Snowflake Interview Discussion Guide

1. Snowflake Architecture (3 Layers)

Three-Layer Architecture:

1. **Database Storage Layer**
   - Stores all data in optimized, compressed, columnar format
   - Managed entirely by Snowflake
   - Independent scaling from compute
2. **Query Processing Layer (Virtual Warehouses)**
   - MPP compute clusters (Virtual Warehouses)
   - Each warehouse is independent
   - Can scale up (larger) or scale out (more clusters)
3. **Cloud Services Layer**
   - Authentication & Access Control
   - Query parsing & optimization
   - Metadata management
   - Infrastructure management

Interview Tip: Emphasize the separation of storage and compute - this allows independent scaling and you only pay for what you use.

---

2. Key Features

Time Travel

What it is: Query, restore, or clone historical data within a retention period (0-90 days).

```
-- Query data from 5 minutes ago
SELECT * FROM my_table AT(OFFSET => -60*5);

-- Query data from specific timestamp
SELECT * FROM my_table AT(TIMESTAMP => '2025-10-13 10:00:00'::timestamp);

-- Restore dropped table
UNDROP TABLE my_table;

-- Restore table to previous state
CREATE TABLE my_table_restored CLONE my_table
AT(TIMESTAMP => '2025-10-13 09:00:00'::timestamp);
```

Notes:

- Standard: 1 day (can set 0-1 days)
- Enterprise: Up to 90 days
- Storage costs apply for Time Travel data

Zero Copy Cloning

Why "Zero Copy"?

- Doesn't physically copy data
- Creates metadata pointers to existing micro-partitions
- Only when cloned object is modified, new micro-partitions are created (copy-on-write)

Cloneable Objects:

- Databases
- Schemas
- Tables (Permanent, Transient, Temporary)
- Streams
- Stages (named stages)
- File Formats
- Sequences
- Tasks
- Pipes

```
-- Clone table
CREATE TABLE orders_clone CLONE orders;

-- Clone database
CREATE DATABASE prod_backup CLONE production;

-- Clone schema
CREATE SCHEMA dev_schema CLONE prod_schema;

-- Clone at specific time
CREATE TABLE orders_backup CLONE orders
AT(TIMESTAMP => '2025-10-13 00:00:00'::timestamp);
```

Interview Note: Cannot clone External Tables, Materialized Views (structure only), or User/Role objects.

---

3. Stages, Tables, and Views

Stages (3 Types)

```
-- 1. User Stage (default, per user)
PUT file://local/file.csv @~;
LIST @~;

-- 2. Table Stage (per table)
PUT file://data.csv @%my_table;

-- 3. Named Stage (Internal or External)
-- Internal Named Stage
CREATE STAGE my_internal_stage;
```

```
-- External Named Stage (S3)
CREATE STAGE my_s3_stage
  URL = 's3://mybucket/path/'
  CREDENTIALS = (AWS_KEY_ID='xxx' AWS_SECRET_KEY='yyy');
```

Table Types

```
-- 1. Permanent (default) - Time Travel + Fail-safe
CREATE TABLE perm_table (id INT, name STRING);

-- 2. Transient - Time Travel, NO Fail-safe
CREATE TRANSIENT TABLE trans_table (id INT, name STRING);

-- 3. Temporary - Session-scoped, Time Travel, NO Fail-safe
CREATE TEMPORARY TABLE temp_table (id INT, name STRING);

-- 4. External - Data stored outside Snowflake
CREATE EXTERNAL TABLE ext_table
  LOCATION = @my_s3_stage
  FILE_FORMAT = (TYPE = PARQUET);

-- 5. Iceberg Tables (new) - Open format, interoperability
CREATE ICEBERG TABLE iceberg_tbl
  EXTERNAL_VOLUME = 'my_volume'
  CATALOG = 'SNOWFLAKE'
  BASE_LOCATION = 's3://bucket/path/';

-- 6. Hybrid Tables (OLTP workload) - Row-based storage
CREATE HYBRID TABLE hybrid_tbl (
  id INT PRIMARY KEY,
  name STRING,
  updated_at TIMESTAMP
);
```

View Types

```
-- 1. Standard View
CREATE VIEW sales_view AS SELECT * FROM sales WHERE region = 'EAST';

-- 2. Secure View (definition hidden)
CREATE SECURE VIEW secure_sales_view AS
SELECT * FROM sales WHERE region = 'EAST';

-- 3. Materialized View (pre-computed, auto-refresh)
CREATE MATERIALIZED VIEW sales_mv AS
SELECT region, SUM(amount) as total_sales
FROM sales
GROUP BY region;
```

---

4. Data Storage (CRITICAL TOPIC)

Micro-Partitioning

How Data is Stored:

- Data automatically divided into micro-partitions (50-500 MB compressed)
- **Columnar storage** within each micro-partition
- Metadata stored for each micro-partition:
    - Number of rows
    - Min/Max values per column
    - Distinct values count
    - NULL counts

What Happens on UPDATE:

```
-- Original: Micro-partition contains 100K rows
-- UPDATE 1 row
UPDATE orders SET status = 'SHIPPED' WHERE order_id = 12345;

-- Result:
-- 1. Entire micro-partition is marked as deleted
-- 2. New micro-partition created with the updated row
-- 3. Old micro-partition kept for Time Travel
```

Micro-Partition Size Range:

- Target: 50-500 MB compressed
- Typically 16 MB compressed (varies by data)

Interview Deep Dive:

```
-- Check table storage
SELECT * FROM TABLE(INFORMATION_SCHEMA.TABLE_STORAGE_METRICS(
  TABLE_NAME => 'ORDERS'
));

-- View clustering information
SELECT SYSTEM$CLUSTERING_INFORMATION('orders', '(order_date)');
```

---

5. Clustering and Clustering Keys

Clustering Keys

```
-- Create table with clustering
CREATE TABLE orders (
  order_id INT,
  order_date DATE,
  customer_id INT,
  amount DECIMAL
) CLUSTER BY (order_date, customer_id);

-- Add clustering to existing table
ALTER TABLE orders CLUSTER BY (order_date);

-- Remove clustering
ALTER TABLE orders DROP CLUSTERING KEY;
```

Studying Query Profile

Steps:

1. Go to **History** tab in Snowflake UI
2. Click on Query ID
3. Review **Profile** tab:
    - Partitions scanned vs. total
    - Bytes scanned
    - Execution time breakdown
    - Spillage (local/remote disk)

Clustering Depth

```
-- Check clustering depth (0-1 is well-clustered)
SELECT SYSTEM$CLUSTERING_DEPTH('orders', '(order_date)');

-- Detailed clustering info
SELECT SYSTEM$CLUSTERING_INFORMATION('orders', '(order_date)');

-- Check if reclustering is needed
-- If average_depth > 2-3, consider manual reclustering
ALTER TABLE orders RECLUSTER;
```

Interview Note:

- Lower depth = better clustering
- Depth of 0-1 is ideal
- Automatic reclustering happens in background (Enterprise edition)

---

6. Caching in Snowflake (3 Types)

1. Query Result Cache (Cloud Services Layer)

- Stores results for 24 hours
- Requires identical query text
- No compute cost for cached results

2. Metadata Cache (Cloud Services Layer)

- Stores metadata (min/max, row counts)
- Used for query optimization
- Always available

3. Data Cache / Warehouse Cache (Virtual Warehouse)

- Caches data files in SSD
- Persists while warehouse is running
- Cleared when warehouse is suspended
- Improves subsequent query performance

```
-- Disable result cache for testing
ALTER SESSION SET USE_CACHED_RESULT = FALSE;

-- Check if query used cache
SELECT query_id, query_text, execution_status,
       partitions_scanned, partitions_total,
       bytes_scanned
FROM TABLE(INFORMATION_SCHEMA.QUERY_HISTORY())
WHERE query_id = 'your_query_id';
```

---

7. Data Loading: COPY INTO, Stages, Snowpipe

Create External Stage with S3 Integration

```
-- 1. Create Storage Integration (ACCOUNTADMIN)
CREATE STORAGE INTEGRATION s3_integration
  TYPE = EXTERNAL_STAGE
  STORAGE_PROVIDER = S3
  ENABLED = TRUE
  STORAGE_AWS_ROLE_ARN = 'arn:aws:iam::123456789:role/snowflake-role'
  STORAGE_ALLOWED_LOCATIONS = ('s3://mybucket/data/');

-- 2. Retrieve external ID and user ARN
DESC STORAGE INTEGRATION s3_integration;
-- Copy STORAGE_AWS_IAM_USER_ARN and STORAGE_AWS_EXTERNAL_ID

-- 3. Update AWS IAM Trust Relationship
-- Add to role's trust policy:
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
      "AWS": "STORAGE_AWS_IAM_USER_ARN"
    },
    "Action": "sts:AssumeRole",
    "Condition": {
      "StringEquals": {
        "sts:ExternalId": "STORAGE_AWS_EXTERNAL_ID"
      }
    }
  }]
}

-- 4. Create External Stage
CREATE STAGE my_s3_stage
  STORAGE_INTEGRATION = s3_integration
  URL = 's3://mybucket/data/'
  FILE_FORMAT = (TYPE = CSV FIELD_DELIMITER = ',' SKIP_HEADER = 1);

-- 5. COPY INTO
COPY INTO orders
FROM @my_s3_stage/orders.csv
FILE_FORMAT = (TYPE = CSV)
ON_ERROR = 'CONTINUE';
```

Snowpipe Setup

```
-- 1. Create Snowpipe
CREATE PIPE orders_pipe
  AUTO_INGEST = TRUE
  AS
  COPY INTO orders
  FROM @my_s3_stage
  FILE_FORMAT = (TYPE = CSV);

-- 2. Get notification channel ARN
SHOW PIPES;
-- Copy notification_channel value

-- 3. Setup SQS in AWS
-- Create S3 Event Notification on bucket:
-- Event: s3:ObjectCreated:*
-- Destination: SNS/SQS topic
-- Subscribe Snowflake notification_channel to SNS

-- 4. Monitor pipe
SELECT SYSTEM$PIPE_STATUS('orders_pipe');

-- Check pipe execution
SELECT * FROM TABLE(INFORMATION_SCHEMA.COPY_HISTORY(
  TABLE_NAME => 'ORDERS',
  START_TIME => DATEADD(HOUR, -1, CURRENT_TIMESTAMP())
));
```

---

8. JSON Parsing (OLD Method)

LATERAL FLATTEN Approach

```
-- Sample JSON data
CREATE TABLE json_data (raw VARIANT);

INSERT INTO json_data
SELECT PARSE_JSON('{
  "customer": {
    "id": 123,
    "name": "John",
    "orders": [
      {"order_id": 1, "amount": 100},
      {"order_id": 2, "amount": 200}
    ]
  }
}');

-- Parse nested JSON using LATERAL FLATTEN
SELECT
  raw:customer.id::INT as customer_id,
  raw:customer.name::STRING as customer_name,
  f.value:order_id::INT as order_id,
  f.value:amount::DECIMAL as amount
FROM json_data,
LATERAL FLATTEN(input => raw:customer.orders) f;

-- Multiple level nesting
SELECT
  raw:customer.id::INT as customer_id,
```

```
  f1.value:order_id::INT as order_id,
  f2.value:item_name::STRING as item_name,
  f2.value:price::DECIMAL as price
FROM json_data,
LATERAL FLATTEN(input => raw:customer.orders) f1,
LATERAL FLATTEN(input => f1.value:items) f2;

-- OLD Method (FLATTEN table function - deprecated but works)
SELECT
  raw:customer.id::INT,
  value:order_id::INT,
  value:amount::DECIMAL
FROM json_data,
TABLE(FLATTEN(raw:customer.orders));
```

---

9. Streams and Tasks

Stream Types

```
-- 1. STANDARD Stream (default) - tracks all DML
CREATE STREAM orders_stream ON TABLE orders;

-- 2. APPEND-ONLY Stream - only INSERT
CREATE STREAM orders_append_stream ON TABLE orders
APPEND_ONLY = TRUE;

-- 3. INSERT-ONLY Stream (External tables/views)
CREATE STREAM ext_stream ON EXTERNAL TABLE ext_orders
INSERT_ONLY = TRUE;

-- Check stream contents
SELECT * FROM orders_stream;

-- Stream metadata columns:
-- METADATA$ACTION: INSERT, DELETE, UPDATE
-- METADATA$ISUPDATE: TRUE for UPDATE operations
-- METADATA$ROW_ID: Unique row identifier
```

SCD Type 1 Using Streams

```
-- Target dimension table
CREATE TABLE dim_customer (
  customer_id INT PRIMARY KEY,
  name STRING,
  email STRING,
  updated_at TIMESTAMP
);

-- Stream on source
CREATE STREAM customer_stream ON TABLE source_customers;

-- Merge for SCD Type 1 (overwrite)
MERGE INTO dim_customer t
USING (
  SELECT customer_id, name, email, CURRENT_TIMESTAMP() as updated_at
  FROM customer_stream
  WHERE METADATA$ACTION = 'INSERT'
```

```
  QUALIFY ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY
METADATA$ROW_ID DESC) = 1
) s
ON t.customer_id = s.customer_id
WHEN MATCHED THEN UPDATE SET
  t.name = s.name,
  t.email = s.email,
  t.updated_at = s.updated_at
WHEN NOT MATCHED THEN INSERT
  (customer_id, name, email, updated_at)
  VALUES (s.customer_id, s.name, s.email, s.updated_at);
```

SCD Type 2 Using Streams

```
-- SCD Type 2 dimension
CREATE TABLE dim_customer_scd2 (
  surrogate_key INT AUTOINCREMENT PRIMARY KEY,
  customer_id INT,
  name STRING,
  email STRING,
  effective_date TIMESTAMP,
  end_date TIMESTAMP,
  is_current BOOLEAN
);

-- Merge for SCD Type 2
MERGE INTO dim_customer_scd2 t
USING (
  SELECT customer_id, name, email
  FROM customer_stream
  WHERE METADATA$ACTION = 'INSERT'
) s
ON t.customer_id = s.customer_id AND t.is_current = TRUE
WHEN MATCHED AND (t.name != s.name OR t.email != s.email) THEN UPDATE SET
  t.end_date = CURRENT_TIMESTAMP(),
  t.is_current = FALSE
WHEN NOT MATCHED THEN INSERT
  (customer_id, name, email, effective_date, end_date, is_current)
  VALUES (s.customer_id, s.name, s.email, CURRENT_TIMESTAMP(), NULL, TRUE);

-- Insert new version for changed records
INSERT INTO dim_customer_scd2 (customer_id, name, email, effective_date,
is_current)
SELECT s.customer_id, s.name, s.email, CURRENT_TIMESTAMP(), TRUE
FROM customer_stream s
JOIN dim_customer_scd2 t
  ON s.customer_id = t.customer_id
  AND t.is_current = FALSE
WHERE METADATA$ACTION = 'INSERT'
  AND (t.name != s.name OR t.email != s.email);
```

Tasks

```
-- How Task Knows Stream Has Data?
-- Use SYSTEM$STREAM_HAS_DATA() in WHEN clause

-- Simple task
CREATE TASK process_orders_task
  WAREHOUSE = compute_wh
```

```
  SCHEDULE = '5 MINUTE'
WHEN
  SYSTEM$STREAM_HAS_DATA('orders_stream')
AS
  MERGE INTO target_orders t
  USING orders_stream s ON t.order_id = s.order_id
  WHEN MATCHED THEN UPDATE SET t.status = s.status
  WHEN NOT MATCHED THEN INSERT VALUES (s.order_id, s.status);

-- Resume task (tasks are suspended by default)
ALTER TASK process_orders_task RESUME;

-- Task tree (multiple streams)
CREATE TASK root_task
  WAREHOUSE = compute_wh
  SCHEDULE = '10 MINUTE'
AS
  CALL validate_data_proc();

CREATE TASK child_task_1
  WAREHOUSE = compute_wh
  AFTER root_task
WHEN
  SYSTEM$STREAM_HAS_DATA('stream_1')
AS
  CALL process_stream_1();

CREATE TASK child_task_2
  WAREHOUSE = compute_wh
  AFTER root_task
WHEN
  SYSTEM$STREAM_HAS_DATA('stream_2')
AS
  CALL process_stream_2();

-- Resume in reverse order (child first, then parent)
ALTER TASK child_task_2 RESUME;
ALTER TASK child_task_1 RESUME;
ALTER TASK root_task RESUME;
```

Managing Multiple Streams:

```
-- Use separate tasks or conditional logic
CREATE TASK multi_stream_task
  WAREHOUSE = compute_wh
  SCHEDULE = '5 MINUTE'
WHEN
  SYSTEM$STREAM_HAS_DATA('stream_1') OR
  SYSTEM$STREAM_HAS_DATA('stream_2')
AS
BEGIN
  IF (SYSTEM$STREAM_HAS_DATA('stream_1')) THEN
    MERGE INTO target1 USING stream_1 ...;
  END IF;

  IF (SYSTEM$STREAM_HAS_DATA('stream_2')) THEN
    MERGE INTO target2 USING stream_2 ...;
  END IF;
END;
```

---

10. Query Optimization

Studying Query Profile

```
-- 1. Query History with Performance Metrics
SELECT
  query_id,
  query_text,
  database_name,
  warehouse_name,
  execution_status,
  total_elapsed_time,
  bytes_scanned,
  rows_produced,
  partitions_scanned,
  partitions_total,
  (partitions_scanned / NULLIF(partitions_total, 0)) * 100 as
partition_scan_pct
FROM TABLE(INFORMATION_SCHEMA.QUERY_HISTORY())
WHERE start_time >= DATEADD('hour', -24, CURRENT_TIMESTAMP())
  AND execution_status = 'SUCCESS'
ORDER BY total_elapsed_time DESC
LIMIT 100;

-- 2. Average Execution Time from ACCOUNT_USAGE
SELECT
  query_type,
  AVG(total_elapsed_time/1000) as avg_seconds,
  MAX(total_elapsed_time/1000) as max_seconds,
  COUNT(*) as execution_count
FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
WHERE start_time >= DATEADD('day', -30, CURRENT_TIMESTAMP())
  AND execution_status = 'SUCCESS'
GROUP BY query_type
ORDER BY avg_seconds DESC;

-- 3. Specific Query Pattern Average
SELECT
  REGEXP_SUBSTR(query_text, 'FROM\\s+(\\w+)', 1, 1, 'e', 1) as table_name,
  AVG(total_elapsed_time/1000) as avg_seconds,
  COUNT(*) as query_count
FROM SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY
WHERE query_text ILIKE '%SELECT%FROM%orders%'
  AND start_time >= DATEADD('day', -7, CURRENT_TIMESTAMP())
GROUP BY table_name;
```

Determining Clustering Key Need

```
-- 1. Check partition pruning efficiency
-- If partitions_scanned / partitions_total > 0.5, consider clustering

-- 2. Check clustering depth
SELECT SYSTEM$CLUSTERING_DEPTH('orders', '(order_date)');
-- If depth > 3-4, reclustering beneficial

-- 3. Analyze query patterns
SELECT
  query_text,
  partitions_scanned,
```

```
  partitions_total,
  bytes_scanned,
  (bytes_scanned / 1024 / 1024 / 1024) as gb_scanned
FROM TABLE(INFORMATION_SCHEMA.QUERY_HISTORY())
WHERE query_text ILIKE '%orders%'
  AND partitions_total > 0
ORDER BY start_time DESC
LIMIT 50;

-- 4. If you see:
-- - High partition scan ratio
-- - Queries filtering on specific columns
-- - Large tables (>1TB)
-- --> Add clustering key on filter columns
```

Optimization Checklist:

- Check for full table scans
- Look for exploding joins (cartesian products)
- Verify clustering on large tables
- Check for spillage to local/remote disk
- Use result cache where possible
- Proper warehouse sizing
- Partition pruning effectiveness

---

11. Masking Policies and Data Sharing

Masking Policies

```
-- Create masking policy
CREATE MASKING POLICY email_mask AS (val STRING)
RETURNS STRING ->
  CASE
    WHEN CURRENT_ROLE() IN ('ADMIN', 'ANALYST') THEN val
    ELSE '***@***' || SPLIT_PART(val, '@', 2)
  END;

-- Apply to column
ALTER TABLE customers
MODIFY COLUMN email
SET MASKING POLICY email_mask;

-- Conditional masking (show full to certain roles)
CREATE MASKING POLICY ssn_mask AS (val STRING)
RETURNS STRING ->
  CASE
    WHEN CURRENT_ROLE() IN ('COMPLIANCE_ROLE') THEN val
    ELSE 'XXX-XX-' || RIGHT(val, 4)
  END;

-- View masked data
SELECT email FROM customers; -- Shows masked based on role

-- Remove policy
ALTER TABLE customers
```

```
MODIFY COLUMN email
UNSET MASKING POLICY;
```

Data Sharing

```
-- As Data Provider:
-- 1. Create share
CREATE SHARE sales_share;

-- 2. Grant usage on database
GRANT USAGE ON DATABASE sales_db TO SHARE sales_share;

-- 3. Grant usage on schema
GRANT USAGE ON SCHEMA sales_db.public TO SHARE sales_share;

-- 4. Grant select on tables
GRANT SELECT ON TABLE sales_db.public.orders TO SHARE sales_share;

-- 5. Add consumer account
ALTER SHARE sales_share
ADD ACCOUNTS = 'CONSUMER_ACCOUNT_ID';

-- As Data Consumer:
-- 1. View available shares
SHOW SHARES;

-- 2. Create database from share
CREATE DATABASE shared_sales_data
FROM SHARE PROVIDER_ACCOUNT.sales_share;

-- 3. Query shared data
SELECT * FROM shared_sales_data.public.orders;
```

---

12. Access Control Models

RBAC (Role-Based Access Control)

```
-- Create hierarchy: SYSADMIN -> MANAGER_ROLE -> ANALYST_ROLE

-- Create roles
CREATE ROLE manager_role;
CREATE ROLE analyst_role;

-- Role hierarchy
GRANT ROLE analyst_role TO ROLE manager_role;
GRANT ROLE manager_role TO ROLE sysadmin;

-- Grant privileges
GRANT USAGE ON DATABASE sales_db TO ROLE analyst_role;
GRANT SELECT ON ALL TABLES IN SCHEMA sales_db.public TO ROLE analyst_role;

-- Grant warehouse access
GRANT USAGE ON WAREHOUSE compute_wh TO ROLE analyst_role;

-- Assign role to user
GRANT ROLE analyst_role TO USER john_doe;
```

```
-- User can have multiple roles, switch between them
USE ROLE analyst_role;
```

UBAC (User-Based Access Control)

```
-- Direct user grants (not recommended, use RBAC instead)
GRANT USAGE ON DATABASE sales_db TO USER john_doe;
GRANT SELECT ON TABLE orders TO USER john_doe;
```

DAC (Discretionary Access Control)

```
-- Object owner can grant access
-- Owner of table 'orders' can:
GRANT SELECT ON TABLE orders TO ROLE analyst_role;
GRANT ALL ON TABLE orders TO ROLE manager_role;

-- Transfer ownership
GRANT OWNERSHIP ON TABLE orders TO ROLE new_owner_role;
```

Best Practices:

1. Use RBAC (role hierarchy)
2. Apply least privilege principle
3. Use functional roles (not user-specific)
4. Separate roles by: Access level, Department, Job function
5. Use masking policies with roles for data governance