INTO {{ ref('my_table') }}
FROM ...
```

5)Snapshot:

Purpose: Materializes a table in a way that retains a version history of the data, allowing you to query the data as it was at different points in time.

Use Case: Useful for slowly changing dimensions or situations where historical data is important.

```
{{ config(
  materialized='snapshot'
) }}
SELECT
  ...
INTO {{ ref('my_snapshot_table') }}
FROM ...
```

20) How to create a test in DBT and Types of Tests in DBT?

To create a test in DBT, define it in the model's `.yml` file.

Use built-in tests like `not_null`, `unique`, and `accepted_values`.

Example: `- name: order_id` with `tests: [not_null, unique]`.

Run all tests using the `dbt test` command.

For custom logic, write SQL tests in the `/tests` folder and reference them in YAML.

Answer: Dbt provides several types of tests that you can use to validate your data. Here are some common test types in dbt:

1)Unique Key Test (`unique`):

Verifies that a specified column or set of columns contains unique values.

```
version: 2

models:
  - name: my_model
    tests:
      - unique:
          columns: [id]
```

2)Not Null Test (`not_null`):

Ensures that specified columns do not contain null values.

```
version: 2

models:
  - name: my_model
    tests:
      - not_null:
          columns: [name, age]
```

3)Accepted Values Test (`accepted_values`):

Validates that the values in a column are among a specified list.

```
version: 2

models:
  - name: my_model
    tests:
      - accepted_values:
          column: status
          values: ['active', 'inactive']
```

4)Relationship Test (`relationship`):

Verifies that the values in a foreign key column match primary key values in the referenced table.

```
version: 2

models:
  - name: orders
    tests:
      - relationship:
          to: ref('customers')
          field: customer_id
```

5)Referential Integrity Test (`referential integrity`):

Checks that foreign key relationships are maintained between two tables.

```
version: 2

models:
  - name: orders
    tests:
      - referential_integrity:
          to: ref('customers')
          field: customer_id
```

6)Custom SQL Test (`custom_sql`):

Allows you to define custom SQL expressions to test specific conditions.

```
version: 2

models:
  - name: my_model
    tests:
      - custom_sql: "column_name > 0"
```

21) What is seed?

Answer: A "seed" refers to a type of dbt model that represents a table or view containing static or reference data. Seeds are typically used to store data that doesn't change often and doesn't require transformation during the ETL (Extract, Transform, Load) process.

Here are some key points about seeds in dbt:

1. **Static Data:** Seeds are used for static or reference data that doesn't change frequently. Examples include lookup tables, reference data, or any data that serves as a fixed input for analysis.
2. **Initial Data Load:** Seeds are often used to load initial data into a data warehouse or data mart. This data is typically loaded once and then used as a stable reference for reporting and analysis.
3. **YAML Configuration:** In dbt, a seed is defined in a YAML file where you specify the source of the data and the destination table or view in your data warehouse. The YAML file also includes configurations for how the data should be loaded.

22) What is Pre-hook and Post-hook?

Answer: Pre-hooks and Post-hooks are mechanisms to execute SQL commands or scripts before and after the execution of dbt models, respectively. dbt is an open-source tool that enables analytics engineers to transform data in their warehouse more effectively.

Here's a brief explanation of pre-hooks and post-hooks:

1)Pre-hooks:

- A pre-hook is a SQL command or script that is executed before running dbt models.
- It allows you to perform setup tasks or run additional SQL commands before the main dbt modeling process.
- Common use cases for pre-hooks include tasks such as creating temporary tables, loading data into staging tables, or performing any other necessary setup before model execution.

2)Post-hooks:

- A post-hook is a SQL command or script that is executed after the successful completion of dbt models.
- It allows you to perform cleanup tasks, log information, or execute additional SQL commands after the models have been successfully executed.
- Common use cases for post-hooks include tasks such as updating metadata tables, logging information about the run, or deleting temporary tables created during the pre-hook.

23) what is snapshots?

Answer: “snapshots” refer to a type of dbt model that is used to track changes over time in a table or view. Snapshots are particularly useful for building historical reporting or analytics, where you want to analyze how data has changed over different points in time.

Here’s how snapshots work in dbt:

1. **Snapshot Tables**: A snapshot table is a table that represents a historical state of another table. For example, if you have a table representing customer information, a snapshot table could be used to capture changes to that information over time.
2. **Unique Identifiers**: To track changes over time, dbt relies on unique identifiers (primary keys) in the underlying data. These identifiers are used to determine which rows have changed, and dbt creates new records in the snapshot table accordingly.
3. **Timestamps**: Snapshots also use timestamp columns to determine when each historical version of a record was valid. This allows you to query the data as it existed at a specific point in time.
4. **Configuring Snapshots**: In dbt, you configure snapshots in your project by creating a separate SQL file for each snapshot table. This file defines the base table or view you’re snapshotting, the primary key, and any other necessary configurations.

24) What is project structure?

In **dbt (data build tool)**, the **project structure** refers to how your dbt project is organized into directories and files. Here's a typical structure:

graphql Copy Edit

```
my_dbt_project/
├── dbt_project.yml         # Project configuration file
├── models/                 # Where your SQL models live
│   ├── staging/            # Staging models (raw -> cleaned)
│   ├── marts/              # Final models (facts, dimensions)
│   └── intermediate/       # Optional: intermediate transformation logic
├── seeds/                  # CSV files loaded as-is into the warehouse
├── snapshots/              # Snapshots for slowly changing dimensions
├── macros/                 # Reusable SQL snippets (Jinja macros)
├── tests/                  # Custom data tests (optional)
├── analyses/               # Ad-hoc or exploratory analysis (optional)
└── target/                 # Auto-generated: compiled project files (ignored in Git)
```

25) What is data refresh?

In **dbt**, **data refresh** refers to the process of re-running dbt models to **rebuild or update the transformed data** in your data warehouse.

This usually happens via:

- `dbt run` – Recomputes the models (SQL files) and materializes them (as tables/views).
- `dbt seed` – Reloads seed CSV data.
- `dbt snapshot` – Updates slowly changing dimension data.
- **Scheduled jobs** – In production, dbt runs are typically scheduled (via Airflow, dbt Cloud, etc.) to refresh data daily, hourly, etc.

In short: data refresh = reprocessing models so that your transformed tables always reflect the latest raw data.

Here are the **generic (built-in) tests** available in dbt:

1. `unique` – Checks if values in a column are unique
2. `not_null` – Ensures no nulls in a column
3. `accepted_values` – Validates that column values belong to a defined set
4. `relationships` – Checks referential integrity between models (like foreign key)

27) How to generate Documents

1. **Add descriptions** in `.yml` files (models, columns).
2. Run `dbt docs generate` – builds docs.
3. Run `dbt docs serve` – opens docs in browser.

Q) Explain the importance of DBT Documentation?

DBT Documentation is important because it **automatically generates detailed docs** for models, sources, tests, and metrics. It includes a **lineage graph**, which visually shows how data flows from source to final model. I use it to help teams understand transformations and dependencies clearly. It improves **collaboration, transparency, and onboarding** for new team members. This documentation becomes a **living reference** for maintaining and scaling data projects.

28) Types of Snapshot Strategies

There are **2 snapshot strategies** in dbt:

1. `timestamp` – Tracks changes using a last-updated timestamp column
   - Good for tables with a reliable `updated_at` column.
2. `check` – Compares selected columns for changes
   - Use when no timestamp column is available.

Defined like this in a snapshot file:

sql Copy Edit

```sql
{% snapshot my_snapshot %}
{
  "target_schema": "snapshots",
  "unique_key": "id",
  "strategy": "timestamp",
  "updated_at": "updated_at_column"
}
SELECT * FROM source_table
{% endsnapshot %}
```

29) Different YMLs in DBT

In dbt, different **YML files** are used for configuration, documentation, and testing. Here are the key ones:

1. `dbt_project.yml`
   – Main config file (project name, models path, materializations, etc.)
2. **Model YMLs** (inside `models/`)
   – Define **tests**, **descriptions**, **tags**, **meta**, etc. for models & columns.
3. **Snapshot YMLs** (inside `snapshots/`)
   – Define **snapshot strategy**, **unique keys**, and configs.
4. **Seed YMLs** (optional, inside `seeds/`)
   – Add **tests** and **descriptions** for seed CSV files.
5. **Source YMLs**
   – Declare **source tables**, **schema**, and apply **tests** on source columns.

30) How to create Permanent Table in Snowflake using DBT?

To create a **permanent table** in Snowflake using dbt, set the **materialization to** `table` and **disable temporary or transient behavior.**

✅ Step-by-step:

1. In your model file (e.g., `orders.sql`), write your SQL as usual.
2. In `dbt_project.yml` or model-level config, set:

yaml Copy Edit

```yaml
models:
  my_project:
    orders:
      materialized: table
      snowflake_warehouse: your_warehouse_name  # optional
```

31) What are Jinja templating in DBT?

Jinja templating in dbt is a way to add logic and dynamic behavior to your SQL models using the Jinja templating engine. It lets you use variables, loops, conditionals, and functions inside your SQL files. This is useful for reusability, automation, and referencing other models (e.g., `{{ ref('model_name') }}`). Jinja runs *before* the SQL is compiled and executed in Snowflake.

32) Difference between DBT and AWS Glue?

"DBT focuses on SQL-based transformations, versioning, and testing, whereas Glue is serverless ETL for managing data pipelines using Python/Scala."

33) How do you handle schema evolution in Snowflake?

Schema evolution in Snowflake is handled by proactively monitoring for changes like added or removed columns. I use **Streams** and **metadata queries** to detect schema changes in source tables. Before loading, I validate the structure to avoid breaking downstream models. If new columns are introduced, I update my **DBT models or SQL scripts** accordingly. This ensures the pipeline remains stable and adaptable to changing data structures.

34) Describe a real-time data pipeline you built?

I used S3 for data storage, Snowpipe for auto-ingestion, Streams for CDC, Tasks for scheduling, and DBT for structured transformation before loading into reporting tables.

35) What is DBT Snapshots?

DBT Snapshots are used to capture and track **historical changes** in data over time, especially for **slowly changing dimensions (SCD Type 2)**. They work by comparing current records to previous versions and storing changes with timestamps. This helps in tracking how a value (like customer status) has changed over time. I've used snapshots to monitor reference/master data like user profiles or product info. It allows accurate historical analysis and auditability in data models.

36) How do you schedule DBT jobs?

To schedule DBT jobs, I use **DBT Cloud's built-in scheduler**, which allows running models at defined intervals (like hourly or daily). It's easy to configure through the UI and supports alerts on failures. For more complex workflows, I integrate DBT with **external schedulers** like **Apache Airflow** or **AWS EventBridge**. These tools help orchestrate DBT alongside other steps like data ingestion or validation. This setup ensures **automated and reliable data pipeline execution.**

37) Difference between ref() and source() functions in DBT?

`ref()` is used to reference another DBT model, ensuring dependency tracking. `source()` is used to refer to raw data tables, usually the starting point of a pipeline.

38) How do you ensure data quality?

I ensure data quality by writing SQL validation queries to check for nulls, duplicates, and incorrect formats.

I use DBT tests like `unique`, `not_null`, and `accepted_values` on critical columns.

Business rules are enforced through conditional checks in transformation logic.

I implement error handling and logging in data pipelines to catch anomalies early.

Regular data audits and monitoring help maintain consistency and accuracy over time.