DBT Developer

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. What are the key differences between ephemeral, view, table, and incremental materialization strategies in DBT, and when would you choose each?

Materialization strategies determine how DBT builds models in your data warehouse:

Ephemeral

- Not materialized in the warehouse; exists only as CTEs in dependent models
- Use for lightweight transformations used by multiple downstream models
- Reduces warehouse object clutter but can lead to complex query plans

View

- Creates a database view; query runs at read time
- Best for lightweight transformations with low query frequency
- No storage cost but higher compute on each query

Table

- Full refresh on every run; drops and recreates the table
- Ideal for smaller datasets or when complete rebuilds are acceptable
- Guarantees data freshness but resource-intensive

Incremental

- Only processes new/changed records since last run
- Essential for large fact tables and event streams
- Requires proper unique keys and merge logic

Selection criteria: Choose based on data volume, query frequency, freshness requirements, and compute costs. Incremental for large growing datasets, tables for dimension tables, views for simple transformations, and ephemeral for shared logic.

2. Explain how DBT's ref() and source() functions work and why they are critical for DAG construction.

ref() and **source()** are DBT's dependency resolution functions that enable automatic DAG construction:

ref() Function

- References other DBT models within your project
- Automatically resolves to the correct schema and table name
- Creates edges in the DAG, ensuring proper execution order
- Environment-aware: resolves differently in dev vs prod

```
SELECT *
FROM {{ ref('stg_customers') }}
WHERE status = 'active'
```

source() Function

- References raw tables defined in sources.yml
- Documents external dependencies and enables freshness checks
- Provides a single point of configuration for source tables

```
SELECT *
FROM {{ source('raw_db', 'customers') }}
WHERE loaded_at > current_date - 7
```

Critical benefits: These functions enable DBT to automatically determine execution order, parallelize independent models, detect circular dependencies, and provide lineage visualization. Without them, you'd need manual dependency management and lose environment portability.

3. How do you implement and optimize an incremental model in DBT? What are the different incremental strategies available?

Incremental models process only new or changed data, dramatically reducing build times for large tables:

Implementation Pattern

```
{{ config(
  materialized='incremental',
  unique_key='event_id'
) }}

SELECT * FROM {{ source('events', 'raw') }}
{% if is_incremental() %}
  WHERE created_at > (SELECT MAX(created_at) FROM {{ this }})
{% endif %}
```

Incremental Strategies

- **append:** Simply adds new rows; no deduplication (fastest, use for immutable events)
- **merge:** Updates existing rows and inserts new ones based on unique_key (default for most warehouses)
- **delete+insert:** Deletes matching records then inserts; useful for partition-based updates
- **insert_overwrite:** Overwrites entire partitions (Bigquery-specific, very efficient)

Optimization Tips

- Use partition columns in incremental filters to minimize scans
- Set appropriate unique_keys to prevent duplicates
- Implement lookback windows for late-arriving data
- Consider full-refresh schedules to prevent drift
- Monitor model timing to ensure incremental remains faster than full refresh

4. What is the purpose of DBT macros, and how would you create a reusable macro for generating surrogate keys?

DBT macros are Jinja templates that generate reusable SQL code, enabling DRY principles and complex logic abstraction:

Purpose and Benefits

- Eliminate repetitive SQL patterns across models
- Encapsulate complex business logic in testable units
- Enable cross-database compatibility through adapter-specific implementations
- Support parameterization for flexible transformations

Surrogate Key Macro Example

```
{% macro generate_surrogate_key(columns) %}
  {{ dbt_utils.surrogate_key(columns) }}
{% endmacro %}

-- Usage in model:
SELECT
  {{ generate_surrogate_key(['customer_id', 'order_date']) }} as sk,
  *
FROM {{ ref('orders') }}
```

Advanced Custom Implementation

```
{% macro hash_key(fields) %}
  md5(concat_ws('|',
    {% for field in fields %}
      coalesce(cast({{ field }} as string), '')
      {{- ',' if not loop.last }}
    {% endfor %}
  ))
{% endmacro %}
```

Best practices: Store macros in the macros/ directory, document parameters clearly, write tests for macro logic, and leverage dbt_utils package for common patterns before creating custom implementations.

5. Explain DBT's testing framework. How do you implement custom schema tests and data tests?

DBT testing ensures data quality through two test types: schema tests (generic) and data tests (singular):

Schema Tests (Generic)

- Applied in YAML configuration files
- Built-in: unique, not_null, accepted_values, relationships
- Run against columns or model configurations

```
models:
  - name: customers
    columns:
      - name: customer_id
        tests:
          - unique
          - not_null
      - name: status
        tests:
          - accepted_values:
              values: ['active', 'inactive']
```

Custom Generic Test

```
-- tests/generic/test_valid_email.sql
{% test valid_email(model, column_name) %}
SELECT *
FROM {{ model }}
WHERE {{ column_name }} NOT LIKE '%_@__%.__%'
{% endtest %}
```

Singular Data Tests

- Stored as SQL files in tests/ directory
- Must return failing rows (0 rows = pass)

```
-- tests/assert_revenue_positive.sql
SELECT order_id, total_amount
FROM {{ ref('fct_orders') }}
WHERE total_amount <= 0
```

Best practices: Test critical business logic, use severity levels (warn/error), implement tests in CI/CD pipelines, and store test results for trend analysis.

6. How does DBT handle snapshots, and what are Type 2 Slowly Changing Dimensions? Provide an implementation example.

DBT snapshots implement Type 2 Slowly Changing Dimensions (SCD) to track historical changes in mutable source data:

Type 2 SCD Concept

- Preserves full history by creating new rows for changes
- Uses validity timestamps (valid_from, valid_to) to track record lifespans
- Maintains current and historical versions simultaneously
- Essential for auditing and historical analysis

Snapshot Implementation

```
{% snapshot customers_snapshot %}
{{config(
  target_schema='snapshots',
  unique_key='customer_id',
  strategy='timestamp',
  updated_at='updated_at'
)}}
SELECT * FROM {{ source('crm', 'customers') }}
{% endsnapshot %}
```

Snapshot Strategies

- **timestamp:** Detects changes via updated_at column (most common)
- **check:** Compares specified columns to detect changes (use when no timestamp available)

Generated Metadata Columns

- dbt_valid_from: When this version became active
- dbt_valid_to: When superseded (NULL for current)
- dbt_scd_id: Unique identifier for each version
- dbt_updated_at: Snapshot execution timestamp

Usage: Query with WHERE dbt_valid_to IS NULL for current state, or join on date ranges for point-in-time analysis.

7. What are DBT packages, and how would you create and distribute a custom package for your organization?

DBT packages are reusable collections of models, macros, and tests that can be shared across projects:

Purpose and Benefits

- Share common transformations across multiple DBT projects
- Leverage community-built solutions (dbt_utils, dbt_expectations)
- Standardize data modeling patterns organization-wide
- Version-control shared business logic

Installing Packages

```
-- packages.yml
packages:
  - package: dbt-labs/dbt_utils
    version: 1.1.1
  - git: "https://github.com/org/custom-pkg"
    revision: main
```

Creating a Custom Package

- Create a separate Git repository with standard DBT structure
- Include macros/, models/, and tests/ directories
- Add dbt_project.yml with package configuration
- Document macros and models in README

```
-- dbt_project.yml in package
name: 'company_metrics'
version: '1.0.0'
config-version: 2
macro-paths: ["macros"]
model-paths: ["models"]
```

Distribution Options

- Git repositories (public or private)
- DBT Hub for public packages
- Internal package registries

Best practices: Semantic versioning, comprehensive documentation, example projects, and automated testing of package code.

8. Explain DBT's compilation and execution phases. How does the Jinja templating engine work during compilation?

DBT execution occurs in two distinct phases: compilation and execution:

Compilation Phase

- Parses all model files and resolves Jinja templates
- Executes ref(), source(), and macro calls to build DAG

- Generates raw SQL from Jinja-templated files
- Validates syntax and dependencies
- Stores compiled SQL in target/compiled/ directory

Jinja Templating Process

```
-- Source model with Jinja
SELECT
  {{ dbt_utils.surrogate_key(['id', 'date']) }} as sk,
  *
FROM {{ ref('staging_orders') }}
{% if target.name == 'prod' %}
WHERE status != 'test'
{% endif %}

-- Compiled SQL output
SELECT
  md5(concat(id, '-', date)) as sk,
  *
FROM prod.staging_orders
WHERE status != 'test'
```

Execution Phase

- Runs compiled SQL against the data warehouse
- Executes models in DAG order, respecting dependencies
- Materializes results according to config
- Logs run metadata and test results

Key insight: Jinja executes at compile time on your machine, not in the warehouse. This enables environment-specific logic, dynamic SQL generation, and metadata-driven transformations without warehouse-specific features.

9. How do you implement incremental models with late-arriving data and handle data quality issues in production pipelines?

Late-arriving data and quality issues require defensive incremental strategies:

Lookback Window Pattern

```
{{ config(
  materialized='incremental',
  unique_key='event_id'
) }}

SELECT * FROM {{ source('events', 'raw') }}
{% if is_incremental() %}
  WHERE created_at > (
    SELECT DATEADD(day, -3, MAX(created_at))
    FROM {{ this }}
  )
{% endif %}
```

- Reprocesses last N days to capture late arrivals
- Balances freshness with processing efficiency
- Configurable lookback based on SLA requirements

Data Quality Safeguards

- **Pre-hook validation:** Check source row counts before processing
- **Post-hook reconciliation:** Compare incremental vs full refresh counts
- **Schema enforcement:** Use schema.yml to validate data types
- **Anomaly detection:** Test for unexpected NULL rates or value distributions

```
{{ config(
  pre_hook="{{ validate_source_freshness() }}",
  post_hook="{{ reconcile_counts() }}"
) }}
```

Production Patterns

- Implement full-refresh fallback for severe data issues
- Use on-run-end hooks for alerting on test failures
- Partition by date for efficient late-arrival handling
- Monitor incremental model drift with periodic full refreshes

10. What are DBT exposures and metrics? How do they integrate with BI tools and improve data governance?

Exposures and metrics document downstream dependencies and standardize business logic:

Exposures

- Define how DBT models are used in dashboards, reports, or applications
- Create bidirectional lineage between transformations and consumption
- Enable impact analysis when modifying upstream models

```
exposures:
  - name: weekly_sales_dashboard
    type: dashboard
    owner:
      name: Analytics Team
      email: analytics@company.com
```

```
    depends_on:
      - ref('fct_sales')
      - ref('dim_products')
    url: https://bi.company.com/sales
    description: Executive sales performance dashboard
```

Metrics (DBT 1.0+)

- Define business metrics once, query anywhere
- Ensure consistent calculation logic across tools
- Support time-based aggregations and dimensions

```
metrics:
  - name: monthly_revenue
    model: ref('fct_orders')
    calculation_method: sum
    expression: order_amount
    timestamp: order_date
    time_grains: [day, week, month]
    dimensions: [region, product_category]
```

Governance Benefits

- Single source of truth for metric definitions
- Automatic documentation in DBT Docs
- Impact analysis when changing core models
- Integration with BI tools via semantic layer

Integration: Exposures connect to Tableau, Looker, Mode; metrics integrate with MetricFlow for semantic querying across tools.

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU (Least Recently Used) Cache with O(1) time complexity for both get and put operations.

LRU Cache Implementation

An **LRU Cache** requires a combination of a **doubly linked list** and a **hash map** to achieve O(1) operations.

- **Hash Map:** Stores key-node pairs for O(1) lookup
- **Doubly Linked List:** Maintains order of usage (most recent at head)
- **Get Operation:** Move accessed node to head
- **Put Operation:** Add to head, evict tail if capacity exceeded

```
class LRUCache:
    def __init__(self, capacity):
        self.cache = {}
        self.cap = capacity
        self.head = Node(0, 0)
        self.tail = Node(0, 0)
        self.head.next, self.tail.prev = self.tail, self.head
```

Time Complexity: O(1) for both get and put operations

2. Given an unsorted array, find all pairs that sum to a target value. What's the optimal approach?

Two Sum - All Pairs Solution

Use a **hash set** to track seen numbers while iterating through the array.

```
def find_pairs(arr, target):
    seen = set()
    pairs = set()
    for num in arr:
        complement = target - num
        if complement in seen:
            pairs.add((min(num, complement), max(num, complement)))
        seen.add(num)
    return list(pairs)
```

- **Time Complexity:** O(n) - single pass through array
- **Space Complexity:** O(n) - hash set storage
- **Key Insight:** For each number, check if (target - number) exists in set
- **Alternative:** Sort array O(n log n) then use two pointers O(n)

3. Explain the difference between a Stack and a Queue, and provide a real-world use case for each.

Stack vs Queue

Stack (LIFO - Last In First Out):

- Operations: push(), pop(), peek() - all O(1)
- Use Case: Function call stack, undo/redo functionality, expression evaluation

```
stack = []
stack.append(1)  # push
stack.pop()      # pop
stack[-1]        # peek
```

Queue (FIFO - First In First Out):

- Operations: enqueue(), dequeue() - all O(1)
- Use Case: Task scheduling, BFS traversal, message queues

```
from collections import deque
queue = deque()
queue.append(1)     # enqueue
queue.popleft()     # dequeue
```

4. Implement a function to find the maximum sum of a subarray using Kadane's Algorithm.

Kadane's Algorithm

Kadane's Algorithm finds the maximum sum contiguous subarray in O(n) time using dynamic programming.

```
def max_subarray_sum(arr):
    max_sum = current_sum = arr[0]
    for num in arr[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)
    return max_sum
```

- **Time Complexity:** O(n) - single pass
- **Space Complexity:** O(1) - constant space

- **Key Insight:** At each position, decide whether to extend existing subarray or start new one
- **Example:** [-2,1,-3,4,-1,2,1,-5,4] returns 6 (subarray [4,-1,2,1])

5. What is a Trie data structure and when would you use it? Implement basic insert and search operations.

Trie (Prefix Tree)

A **Trie** is a tree-like data structure for storing strings, optimized for prefix-based searches.

- **Use Cases:** Autocomplete, spell checking, IP routing, dictionary implementation
- **Advantages:** O(m) search time where m is key length, prefix matching

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

class Trie:
    def insert(self, word):
        node = self.root
        for char in word:
            node = node.children.setdefault(char, TrieNode())
        node.is_end = True
```

Space Complexity: O(ALPHABET_SIZE * N * M) where N is number of keys and M is average length

6. Implement a sliding window algorithm to find the maximum sum of k consecutive elements in an array.

Sliding Window Technique

The **sliding window** technique optimizes problems involving contiguous subarrays by avoiding redundant calculations.

```
def max_sum_k_consecutive(arr, k):
    if len(arr) < k:
        return None
    window_sum = sum(arr[:k])
    max_sum = window_sum
    for i in range(k, len(arr)):
        window_sum = window_sum - arr[i-k] + arr[i]
        max_sum = max(max_sum, window_sum)
    return max_sum
```

- **Time Complexity:** O(n) - single pass after initial window
- **Space Complexity:** O(1)
- **Key Insight:** Subtract element leaving window, add element entering window
- **Brute Force:** Would be O(n*k) calculating sum for each window

7. Explain hash collisions and describe two collision resolution techniques with their trade-offs.

Hash Collision Resolution

A **hash collision** occurs when two keys produce the same hash value.

1. Chaining (Separate Chaining):

- Each bucket contains a linked list of entries
- **Pros:** Simple, handles high load factors well
- **Cons:** Extra memory for pointers, cache unfriendly
- **Time:** O(1) average, O(n) worst case

2. Open Addressing (Linear Probing):

- Find next available slot in array
- **Pros:** Better cache performance, no extra memory
- **Cons:** Clustering issues, requires good hash function
- **Time:** O(1) average, degrades with high load factor

Load Factor: n/m (entries/buckets) - resize when exceeds 0.7-0.75

8. Implement a function to detect a cycle in a linked list using Floyd's Cycle Detection Algorithm.

Floyd's Cycle Detection (Tortoise and Hare)

Uses **two pointers** moving at different speeds to detect cycles in O(n) time with O(1) space.

```
def has_cycle(head):
    if not head:
        return False
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

- **Time Complexity:** O(n)
- **Space Complexity:** O(1)
- **Key Insight:** If cycle exists, fast pointer will eventually catch slow pointer
- **Extension:** To find cycle start, reset one pointer to head and move both one step at a time

9. What is the difference between a Binary Search Tree and a Balanced Binary Search Tree? Why does balancing matter?

BST vs Balanced BST

Binary Search Tree (BST):

- Left subtree < node < right subtree
- **Operations:** Search, insert, delete
- **Time Complexity:** O(h) where h is height
- **Worst Case:** O(n) when tree becomes skewed (linked list)

Balanced BST (AVL, Red-Black Tree):

- Maintains height balance: h = O(log n)
- **Guaranteed Time:** O(log n) for all operations
- **Self-Balancing:** Rotations after insertions/deletions

Why Balancing Matters: Unbalanced BST with n nodes inserted in sorted order degrades to O(n) operations. Balanced trees guarantee O(log n) performance for databases, file systems, and memory management.

10. Implement a function to find the kth largest element in an unsorted array. What are different approaches and their complexities?

Kth Largest Element

Approach 1: Min Heap (Optimal for small k)

```
import heapq
def find_kth_largest(nums, k):
    heap = nums[:k]
    heapq.heapify(heap)
    for num in nums[k:]:
        if num > heap[0]:
            heapq.heapreplace(heap, num)
    return heap[0]
```

Time: O(n log k), **Space:** O(k)

Approach 2: QuickSelect (Average O(n))

- Partition array like quicksort
- Recursively search only one partition
- **Time:** O(n) average, $O(n^2)$ worst case

Approach 3: Sorting - O(n log n) - Simple but not optimal

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. How would you design a scalable data transformation pipeline using dbt for a company processing 100TB+ of data daily?

Architecture Overview

For a large-scale dbt pipeline processing 100TB+ daily, consider the following design:

- **Incremental Processing:** Use dbt incremental models with appropriate unique keys and merge strategies to process only new/changed data
- **Partitioning Strategy:** Leverage warehouse-native partitioning (date-based) to limit data scanned
- **Modular Design:** Break transformations into staging, intermediate, and mart layers
- **Parallel Execution:** Configure dbt to run independent models concurrently using thread pools
- **Resource Management:** Use dbt model configs to specify warehouse sizes/clusters per model complexity

Example Configuration

```
{{ config(
  materialized='incremental',
  unique_key='event_id',
  partition_by={'field': 'event_date', 'data_type': 'date'},
  cluster_by=['user_id'],
  incremental_strategy='merge'
) }}
```

Scalability Considerations

- **State Management:** Use dbt state files with --defer flag for CI/CD to test only modified models
- **Orchestration:** Integrate with Airflow/Prefect for dependency management across multiple dbt projects
- **Monitoring:** Implement dbt artifacts analysis for performance tracking and SLA monitoring
- **Data Quality:** Use dbt tests and custom data quality checks at each layer

2. Design a real-time analytics system where dbt models need to support both batch and streaming data sources. How would you architect this?

Hybrid Architecture Design

A Lambda/Kappa architecture approach works well for combining batch and streaming with dbt:

- **Streaming Layer:** Use tools like Kafka/Kinesis → Stream processor (Flink/Spark Streaming) → Data Warehouse staging tables
- **Batch Layer:** Traditional ETL → Data Warehouse → dbt transformations
- **Serving Layer:** dbt models that union/merge both sources with deduplication logic

Implementation Strategy

```
-- Unified model combining streams
WITH streaming_data AS (
  SELECT * FROM {{ ref('streaming_events') }}
  WHERE updated_at > current_timestamp - interval '1 hour'
),
batch_data AS (
  SELECT * FROM {{ ref('batch_events') }}
  WHERE event_date = current_date
)
SELECT * FROM streaming_data
UNION ALL
SELECT * FROM batch_data
```

Key Design Decisions

- **Microbatch Pattern:** Run dbt every 5-15 minutes on streaming staging tables for near-real-time
- **Watermarking:** Implement event time vs processing time logic to handle late-arriving data
- **Deduplication:** Use window functions or merge strategies to handle duplicate events across layers
- **SLA Tiering:** Different refresh rates for different mart layers (real-time vs hourly vs daily)

3. How would you design a multi-tenant dbt architecture where each tenant's data must be isolated, but transformations are shared?

Multi-Tenant Design Patterns

There are several approaches to multi-tenant dbt architectures:

- **Schema-per-Tenant:** Each tenant gets dedicated schemas with shared transformation logic
- **Table-per-Tenant:** Use tenant_id column with row-level security
- **Database-per-Tenant:** Complete isolation with dbt project per tenant or dynamic target configuration

Recommended Approach: Schema-per-Tenant with Macros

```
-- Macro to generate tenant-specific refs
{% macro tenant_ref(model_name, tenant_id) %}
```

```
  {{ ref(model_name) }}_{{ tenant_id }}
{% endmacro %}

-- Model using tenant macro
SELECT *
FROM {{ tenant_ref('raw_orders', var('tenant_id')) }}
WHERE status = 'completed'
```

Orchestration Strategy

- **Dynamic Execution:** Use Airflow to loop through tenant list and execute dbt with --vars '{tenant_id: xyz}'
- **Shared Models:** Create base transformation logic that's tenant-agnostic
- **Custom Schemas:** Use dbt custom schema configurations to route models to tenant-specific schemas
- **Access Control:** Implement warehouse-level grants to ensure tenant isolation
- **Performance:** Run tenants in parallel with separate thread pools to optimize execution time

4. Design a dbt project structure for a company with multiple business units that need both shared and unit-specific metrics. How do you handle dependencies and governance?

Project Structure Design

A well-organized multi-business-unit dbt project requires clear separation and dependency management:

- **Core dbt Project:** Contains shared staging models, utilities, and enterprise-wide dimensions
- **Business Unit Projects:** Separate dbt projects that depend on core using dbt packages or cross-project refs
- **Metrics Layer:** Centralized semantic layer using dbt metrics for consistent definitions

Directory Structure

```
dbt_projects/
├── core/
│   ├── models/staging/
│   ├── models/dimensions/
│   └── macros/
├── finance/
│   ├── models/marts/
│   └── dependencies.yml
└── marketing/
    ├── models/marts/
    └── dependencies.yml
```

Dependency Management

- **dbt Packages:** Publish core project as internal package, referenced in packages.yml of BU projects
- **Cross-Database Refs:** Use two-argument ref() for cross-project dependencies: ref('core', 'dim_customers')
- **API Contracts:** Define and version model contracts in core project to prevent breaking changes
- **Governance Layer:** Implement dbt meta tags for data classification, ownership, and SLA requirements

Metric Governance

```
-- metrics.yml in core project
metrics:
  - name: monthly_revenue
    label: Monthly Revenue
    model: ref('fct_orders')
    calculation_method: sum
    expression: order_amount
    timestamp: order_date
    time_grains: [day, month, quarter]
```

5. How would you implement a Change Data Capture (CDC) system in dbt to track slowly changing dimensions (SCD Type 2) for a customer dimension with millions of records?

SCD Type 2 Implementation Strategy

Implementing efficient CDC for large dimensions requires careful design:

- **Snapshot Strategy:** Use dbt snapshots with timestamp strategy for automatic SCD Type 2 tracking
- **Incremental Processing:** Process only changed records using source system CDC feeds
- **Surrogate Keys:** Generate deterministic surrogate keys for version tracking

dbt Snapshot Configuration

```
{% snapshot customers_snapshot %}
{{config(
  target_schema='snapshots',
  unique_key='customer_id',
  strategy='timestamp',
  updated_at='updated_timestamp',
  invalidate_hard_deletes=True
)}}
SELECT * FROM {{ source('erp', 'customers') }}
{% endsnapshot %}
```

Performance Optimization

- **Partition by Date:** Partition snapshot tables by dbt_valid_from to improve query performance
- **Incremental Source:** Only read changed records from source using watermark timestamps
- **Clustering:** Cluster by customer_id and dbt_valid_from for efficient lookups
- **Merge Optimization:** Use warehouse-specific merge optimizations (COPY grants, cluster keys)

Handling Large Volumes

- **Micro-batching:** Run snapshots multiple times per day to reduce processing windows
- **Parallel Processing:** Partition customers by region/segment and process in parallel
- **Archival Strategy:** Move historical versions to cold storage after retention period

6. Design a dbt testing and data quality framework that can handle complex business rules, cross-table validations, and generate automated alerts. What's your approach?

Comprehensive Data Quality Framework

A robust dbt testing framework should include multiple layers of validation:

- **Generic Tests:** Standard dbt tests (unique, not_null, relationships, accepted_values)
- **Singular Tests:** Custom SQL tests for complex business logic
- **Custom Generic Tests:** Reusable test macros for common patterns
- **dbt Expectations:** Package for advanced statistical and pattern-based tests

Custom Generic Test Example

```
-- tests/generic/revenue_reconciliation.sql
{% test revenue_reconciliation(model, source_table) %}
SELECT
  'revenue_mismatch' as error_type,
  ABS(a.total - b.total) as difference
FROM (SELECT SUM(amount) as total FROM {{model}}) a
CROSS JOIN (SELECT SUM(amount) as total FROM {{source_table}}) b
WHERE ABS(a.total - b.total) > 0.01
{% endtest %}
```

Multi-Layer Testing Strategy

- **Source Tests:** Validate raw data quality at ingestion (freshness, volume anomalies)
- **Staging Tests:** Data type validation, format checks, referential integrity
- **Intermediate Tests:** Business logic validation, calculation accuracy
- **Mart Tests:** Aggregate reconciliation, metric consistency, SLA compliance

Alerting and Monitoring

- **Test Severity:** Use severity levels (warn vs error) in test configs
- **Elementary Data:** Integrate Elementary for automated anomaly detection and Slack alerts
- **Test Results Tracking:** Parse test results from run_results.json and send to monitoring system
- **Business Rule Engine:** Store complex rules in YAML, generate tests dynamically using macros

7. How would you design a dbt deployment strategy supporting multiple environments (dev, staging, prod) with proper CI/CD, including slim CI and blue-green deployments?

Multi-Environment CI/CD Architecture

A production-grade dbt deployment requires sophisticated CI/CD pipelines:

- **Environment Strategy:** Separate targets in profiles.yml for dev, staging, and prod
- **Git Workflow:** Feature branches → PR → staging → main → production
- **Slim CI:** Use dbt state comparison to test only modified models and their downstream dependencies
- **Blue-Green Deployment:** Deploy to alternate schema, validate, then swap

Slim CI Pipeline Example

```
# .github/workflows/slim_ci.yml
- name: Run Slim CI
  run: |
    dbt deps
    dbt build --select state:modified+ \
      --defer --state ./prod-state/ \
      --target ci \
      --profiles-dir .
- name: Run tests
  run: dbt test --select state:modified+
```

Blue-Green Deployment Strategy

- **Schema Cloning:** Create prod_blue and prod_green schemas
- **Deployment Process:** Deploy to inactive schema (green), run full test suite, swap views
- **Rollback Capability:** Keep previous schema active for instant rollback
- **Zero Downtime:** Use views in a presentation layer that point to active schema

Advanced CI/CD Features

- **Automated Docs:** Generate and publish dbt docs on every merge to main
- **Performance Regression:** Compare model execution times against baseline
- **Schema Change Detection:** Alert on breaking schema changes in production models
- **Automated Rollback:** Trigger rollback pipeline if critical tests fail post-deployment

8. Design a dbt solution for handling late-arriving facts and slowly changing dimensions in a data warehouse serving real-time dashboards. How do you ensure consistency?

Late-Arriving Data Architecture

Handling late-arriving data requires temporal consistency mechanisms:

- **Event Time vs Processing Time:** Store both event_timestamp and processed_timestamp
- **Lookback Windows:** Reprocess recent partitions to incorporate late data
- **Idempotent Models:** Design models to produce same results regardless of run time
- **Version Tracking:** Use SCD Type 2 for dimensions to maintain historical accuracy

Incremental Model with Lookback

```
{{ config(materialized='incremental') }}
SELECT *
FROM {{ source('events', 'transactions') }}
WHERE event_timestamp >=
  {% if is_incremental() %}
    (SELECT MAX(event_timestamp) - interval '3 days'
     FROM {{ this }})
  {% else %}
    '2020-01-01'
  {% endif %}
```

Consistency Mechanisms

- **Fact-Dimension Alignment:** Use point-in-time joins to match facts with correct dimension versions
- **Restatement Logic:** Flag and version fact records that were updated due to late arrivals
- **Watermark Tables:** Track processing watermarks separately from data timestamps
- **Reconciliation Jobs:** Periodic full refresh of critical aggregates to ensure accuracy

Real-Time Dashboard Strategy

- **Hybrid Materialization:** Use views for latest data, incremental tables for historical
- **Cache Warming:** Pre-aggregate common queries in materialized tables
- **Freshness Indicators:** Include metadata showing last update time and data latency
- **Eventually Consistent:** Accept short-term inconsistency with clear SLAs for stabilization

9. How would you architect a dbt project to support both analytical workloads and operational reporting with different latency requirements (real-time vs batch)?

Dual-Speed Architecture Design

Supporting different latency SLAs requires a tiered architecture:

- **Hot Path (Real-time):** Lightweight transformations, materialized as views or tables with frequent refresh
- **Cold Path (Batch):** Complex transformations, incremental models, daily/hourly refresh
- **Serving Layer:** Unified interface combining both paths based on query patterns

Model Organization

```
models/
├── staging/          # Shared staging (hourly)
├── intermediate/     # Complex logic (daily)
├── marts/
│   ├── operational/  # Real-time (5-15 min)
│   └── analytical/   # Batch (hourly/daily)
└── serving/          # Unified views
```

Differential Refresh Strategy

- **Tag-Based Execution:** Use dbt tags to group models by refresh frequency
- **Separate DAGs:** Create multiple Airflow DAGs with different schedules for each tier
- **Materialization Strategy:** Views for operational, incremental/tables for analytical
- **Resource Allocation:** Different warehouse sizes for real-time vs batch workloads

Operational Model Example

```
{{ config(
  materialized='table',
  tags=['operational', 'real_time'],
  on_schema_change='sync_all_columns'
) }}
SELECT
  order_id,
  customer_id,
  order_status,
  total_amount
FROM {{ ref('stg_orders') }}
WHERE created_at >= current_timestamp - interval '24 hours'
```

Performance Optimization

- **Partition Pruning:** Operational models only process recent partitions
- **Aggregate Rollups:** Pre-compute common operational metrics
- **Caching Layer:** Use BI tool caching for frequently accessed operational reports

10. Design a dbt architecture for a data mesh implementation where domain teams own their data products but need to share and discover data across domains. How do you handle governance and lineage?

Data Mesh Architecture with dbt

Implementing data mesh principles with dbt requires federated ownership with centralized governance:

- **Domain Projects:** Each domain owns a separate dbt project with full autonomy
- **Shared Core:** Central dbt project with common utilities, macros, and enterprise dimensions

- **Data Products:** Well-defined, documented, and tested models exposed as domain APIs
- **Discovery Layer:** Centralized data catalog aggregating metadata from all domains

Cross-Domain Dependencies

```
-- packages.yml in consumer domain
packages:
  - git: "https://github.com/company/sales-domain"
    revision: v1.2.0
  - git: "https://github.com/company/customer-domain"
    revision: v2.1.0

-- Using cross-domain data product
SELECT * FROM {{ ref('sales_domain', 'fct_orders') }}
```

Governance Framework

- **Data Contracts:** Define schema contracts using dbt model contracts with version enforcement
- **Quality Gates:** Mandatory tests and documentation for published data products
- **Access Policies:** Domain teams define and enforce access policies via warehouse grants
- **SLA Definitions:** Document refresh schedules and data freshness guarantees in model meta

Lineage and Discovery

- **Unified Catalog:** Aggregate dbt artifacts (manifest.json) from all domains into central catalog
- **Cross-Project Lineage:** Use dbt Cloud Discovery API or parse manifests to build complete lineage graph
- **Metadata Enrichment:** Use meta tags for domain, owner, classification, and business glossary terms
- **Automated Documentation:** Generate domain-specific docs sites with cross-linking to dependencies

Platform Engineering

- **Template Projects:** Provide standardized dbt project templates for new domains
- **Shared CI/CD:** Centralized pipelines with domain-specific customization hooks
- **Observability:** Cross-domain monitoring dashboard showing health of all data products

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. How do you create a custom incremental materialization strategy in DBT?

Custom Incremental Materialization

To create a custom incremental strategy, you define a macro that implements the logic for merging new data. Here's an example:

```
{% macro get_custom_incremental_sql(target_relation, temp_relation, unique_key, dest_columns, strategy) %}
  {% if strategy == 'custom_merge' %}
    merge into {{ target_relation }} as target
    using {{ temp_relation }} as source
    on target.{{ unique_key }} = source.{{ unique_key }}
    when matched then update set {% for col in dest_columns %}{{ col }} = source.{{ col }}{% if not loop.last %}, {% endif %}{% endfor %}
    when not matched then insert ({% for col in dest_columns %}{{ col }}{% if not loop.last %}, {% endif %}{% endfor %})
    values ({% for col in dest_columns %}source.{{ col }}{% if not loop.last %}, {% endif %}{% endfor %})
  {% endif %}
{% endmacro %}
```

Key points:

- Use **macros** to define custom logic
- Reference target and temp relations for merge operations
- Specify the strategy in your model's config block
- Handle unique_key for proper deduplication

2. Write a DBT macro to dynamically pivot columns based on database values.

Dynamic Pivot Macro

Here's a macro that generates pivot SQL dynamically:

```
{% macro pivot_columns(column_name, value_column, relation) %}
  {% set query %}
    select distinct {{ column_name }} from {{ relation }} order by 1
  {% endset %}
  {% set results = run_query(query) %}
  {% if execute %}
    {% set pivot_values = results.columns[0].values() %}
    {% for value in pivot_values %}
      sum(case when {{ column_name }} = '{{ value }}' then {{ value_column }} else 0 end) as {{ value }}
      {%- if not loop.last %}, {% endif %}
    {% endfor %}
  {% endif %}
{% endmacro %}
```

Usage:

- Call this macro in your model SELECT statement
- The **run_query** function executes at compile time
- Use **execute** flag to prevent errors during parsing
- Returns dynamically generated CASE statements

3. How do you debug a failing DBT test and identify the problematic records?

Debugging DBT Tests

To identify failing records in a DBT test:

Method 1: Store Failures

```
-- In dbt_project.yml
tests:
  +store_failures: true
  +schema: dbt_test_failures
```

This creates tables with failing records that you can query directly.

Method 2: Custom Test with Detailed Output

```
{% test detailed_uniqueness(model, column_name) %}
  select {{ column_name }}, count(*) as occurrences
  from {{ model }}
  group by {{ column_name }}
  having count(*) > 1
  order by occurrences desc
{% endtest %}
```

Method 3: Use DBT Commands

- **dbt test --store-failures**: Persist failures to database
- **dbt test --select test_name**: Run specific test
- Query the test failures schema to analyze patterns

- Use **dbt show** to preview compiled SQL

4. Explain how to implement slowly changing dimension (SCD) Type 2 in DBT with snapshots.

SCD Type 2 with DBT Snapshots

DBT snapshots automatically handle SCD Type 2 logic:

```
{% snapshot orders_snapshot %}
  {{ config(
    target_schema='snapshots',
    unique_key='order_id',
    strategy='timestamp',
    updated_at='updated_at',
    invalidate_hard_deletes=True
  ) }}
  select * from {{ source('raw', 'orders') }}
{% endsnapshot %}
```

Key Configuration Options:

- **strategy='timestamp'**: Tracks changes based on updated_at column
- **strategy='check'**: Compares specified columns for changes
- **unique_key**: Identifies the business key
- **invalidate_hard_deletes**: Closes records when source row is deleted
- DBT adds dbt_valid_from, dbt_valid_to, dbt_scd_id columns automatically

Run with: **dbt snapshot**

5. How do you optimize a DBT model that's causing warehouse performance issues?

DBT Model Optimization Strategies

1. Materialization Strategy

- Use **incremental** models for large tables
- Switch from view to table for frequently queried models
- Consider **ephemeral** for simple transformations

2. Query Optimization

```
{{ config(
  materialized='incremental',
  unique_key='id',
  cluster_by=['date_column'],
  partition_by={'field': 'date_column', 'data_type': 'date'}
) }}
```

3. Reduce Data Processing

- Filter early in CTEs to reduce row counts
- Use **where** clause in source/ref for incremental logic
- Avoid SELECT * - specify needed columns only

4. Leverage Warehouse Features

- Add appropriate indexes via post-hooks
- Use clustering keys on Snowflake/BigQuery
- Enable query result caching

6. Write a DBT macro to generate surrogate keys from multiple columns.

Surrogate Key Generation Macro

DBT provides the **dbt_utils.generate_surrogate_key** macro, but here's a custom implementation:

```
{% macro generate_surrogate_key(columns) %}
  {{ dbt_utils.surrogate_key(columns) }}
{% endmacro %}

-- Custom MD5-based implementation:
{% macro custom_surrogate_key(columns) %}
  md5({% for col in columns %}coalesce(cast({{ col }} as string), ''){% if not loop.last %} || '|' || {% endif %}{% endfor %})
{% endmacro %}
```

Usage in Model:

```
select
  {{ custom_surrogate_key(['customer_id', 'order_date']) }} as sk_key,
  *
from {{ ref('orders') }}
```

Best Practices:

- Use **coalesce** to handle NULLs
- Add delimiter between concatenated values
- Cast all columns to string for consistency

7. How do you implement cross-database joins in DBT when working with multiple data sources?

Cross-Database Joins in DBT

Approach 1: Use Database-Specific Features

```
-- Snowflake example with database.schema notation
select
  o.*,
  c.customer_name
from {{ source('sales_db', 'orders') }} o
left join {{ source('crm_db', 'customers') }} c
  on o.customer_id = c.id
```

Approach 2: Stage Data First

- Create staging models that pull data into a common database
- Perform joins in downstream models
- More maintainable and performant

Approach 3: Use External Tables or Federation

- BigQuery: Use federated queries
- Snowflake: Use external tables or shares
- Redshift: Use Redshift Spectrum

Best Practice: Define sources clearly in schema.yml with proper database and schema references to maintain portability.

8. Explain how to use DBT exposures and implement data quality monitoring.

DBT Exposures and Data Quality

Defining Exposures:

```
exposures:
  - name: revenue_dashboard
    type: dashboard
    maturity: high
    owner:
      name: Analytics Team
      email: analytics@company.com
    depends_on:
      - ref('fct_revenue')
      - ref('dim_customers')
    description: Executive revenue dashboard
```

Data Quality Monitoring:

- **dbt test**: Built-in tests (unique, not_null, relationships, accepted_values)
- **dbt_expectations package**: Advanced statistical tests
- **Custom tests**: Business logic validation
- **dbt test --store-failures**: Persist and track failures over time

Monitoring Strategy:

- Schedule tests with your orchestrator
- Alert on test failures via webhooks
- Track test results in metadata tables
- Use exposures to understand downstream impact

9. How do you handle late-arriving data in DBT incremental models?

Handling Late-Arriving Data

Strategy 1: Lookback Window

```
{{ config(materialized='incremental', unique_key='id') }}
select * from {{ source('raw', 'events') }}
{% if is_incremental() %}
  where event_date >= (select max(event_date) - interval '3 days' from {{ this }})
{% endif %}
```

Strategy 2: Update Strategy with Merge

```
{{ config(
  materialized='incremental',
  unique_key='transaction_id',
  incremental_strategy='merge',
  merge_update_columns=['status', 'updated_at']
) }}
```

Best Practices:

- Define appropriate **lookback windows** based on SLA
- Use **merge strategy** instead of append for updates
- Partition by date for efficient lookback queries
- Monitor late-arrival metrics to tune window size
- Consider separate reconciliation jobs for critical data

10. Write a DBT macro to implement row-level security by dynamically filtering data based on user context.

Row-Level Security Macro

Here's a macro that applies dynamic filtering based on user roles:

```
{% macro apply_rls(model_name, user_column, allowed_values) %}
  {% if target.name == 'prod' %}
    select * from {{ ref(model_name) }}
    where {{ user_column }} in (
```

```
      {% for value in allowed_values %}
        '{{ value }}'{% if not loop.last %}, {% endif %}
      {% endfor %}
    )
  {% else %}
    select * from {{ ref(model_name) }}
  {% endif %}
{% endmacro %}
```

Advanced Implementation with Variables:

```
-- In model:
select * from {{ ref('sensitive_data') }}
{% if var('user_role', 'admin') != 'admin' %}
  where region = '{{ var('user_region') }}'
{% endif %}
```

Best Practices:

- Use **target.name** to apply RLS only in production
- Pass user context via DBT variables or environment vars
- Consider database-native RLS features (Snowflake, BigQuery)
- Create separate secured views for different user groups

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to optimize a poorly performing dbt project. What was your approach?

Situation: At my previous company, our dbt project had grown to over 500 models with transformation runs taking 4+ hours, causing delays in data availability for business users.

Task: I was tasked with reducing runtime by at least 50% while maintaining data quality and accuracy.

Action: I conducted a comprehensive audit using dbt artifacts and identified bottlenecks. I implemented incremental models for large fact tables, optimized SQL queries by removing unnecessary joins and CTEs, introduced model tagging for selective runs, and restructured the DAG to improve parallelization. I also configured appropriate materialization strategies and added data quality tests.

Result: Runtime decreased from 4 hours to 1.5 hours (62.5% improvement), and we achieved better resource utilization. The business received data 2.5 hours earlier daily, significantly improving decision-making speed.

2. Describe a situation where you had to handle a data quality issue discovered in production. How did you resolve it?

Situation: A critical revenue reporting model in production was showing a 15% discrepancy compared to the source system, which was discovered during a quarterly business review.

Task: I needed to identify the root cause, implement a fix, and prevent similar issues in the future without disrupting ongoing operations.

Action: I immediately created a hotfix branch and used dbt's --select flag to isolate affected models. After analysis, I found a logic error in a date filter that excluded certain transactions. I corrected the SQL, added comprehensive dbt tests including custom schema tests for data reconciliation, implemented row count and sum checks against source tables, and created documentation explaining the business logic. I also established a peer review process for critical models.

Result: The issue was resolved within 6 hours, historical data was backfilled accurately, and we implemented 12 new data quality tests. No similar incidents occurred in the following year, and stakeholder confidence was restored.

3. Tell me about a time when you had to mentor junior developers on dbt best practices. What was your approach?

Situation: Our team expanded rapidly with three junior data engineers who had SQL experience but were new to dbt and analytics engineering concepts.

Task: I was assigned to mentor them and bring them up to speed within two months so they could contribute independently to our dbt project.

Action: I created a structured onboarding program starting with dbt fundamentals, then progressing to our specific project structure and conventions. I conducted weekly pair programming sessions, established code review guidelines with detailed feedback, created internal documentation with real examples from our codebase, and assigned progressively complex tasks with clear acceptance criteria. I also encouraged questions and maintained an open-door policy.

Result: All three developers became productive contributors within 6 weeks, delivering quality models independently. Two of them later became dbt champions who mentored subsequent hires. The documentation I created became our standard onboarding resource, reducing future onboarding time by 40%.

4. Describe a situation where you had to make a difficult technical decision regarding dbt architecture. How did you approach it?

Situation: Our dbt project was becoming unwieldy with mixed staging, intermediate, and mart models in a single project. We needed to decide between maintaining a monolithic structure or splitting into multiple dbt projects.

Task: I needed to evaluate both approaches, considering team size, deployment complexity, code reusability, and maintenance overhead, then make a recommendation to leadership.

Action: I researched best practices, consulted dbt community resources, and created a detailed comparison matrix evaluating factors like CI/CD complexity, cross-project dependencies, development experience, and scalability. I prototyped both approaches with a subset of our models, gathered feedback from the team through workshops, and documented trade-offs. I recommended a modular monorepo approach using dbt packages for shared utilities.

Result: Leadership approved the recommendation. We successfully restructured the project into logical subdirectories with clear ownership boundaries while maintaining a single deployment pipeline. Development velocity increased by 30%, and merge conflicts decreased significantly. The structure scaled well as we grew from 500 to 1200+ models.

5. Tell me about a time when you had to balance technical debt with feature delivery in a dbt project.

Situation: Our dbt project had accumulated significant technical debt including outdated models, missing tests, and inconsistent naming conventions, while simultaneously facing pressure to deliver new business-critical dashboards.

Task: I needed to address technical debt without compromising feature delivery timelines or team morale.

Action: I categorized technical debt by severity and impact, then proposed a hybrid approach to leadership: allocating 70% capacity to new features and 30% to debt reduction. I integrated debt reduction into feature work by requiring that any modified models be brought up to current standards. I created a tracking system for debt items, automated what could be automated (linting, formatting), and celebrated debt reduction milestones. I also documented the long-term cost of ignoring technical debt.

Result: Over six months, we delivered all committed features on time while reducing technical debt by 60%. Test coverage increased from 45% to 85%, and model runtime improved by 25%. Leadership appreciated the transparent approach and made technical health a permanent part of sprint planning.

6. Describe a situation where you had to collaborate with stakeholders who had conflicting requirements for a dbt data

model.

Situation: The marketing and finance teams needed a customer metrics model, but marketing wanted real-time data with detailed segmentation while finance required point-in-time accuracy with strict reconciliation to the general ledger.

Task: I needed to design a solution that satisfied both teams' requirements without duplicating effort or creating inconsistent metrics.

Action: I organized a requirements gathering workshop with both stakeholders to understand their underlying needs. I proposed a layered approach: creating a foundational staging model with point-in-time logic for finance, then building separate mart models for each team's specific use cases using the same base. I documented the lineage and business logic clearly, created a shared metrics layer using dbt metrics, and established a governance process for future changes. I also provided training on how to interpret each model.

Result: Both teams were satisfied with the solution. Finance achieved 100% reconciliation accuracy, and marketing received their detailed segmentation with acceptable latency. The shared foundation reduced maintenance by 40% compared to separate pipelines, and the approach became our standard pattern for handling similar conflicts.

7. Tell me about a time when you identified and implemented a significant improvement to your team's dbt workflow or processes.

Situation: Our team was experiencing frequent deployment failures and lengthy code review cycles, with changes taking 3-5 days from PR creation to production deployment.

Task: I was asked to identify bottlenecks and propose improvements to accelerate our delivery process while maintaining quality standards.

Action: I analyzed our Git history and CI/CD logs to identify patterns. I implemented slim CI using dbt's state comparison to run only modified models and their children, reducing CI time from 45 minutes to 8 minutes. I created PR templates with checklists, established clear code review guidelines with 24-hour SLA, implemented automated linting and formatting with pre-commit hooks, and set up dbt Cloud jobs for automated testing. I also introduced feature flags for gradual rollouts.

Result: Average PR-to-production time decreased from 3-5 days to 1 day. Deployment failures dropped by 70%, and developer satisfaction scores improved significantly. The faster feedback loop enabled more experimentation and innovation, resulting in 50% more features delivered per quarter.

8. Describe a time when you had to debug a complex issue in a dbt project that others couldn't solve.

Situation: A critical incremental model was producing duplicate records intermittently, but the issue couldn't be reproduced in development environments. Two developers had already spent a week investigating without success.

Task: I was escalated to identify the root cause and implement a permanent fix before it impacted monthly financial reporting.

Action: I took a systematic approach: first, I examined the compiled SQL and compared production vs. development configurations. I analyzed dbt run artifacts and logs across multiple failed runs to identify patterns. I discovered that the issue occurred only when the model ran concurrently with upstream source refreshes, causing race conditions. The unique_key logic was also insufficient for the data's grain. I redesigned the incremental logic with proper windowing functions, added a merge key combining multiple columns, and implemented upstream dependency checks.

Result: The duplicate issue was completely resolved. I documented the debugging methodology and created a runbook for similar issues. The solution prevented an estimated $200K in potential reporting errors, and my debugging approach was adopted as a team standard for complex investigations.

9. Tell me about a time when you had to advocate for adopting a new dbt feature or practice that faced resistance from your team.

Situation: I wanted to introduce dbt's semantic layer and metrics framework to standardize metric definitions across the organization, but the team was skeptical about the learning curve and concerned about migration effort.

Task: I needed to demonstrate the value proposition and gain team buy-in for this significant architectural change.

Action: I created a proof-of-concept implementing 10 key business metrics using the new framework, showing how it eliminated duplicate metric logic across 15 different models. I presented a cost-benefit analysis highlighting reduced maintenance burden and improved consistency. I organized a lunch-and-learn session demonstrating the developer experience, addressed concerns about migration with a phased rollout plan, and volunteered to lead the initial implementation. I also identified quick wins to build momentum.

Result: The team approved a pilot program. After successfully implementing metrics for one business domain, adoption expanded organically. Within six months, we had 50+ standardized metrics, eliminated 30+ redundant calculations, and reduced metric-related bugs by 80%. Stakeholders praised the consistency, and the approach became our standard for metric management.

10. Describe a situation where you had to recover from a failed dbt deployment or migration. What was your approach?

Situation: During a major dbt version upgrade from 0.21 to 1.0, our production deployment failed mid-execution due to breaking changes in package dependencies, leaving the warehouse in an inconsistent state during business hours.

Task: I needed to quickly restore service, ensure data integrity, and complete the migration without extended downtime.

Action: I immediately activated our incident response protocol and communicated status to stakeholders. I rolled back to the previous dbt version using our versioned Docker images and restored the last known good state. I then created a detailed migration plan: set up a parallel environment to test the upgrade thoroughly, identified all breaking changes and updated code accordingly, created a comprehensive rollback plan, and scheduled the migration during a maintenance window. I also implemented blue-green deployment strategy for future migrations.

Result: Production was restored within 45 minutes with no data loss. The planned migration completed successfully two weeks later with zero downtime. I documented the incident and lessons learned, which led to implementing automated compatibility checks in CI/CD and a formal change management process. No similar incidents occurred in subsequent major upgrades.