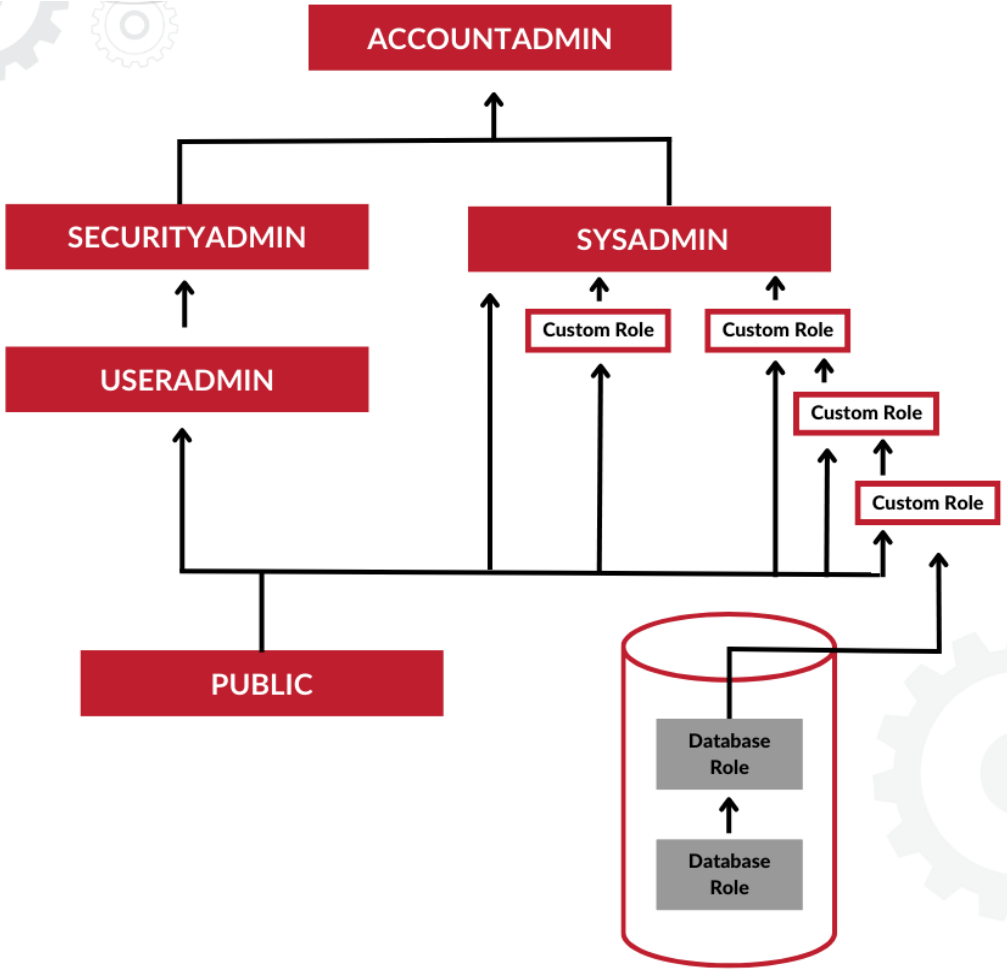
System Defined Roles

•**ACCOUNTADMIN**: The top-level role with control over all aspects of the account. It inherits privileges from the SECURITYADMIN and SYSADMIN roles and is intended for account-level tasks, not day-to-day object creation.

•**SECURITYADMIN**: A role that can manage any object grant globally and create/manage users and roles. It is granted the global MANAGE GRANTS privilege and inherits the privileges of the USERADMIN role.



•**SYSADMIN**: The system administrator role with privileges to create and manage warehouses, databases, and all database objects (schemas, tables, etc.). It is a best practice to have custom roles roll up to the SYSADMIN role in a role hierarchy.

•**USERADMIN**: A role dedicated solely to user and role management. It can create users and roles and manage those it owns, without the ability to manage grants globally.

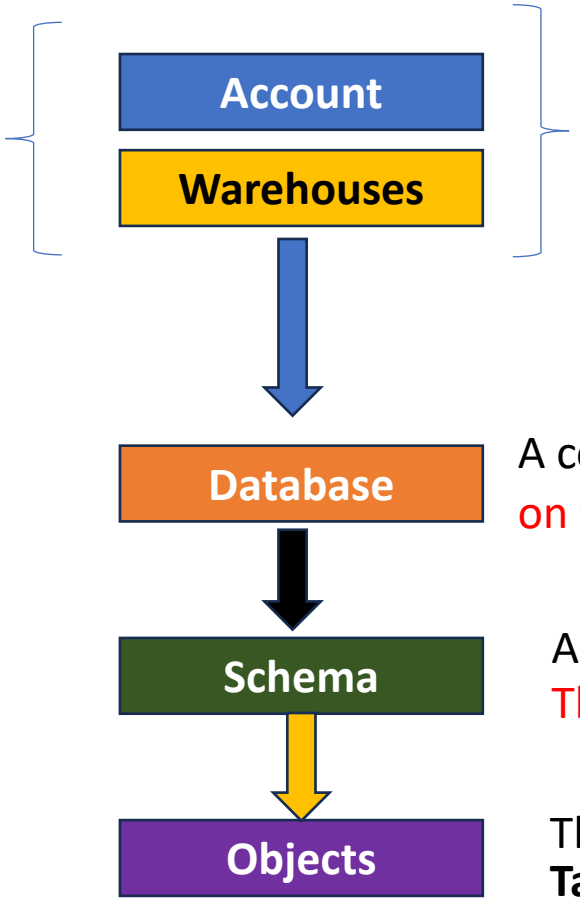
•**ORGADMIN**: A role used to manage operations at the organization level, such as managing accounts. This role is not part of the standard ACCOUNTADMIN hierarchy.

•**PUBLIC**: A pseudo-role automatically granted to every user and every other role in the account. Objects owned by the PUBLIC role are available to all users by default, so caution is advised when granting it privileges on sensitive data.

Securable object Hierarchy

Securable objects are organized in a strict logical hierarchy. To access a specific object, a user must have the necessary privileges on the object itself, as well as the USAGE privilege on all its parent container objects in the hierarchy.

The hierarchy of securable objects in Snowflake is as follows, from the highest level to the lowest:



A container for schemas. To access objects within a database, the `USAGE` privilege must be granted on the database.

A container for specific database objects such as tables, views, stages, and functions.
The `USAGE` privilege is required on the schema to access its contained objects.

The actual resources where data is stored or operations are performed. Examples include:
Tables, Views, Stages, File Formats, Functions etc.

Access control framework

Snowflake’s approach to access control combines aspects from the following models:

- **Discretionary Access Control (DAC):** Each object has an owner, who can in turn grant access to that object.

- **Role-based Access Control (RBAC):** Access privileges are assigned to roles, which are in turn assigned to users.

RBAC is the administrative skeleton of Snowflake. Access is granted to Roles, and then those Roles are assigned to Users.

Logic: "If you are in the Marketing Team, you get the Marketing Role."
Centralized: Managed typically by SECURITYADMIN or USERADMIN.
Inheritance: Uses a hierarchy. If the ANALYST role is granted to the MANAGER role, the Manager automatically gets every permission the Analyst has.

DAC is centered around Ownership. Every object in Snowflake (Table, Schema, Warehouse) has an Owner (a role with the OWNERSHIP privilege).

Logic: "I created this table, so I decide who else can see it."
Decentralized: The owner of an object can grant SELECT, INSERT, or USAGE to any other role at their own discretion.
Control: If you create a table using the DEV_ROLE, that role owns the table and can "discretionally" share it with the QA_ROLE.

Network Policies

Snowflake network policies are a security feature used to control inbound network traffic to the Snowflake service and internal stages by defining allowed and blocked network origins.

They work in conjunction with network rules to enforce IP-based restrictions and private connectivity requirements.

Syntax

```
CREATE NETWORK POLICY allow_vpceid_block_public_policy
  ALLOWED_NETWORK_RULE_LIST = ('allow_vpceid_access')
  BLOCKED_NETWORK_RULE_LIST = ('block_public_access');

DESC NETWORK POLICY rule_based_policy;

ALTER ACCOUNT SET NETWORK_POLICY = <policy_name>;
```


Note
Only security administrators (i.e. users with the SECURITYADMIN role) or higher *or* a role with the global CREATE NETWORK POLICY privilege can create network policies.

Authentication

In 2026, Snowflake has fully modernized its authentication landscape. Password-only access for human users is being phased out in favor of **Strong Authentication**, and service accounts are moving toward purely programmatic methods.''

1. Multi-Factor Authentication (MFA)

MFA is the "second lock" on the door. It is very important & adviceable for all users accessing Snowflake via the UI (Snowsight).

2. Federated Authentication & Single Sign-On (SSO)

Federated Authentication separates **Identity Provider (IdP)** from the service (Snowflake). SSO is the *result* of this setup, allowing users to log in once to their corporate network and access Snowflake without a separate password. Snowflake redirects the user to the IdP (like Okta or Azure AD). Once the IdP verifies the user, it sends a digital "token" back to Snowflake to grant entry.

3. OAuth

OAuth is used for **secure application-to-application** communication. Instead of sharing a password with a third-party tool (like Tableau or a custom web app), the tool receives a temporary **Access Token**.

Two Types:

Snowflake OAuth: Snowflake acts as the "Authorization Server."

External OAuth: Your corporate IdP (Okta/Azure) issues the tokens.

New for 2026: Snowflake OAuth for Local Applications is now the preferred method for desktop scripts and local apps, removing the need for users to store any secrets locally.

4. Key-Pair Authentication

This is the "Gold Standard" for automated service accounts. It uses a **Public Key** (stored in Snowflake) and a **Private Key** (kept secure by the developer/system).

Domain 2.0: Account Management and Data Governance

2.1 Explain Snowflake security model and principles

- Role-Based Access Control (RBAC) ✔
- Securable object hierarchy ✔
- Discretionary access control (DAC) ✔
- Network Policies ✔
- Authentication ✔
  - Multi-Factor Authentication (MFA)
  - Federated Authentication
  - Single Sign-on (SSO)
  - OAuth
  - Key-pair authentication
- System-defined roles ✔
- Functional roles
  - Account roles
  - Database roles
  - Custom roles
- Secondary roles
- Account identifiers
- Logging and tracing

2.2 Define data governance features and how they are used

- Data masking
  - Row-level security
  - Column-level security
- Object tagging
- Privacy policies
- Trust Center
- Encryption key management
- Alerts
- Notifications
- Data replication and failover
- Data lineage

2.3 Explain monitoring and cost management

- Resource Monitors
  - Cost and warehouse monitoring
- Calculating virtual warehouse credit usage
- ACCOUNT_USAGE schema

Account identifiers

An account identifier uniquely identifies a Snowflake account within your organization, as well as throughout the global network of Snowflake-supported cloud platforms and cloud regions.

The preferred account identifier consists of the *name* of the account prefixed by its organization (e.g. myorg-account123). You can also use the Snowflake-assigned *locator* as the account identifier; however, the use of this legacy format is *discouraged*.

Account identifiers are required in Snowflake wherever you need to specify the account you are using, including:

•URLs for accessing any of the Snowflake web interfaces.

•Snowflake CLI, SnowSQL, and other clients (connectors, drivers, etc.) for connecting to Snowflake.

•Third-party applications and services that comprise the Snowflake ecosystem.

•Security features for protecting Snowflake internal operations and communication/interaction with external systems.

•Global features such as Secure Data Sharing and Replication and Failover/Failback.

For example, the URL for an account uses the following format:
account_identifier.snowflakecomputing.com

Logging and tracing

Snowflake provides built-in capabilities for **logging and tracing** applications, which consolidate data into a central **event table** for analysis and monitoring. This observability framework, based on the OpenTelemetry standard, is available for code written in Python, Java, Scala, JavaScript, and Snowflake Scripting.

Key Components

Event Tables: A special, predefined table in your account where all log messages and trace events are automatically collected. An account can have only one active event table at a time.

Telemetry Levels: You control the verbosity of collected data by setting LOG_LEVEL and TRACE_LEVEL parameters. These can be set at the account, database, schema, session, or specific object (function/procedure) level, with more verbose levels taking precedence.

APIs: Developers use language-specific APIs within their handler code (e.g., Python's logging module, Snowflake's telemetry package, Java's SLF4J API, or Snowflake Scripting's SYSTEM$LOG function) to emit log messages and trace events.

Data Masking

CEO & Finance Dept

Name	FName	City	Age	Salary
Smith	John	3	35	$280
Doe	Jane	1	28	$325
Brown	Scott	3	41	$265
Howard	Shemp	4	48	$359
Taylor	Tom	2	22	$250

Support Agent/HR Team

Name	FName	City
Smith	John	3
Doe	Jane	1
Brown	Scott	3
Howard	Shemp	4
Taylor	Tom	2

Data Analysts

Name	FName	
Smith	John	
Doe	Jane	
Brown	Scott	
Howard	Shemp	
Taylor	Tom	2

Snowflake primarily uses Dynamic Data Masking for protecting sensitive data. This feature obscures data in real-time based on the user's role and access permissions, without altering the underlying data in storage.

Row-level security

Snowflake implements row-level security (RLS) primarily through **row access policies**, which are schema-level objects that dynamically determine which rows a user can view in a table or view at query runtime. This ensures users only see data relevant to their roles or permissions without needing to create multiple copies of tables or complex views.

SELECT * FROM COMPANY.public.employees

EMPLOYEE_ID	EMPL_JOIN_DATE	DEPT	SALARY	MANAGER_ID
1	2014-10-01	HR	40000	4
2	2014-09-01	Tech	50000	9
3	2018-09-01	Marketing	30000	5
4	2017-09-01	HR	10000	5
5	2019-09-01	HR	35000	9
6	2015-09-01	Tech	90000	4
7	2016-09-01	Marketing	20000	1

HR TEAM

	EMPLOYEE_ID	EMPL_JOIN_DATE	DEPT	SALARY	MANAGER_ID
1	1	2014-10-01	HR	40000	4
2	4	2017-09-01	HR	10000	5
3	5	2019-09-01	HR	35000	9

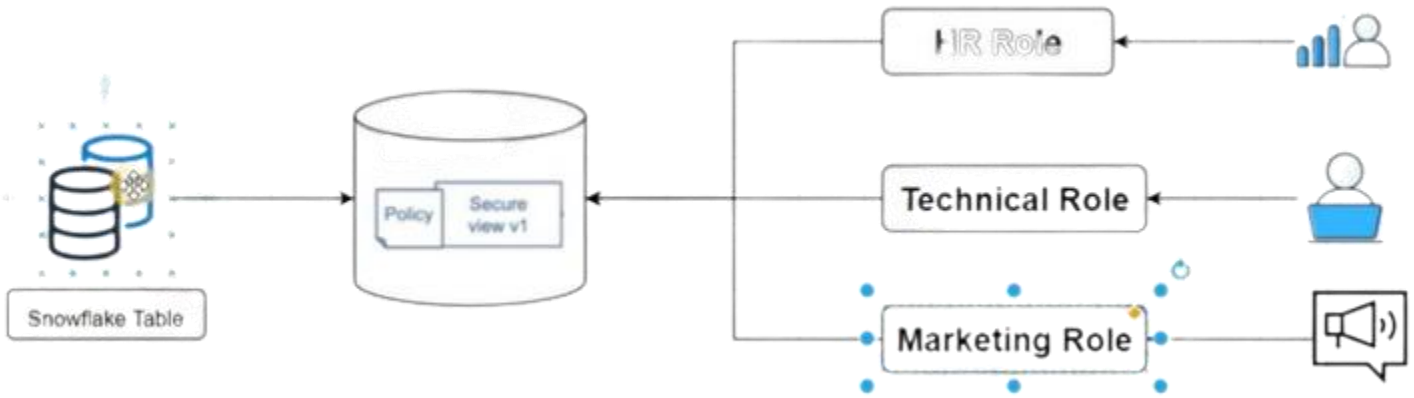
TECH TEAM

	EMPLOYEE_ID	EMPL_JOIN_DATE	DEPT	SALARY	MANAGER_ID
	2	2014-09-01	Tech	50000	9
	6	2015-09-01	Tech	90000	4

Marketing

EMPLOYEE_ID	EMPL_JOIN_DATE	DEPT	SALARY	MANAGER_ID
3	2018-09-01	Marketing	30000	5
7	2016-09-01	Marketing	20000	1

create or replace secure view vw_employee as select e.* from Company.public.employees e where upper(e.DEPT)=upper(current_role());



MARKETING • COMPUTE_WH (X-Small) Share

COMPANY.PUBLIC Settings Open in Workspaces

```
10                          (2,'2014-09-01','Tech',50000,9),
11                          (3,'2018-09-01','Marketing',30000,5),
12                          (4,'2017-09-01','HR',10000,5),
13                          (5,'2019-09-01','HR',35000,9),
14                          (6,'2015-09-01','Tech',90000,4),
15                          (7,'2016-09-01','Marketing',20000,1);
16
17  SELECT * FROM COMPANY.public.employees;
```


Results Chart

	EMPLOYEE_ID	EMPL_JOIN_DATE	DEPT	SALARY	...	MANAGER_ID
1	3	2018-09-01	Marketing	30000		5
2	7	2016-09-01	Marketing	20000		1

Query Details

Query duration 59ms

Rows 2

Query ID 01c266fe-0009-3858-...

EMPLOYEE_ID

Column-level security

Column-level security (CLS) restricts user access to specific columns within a database table, providing granular data protection for sensitive information like PII, financial data, or health records. It ensures only authorized users can view or modify particular fields, regardless of the application or query method used.

Column-level Security in Snowflake allows the application of a masking policy to a column within a table or view. Currently, Column-level Security includes two features:

1. Dynamic Data Masking
2. External Tokenization

Dynamic Data Masking is a Column-level Security feature that uses masking policies to selectively mask plain-text data in table and view columns at query time.

External Tokenization enables accounts to tokenize data before loading it into Snowflake and detokenize the data at query runtime. Tokenization is the process of removing sensitive data by replacing it with an undecipherable token.

ID	Email	Salary
2	name@cxample	70000

ID	Email	Salary
1	****	

```
CREATE OR REPLACE MASKING POLICY customer_accbal
   as (val number) returns number->
CASE WHEN CURRENT_ROLE() in ('SALES_ADMIN', 'MARKET_ADMIN') THEN val ELSE '####'  END;

ALTER TABLE PUBLIC.CUSTOMER MODIFY COLUMN C_ACCTBAL  SET MASKING POLICY customer_accbal;
```

Object Tagging

A tag is a schema-level object that can be assigned to another Snowflake object. You can set a maximum of 50 tags on a single object, including tables and views.

If you have reached the limit on tags and want to drop one, execute an ALTER <object> UNSET TAG statement.

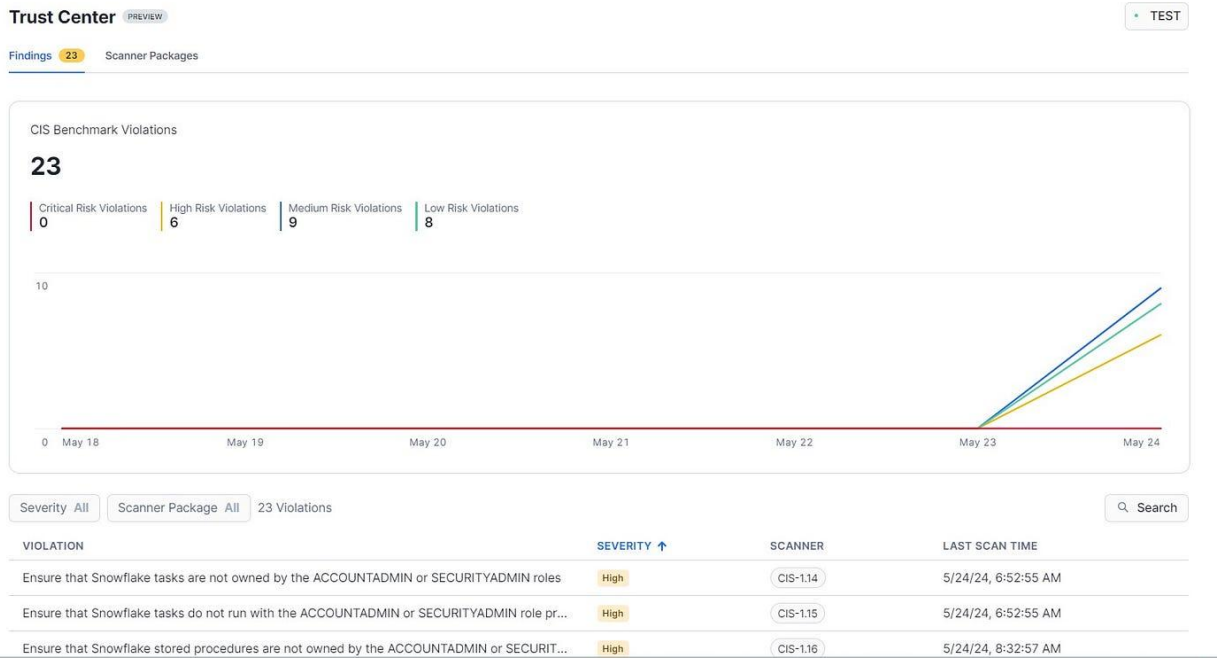
Privacy Policies

Ex:-

```
CREATE PRIVACY POLICY my_priv_policy
  AS ( ) RETURNS PRIVACY_BUDGET ->
  PRIVACY  BUDGET(BUDGET  NAME=> 'analysts');
```

Data-2-Dollar$

Trust Center

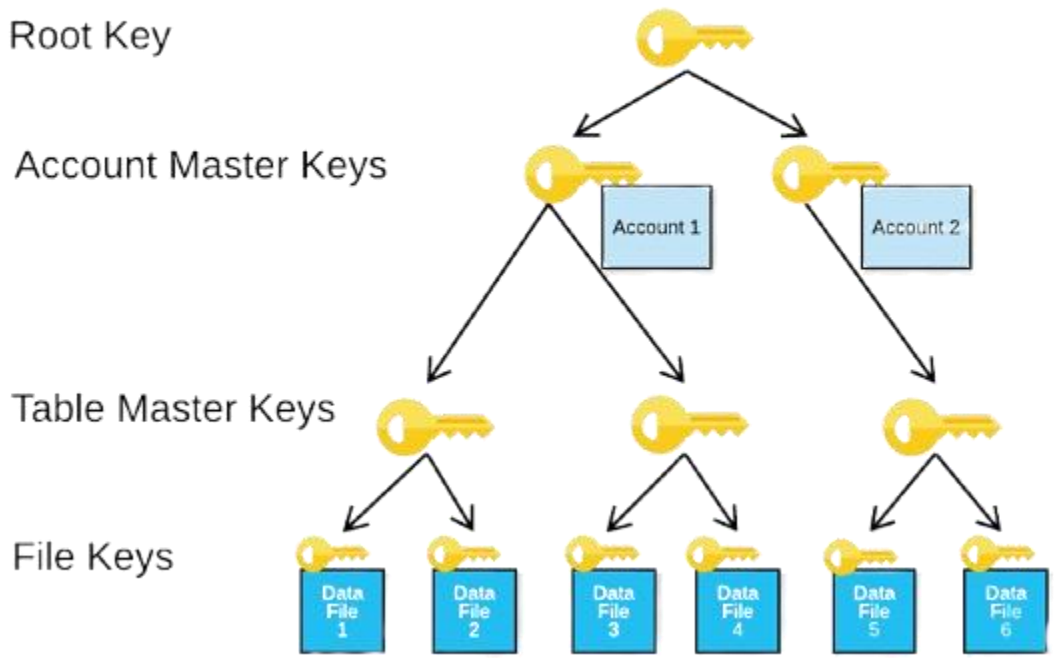


Encryption Key Management

Snowflake manages data encryption keys to protect customer data. This management can occur automatically without any need for customer intervention.

Customers can also use the key management service in the cloud platform that hosts their Snowflake account to maintain their own additional encryption key.
When enabled, the combination of a Snowflake-maintained key and a customer-managed key creates a composite master key to protect customer data in Snowflake. This is called Tri-Secret Secure.

All Snowflake customer data is encrypted by default. Snowflake uses strong AES encryption with a hierarchical key model rooted in a cloud-provider-hosted hardware security module.



Encryption key rotation

- Keys in the Snowflake-managed key hierarchy are automatically rotated by Snowflake when they are more than 30 days old.
- Active keys are retired, and new keys are created.
- When Snowflake determines the retired key is no longer needed, the key is automatically destroyed.

Alerts and Notifications

You can use Snowflake alerts to send notifications and perform actions automatically. In SQL, you can send a notification to an email address or queue by calling a built-in stored procedure.

Snowflake Alerts

If you need to send a notification or perform an action when data in Snowflake meets certain conditions, you can set up a Snowflake Alert.

Notifications in Snowflake

You can configure Snowflake to send notifications about Snowpipe and task errors to a cloud provider queue (Amazon SNS, Microsoft Azure Event Grid, or Google Cloud Pub/Sub).
You can also use a SQL statement to send a notification to an email address, a cloud provider queue, or a webhook.

Data replication and failover

Replication is the act of copying your data from a **Primary** account to a **Secondary** account. It's not just the tables; it's your users, roles, and even your security policies. Snowflake keeps them in sync so the backup always looks exactly like the original.

"Failover is the 'Big Red Button.' If your main region goes down, you promote your Secondary account to be the new **Primary**. Your users are automatically redirected, and business continues like nothing happened. This is what we call **Business Continuity.**

"Once the original region is fixed, you **'Failback.**' You sync any new data created during the outage back to the original site and switch the Primary status back to where it started."

Data lineage

Data Lineage tracks the life cycle of data: where it originated (**Upstream**), how it was transformed, and where it ended up (**Downstream**).

The Two Types You Must Know

1.Object Dependencies: "Who is my parent?" (e.g., A **View** depends on a **Table**).

2.Data Movement: "Where did my data go?" (e.g., A `MERGE` or `CTAS` operation that physically moves data from Table A to Table B).

The "Big Three" Metadata Views

View Name	Level of Detail	Exam Keyword
OBJECT_DEPENDENCIES	Table/View Level	**Impact Analysis** (e.g., "If I drop this table, what breaks?")
ACCESS_HISTORY	Column Level	**Sensitive Data Tracking** (The most granular view)
QUERY_HISTORY	Query Level	**Transformation Logic** (The actual SQL used to change the data)

Importants:-

•**Latency:** Remember that `ACCOUNT_USAGE` views (like `ACCESS_HISTORY`) have a latency of **up to 3 hours**. They are not real-time.

•**Retention:** `ACCOUNT_USAGE` stores lineage data for **365 days** (1 year).

•**Privileges:** To see lineage in the UI, a user typically needs the **`VIEW LINEAGE`** privilege.

Resource Monitors

Resource Monitors track credit usage by **Virtual Warehouses** and their supporting **Cloud Services**.
The Three Actions : When a threshold (percentage of quota) is hit, Snowflake can perform these actions:

- **Notify:** Sends an alert to all account admins.
- **Suspend:** Stops the warehouse from taking new queries, but allows currently running queries to finish.
- **Suspend Immediate:** Instantly cancels all running queries and shuts down the warehouse.

Warehouse Monitoring Metrics

WAREHOUSE_METERING_HISTORY: Best for **Cost**. Shows exactly how many credits each warehouse consumed by the hour.
QUERY_HISTORY: Best for **Granular Performance**. Shows execution time, queued time, and bytes scanned for individual queries.
WAREHOUSE_LOAD_HISTORY: Best for **Concurrency**. Shows if queries are **Running**, **Queued (Overloaded)**, or **Provisioning**.

Importants:-
Reset Frequency: Monitors can reset Daily, Weekly, Monthly (default), Yearly, or Never.

Role Required: Only the ACCOUNTADMIN role can create resource monitors by default.

Cloud Services: Resource monitors track cloud services usage but only trigger suspension based on Warehouse (Compute) usage.

Grace Period: Credit usage resets to 0 at 12:00 AM UTC on the scheduled reset day, regardless of your time zone.

Calculating virtual warehouse credit usage

1. The "Per-Second" Billing Rule

Snowflake bills for warehouse usage by the **second**, but there is a catch, there is a **60-second minimum** every time a warehouse starts or is resized to a larger size.

2. The T-Shirt Size Multiplier

Size	Credits Per Hour
X-Small	1
Small	2
Medium	4
Large	8
X-Large	16

3. Multi-Cluster Warehouses (The Addition Rule)

If you are using a Multi-Cluster Warehouse (Enterprise Edition+), the math is simply: **Total Credits = (Credits for Size) × (Number of Clusters Running)**

4. Cloud Services "Free" Tier

Snowflake charges for **Cloud Services** (like metadata hits or security checks), but only if they exceed **10% of your daily warehouse usage**.

ACCOUNT_USAGE schema

Feature	ACCOUNT_USAGE (The Archive)	INFORMATION_SCHEMA (The Real-Time)
Data Retention	**365 Days** (1 Year)	7 to 14 Days (Short-term)
Latency	**Up to 3 Hours** (Delayed)	Real-time (Immediate)
Dropped Objects	Includes them in history	Only shows current objects

Top 5 Views You MUST Know

1. **`QUERY_HISTORY`**: Tracks every query run in the last year. Used to find "expensive" queries or slow performance.
2. **`WAREHOUSE_METERING_HISTORY`**: The "Billing" view. Shows credit consumption per hour per warehouse.
3. **`LOGIN_HISTORY`**: The "Security" view. Tracks successful and failed logins (great for spotting brute-force attacks).
4. **`STORAGE_USAGE`**: Shows your data footprint (Database, Stage, and Fail-safe storage) for billing.
5. **`ACCESS_HISTORY`**: (Enterprise Edition+) Tracks which users accessed which specific columns. Crucial for data lineage.

Data-2-Dollar$

Internal stages

3 Types of Internal Stages:

User Stage

Table Stage

Named Internal Stage

User Stage (@~)

- A user stage is allocated to each user for storing files.
- To store files that are staged and managed by a single user but can be loaded into multiple tables.
- User stages cannot be altered or dropped.
- This option is not appropriate if multiple users require access to the files.

Table Stage (@%)

- A table stage is available for each table created in Snowflake.
- Table stages have the same name as the table.

  e.g. a table named mytable has a stage referenced as @%mytable
- To store files that are staged and managed by one or more users but only loaded into a single table.
- Table stages cannot be altered or dropped.
- Note that a table stage is not a separate database object, rather it is an implicit stage tied to the table itself.

Named Internal Stage (@)

- A named internal stage is a database object created in a schema
- To store files that are staged and managed by one or more users and loaded into one or more tables.
- Create stages using the CREATE STAGE command (similar to external stages)

External Stages

- An external stage is the external cloud storage location where data files are stored.
- Loading data from any of the following cloud storage services is supported regardless of the cloud platform that hosts your Snowflake account:
  - Amazon S3
  - Google Cloud Storage
  - Microsoft Azure
- An external stage is a database object created in a schema.
- This object stores the URL to files in cloud storage.
- Also stores the settings used to access the cloud storage account (Storage Integration Object).
- Create stages using the CREATE STAGE command.


```
CREATE OR REPLACE STAGE MYDB.external_stages.sample_stage
    url='s3://bucketsnowflakes3';
```

Server-side encryption

In Snowflake ,by default, **all data at rest is automatically encrypted** using AES-256 bit encryption, and all data in transit is protected via TLS 1.2.

1. The Hierarchical Key Model

- **Root Key:** The "master of masters," stored securely in a Hardware Security Module (HSM) provided by the cloud host (AWS, Azure, or GCP). It never leaves the HSM.
- **Account Master Keys (AMK):** Unique to your specific Snowflake account.
- **Table Master Keys:** Unique to each individual table.
- **File Keys:** Every single micro-partition file (the small files Snowflake uses to store data) has its own unique encryption key.

2. Automatic Maintenance

- **Key Rotation**
- **Periodic Rekeying**

3. Advanced Control: Tri-Secret Secure

- **Snowflake-managed key.**
- **Customer-managed key (CMK) (stored in your own AWS KMS, Azure Key Vault, or Google KMS).**
- **User Credentials.**

4. Encryption for Staging Data: You can choose to use Snowflake’s SSE or your own cloud provider's encryption

Directory Tables

Directory Tables allow you to query a stage (like an S3 bucket or internal stage) as if it were a database table.A Directory Table is not a separate object; it is a feature you **enable** on a Snowflake Stage. Once enabled, it stores a snapshot of the files in that stage, including:

Relative Path: Where the file is located.
File Size: How big it is.
Last Modified: When it was uploaded.
ETag: The file's unique version identifier.
File URL: A scoped URL used to download or view the file.

Why use them?

Directory Tables are the backbone of **Unstructured Data Management** in Snowflake. Common use cases include:

Data Pipelines: Using a SELECT statement on the Directory Table to find only files uploaded in the last hour to trigger a processing job.

AI/ML: Passing file URLs to Snowflake **Cortex AI** functions or Python scripts (Snowpark) to perform OCR on PDFs or image recognition.

Governance: Quickly auditing how much storage your unstructured files are taking up across different departments.

```
CREATE STAGE my_external_stage
  URL = 's3://my-bucket/files/'
  STORAGE_INTEGRATION = my_s3_int
  DIRECTORY = (ENABLE = TRUE);

SELECT * FROM DIRECTORY(@my_external_stage);
```

Copy into Command

```
COPY INTO <table_name>
FROM  @ExternalStage
FILES = ( '<file_name>','<file_name2>')
FILE_FORMAT = <file_format_name>
```

Copy Options

VALIDATION_MODE
RETURN_FAILED_ONLY
ON_ERROR
FORCE
SIZE_LIMIT
TRUNCATECOLUMNS
ENFORCE_LENGTH
PURGE
LOAD_UNCERTAIN_FILES

RETURN_FAILED_ONLY

```
COPY INTO <table_name>
FROM  @ExternalStage
FILES = ( '<file_name>','<file_name2>')
FILE_FORMAT = <file_format_name>
RETURN_FAILED_ONLY = TRUE | FALSE ;
```

→ Specifies whether to return only files that have failed to load in the statement result
→ Default is FALSE

VALIDATION MODE

```
COPY INTO <table_name>
FROM  @ExternalStage
FILES = ( '<file_name>','<file_name2>')
FILE_FORMAT = <file_format_name>
VALIDATION_MODE =
RETURN_n_ROWS|RETURN_ERRORS|RETURN_ALL_ERRORS;
```

→ Validate the data files instead of loading them into the table
→ RETURN_ERRORS gives all errors in the files
→RETURN_ALL_ERRORS gives all errors from previously loaded files if we have used ON_ERROR = CONTINUE
→ RETURN_N_ROWS displays first N records and fails at the first error record

FORCE & SIZE LIMIT

```
COPY INTO <table_name>
FROM  @ExternalStage
FILES = ( '<file_name>','<file_name2>')
FILE_FORMAT = <file_format_name>
FORCE = TRUE | FALSE
SIZE_LIMIT = <NUMBER>;
```

→ To load all the files, regardless of whether they've been loaded previously

→ Default is False, if we don't specify this property Copy command will not fail but it skips loading the data

→Specify maximum size in bytes of data loaded in that command

→ When the threshold is exceeded, the COPY operation stops loading

→ In case of multiple files of same pattern also it will stop after the size limit, that means some files can be loaded fully and one file will be loaded partially

ON_ERROR

```
COPY INTO <table_name>
FROM  @ExternalStage
FILES = ( '<file_name>','<file_name2>')
FILE_FORMAT = <file_format_name>
ON_ERROR = CONTINUE
| SKIP_FILE | SKIP_FILE_num |
                SKIP_FILE_num% | ABORT_STATEMENT
```

→ CONTINUE – To skip error records and load remaining records

→ SKIP_FILE – To skip the files that contain errors

→ SKIP_FILE_Num – Skip a file when the number of error rows found in the file is equal to or exceeds the specified number.

→ SKIP_FILE_Num% – Skip a file when the percentage of error rows found in the file exceeds the specified percentage.

→ Default is ABORT_STATEMENT, Abort the load operation if any error is found in a data file.

PURGE

```
COPY INTO <table_name>
FROM  @ExternalStage
FILES = (
'<file_name>','<file_name2>')
FILE_FORMAT = <file_format_name>
PURGE = TRUE | FALSE
```

→ Specifies whether to remove the data files from the stage automatically after the data is loaded successfully.

→ Default is FALSE

TRUNCATE_COLUMNS or ENFORCE_LENGTH

```
COPY INTO <table_name>
FROM  @ExternalStage
FILES = ( '<file_name>','<file_name2>')
FILE_FORMAT = <file_format_name>
TRUNCATECOLUMNS = TRUE | FALSE - Default is
False ;
(Or) ENFORCE_LENGTH = TRUE | FALSE - Default is
TRUE ;
```

→ Specifies whether to truncate text strings that exceeds the target column length

→ Default is FALSE, that means if we don't specify this option, and if text strings that exceeds the target column length, then Copy command will fail.

LOAD UNCERTAIN FILES

```
COPY INTO <table_name>
FROM  @ExternalStage
FILES = ( '<file_name>','<file_name2>')
FILE_FORMAT = <file_format_name>
LOAD_UNCERTAIN_FILES = TRUE | FALSE
```

→specifies to load files for which the load status is unknown. The COPY command skips these files by default.

Note: The load status is unknown if ***all*** of the following conditions are true:

- The file's LAST_MODIFIED date is older than 64 days.
- The initial set of data was loaded into the table more than 64 days earlier.
- If the file was already loaded successfully into the table, this event occurred more than 64 days earlier.

Error handling options

1. Error Handling in Data Loading (`COPY INTO`)

When bulk loading data, the `ON_ERROR` copy option is your most powerful tool. It determines how the system reacts when it encounters a malformed row (e.g., a string in a date column).

2. Handling Errors in SQL Scripting

If you are writing Snowflake Scripting (blocks of SQL) or Stored Procedures, you use **Exception Handling** similar to Python or Java.

Example:

```
BEGIN
   INSERT INTO target_table SELECT * FROM source_table;
EXCEPTION
   WHEN OTHER THEN
      -- Capture the error code and message
      RETURN 'Error ' || SQLCODE || ': ' || SQLERRM;
END;
```

Streams

- A stream object records DML changes made to tables including inserts, updates, and deletes.
- Also Stream stores metadata about each change, so that actions can be taken using this metadata.
- We call this process as change data capture (CDC).
- Streams tracks all row level changes to a source table using offset but doesn't store the changed data.
- Once these changes are consumed by the Target table, this offset moves to the next point.
- Streams can be combined with tasks to set continuous data pipelines.
- Snowpipe + Stream + Task ➔ Continuous Data Load.

Along with the changes made to the source table, Streams maintain 3 metadata fields.

1. METADATA$ACTION: Indicates the DML operation (INSERT, DELETE) recorded.

2. METADATA$ISUPDATE: Indicates whether the operation was part of an UPDATE statement. Updates to rows in the source object are represented as a pair of DELETE and INSERT records in the stream with a metadata column METADATA$ISUPDATE values set to TRUE.

3. METADATA$ROW_ID: Specifies the unique and immutable ID for the row, which can be used to track changes to specific rows over time.

How a Stream works? or Data flow in a Stream

Consuming data from Streams

We use **merge** statement for consuming the changes from stream and applying the same on Target tables.

// To identify **Insert** records

```
WHERE metadata$action = 'INSERT AND metadata$isupdate = 'FALSE';
```

// To identify **Update** records


```
WHERE metadata$action = 'INSERT' AND metadata$isupdate = 'TRUE';
```

To identify **Delete** records

```
WHERE metadata$action = 'DELETE' AND metadata$isupdate = 'FALSE';
```

Note: If we want to consume these changes to multiple tables then we have to create multiple streams.

Types of Streams

1. Standard Streams: A Standard stream tracks all DML changes to the source object, including inserts, updates, and deletes (including table truncates).

```
CREATE OR REPLACE STREAM my_stream ON TABLE my_table;
```

2. Append-only Streams: An Append-only stream tracks row inserts only. Update and delete operations (including table truncates) are not recorded.

```
CREATE OR REPLACE STREAM my_stream ON TABLE my_table
APPEND_ONLY = TRUE;
```


3. Insert-only Streams: Supported for **External tables** only. An insert-only stream tracks row inserts only. They do not record delete operations.

```
CREATE OR REPLACE STREAM my_stream ON EXTERNAL TABLE my_table
INSERT_ONLY = TRUE;
```

Tasks

- We use tasks for scheduling in Snowflake.
- We can schedule
  - SQL queries
  - Stored procedures
- Tasks can be combined with streams for implementing the continuous change data captures(CDC).
- We can maintain a DAG of tasks to keep the dependencies between tasks.
- Tasks require compute resources to execute SQL code, we can choose either of
  - Snowflake managed compute resources(Serverless)
  - User managed (Virtual warehouses)


```
CREATE OR REPLACE TASK task_name
        WAREHOUSE = 'warehouse name'
        SCHEDULE = 'Time or Cron'
        AFTER dep_task_name
```

Snowflake Openflow

Snowflake Openflow is a managed, unified data integration service designed to simplify the movement of data from virtually any source into the Snowflake AI Data Cloud.

Built on the open-source **Apache NiFi 2.0** framework, it provides a visual, low-code environment to build, scale, and govern complex data pipelines—eliminating the need for fragmented third-party ETL tools or custom scripting.

Key Capabilities

Unified Multi-Modal Ingestion

Real-Time Streaming & CDC

Massive Throughput

AI-Ready Pipelines

Core Architecture: Control vs. Data Plane

Control Plane (Snowflake-Managed): Hosted within the Snowsight UI. This is your "mission control" where you use a drag-and-drop canvas to design, monitor, and manage your data flows.

Data Plane (Execution Layer): This is where the actual data processing happens. Snowflake offers two deployment models:

Snowflake-Managed: Runs within Snowflake's infrastructure (using Snowpark Container Services).

Bring Your Own Cloud (BYOC): The runtime executes within your VPC (AWS or Azure). This allows you to process sensitive data locally, complying with strict security policies while Snowflake still manages the "logic" of the flow.

Snowflake drivers & Connectors

Snowflake Drivers

Driver	Description	Best For
Python Connector	A pure-Python package with no dependencies on JDBC/ODBC. Supports Pandas DataFrames.	Data science, ETL scripts, and web apps.
JDBC Driver	A Type 4 driver that supports standard Java database connectivity.	Java/Scala applications and legacy BI tools.
ODBC Driver	Supports the ODBC 3.8 specification for Windows, Linux, and macOS.	Desktop BI tools (Tableau, Power BI) and C++ apps.
Go Driver	Implements the standard database/sql package.	High-performance backend microservices.
.NET Driver	A managed driver for the .NET framework and .NET Core.	Windows-based enterprise applications.
Node.js Driver	A JavaScript driver for asynchronous data access.	Server-side web applications and Lambda functions.

Snowflake Connectors

Snowflake Connector for Spark

Snowflake Connector for Kafka

Snowflake Connector for Kinesis

API integration

API integration in Snowflake works in two directions: **Inbound** (external apps talking to Snowflake) and **Outbound** (Snowflake talking to external APIs).

1. Inbound: Snowflake SQL API

The **SQL API** is a RESTful interface that allows custom applications to interact with Snowflake without using traditional drivers (JDBC/ODBC).

2. Outbound: Snowflake External Access

This is the modern way to make Snowflake "talk" to the outside world. It allows you to call external APIs (like OpenAI, Salesforce, or Slack) directly from within **Python/Java UDFs** or **Stored Procedures**.

```
CREATE [ OR REPLACE ] API INTEGRATION [ IF NOT EXISTS ] <integration_name>
  API_PROVIDER = { aws_api_gateway | aws_private_api_gateway |
aws_gov_api_gateway | aws_gov_private_api_gateway }
  API_AWS_ROLE_ARN = '<iam_role>'
  [ API_KEY = '<api_key>' ]
  API_ALLOWED_PREFIXES = ('<...>')
  ENABLED = { TRUE | FALSE }
  [ COMMENT = '<string_literal>' ]
  ;
```

Git integration

Data-2-Dollar$