Top 10 dbt Interview Questions with Detailed Answers

Question 1: Explain the Medallion Architecture and How to Implement It in dbt

Explanation

The Medallion Architecture (also called Bronze-Silver-Gold) is a layered data architecture pattern that structures data transformation in three distinct stages:

- **Bronze Layer**: Contains raw, unprocessed data directly from source systems without any transformations
- **Silver Layer**: Contains cleaned, deduplicated, validated, and slightly transformed data
- **Gold Layer**: Contains business-ready, aggregated data optimized for reporting and analytics

This approach ensures data quality improves at each layer and maintains clear separation of concerns. Each layer has specific responsibilities making the pipeline modular and maintainable.

Code Implementation

yaml

```yaml
# dbt_project.yml - Project Configuration
models:
  my_project:
    bronze:
      materialized: table
      schema: bronze
      +pre-hook: "{{ log('Loading bronze layer', info=true) }}"
    silver:
      materialized: incremental
      schema: silver
      +pre-hook: "{{ log('Transforming to silver', info=true) }}"
    gold:
      materialized: table
      schema: gold
      +pre-hook: "{{ log('Building gold layer', info=true) }}"
```

sql

```sql
-- models/silver/stg_customers.sql
{{ config(
    materialized='incremental',
    unique_key='customer_id',
    on_schema_change='fail',
    tags=['critical', 'daily']
) }}

SELECT
    customer_id,
    LOWER(TRIM(customer_name)) as customer_name,
    LOWER(email) as email,
    CAST(created_at AS TIMESTAMP) as created_at,
    _dbt_ingestion_time
FROM {{ ref('bronze_raw_customers') }}
WHERE customer_id IS NOT NULL

{% if execute %}
    {% if var('full_refresh') is false %}
        WHERE _dbt_ingestion_time > (SELECT MAX(_dbt_ingestion_time) FROM {{ this }})
    {% endif %}
{% endif %}
```

Question 2: How Do You Handle Slowly Changing Dimensions (SCD Type 2)?

Explanation

SCD Type 2 tracks the complete history of changes to a dimension by creating new rows with effective date ranges. When an attribute changes, the old record is closed (marked with an end date) and a new record is created with updated values. This allows analysis of how customers/products changed over time and is commonly used in financial reporting where historical context is critical.

Code Implementation

sql

```sql
-- models/gold/dim_customers_scd2.sql
{{ config(
    materialized='incremental',
    unique_key=['customer_id', 'dbt_valid_from'],
    on_schema_change='fail'
) }}

WITH source_data AS (
    SELECT
        customer_id,
        customer_name,
        email,
        address,
        phone,
        CURRENT_TIMESTAMP as dbt_valid_from,
        CAST(NULL AS TIMESTAMP) as dbt_valid_to,
        TRUE as is_current
    FROM {{ ref('stg_customers') }}
),

existing_records AS (
    SELECT * FROM {{ this }}
    WHERE is_current = TRUE
),

changes_detected AS (
    SELECT
        s.customer_id,
        s.customer_name,
        s.email,
        s.address,
        s.phone,
        s.dbt_valid_from,
        COALESCE(e.dbt_valid_from, s.dbt_valid_from) as prior_valid_from,
        CASE
            WHEN e.customer_id IS NULL THEN 'INSERT'
            WHEN s.customer_name != e.customer_name
                OR s.email != e.email
                OR s.address != e.address
                OR s.phone != e.phone THEN 'UPDATE'
            ELSE 'NO_CHANGE'
        END as change_type
    FROM source_data s
    LEFT JOIN existing_records e ON s.customer_id = e.customer_id
```

```sql
),

handle_updates AS (
    SELECT
        customer_id,
        customer_name,
        email,
        address,
        phone,
        prior_valid_from as dbt_valid_from,
        dbt_valid_from as dbt_valid_to,
        FALSE as is_current
    FROM changes_detected
    WHERE change_type = 'UPDATE'
    UNION ALL
    SELECT
        customer_id,
        customer_name,
        email,
        address,
        phone,
        dbt_valid_from,
        dbt_valid_to,
        is_current
    FROM changes_detected
    WHERE change_type IN ('INSERT', 'UPDATE')
)

SELECT * FROM handle_updates
```

Question 3: What Are Star Schema and Fact/Dimension Tables in dbt?

Explanation

A star schema is a dimensional modeling technique where data is organized around fact tables (containing metrics and measures) and dimension tables (containing descriptive attributes). Fact tables store quantitative data (sales amounts, counts) and foreign keys to dimensions. Dimension tables store descriptive data (customer names, product categories). This structure optimizes query performance and makes data easier to understand for analytics.

Code Implementation

sql

```sql
-- models/gold/fct_orders.sql (Fact Table)
{{ config(
    materialized='table',
    tags=['fact_table'],
    persist_docs={"relation": true, "columns": true}
) }}

SELECT
    order_id,
    customer_key,
    product_key,
    store_key,
    date_key,
    quantity,
    unit_price,
    discount_amount,
    tax_amount,
    total_amount,
    CURRENT_TIMESTAMP as _dbt_generated_at
FROM {{ ref('stg_orders') }}
WHERE order_status != 'cancelled'
```

sql

```sql
-- models/gold/dim_customers.sql (Dimension Table)
{{ config(
    materialized='table',
    tags=['dimension_table']
) }}

SELECT
    {{ dbt_utils.generate_surrogate_key(['customer_id']) }} as customer_key,
    customer_id,
    customer_name,
    email,
    phone,
    country,
    city,
    customer_segment,
    CURRENT_TIMESTAMP as created_at
FROM {{ ref('stg_customers') }}
WHERE is_active = TRUE
```

Question 4: Explain Snapshots and When to Use Them

Explanation

Snapshots capture the state of a table at specific points in time and track how rows change over time. They create point-in-time copies using a timestamp strategy (detecting changes based on an updated_at column) or a check strategy (detecting any column changes). Snapshots are useful for auditing, compliance, and understanding data changes without needing to build complex SCD logic manually.

Code Implementation

sql

```sql
-- snapshots/snap_customers.sql
{% snapshot snap_customers %}
  {{
    config(
      target_schema='snapshots',
      unique_key='customer_id',
      strategy='timestamp',
      updated_at='updated_at',
      tags=['snapshots']
    )
  }}

  SELECT
    customer_id,
    customer_name,
    email,
    phone,
    address,
    status,
    updated_at
  FROM {{ source('raw_database', 'customers') }}
  WHERE deleted_at IS NULL

{% endsnapshot %}

-- Access snapshot data with built-in columns:
-- dbt_valid_from: When record was captured
-- dbt_valid_to: When record was replaced (NULL if current)
-- dbt_scd_id: Unique identifier for each snapshot version
-- SELECT * FROM {{ snapshot('snap_customers') }}
```

Question 5: What's the Difference Between `ref()` and `source()` and When to Use Each?

Explanation

`ref()` creates a dependency on other dbt models - dbt knows these are managed data transformations and creates a lineage graph. `source()` references external data outside dbt's control, typically raw data from operational systems. Using `source()` helps document where data originates and allows testing raw data quality. Using `ref()` ensures models run in correct order based on dependencies.

Code Implementation

sql

```sql
-- models/silver/stg_customers.sql
{{ config(materialized='view') }}

SELECT
    customer_id,
    LOWER(TRIM(customer_name)) as customer_name,
    LOWER(email) as email,
    CAST(created_at AS TIMESTAMP) as created_at
FROM {{ source('raw_data', 'customers') }}
WHERE customer_id IS NOT NULL
```

yaml

```yaml
# models/sources.yml
sources:
  - name: raw_data
    description: Raw data from operational systems
    database: analytics_raw
    schema: raw
    tables:
      - name: customers
        description: Raw customer data from CRM
        columns:
          - name: customer_id
            description: Unique customer identifier
            tests:
              - unique
              - not_null
```

sql

```sql
-- models/gold/fct_orders.sql
{{ config(materialized='table') }}

SELECT
    order_id,
    {{ ref('stg_customers') }}.customer_id,
    {{ ref('dim_products') }}.product_id,
    order_date,
    amount
FROM {{ ref('stg_customers') }}
JOIN {{ ref('dim_products') }} USING (product_id)
```

Question 6: How Do You Implement Recursive Models?

Explanation

Recursive models use Common Table Expressions (CTEs) with recursion to handle hierarchical data like organizational charts or category hierarchies. They work by defining a base case (root nodes) and then recursively finding child records. This is useful for traversing trees without knowing the depth in advance. Always include termination conditions to prevent infinite loops.

Code Implementation

sql

```sql
-- models/bronze/recursive_hierarchy.sql
{{ config(
    materialized='table',
    tags=['hierarchical']
) }}

WITH RECURSIVE category_hierarchy AS (
    -- Base case: root categories (no parent)
    SELECT
        category_id,
        parent_category_id,
        category_name,
        0 as hierarchy_level,
        CAST(category_id AS VARCHAR(500)) as path
    FROM {{ ref('stg_categories') }}
    WHERE parent_category_id IS NULL

    UNION ALL

    -- Recursive case: find all children at each level
    SELECT
        c.category_id,
        c.parent_category_id,
        c.category_name,
        h.hierarchy_level + 1,
        CONCAT(h.path, ' > ', c.category_id)
    FROM {{ ref('stg_categories') }} c
    INNER JOIN category_hierarchy h
        ON c.parent_category_id = h.category_id
    WHERE h.hierarchy_level < 10  -- Prevent infinite recursion
)

SELECT
    *,
    CURRENT_TIMESTAMP as loaded_at
FROM category_hierarchy
```

Question 7: How Do You Use Incremental Models Effectively?

Explanation

Incremental models only process new or changed data instead of rebuilding from scratch. This drastically reduces run time and costs. You specify a unique key and an incremental strategy (append, delete+insert, merge). The first run creates the table; subsequent runs add only new records. This is essential for large tables in production environments.

Code Implementation

sql

```sql
-- models/silver/stg_events_incremental.sql
{{ config(
    materialized='incremental',
    unique_key='event_id',
    incremental_strategy='merge',
    merge_exclude_columns=['_dbt_ingestion_time']
) }}

SELECT
    event_id,
    user_id,
    event_type,
    event_timestamp,
    event_properties,
    CURRENT_TIMESTAMP as _dbt_ingestion_time
FROM {{ source('raw_events', 'events') }}

-- Only process new events on incremental runs
{% if execute %}
  {% if var('full_refresh') is false %}
    WHERE event_timestamp > (
      SELECT COALESCE(MAX(event_timestamp), CAST('2000-01-01' AS TIMESTAMP))
      FROM {{ this }}
    )
  {% endif %}
{% endif %}
```

Question 8: How Do You Write Custom Macros in dbt?

Explanation

Macros are reusable Jinja2 functions that generate SQL. They enable code reuse, parameterization, and dynamic SQL generation. Custom macros can encapsulate complex logic like surrogate key generation, data quality checks, or transformation patterns. Macros are compiled at run time and inserted into models as SQL.

Code Implementation

sql

```
-- macros/generate_date_spine.sql
{% macro generate_date_spine(start_date, end_date) %}
  {% if execute %}
    {% set days = var('date_spine_days', 365) %}
  {% else %}
    {% set days = 365 %}
  {% endif %}

  WITH date_range AS (
    SELECT DATEADD('day', seq, '{{ start_date }}') as date_day
    FROM (
      SELECT row_number() over (order by null) - 1 as seq
      FROM TABLE(GENERATOR(rowcount => {{ days }}))
    )
  )
  SELECT * FROM date_range
  WHERE date_day <= '{{ end_date }}'
{% endmacro %}

-- Usage in model:
-- models/date_spine.sql
{{ generate_date_spine('2020-01-01', '2025-12-31') }}
```

sql

```
-- macros/cents_to_dollars.sql
{% macro cents_to_dollars(column_name) %}
  ROUND({{ column_name }} / 100.0, 2)
{% endmacro %}

-- Usage:
-- SELECT {{ cents_to_dollars('amount_cents') }} as amount FROM orders
```

Question 9: How Do You Handle Data Freshness Checks?

Explanation

Data freshness monitoring ensures your data warehouse receives timely updates. Use the `freshness` property to define when data should be considered stale. dbt automatically checks if source data is fresh and warns/fails if updates are delayed. This catches pipeline failures early before they impact downstream users.

Code Implementation

yaml

```yaml
# models/sources.yml
sources:
  - name: operational_db
    database: raw_database
    schema: public
    freshness:
      warn_after: {count: 12, period: hour}
      error_after: {count: 24, period: hour}
    loaded_at_field: updated_at

    tables:
      - name: customers
        description: Customer source table
        columns:
          - name: customer_id
          - name: email
          - name: updated_at

      - name: orders
        freshness:  # Override source-level freshness
          warn_after: {count: 2, period: hour}
          error_after: {count: 4, period: hour}
        columns:
          - name: order_id
          - name: updated_at
```

bash

```bash
# Run freshness checks
dbt source freshness

# Output shows which sources are stale
# Example: Source 'operational_db.orders' last loaded 6 hours ago (ERROR)
```

Question 10: How Do You Set Up Multi-Environment Deployments?

Explanation

Production, staging, and development environments allow safe testing before deploying changes. Use dbt profiles to configure different databases/schemas for each environment. Separate CI/CD pipelines validate changes before they reach production. This reduces risk and allows developers to test independently.

Code Implementation

yaml

```yaml
# profiles.yml (never commit to version control)
my_project:
  outputs:
    dev:
      type: snowflake
      account: myaccount.us-east-1
      user: dev_user
      schema: analytics_dev
      database: analytics_dev
      warehouse: dev_wh
      threads: 4

    staging:
      type: snowflake
      account: myaccount.us-east-1
      user: staging_user
      schema: analytics_staging
      database: analytics_prod
      warehouse: compute_wh
      threads: 6

    prod:
      type: snowflake
      account: myaccount.us-east-1
      user: prod_user
      schema: analytics
      database: analytics_prod
      warehouse: prod_wh
      threads: 8  # More threads for production performance

  target: dev  # Default environment
```

bash

```
# Deploy to different environments
dbt run --target dev      # Development
dbt run --target staging  # Staging
dbt run --target prod     # Production

# Run with specific target and select models
dbt run --target prod -s tag:critical

# Test before production deployment
dbt test --target staging -s state:modified+
```

yaml

```yaml
# .github/workflows/dbt-ci.yml - CI/CD Pipeline
name: dbt CI/CD

on:
  pull_request:
    branches: [main]
  push:
    branches: [main]

jobs:
  dbt-validation:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install dbt
        run: pip install dbt-snowflake

      - name: Test on staging
        run: dbt run --target staging --select state:modified+

      - name: Run tests on staging
        run: dbt test --target staging --select state:modified+

      - name: Deploy to production (on main merge)
        if: github.ref == 'refs/heads/main'
        run: dbt run --target prod
```

Summary

These 10 questions cover the most important aspects of dbt interview preparation:

1. **Architecture** - Medallion pattern for data layering
2. **Advanced Modeling** - SCD Type 2 for dimension history tracking
3. **Schema Design** - Star schema for analytical databases
4. **Data Versioning** - Snapshots for point-in-time data capture
5. **Dependencies** - Understanding ref() vs source()
6. **Complex Queries** - Recursive models for hierarchies
7. **Performance** - Incremental models for efficiency
8. **Code Reusability** - Custom macros for DRY principles
9. **Data Quality** - Freshness monitoring for reliability

10. **Operations** - Multi-environment deployments for safety

Focus on understanding the concepts deeply and be prepared to explain the trade-offs of different approaches during your interview.