[**50 Free Questions - SnowFlake SnowPro Core Certification (whizlabs.com)**](https://www.whizlabs.com/blog/snowflake-snowpro-core-certification-free-questions/)

[**Understanding access control & roles on Snowflake - YouTube**](https://www.youtube.com/watch?v=b-YRXJgjDC8&list=PLy4OcwImJzBIX77cmNYiXIJ3tBhpNSUKI&index=2)

**Introduction:**

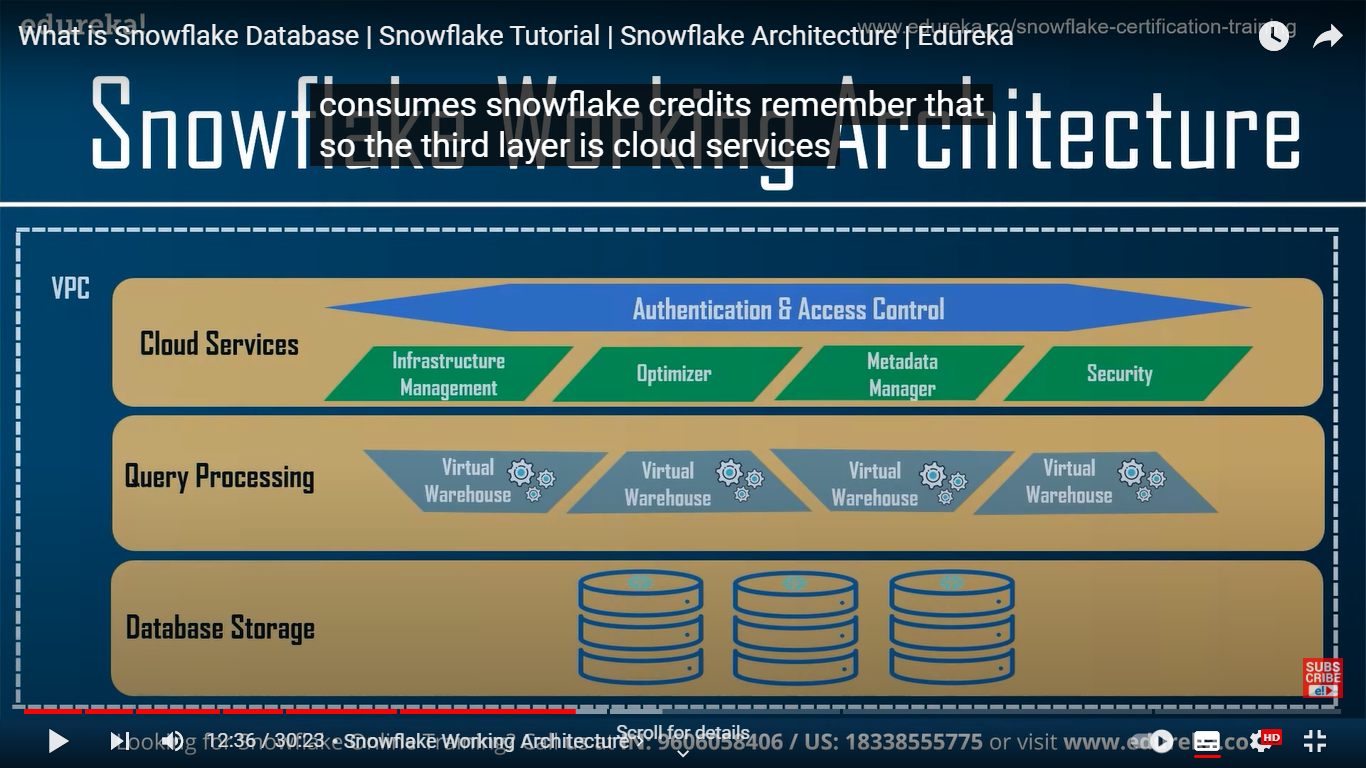
1. Data platform as a cloud services (AWS, GCP, Azure)
2. Software as a service
3. No hardware
4. Virtual compute instances for compute needs
5. Storage service for persistent storage of data

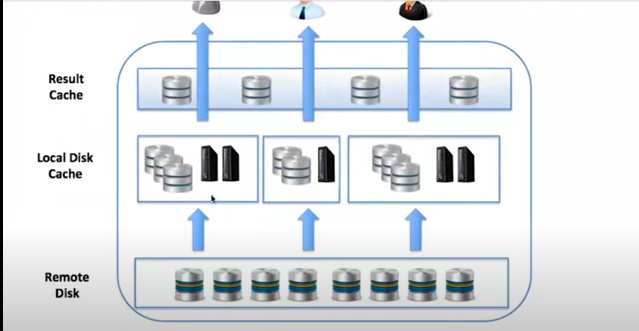
**Overview:**

* Snowflake is a fully managed cloud based data warehouse.
* Data storage and data analytic solutions
* It is software as a Service as a service (Saas) platform to load, analyse and report on massive data volumes.

**Architecture:**

* Data Storage, process and analytics
* A hybrid of traditional shared-disk and shared-nothing database architecture.

****

****

**Storage:**

* All the data stored as an internal, optimised, **compressed columnar format** (micro-partitions- represents logical structure of table)
* Can't access the data directly, only through SQL, etc

**Query Processing:**

* Here queries are being processed
* Uses virtual warehouses -an MPP compute cluster
* **Massively parallel processing(MPP)** is **a means of crunching huge amounts of data by distributing the processing over hundreds or thousands of processors**
* Each processor in an MPP system has its own memory, disks, applications, and instances of the operating system.

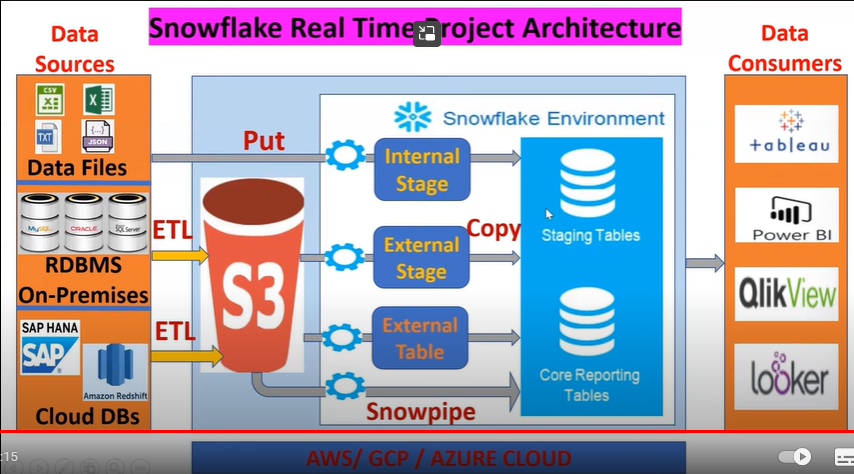
**Cloud Services:**

* Command centre of SF - coordinates and ties all activities together within SF
* Gets provisioned by snowflake and within AWS, Azure or GCP
* We don't have access to build or do modifications, but services this layer has:

1. Authentication
2. Infrastructure management
3. Metadata management
4. Query parsing and optimisation
5. Access control

**Architecture Flow of Snowflake: (example of real time project)**

1. **Using S3**



Step1: Data are taken in the form of txt, xml, json or csv format id data is in RDBMS then its compressed into csv, txt, xml or json format then it is passed

Step2: Then through ETL(extract transform and load) the data is passed into snowflake env

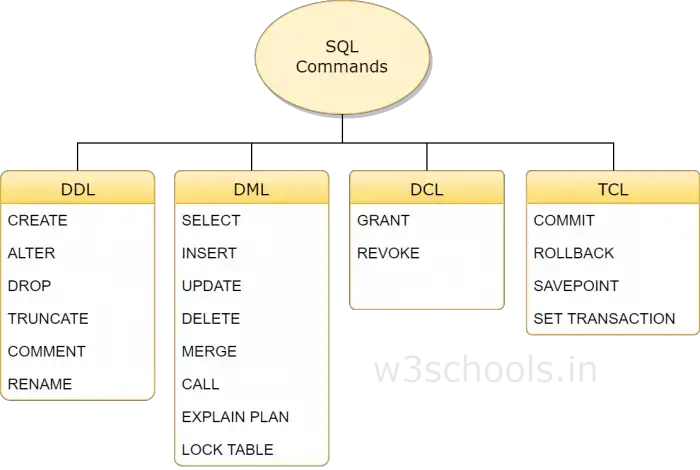
Step3: For passing the data there are four possible ways through which it can be passed

1. Through **PUT** command where using the internal stage we can pass the data into snow env.
2. Through **S3** then using **COPY** command
3. Through **S3** then using external staging
4. Through **S3** then using snowpipe

Note: While passing the data snowflake create an virtual warehouses

Now here some challenges are their with S3

* Does not support updating the data (**only overwriting the data**)



**Using DDL : Data definition language with S3:** here we will pay for compute cost

Step1: create database

Step2: create schema

Step3: Create Table

Step4: Load Data in tables

* Loading from data files
* Using Ingestion tools

**Snowflake Virtual Warehouse- VW**

Comes in **various Sizes**: (These are amazon EC2 instances)

* **X-small – Single Node**
* **Small – Two Nodes**
* **Medium – Four Nodes**
* **Large – Eight Nodes**
* **X Large – Sixteen Nodes**

**Multi- Cluster VW:**

In concurrent queries if the nodes are occupied then next queries are queued,but in multi-cluster VW it will automatically detect queues and create new nodes.

**Connecting to snowflake:**

* **web /CLI/ODBC and JDBC/ Native** Connectors **(Kafka/python/spark/Node Js/.net/php) snowflake driver for go language**
* 3rd part connects (**Tableau, fivetran,etc)**

**Regions / Pricing**

* **On-demand storage- 40$/TB**
* **Capacity Storage- 23$/TB**

Computation/ warehouse cost:

* Charged for active warehouse per hour, depend on the size of warehouse
* **Minimum 60s** on resuming, **per 1s after**

Cost of storage(Temp,Transit, perm Tables & time-travel data & fail-safe)

**Supported regions:**

* Each SF in single region
* **17 regions in total**
* **AWS - 9 regions, GCP - 1, Azure - 7**

**WareHouse:**

**Scale-up and scale-out:**

* **Scale-up:** Resizing to a larger warehouse is to accommodate more complex queries/workload
* **Scale-Out**: To accommodate more concurrent queries/consumption. Size and complexity of query determines the concurrency.

**Data Loading:**

* **File Location**
* On local
* On cloud

1. Aws: can load directly from S3 into SF
2. Azure: can load directly from blob storage into SF
3. GCP: Can load directly from GCS into SF

* **File Type**
* Structured

1. Delimited files(CSV, TSV etc)

* Semi-structured

1. JSON: SF can auto detect if snappy compressed
2. ORC: SF can auto detect if snappy compressed or zlib
3. Parquest: Sf can auto detect if snappy compressed
4. XML: in preview

If file is uncompressed, on load to SF it is gzip,

**SF auto compresses** data from local fs to gzip - can specify compression type on loading compressed data

**Encryption for load:**

* **Loading unencrypted files**:: SF **auto encrypts files using 126-bits keys!** (or 256-keys -requires configuring)
* **Loading encrypted files**:: provide your **key to SF on load**

**Types of Loading**

Bulk Loading:

* Loading from stages
* Copy command
* Transformations possible

Continuous Loading:

* To load small volumes of data
* Automatically once they are added to stages
* Snowpipe (serverless feature)

**Staging Data:**

1. Organising Data by Path- snowflake staging

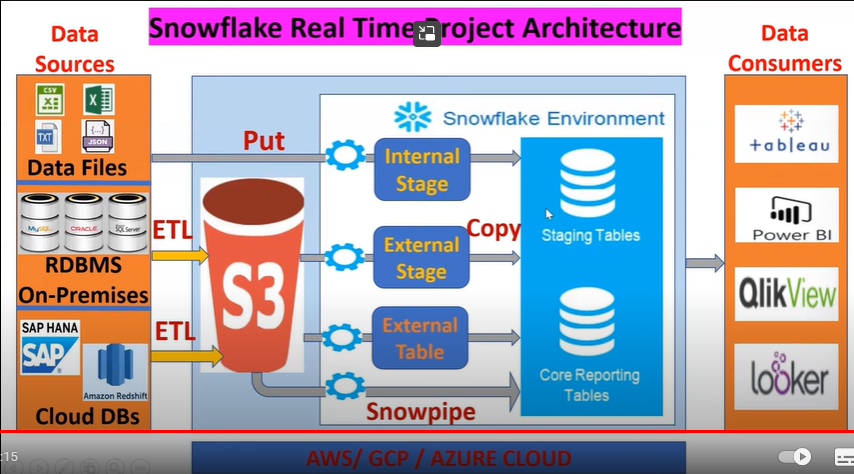
* Both internal and external ref can include path in the cloud storage
* SF recommends (for file organisation in cloud storage):

1. **Partition into logical path**- includes identifying details with data like date, source identifiers, geographical location, etc)
2. **Organising file by Path** - allow copy fraction of partitioned data in SF in one command (COPY statements)

- Named Stage operation :

a. **Staging** = place where the location/path of the data is stored to assist in processing the upload of files.

* Note: file uploaded to snowflake staging area using **PUT** are automatically encrypted with **128-bit or 256-bit** (CLIENT\_ENCRPTION\_KEY\_SIZE to specify) keys



**Loading Data:**

1. **Copy Command** : parallel execution

* Support loading by internal stage or S3 bucket path
* Specifying specific list of files to upload (**1000 files max at a tim**e)
* Identify files through pattern matching
* **Validating Data**:

1. Use **VALIDATION\_MODE**: validate error on load- does not load into table
2. **ON\_ERROR** to run actions to follow

* When **COPY command runs, SF sets load status in the tables metadata**

1. Prevents parallel COPY from loading the same file
2. On complete, SF sets the load status in the tables metadata
3. Metadata **(expires after 64 days):**

* Name of file
* File size
* Etag of the file
* # rows parsed in file
* Timestamp of last load
* Information on errors during load

- SF recommends removing the Data from the stage once the load is completed to avoid reloading again **- use REMOVE** command

**2. Semi-structured**

* SF loads semi-structured data to **variant type** column
* Can load ss data into multiple columns but ss data must be stored as field in structured data
* Use **FLATTEN** to explode compounded values into multiple rows

**3. Data Transformation during Load**

* Supported

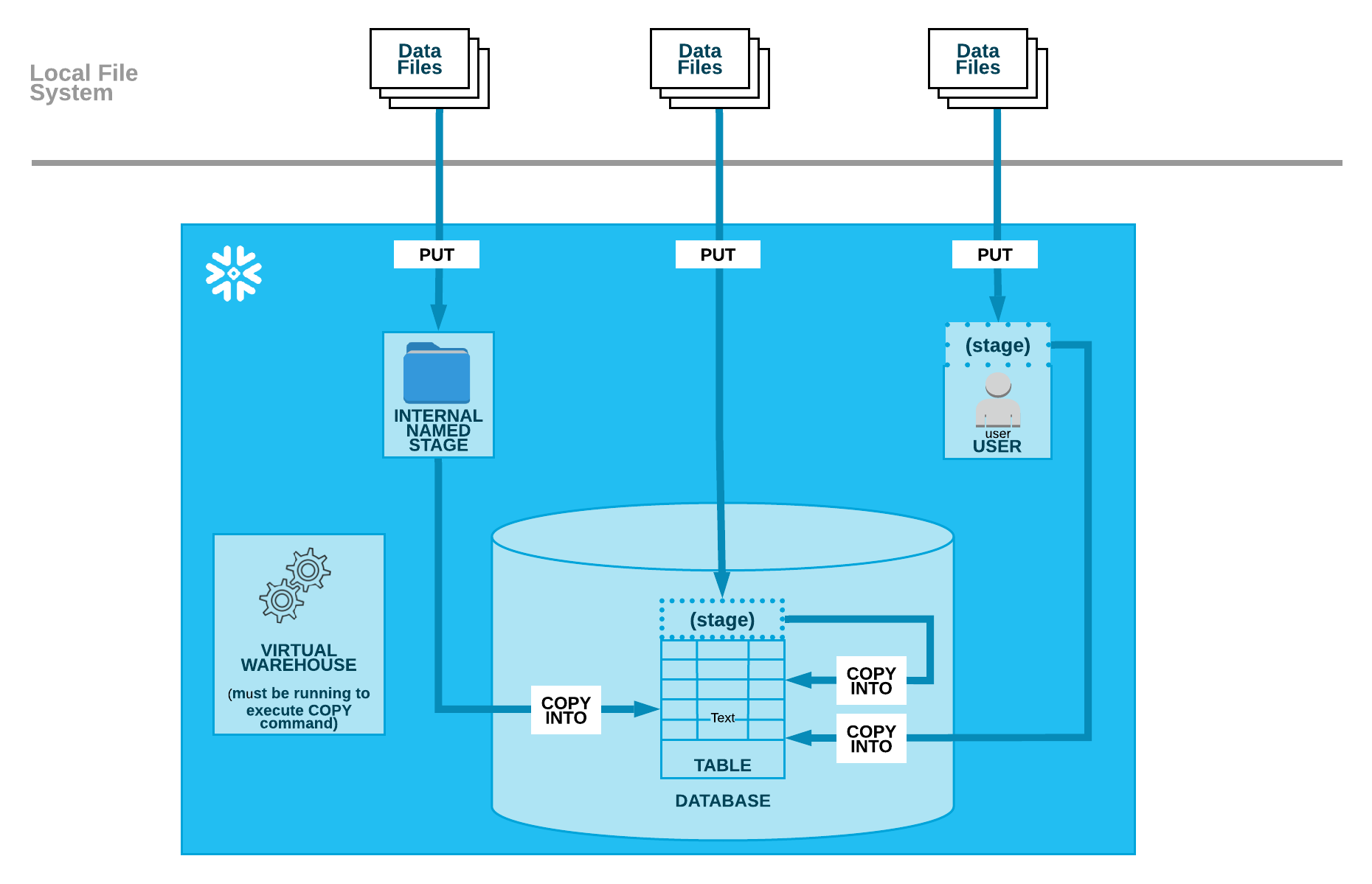
1. Sequence, substring, to\_binary, to\_decimal

* Command not supported

1. WHERE, FLATTERN, JOIN, GROUP BY, DISTINCT
2. VALIDATION\_MODE
3. CURRENT\_TIME

**Bulk Loading:**

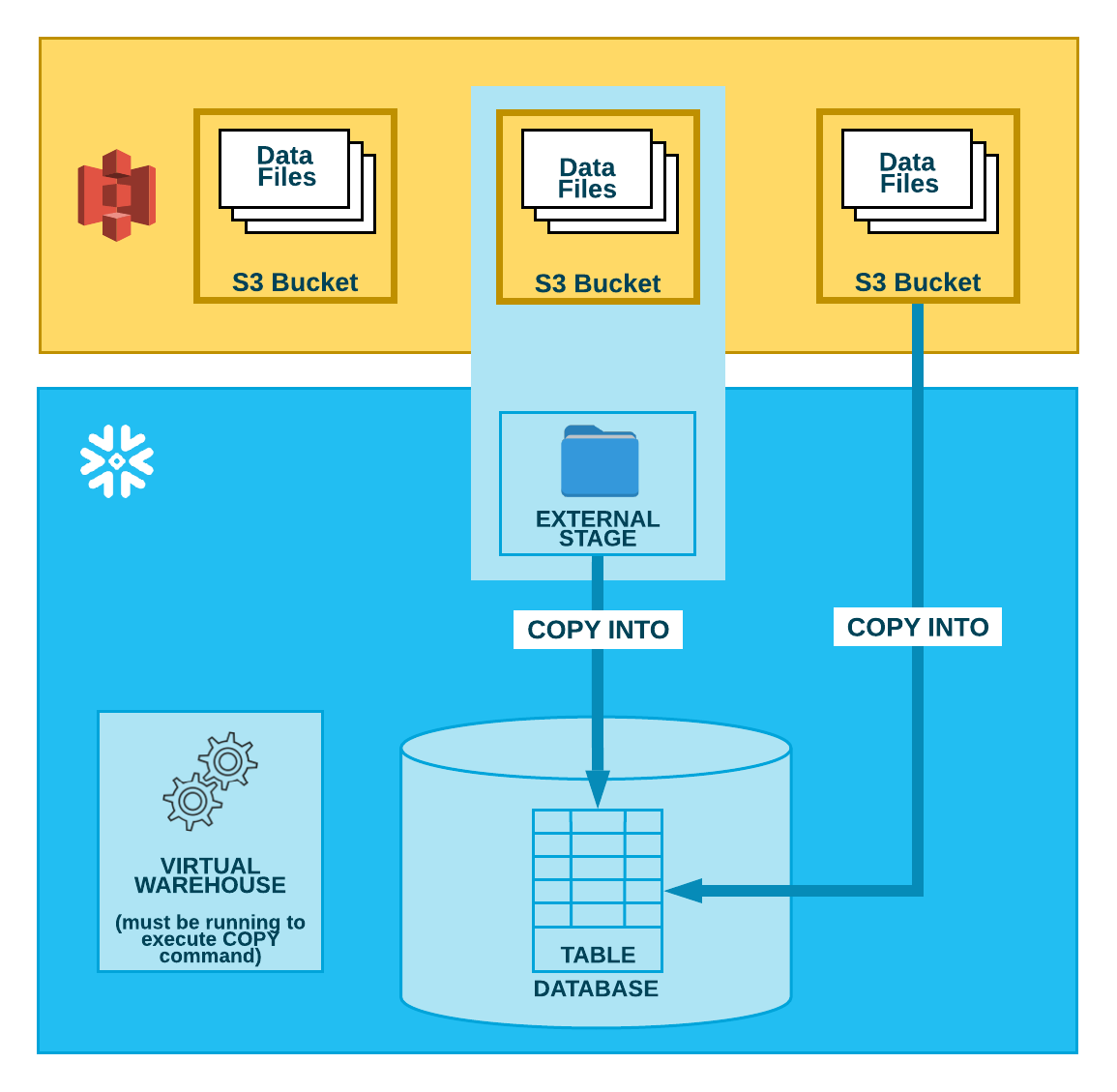
1. Bulk loading from a local file system:



* Step 1: upload one or more data files to a snowflake stage (various types) using the **PUT** command from local machine
* Step2: Use the **COPY INTO command** to load the contents from stage to table

**Bulk Loading from AWS S3**

* Creating S3 bucket
* Uploading files in S3
* Creating policies: multiple options
* Creating integration object
* Loading from S3



* SF uses **S3 Gateway** endpoint.
* If **region bucket = SF region -no route through public internet**
* Securely access S3:

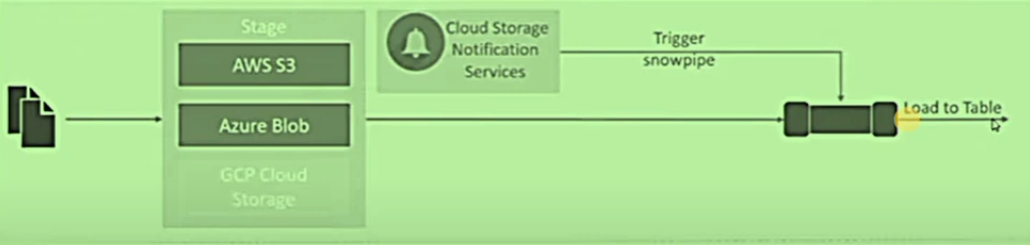
1. Configure storage integration object - delegate authentication responsibility for external cloud storage to SF identity and IAM
2. Configure AWS IAM role → allow S3 bucket access
3. Configure IAM user, provide Key and Secret Key

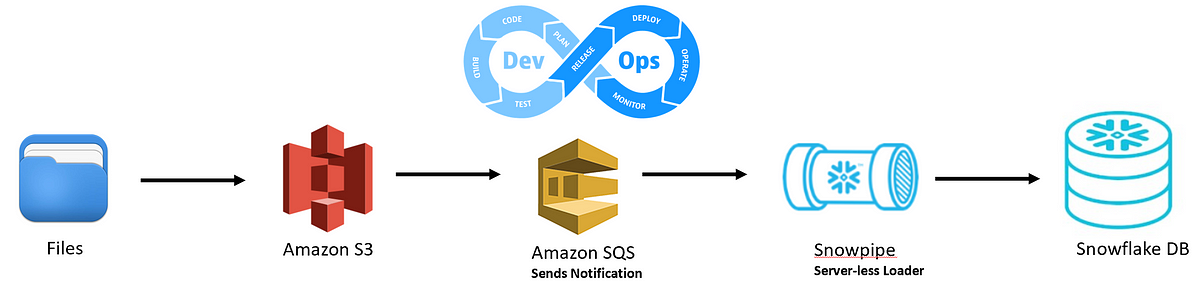
**Snowpipe**

* **Loades data in small volumes every hour or minutes.**
* Enables loading data from files as soon as they’re available in a stage (incremental loading), rather than manually executing **COPY statements** on a schedule to load larger batches (bulk loading)
* Loads data within minutes after files are added to stage and submitted for ingestion.
* Micro file size is **100-250 MB.**
* Uses pipe (first-class SF object) which contains the **COPY** command.
* This is like streaming data into a table as long as you have created a **named stage.**
* Generally loads **older files** first but there is no guarantee that files are loaded in the same order they are staged
* Snowpipe uses **file loading metadata** associated with each pipe object to prevent reloading the same files
* Different mechanisms for detecting the staged files are:

1. Automating Snowpipe using cloud messaging
2. Calling Snowpipe **REST** endpoints

* SF charges are calculated by per-second/per-core granularity.





Steps in **creating snowpipe (we can use DDL command for this)**

1. Create storage integration object
2. Create a stage using storage integration object
3. Create a pipe by using the copy command
4. Setup event notification at cloud storage providers end.

link for the snowpipe better understanding:

[KSnow: Load continuous data into Snowflake using Snowpipe - Knoldus Blogs](https://blog.knoldus.com/ksnow-load-continuous-data-into-snowflake-using-snowpipe/)

**COPY command vs snowpipe**

|  | **Copy Command** | **Snowpipe** |
| --- | --- | --- |
| **Authentication** | Security options supported by the clients for authenticating and initiating a user session | When calling the REST endpoints: key pair authentication with **JSON web Token (JWT)**. JWTs are **signed using a public/private key pair** with **RSA encryption**. |
| **Transactions** | Adds data to a table in transactions alongside any other SQL statements submitted manually by users.  Loads are always performed in a single transaction | Adds data to a table in transactions controlled by Snowflake with no opportunity to involve other statements in the transaction.  Loads are combined or split into a single or multiple transactions based on the number and size of the rows in each data file.  Rows of partially loaded files (based on the ON\_ERROR copy option setting) can also be combined or split into one or more transactions. |
| **Load History** | Stored in the metadata for the target table for **64 days**. Available upon completion of the COPY statement as statement results. | Stored in the metadata for the pipe for **14 days,** Must be requested from snowflake  (via a REST endpoint, SQL table function, or **ACCOUNT\_USAGE view)** |
| **Compute Resources** | Requires a user-specified warehouse to execute **COPY** statements. | Uses **snowflake-supplied compute** resources |
| **Cost** | Billed for the amount of time each virtual warehouse is active | Billed according to the compute resources used in the snowpipe warehouse while loading the files. |

**Unloading Data:**

Like data loading we can export data in bulk i.e unloading data from a database table into flat, delimited text files.

Step1: **COPY INTO** command

* Supporting queries with **SELECT statement**
* Unloading into SIngle or Multiple FIles (**MAX\_FILE\_SIZE: default 16MB** and max **size 5G** for **AWS/GCP and 256MB for Azure**)
* Partitioned data unloading
* Parameter **SINGLE = TRUE** to force unloading into a single file

Step2: Download the file from stage.

**Format and restrictions:**

* Allowed **formats** for export (only **UTF-8 as encoding allowed**)

1. Delimited Files (CSV etc)
2. JSON
3. Parquet

* Allowed **Compression** for export

1. Gzip (default)
2. Bzip2
3. Brotli
4. Zstandard

* Encryption for export

1. Internal stage - **128-bit or 256-bit** - gets decrypted when downloaded
2. External- customer supply encryption key

**Considerations:**

* **Empty string and null**

1. FIELD\_OPTIONALLY\_ENCLOSED\_BY
2. EMPTY\_FIELD\_AS\_NULL
3. NULL\_IF

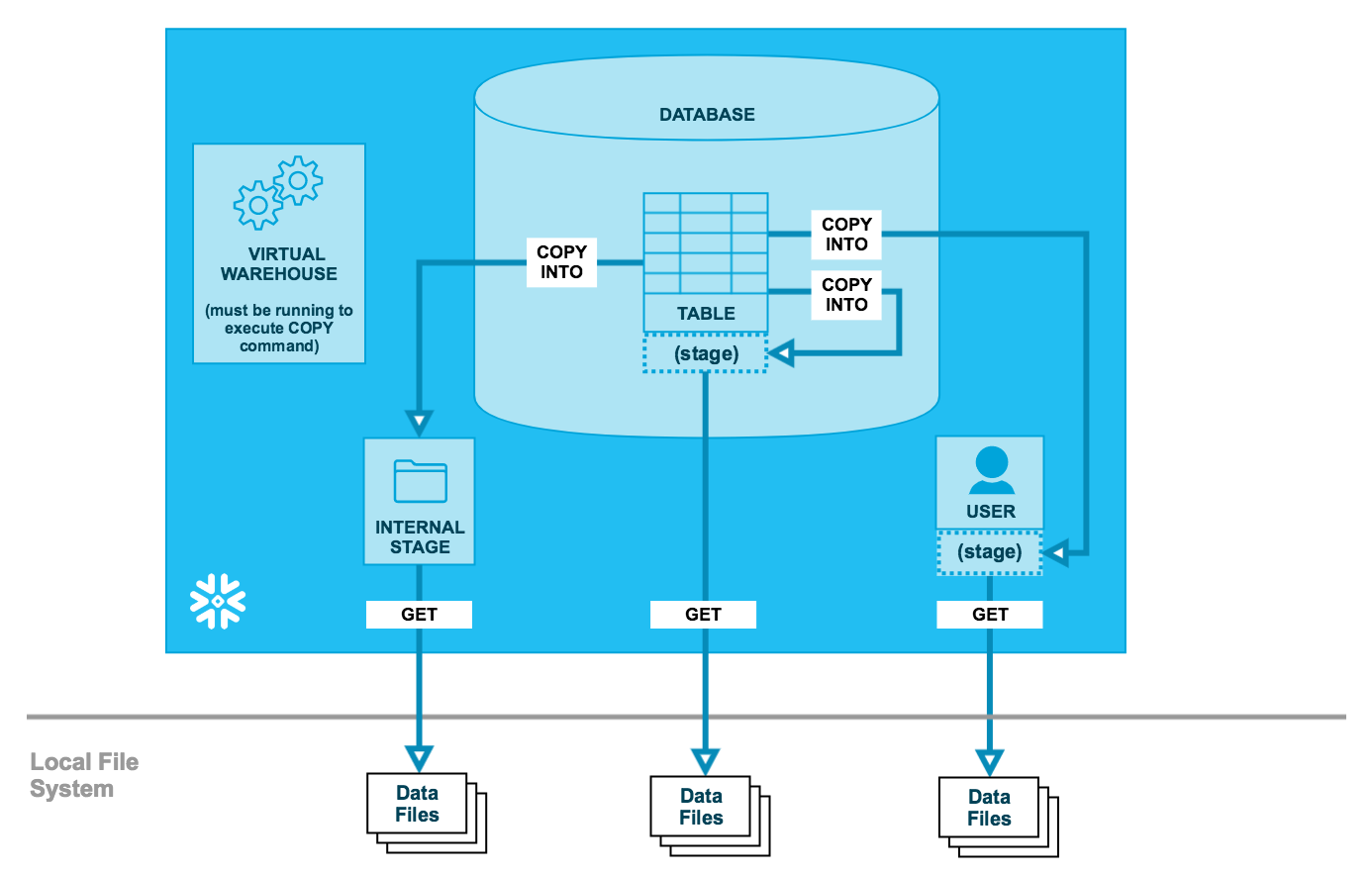
* **Unloading a single file**

1. MAX\_FILE\_SIZE default = **16MB**
2. Max is **5gb for AWS, GCP, and 256mb for Azure**

* **Unloading Relational table to JSON**

1. User **OBJECT\_OBSTRUCT** obstruct in COPY command to convert rows of relational tables to **VARIANT** columns then unload the data as per usual.

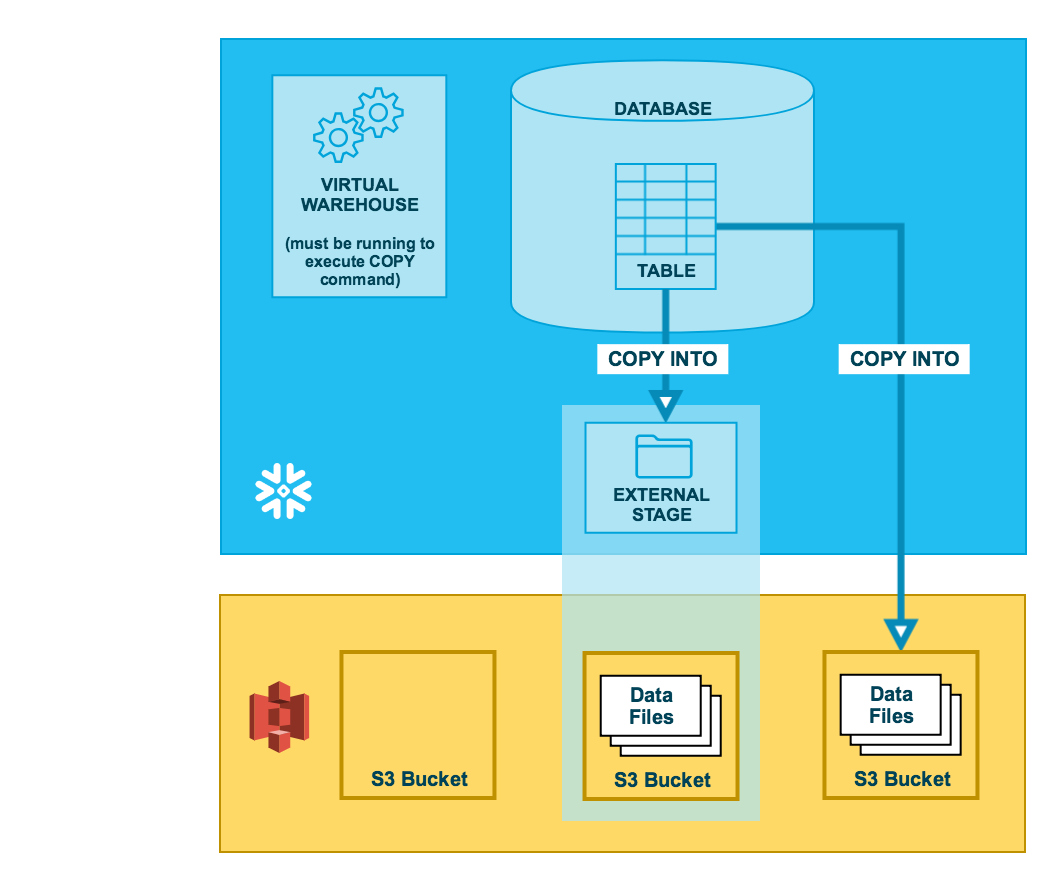
**Unloading into Snowflake Stage**

****

**Step1** : **COPY INTO** command to copy the data from into one or more files in a snowflake stage. Here we will require virtual warehousing.

**Step2**: **GET command** to download the data

**Unloading into Amazon S3**



Step1: use the **COPY INTO** command to copy data into **S3 bucket**

Step2: use the interface/tools provided by amazon to download the files from the S3 bucket.

**Virtual Warehouse**

1. **Cluster of resource**

* Provides CPU, memory and temp storage
* Is active on usage of **SELECT** and **DML (DELETE, INSERT, COPY INTO etc)**
* Can be stopped at any time and resized at any time(even while running)
* Warehouse sizes = **T-shirt sizes** (generally query performance scales linearly with vwh size)
* **When creating a VWH you can specify**:

1. **Auto-suspend (default: 10mins)** - suspends warehouse if active after certain time
2. **Auto-resume** - auto resumes warehouse whenever a statement that requires active vwh is required

* **Query Caching:**

1. VWH maintains cache of table data - improves query performance
2. The larger the vwh, larger the cache - Cache dropped when vwh suspended

1. **Choosing Virtual warehouse**

* SF recommends experimenting by running the same queries on different VWH
* Monitoring warehouse load- WebUI: In warehouse (view queued and running queries)

1. **Query processing and concurrency**

* When queries submitted, VWH calls and reserves resource
* Query is queued if VWH doesn't have enough resource

1. STATEMENT\_QUEUED\_TIMEOUT\_IN\_SECONDS
2. STATEMENT\_TIMEOUT\_IN\_SECONDS

Both can be used to control query processing and concurrency

- Size and complexity of query determines the concurrency (also # queries being executed )

1. **Multi-cluster warehouse;**

In concurrent queries if the nodes are occupied then next queries are queued,but in multi-cluster VW it will automatically detect queues and create new nodes.

* Up to **10 server** clusters
* **Auto-suspends and auto-resume** is of whole cluster not 1 server
* Can resize anytime
* Multi-cluster mode:

1. **Maximised** - associated **same min and max for #cluster**

* Sf starts all cluster at start would be good for expected workload
* **Higher max and min:** specified number of clusters start immediately.
* **Lower max and min:** specified number of clusters shut down when they finish executing statements and the auto-suspend period elapses.

1. **Auto-scale** -**different min and max for #cluster**

* To help control this, Sf provides scaling policy

1. **Standard:**
2. **Des**: Prevents queuing by favouring starting additional clusters over conserving credits.
3. **Cluster starts**: immediately when either a query is queued or the system detects that there’s one more query that's running.
4. **Cluster shuts down**: After **2 to 3 consecutive checks**(**1 minute interval**), which determine whether the load on the least-loaded cluster could be redistributed to the other cluster.
5. **Economy:**
6. **Des**: conserves credits by running clusters fully loaded, rather than starting additional clusters.
7. **Cluster starts**: only if the system estimates there’s enough query load to keep the cluster busy for at least 6 minutes**.**
8. **Cluster shuts down**: After **5 to 6 consecutive checks(1 minute intervals),** which determine whether the load on the least-loaded cluster could be redistributed to the other cluster.

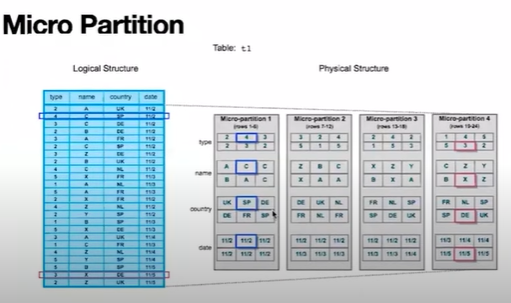
* **Higher max:** if *new\_max\_clusters > running \_clusters,* no changes until additional clusters are needed
* **Lower max:** if *new\_max\_clusters < running\_clusters,* excess clusters shut down when they finish executing statement and the **scaling policy** conditions are met
* **Higher min:** if *new\_min\_clusters > running\_clusters,* additional clusters immediately started to meet the minimum.
* **Lower min:** if *new\_min\_clusters < running\_clusters,* excess clusters shut down when they finish executing statements and the scaling policy condition.

**Features:** [**Snowflake Tutorial for Beginners | Snowflake Overview | Snowflake Database Architecture | HKR - YouTube**](https://www.youtube.com/watch?v=AKzxj95qBV8)

**Micro-Partitions**: divides the table

* All of SF tables are divided into micro partition
* Each micro-partitions are compressed columnar data
* Max size of **16mb compressed** and **50mb-500mb** **of uncompressed data**
* Data is immutable , can’t be changed
* Customers can query this without needing an active VWH like

1. Range of each columns
2. No. of distinct values
3. Count NULL



**Clustering Key should be consider when:**

* SF recommends the use of clustering key once your table grows really large
* Clustering keys are a subset of columns designed to **colocate data in the table in the same micro-partitions**.
* Table contains multiple terabytes (TB) of data
* When to use clustering:

1. Queries are running slower
2. Clustering ratio for the table is very low and clustering depth is very large.

**Tables**

Types of tables

1. **Temporary Tables**

* **Syntax: *CREATE TEMPORARY TABLE***
* Used to store data temporarily
* Non-permanent data and exits only with the session
* data is purged after session ends
* Not recoverable
* Contributes to overall storage cost.
* Belongs to DB and Schema- can have same name as another non-temp table within the same
* Default **1 day retention period**

1. **Transient Tables (DB and Schema)**

* **Syntax: *CREATE TRANSIENT TABLE***
* Persists/ exits until dropped
* Contributes to overall storage charges
* Have all functionality as a permanent table but with **1 day retention period**.
* no **Fail-safe mode** (no FS storage cost)
* Only for data that doesn't need to be protected

1. **Permanent Table**

* **Syntax: *CREATE TABLE***
* **Have 7 days fail safe**
* Default 1 day time travel **(up to 90 days for >=Enterprise edition**)

1. **External Table**

* Allows access to data stored in external stage through **COPY INTO** command
* Access files stored in external stage areas such as amazon, s3 or gcp bucket or azure blob storage.
* Are **read-only tables** , no DML operations can be performed
* Can use external tables for query and join operations
* **Views and materialised views can be created** against external tables.
* We can **analyse the data without storing it in SF.**

**Metadata of external tables: it can store information for:**

1. **VALUE:** a **VARIANT** type column that represents a single row in the external file.

\*variant: it is a column which stores all the files.

1. **METADATA$FILENAME:** A pseudocolumn that identifies the name of each staged data file included in the external table, including its path in the stage.
2. **METADATA$FILE\_ROW\_NUMBER:** a pseudocolumn that shows the row number for each record in a staged data file.

**Creating Tables**

* SF does not enforce any constraints (primary key, unique key, foreign key) beside the **NOT NULL constraint**
* For creating external tables,

1. Create file format object
2. Create stage object referring to cloud storage location
3. Create external stage

**Views**

Types of views

1. **Non-materialized view**

* Referred to as views
* Named definition of query
* Result are not shared

1. **Materialised view**

* Result are stored
* Faster performance than views
* Contributes towards storage cost
* Can query only **single table, Joins, self-joins** are not supported

1. **Secure View**

* Both **non-materialised and materialised view** can be defined as **secure view**
* Improves **data privacy**- hides view definition and details from **unauthorised viewers**
* Performance is impacted

**Semi Structured Data support**

**Access Control:**

1. **Network Policy:**

* Allows access based on IP whitelist or restrictions to IP blacklist
* Apply through SQL or WebUI
* Only **ACCOUNTADMIN or SECURITYADMIN** can **modify, drop or create** these

1. **Private Link**

* AWS & Azure

1. **MFA(multi factor authentication)**

* Powered by Duo system and enrolled on all accounts, customers need to enable it
* Recommended enabling MFA on **ACCOUNTADMIN**

1. **Federated Auth and SSO**

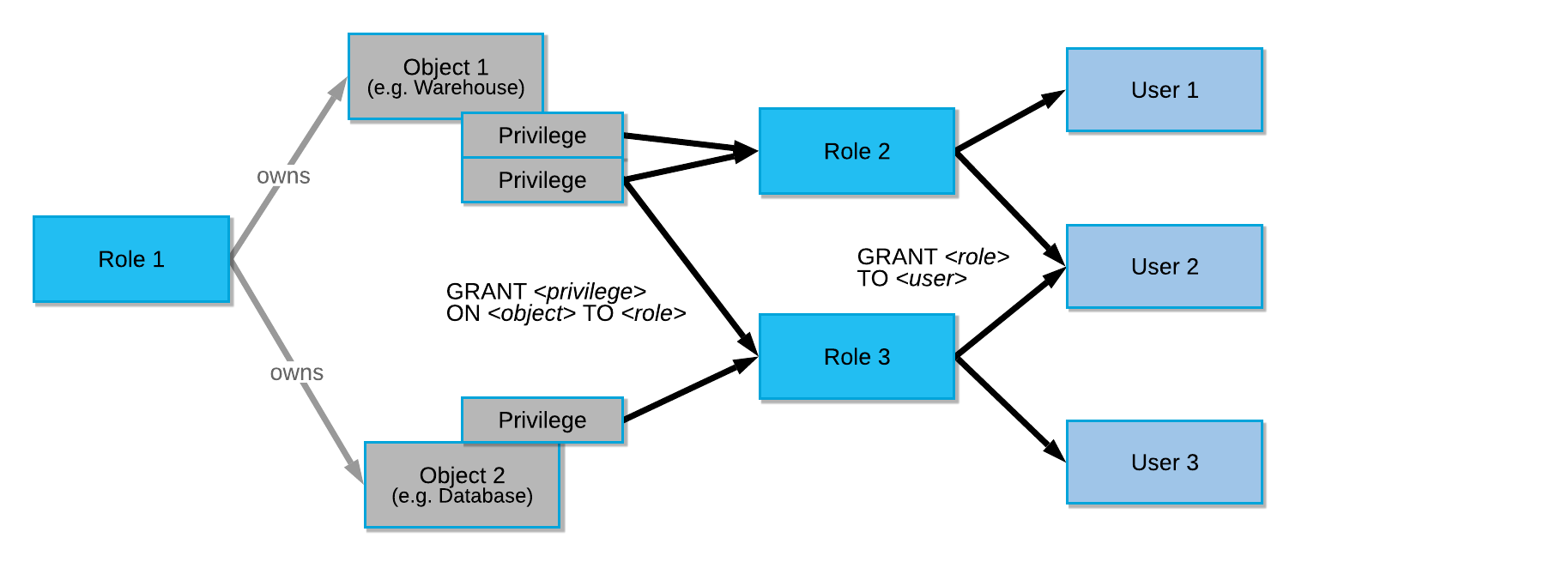
* User authentication through external **SAML 2.0-compliant identity provider (IdP)**
* SAML 2.0 (**Security Assertion Markup Language**) is an open standard created to provide cross-domain **single sign-on (SSO)**. In other words, it allows a user to **authenticate in a system and gain access to another system by providing proof of their authentication.**
* For SSO, a token is passed to the application that the user is trying to login to authenticate which in turns will open access when verified.

1. **Access control Models**

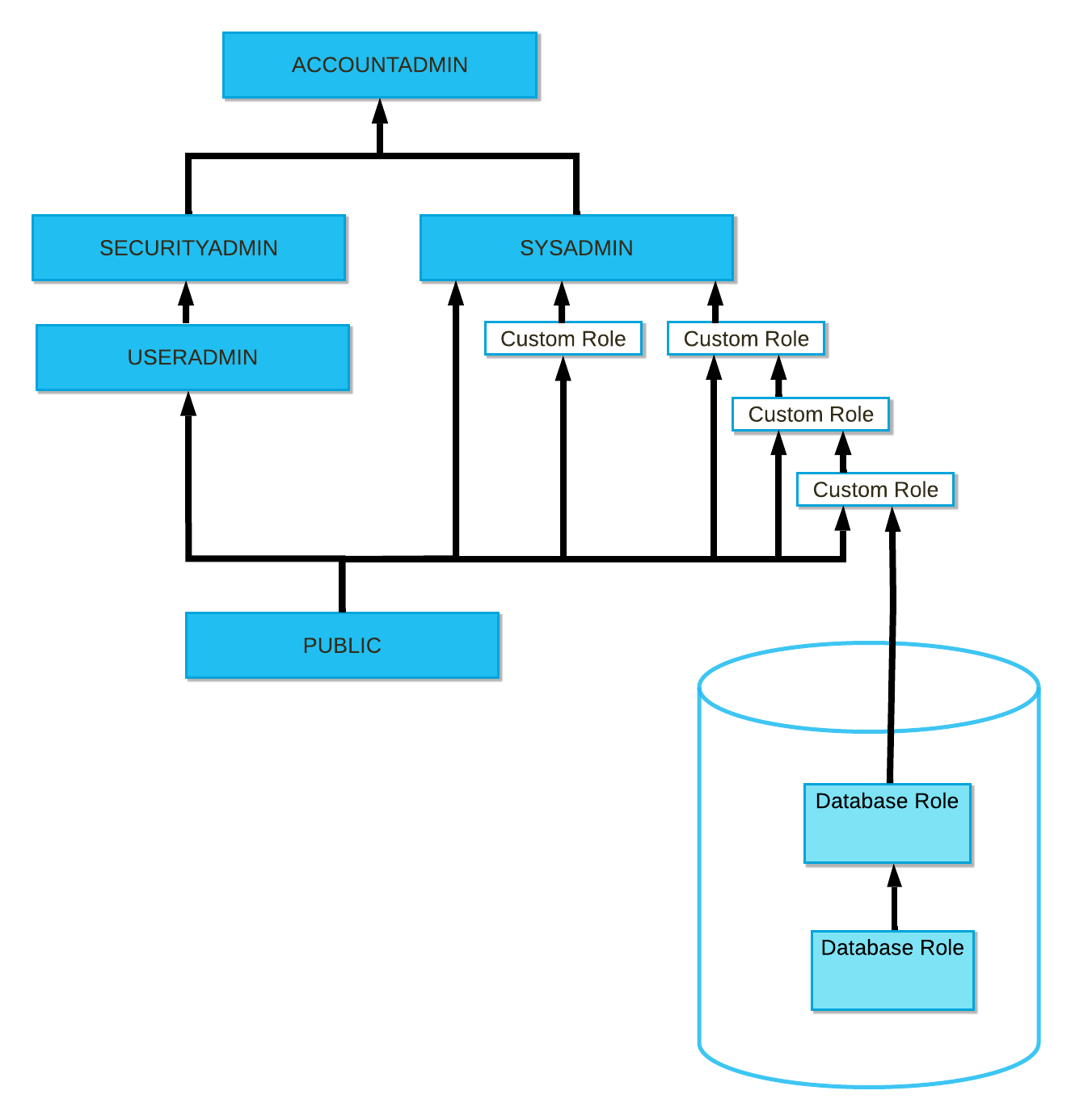
* SF approach:

1. **Discretionary Access Control (DAC**): all objects have an owner, owner can grant access to their objects.
2. **Role based access control (RBAC):** privileges are assigned to roles, then roles to users.

* Roles are entities that contain granted privileges (**level of access to object**) which are **system defined roles**

****

1. **System Defined Roles**

****

1. **ACCOUNTADMIN:**

* encapsulates **SECURITYADMIN and SYSTEMADMIN**

1. **SECURITYADMIN:**

* creates, modify and drops user, roles, networks, monitor or grants

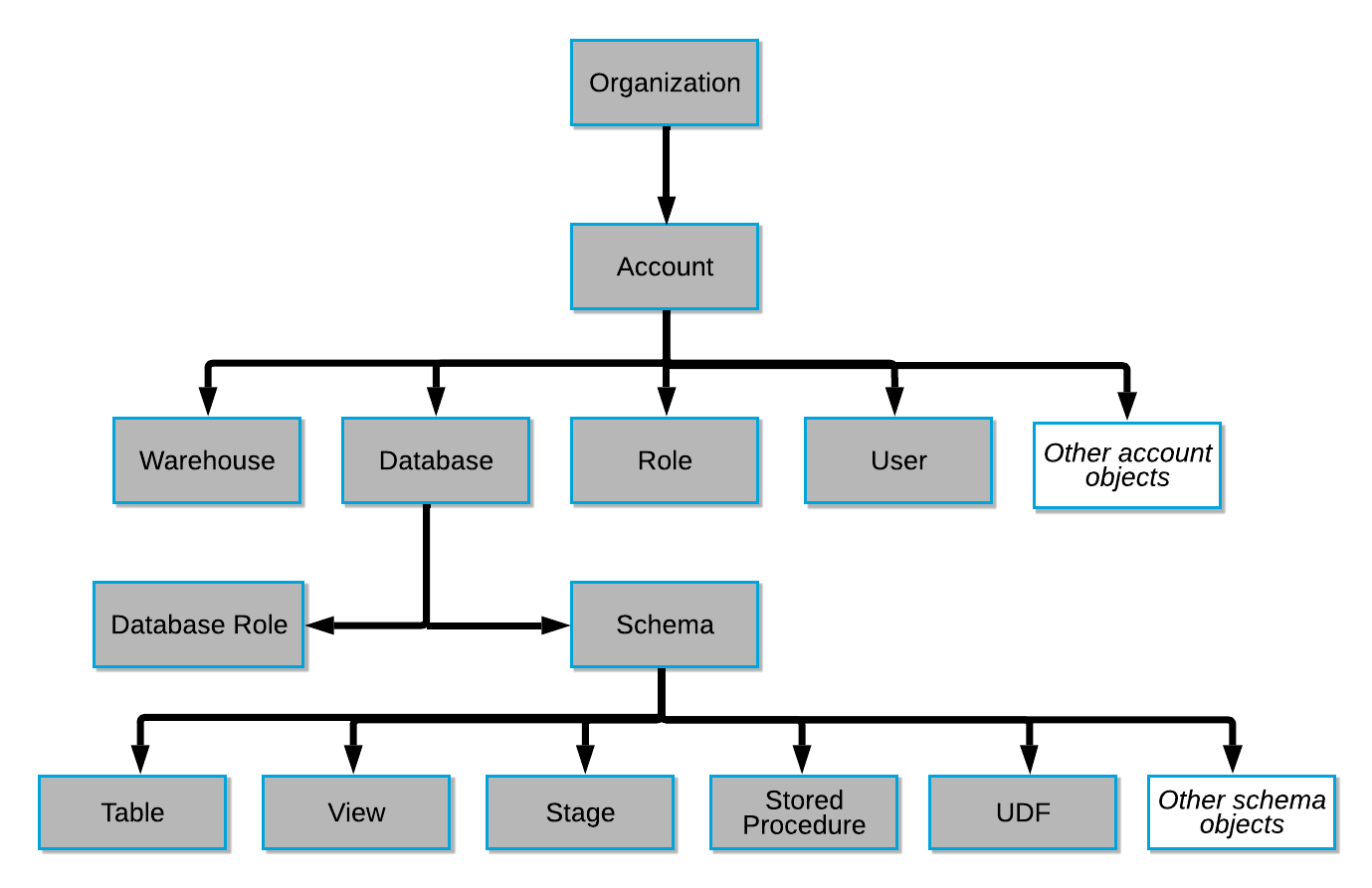
1. **SYSADMIN:**

* has privileges to create VWH, and db and its objects
* Recommended assigning all custom roles to **SYSADMIN** so it has control over all objects created

1. **PUBLIC:**

* Automatically grants to all users
* Can own secured object
* Used where explicit access control is not required.

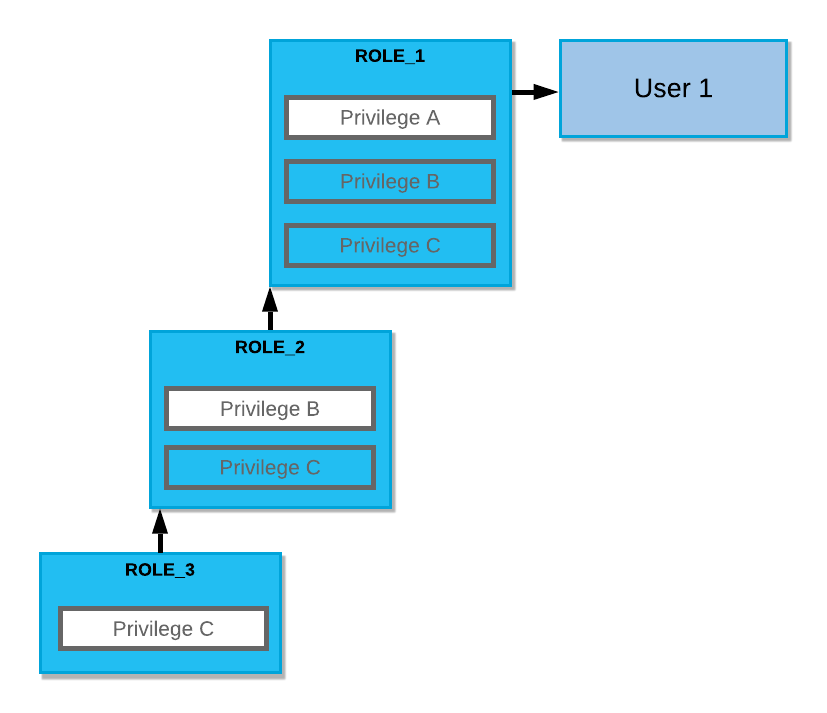
1. **ORGADMIN:** organisation administration, organisation level operation



1. **CUSTOM ROLES:**

* **Created by** **SECURITYADMIN**
* **Assigned by SYSADMIN**
* Create custom roles with least privilege and role them up
* Custom roles are normally created by **SECURITYADMIN and then assigned to SYSADMIN**
* **In below scenario**:

1. Role 2 inherits Privilege C. c. User 1 has all three privileges.
2. Role 1 inherits Privileges B and C

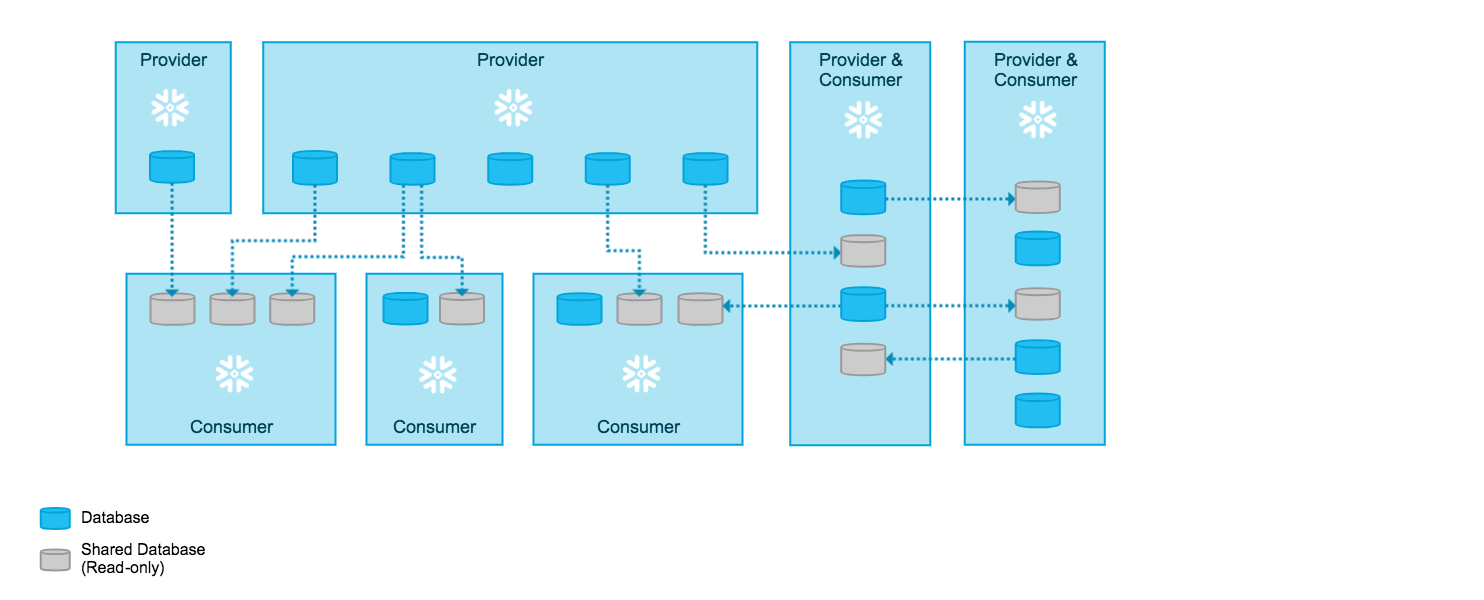
****

**Data Sharing:**

Snowflake database objects can be shared are:

1. Tables
2. External Tables
3. Secure views
4. Secure materialised views
5. Secure UDFs

All objects shared are **read-only**, no actual data is copied or transferred, all sharing is accomplished through a unique **service layer and metadata store.**

****

- Provider - who is sharing the data by creating a shared object.

- Consumer - who is consuming or using the shared data.

1. No storage cost
2. Compute resource cost only
3. Only **ACCOUNTADMIN** can create sharing
4. The sharer is the **provider** while the user of the shared data is the **customer**
5. **Secure data** sharing enables **account-to-account sharing of databases, tables and views. (no actual data is copied or transferred )**
6. Sharing accomplished using SF **service layer and metadata store**
7. User creates **share of DB** then grants **object level access(read only DB is created on consumer side)**
8. Type of share includes **Share and Reader**
9. No limit of number of shares or account buts **1 database per share**

- Data can be shared with snowflake users and non-snowflake users as well.

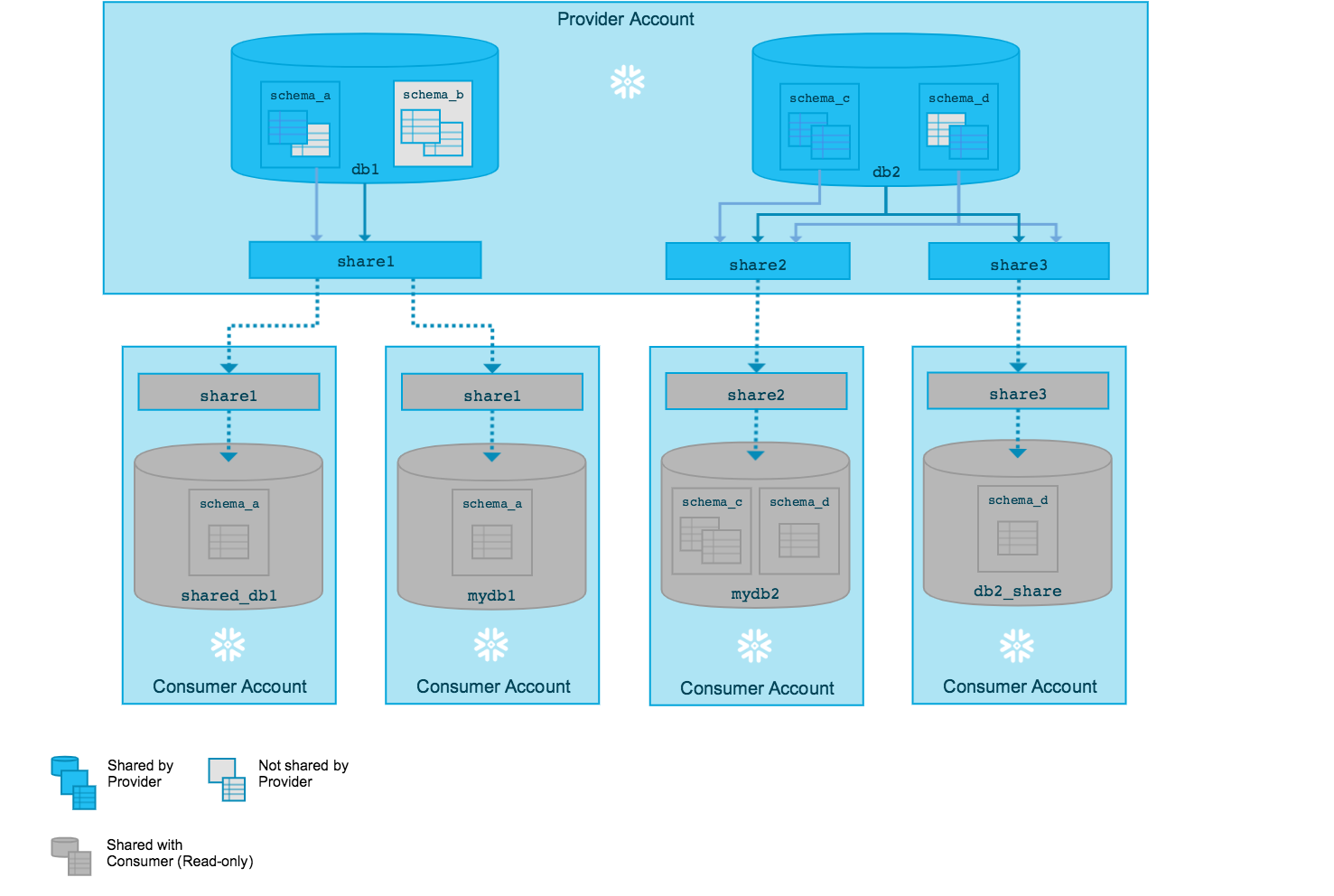
- For **non -snowflake users, we have to create a** **reader account** and share the data.

**Share:**

named database objects that contain all of the information required to share a database like object grant and consumer account details.

* Snowflake database objects can be shared are:

1. Tables
2. External Tables
3. Secure views
4. Secure materialised views
5. Secure UDFs



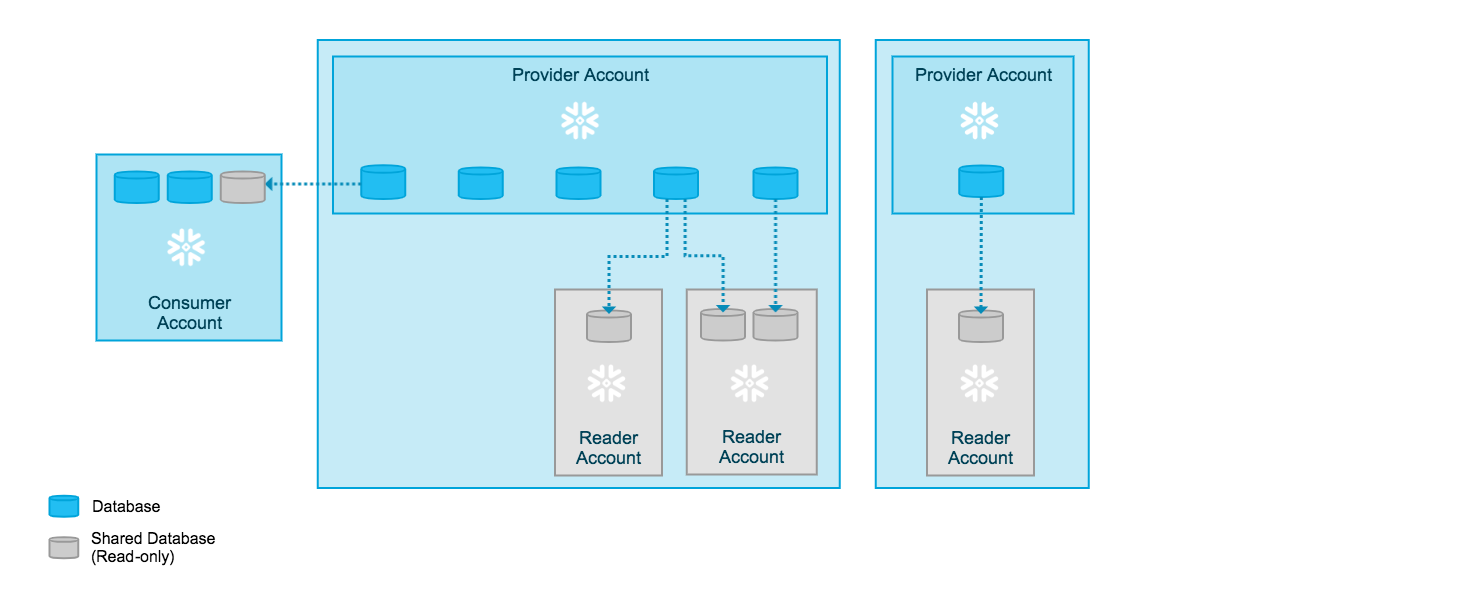
* **Between SF account:**
* Each share consists of

1. Privilege that **grants db access** to the database and schema containing the objects to share.
2. Privilege that **grant access to specific object (tables, views)** in the database
3. Consumer accounts with which db and object are shared.

* Can perform DML operations

**Reader Account:**  for non-snowflake user

For third party access



* Share data with **user who isn’t on SF**
* Objects can **only be read** not modified
* **DML** tasks not allowed
* Consumer can only consume the data

**Security:**

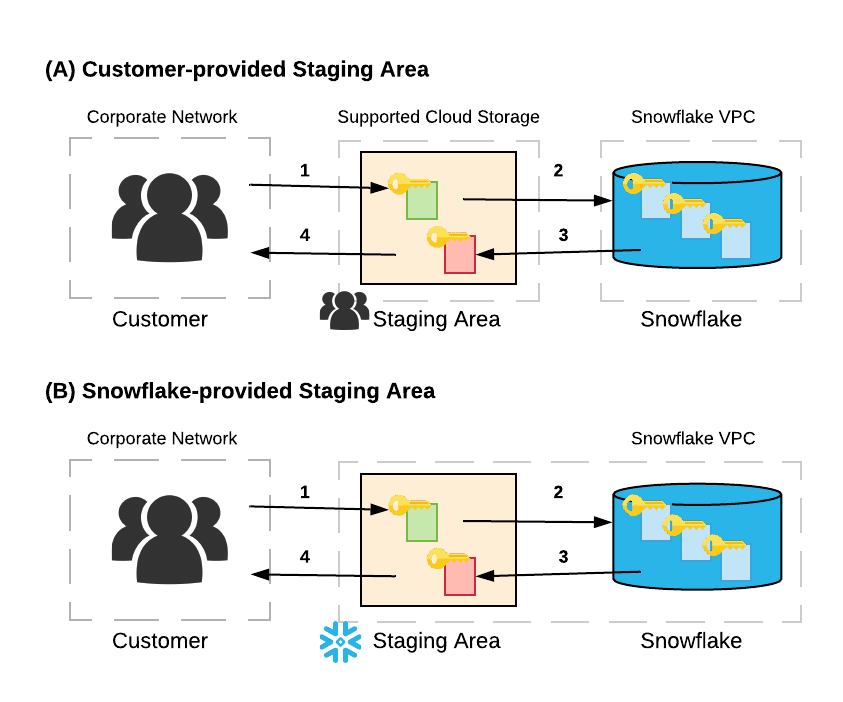
1. **End-to end encryption**

* SF **encrypts all data by default**, no additional cost
* No one other than the customer or runtime components can **read the data**
* Data is always protected as it is in an **encrypted state**
* Customer provides encrypted data to the external staging area (S3, etc)

a. Provide SF with the **encryption master key** when creating the **Named Stage**

b. **Client side encryption**

* Customer provides **unencrypted data to SF internal staging area**, SF will **automatically encrypt the data.**



From above image:

Sf supports both internal and external stages for data files, data is end-to-end encryption

1. A user uploads one or more data files to a stage.

- Image A user may optionally encrypt the data files

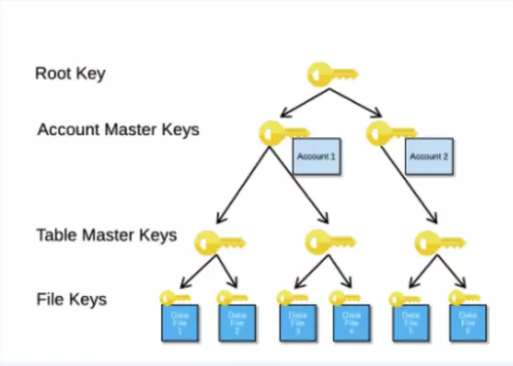
- Image B- automatically encrypted by the snowflake client on the user’s local machine

1. The user loads the data from stage into a table
2. The user can upload query results into an external or internal stage
3. The user downloads data files from the stage and decrypts the data on the client side .
4. **Key Rotation Snowflake Managed keys:**

* SF encrypts all data by default and keys are rotated regularly
* Uses hierarchy key model for its **Encryption key management** -comprises of:

1. **Root key**
2. **Account Master Key (auto-rotate if> 30 days old)**
3. **Table Master Key (auto-rotate if> 30 days old)**
4. **File Keys**

Hierarchical key model, all snowflake-managed keys are automatically rotated by snowflake when they are **more than 30 days.**

****

1. **Tri-secret secure and customer-managed keys (Business Critical Edition)**

* Combines the customer key with snowflake’s maintained key

1. Create composite master key then use it to encrypts all data in the account
2. If customer key or snowflake key is revoked, data cannot be decrypted

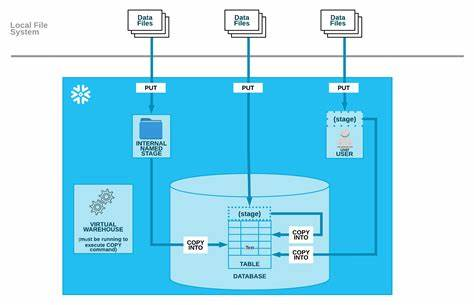
In **security** it has two editions:

| **Enterprise Edition** | **Business Critical Edition** |
| --- | --- |
| Configure session policies | Private communication between the VPC and SF |
| Periodic rekeying of encrypted data | Private communication to internal stage |
|  | Support for encrypting data using customer-managed keys |
|  | HIPAA, PCI DSS, HITRUST CSF, FredRAMP Moderate, IRAP protected compliance |

**Staging :**

* Is a space where we can load and unload the file
* Space provided by snowflake to load the data files before loading it into table
* Internal and external staging; generate an copy of table
* Here each **table and user are allocated to an internal named stage for staging data file to be loaded**
* In **Loading the Data:**
* Must specify internal stage in **PUT command** when **uploading file** to SF
* Must specify the same Stage in **COPY INTO** when **loading data** into a table
* In **Unloadin Data**
* Must specify an internal stage in **GET command** when **unloading file** from SF
* Must specify the same Stage in **COPY INTO** when **unloading data** from a table to a staged file.

1. **Internal staging**



1. **User Stages (@~)**

* Accessed by single user but need copying to multiple tables
* Can’t be altered or dropped
* Can't set file format, need to specify in COPY command to table

1. **Table stages (@%)**

* Accessed by multiple users and copied to **single table**
* Can’t be altered or dropped
* Can’t set file format, need to specify in COPY command to table
* No transformation while loading

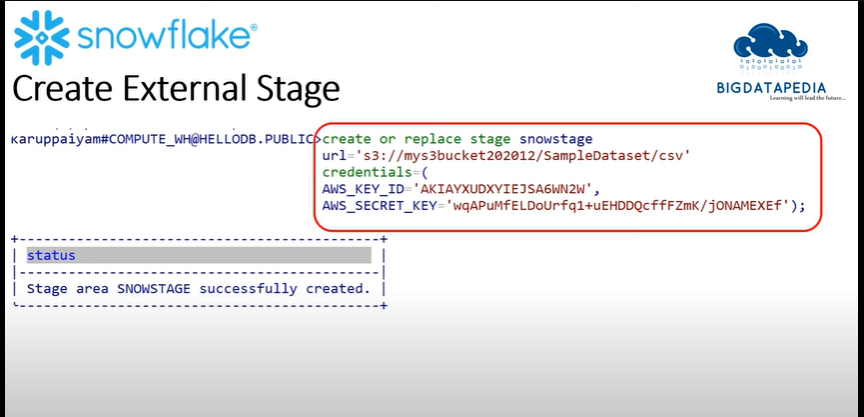
1. **Internal Named stages (@)**

* A database object
* Can load data into any tables (Needs user with privilege)
* Ownership of stage can be transferred

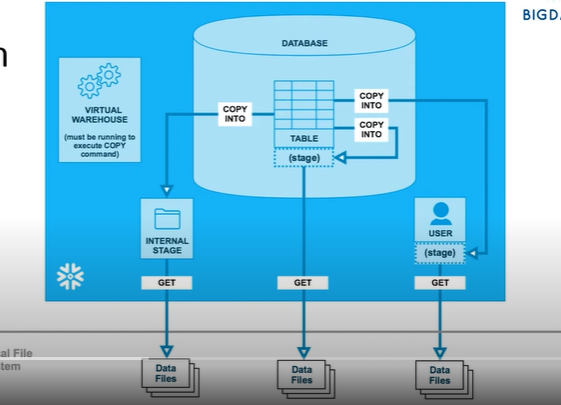
1. **External Staging**
2. **Aws**

* SF uses S3 gateway endpoint. If region bucket = SF region - no route through public internet.
* Securely access S3

1. Configure storage integration object - delegate authentication responsibility for external cloud storage to SF identity and IAM
2. Configure AWS IAM role -> allow S3 bucket access
3. Configure IAM user, provide key and secret key



1. Azure
2. GCP



**Time Travel:**

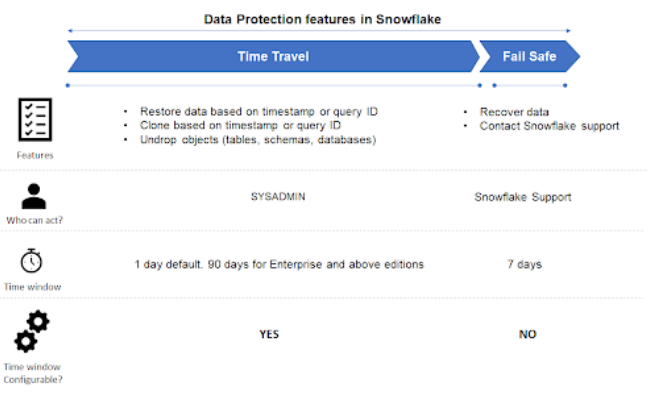
* can access the deleted or prev data at any point within a defined period (historical data: data has been changed or deleted)
* Ex: *drop table data*, now to receive *undrop table data*
* We can also set rendition data, **default is 90 days**
* **(Standard edition: 0-1 days and enterprise edition: 0-90 days)**
* Can be useful if data is deleted, dropped or updated
* Allow db to query, clone or restore data to tables, schema or db for up to 90 days
* Can use this to access a snapshot of data at a point in time.

Time travel options:

1. TIMESTAMP (<timestamp>) → specify an exact date and time to use for time travel.
2. OFFSET (<time\_difference>) → specifies the difference in seconds from the current time to use for time travel
3. STATEMENT (<id>) → specifies the query ID of a statement to use as the reference point for time travel
4. STREAM (<name>) → specifies the identifier (i.e. name) for an existing stream on the queried table or view.

**Fail Safe**

* Allows disaster recovery of historical data -only accessible by snowflake
* **0-7 days**, post the end of the time travel period.
* \* not provided as a means for accessing historical data after the time travel retention period has ended.

****

**Zero copy clone**: can clone the table but the the data in the table

Ex:

*Demo demo1* → DB

Insert *demo 1* → DB & *DB1 (micro partition)*

Ex2: if we have 10 micro partition and want to change on 10th partition

*Demo Demo1 →* DB [10]

Insert *demo 1* → 10

Now *Demo Demo1 → DB (9)*

For 10th *Demo →* DB(10)

*Demo1 →* DB(10)

* Allows customers to **clone their table**
* SF references the original data (micro-partition) hence **zero copy (no additional cost)**
* When a change is made to the cloned table (**new data added, deleted or updated**) then a new micro-partition is created (incurs storage cost).
* **Clone objects do not inherit the source’s granted privileges**
* Its a quick way to **snapshot any table, schema, or database**
* Cloned object **does not inherit the sources privileges**

1. **Only child object retain privileges of the source**
2. **If DB is cloned -> schema and table retain;**
3. **If schema is cloned -> table retains.**

* Objects for clone:

1. **Database objects**
2. **External named stages**
3. **Snowpipes**
4. **Sequences**
5. **Streams**
6. **Tasks**

* A snapshot of data present in the source object is taken when the clone is created, and is made available to the cloned object.
* The cloned object is writable, and is independent of the clone source.
* Any **DML updates to the cloned table will not affect the source object.**
* Saves storage cost.
* **Tables are cloned** but **internal named stages are** **Not cloned**

**Cache:**

* SF caches different data to improve query performance and assist in reducing cost

1. **Metadata Cache**: cloud services layer

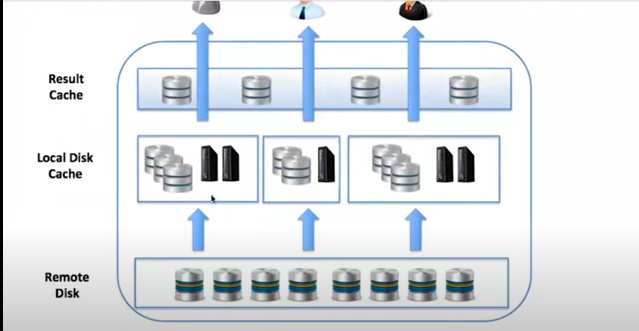
* Improves compile time for queries against commonly used tables

1. **Result Cache** - cloud services layer

* Holds the query results
* Works until underlying data has not changed. (**query must be same**)
* If customers run the **exact same within 24 hrs**, result cache is used and no warehouse is required to be active

1. **Local Disk cache** or **warehouse cache** -virtual warehouse layer

* Caches the data used by the sql query in its local **SSD and memory**
* Improves query performance if the same data was used
* Cache is deleted **if the warehouse is suspended**.

****

**Caching:**

* Cache is temporary storage location that stores copies of files or data,
* 2 types of caches:

1. Query result cache
2. Local disk cache

**File Format:**

Is a named database object that can be used to simplify the processing of accessing the staged data and streamlines loading data into and unloading data out of the database table.

**Format file options:**

**A. Types**

1. .csv (comma separated value)
2. .psv (pipe separated value)
3. .tsv (tab separated value)
4. .json
5. .xml
6. Parquet
7. Avro
8. Orc

**B. Compression**:

1. AUTO
2. GZIP
3. BZ2
4. BROTLI
5. ZSTD
6. DEFLATE
7. RAW\_DEFLATE

Use → data loading, data unloading and external tables

Default → auto

**C. Record Delimiter=** ‘\n’, ‘\r’: for new line character

**D.** **Field Delimiter**= ‘,’, ‘|’, ‘\t’: here delimiter is different for different file format like ‘/t’, '|', '~|~'

**E. Field Extension**= ‘.csv’

**F. skip\_header** = ‘1’ | ‘none’: it will skip header ie. it will not consider that header into data.

**G. SKIP\_BLANK\_LINES** = TRUE | FALSE - - - - boolean that specifies to skip blank lines encountered in the data files, otherwise blank lines produce an end-of-record error

**H. FIELD\_OPTIONALLY\_ENCLOSED\_BY** = ‘none’ | ‘\042’: look for the problem of double quotes.

**I. ERROR\_ON\_COLUMN\_COUNT\_MISMATCH** = TRUE | FALSE, it will match the column according to the data.

**Creating a file format (example):**

create or replace file format emp.emp.coding\_ff

type = 'csv'

compression= 'auto'

field\_delimiter = '\t'

record\_delimiter = '\n'

field\_optionally\_enclosed\_by = '\042'

skip\_header = 1;

* **Staging data files from a local file system**

**Linux or macOS:**

* put file:///data/data.csv@~/staged; → **for user stage**
* put file:///data/data.csv@%mytable; → **for table stage**
* put file:///data/data.csv@my\_staged; → **for named table stage**

**Windows:**

* put file://c:\data\data.csv@~/staged; → **for user stage**
* put file://c:\data\data.csv@%mytable; → **for table stage**
* put file://c:\data\data.csv@my\_stage; → **for named table stage**

**Listing Files:**

* list @~; → **for user stage**
* list @%mytable; → **for table stage**
* list @my\_stage; → **for named table stage**
* **Copying data from internal stage**

**User stage:**

* To load all files prefixed with staged in your user stage,

*copy into mytable from @~/staged*

*file\_format = (type = csv field\_delimiter = ‘|’ skip header = 1);*

**Table stage:**

* To load all files in the stage for the customer table,

*copy into mytable from @%customer*

*file\_format = (format\_name = mys\_csv\_format);*

**Named stage:**

* To load all files from the my\_stage named stage

*copy into mytable from @my\_stage;*

**Data Sampling**

Selecting a part of data or subset of records from a table for:

1. Query building and testing
2. Data analysis or understanding

Useful in dev environment where we use small warehouse and occupy less storage

-can sample a fraction or % of rows

-can sample a fixed number of rows

Sampling Methods

1. Bernoulli or row: default it will be selected

* where the probability of including a row is p/100
* We can say this gives almost p% of data
* Good for smaller tables

1. System or Block-

* Where the probability of including a block is p/100
* -we can say this gives data from p% of blocks
* Good for larger tables

Example: if a table contains 4 million records stored in 600 micro partitions and I need 10% of data for my testing.

Bernoulli or row - it will fetch 10% of 4 mil = 4 lakh rows

System or block - it will fetch data from 10% of 600 micro partition = 60 micro partitions

**--some example for data sampling**

1. select \* from table\_sample row(10); --return a sample with 10% of rows

2. select \* from tablename tablesample block(20) --return a sample with data from 20% block

3. select \* from tablename sample system (10) seed (111);

--return a sample with data from 10% of blocks and same data set if use seed11 next time

4. select \* from tablename tablesample (100) --return an entire table, including all rows into the sample

5. select \* from tablename tablesample row (0); -- return an empty sample

6. select \* from tablename sample (10 rows); --return a fixed size sample of 10 rows

**Data Masking**

Data masking is a security feature that allows organisations to mask sensitive data in their database tables, views, and query results in real time.

**UDF**

User defined function

allows you to perform operations that are not available through the built-in, system defined function.

SF UDF's can return scalar(just a value or a string) and tabular results.

2 types of UDF available in SF

a. SCALAR

b. TABULAR

**SCALAR:** returns output for each input we are passing (varchar or any string or value).

**TABULAR:** returns a table (can return zero, one or multiple rows)

SF supports 4 languages for writing UDFs

1. **SQL**
2. **JavaScript**
3. **Java**
4. **Python**

Snowflake supports UDF overloading: means support functions with the same name but different params.

– **Proc\_Calculate\_Area()** is different from **Proc\_Calculate\_Area(Radius FLOAT)**

**– Proc\_Calculate\_Area (Radius FLOAT) is different from**

**– Proc\_Calculate\_Area (Length number, width number)**

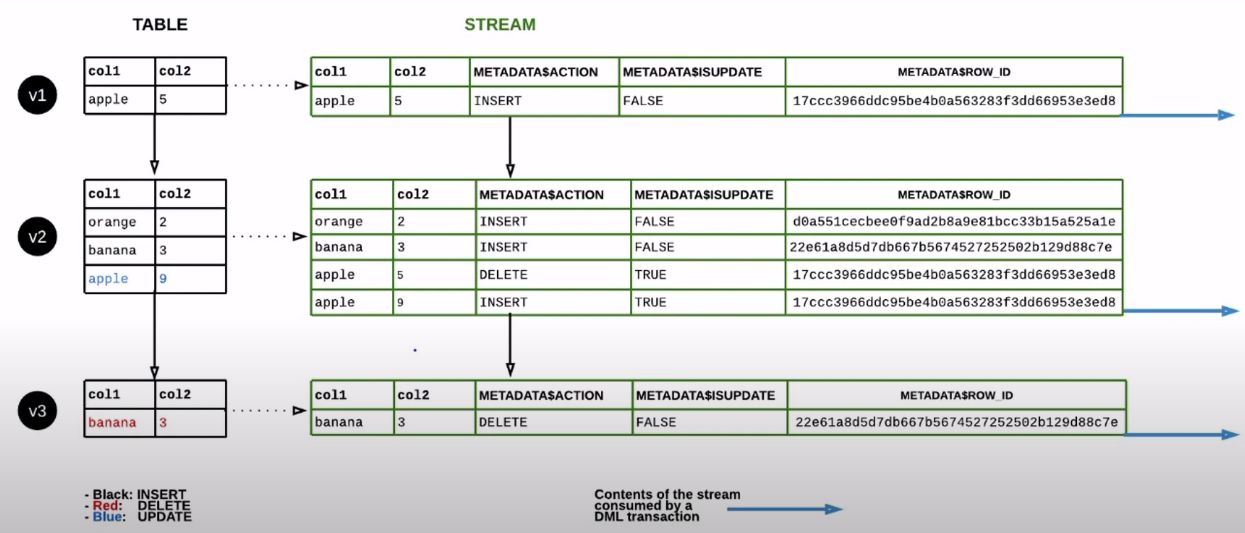
**Stream**

A stream object records DML changes made to tables including inserts, updates, and deletes

* It **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 doesnt store changed data
* Once 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

Streams maintain 3 metadata fields.

1. **METADATA$ACTION:** indicates the DML operation (insert, delete) recorded
2. **METADATA$ISUPDATE:** indicate whether the operation was part of an **UPDATE statement**. Updates rows in the source object are represented as pair or 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.



Use merge statement for consuming the changes from stream and applying the same on target table

//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’;

-**Types of Stream**

1. **Standard Streams**: **track all DML changes to the source objec**t, including inserts, updates and deletes (also table truncates)

Syntax: **CREATE OR REPLACE STREAM my\_stream ON TABLE my\_table;**

1. **Append-only Streams**: **an append-only stream tracks row inserts only**. Update and delete operations (including table truncate) are not recorded

Syntax: **CREATE OR REPLACE STREAM my\_stream ON TABLE my\_table APPEND\_ONLY = TRUE;**

1. **Insert-only Streams**: **supported for external tables only**. An insert only stream tracks row inserts only. They do not record delete operations

Syntax: **CREATE OR REPLACE STREAM my\_stream ON EXTERNAL TABLE my\_table INSERT\_ONLY = TRUE;**

* **Standard and Append only streams** can be on- Permanent, Transient and Temporary table
* **Insert only on** Ext table

**APPEND ONLY STREAM:** just capture the data which is being inserted, update or deleting will not be updated

**Task**

* We use **tasks for scheduling** in SF
* We can schedule   
  - Sql queries  
  - Stored procedures
* Its combined with streams for implementing the continuous **change data captures(CDC)**
* We can maintain a DAG of tasks to keep the dependencies between tasks.
* Task require compute resources to execute SQL code

- snowflake managed compute resources(serverless)

- user managed (virtual warehouses)

- **will take time in minutes only.**

**- we can set task schedule time till 11520 min that is 8 days**

Syntax:

create or replace task table\_name

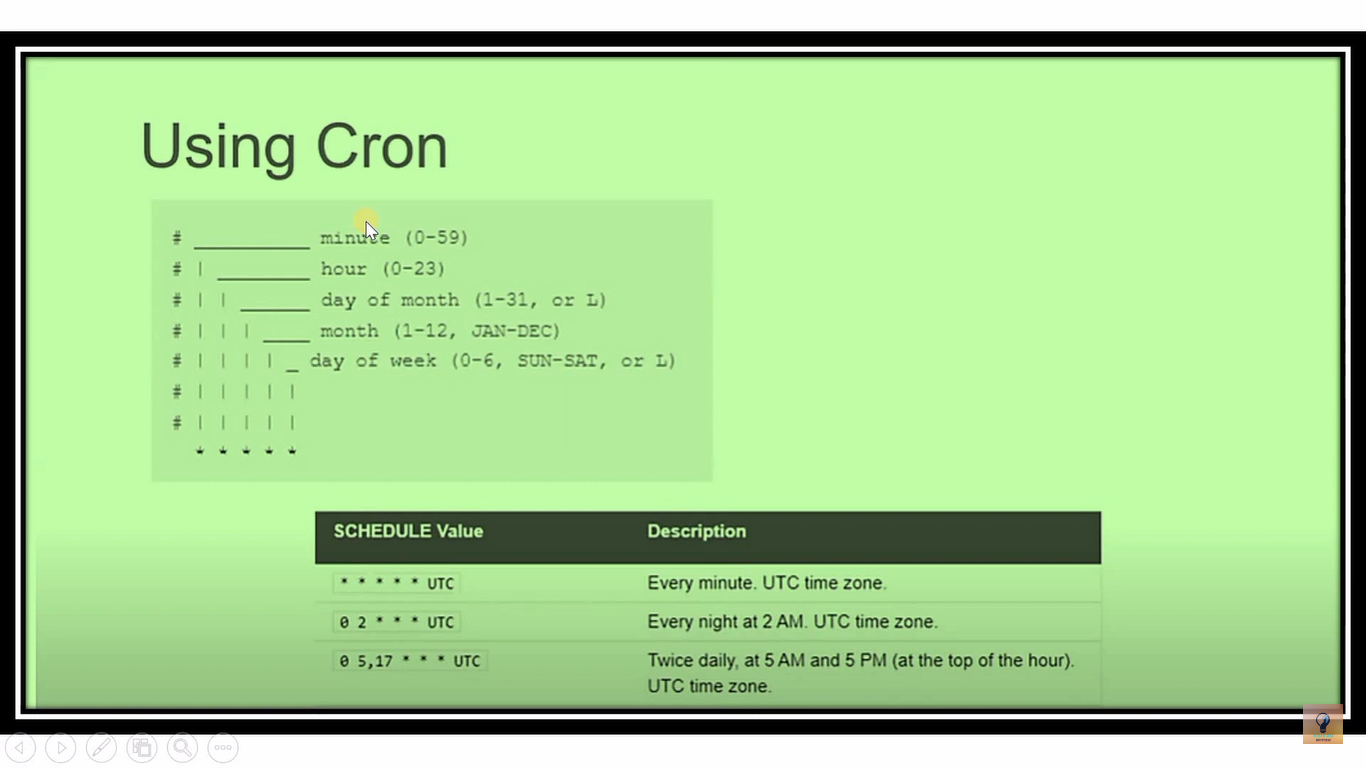
warehouse = warehouse\_name

schedule = ‘time or cons’

AS

Insert into customers(create\_date) values(current\_timestamp);

\*\*Note: **what are cons?**

****

Example1: ‘Using CRON 30 9 1 12 6 UTC’

–it will be 30:min 9:hr 1:day 12:month 6 week(can be 0-6, sun-sat, or L)

Example2: 1) \* \* \* \* \* UTC: every minute. UTC time zone

2) 0 2 \* \* \* UTC: every night at 2AM. UTC time zone.

3) 0 5, 17 \* \* \* UTC: twice daily at 5 AM an 5 PM (at the top of the hour), UTC time zone

(,17 is 24hr format)

**DAG of Task**

**Directed Acyclic Graph**

* To maintain dependencies between tasks
* A root task followed by child tasks
* Just schedule root task, child tasks will be executed in order

Example:

Create or replace task task\_a

Warehouse = warehouse\_name

Schedule = ‘using cron 30 9 \* \* \* UTC’

As ‘SQL Query 1’;

Create or replace task task\_b

Warehouse = warehouse\_name

After task\_a

As ‘SQL Query 2’;

Create or replace task task\_c

Warehouse = warehouse\_name

After task\_a

As ‘SQL Query 3’;

Create or replace task task\_d

Warehouse = warehouse\_name

As ‘SQL Query 4’;

Alter task task\_d add after task\_b;

Alter task task\_d add after task\_c;

To get corn value we can use some links:

[Crontab.guru - The cron schedule expression editor](https://crontab.guru/)

[Cron expression generator by Cronhub](https://crontab.cronhub.io/)

**Stored Procedures**

Stored procedure is created once and can be executed many times.

* Create with CREATE PROCEDURE
* Executed with CALL command
* Languages it can be written are:

- java (using snowspark or udf )

- javascript

- python (using snowspark or udf)

- scala (using snowspark)

- writing stored procedure in snowflake scripting

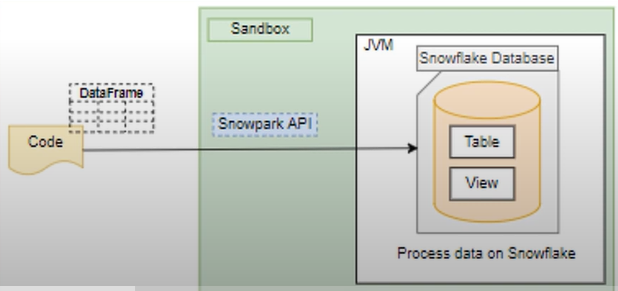
From a stored procedure, we can return a single value or tabular data.

Benefits of stored procedures:

* Procedural logic (branching and looping), which SQL does not support (eg: while loop, for loop, etc, if else statements, etc)
* Error handling- try-catch blocks, raise\_error function
* Dynamically creating a sql statement and executing it.

**Snowspark**

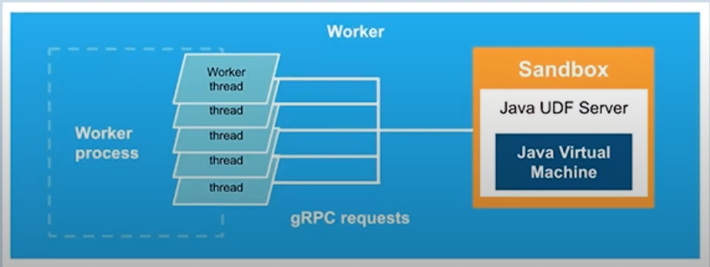
* API for querying and processing data which can interact with scala java and python.
* The api is automatically deployed to a sandbox that is equivalent to a docker container.
* Client application using the API library can process data snowflake without moving data to where the application code is run



**Architecture:**

* Snowpark API is deployed to sandboxes that are similar to docker containers this means:

1. Sandboxes(env) are more locked down
2. Sandboxes avoid modifications which otherwise might be possible with a VM
3. Snowpark API requests process outside of the sandbox



Architecture- **API Request Evaluation**

* Strict

1. Evaluates on every use
2. Evaluate immediately

* Lazy

1. Evaluates on first use
2. Evaluates when action is performed, not when object are constructed at run -time

* Applies to

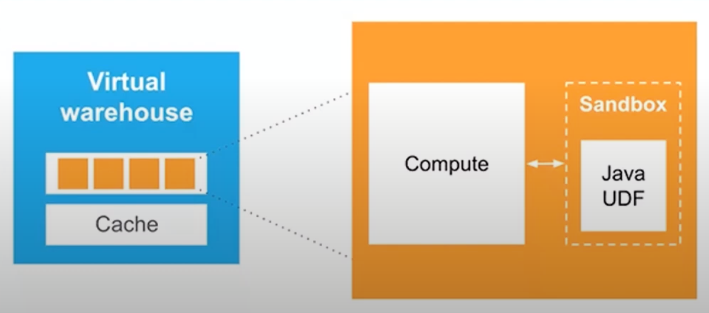
1. Variable assignment
2. DataFrame methods

Architecture- **Sandbox- Restriction**

* Cannot access the network from your application code
* Cannot access files outside of snowflake/ tmp location and size is limited
* Cannot spawn additional processes
* Cannot access a native library with code compiled for specific architecture
* There is a limited amount of time allowed to process each row

Architecture- **Sizing and compute**

* Snowpark API request operates and compute on Virtual warehouses
* Snowpark sandboxes follow your existing elasticity, scalability and consumption- based pricing selections



Snowpark API- **DataFrame-Based**

* A dataFrame is used to represent a set of data (they are automatically created with no additional configuration)
* A API provides methods to operate on that data

1. Data retrieval
2. Column manipulation
3. filtering

* The snowpark API generates the equivalent SQL statement and pushes it down to snowflake

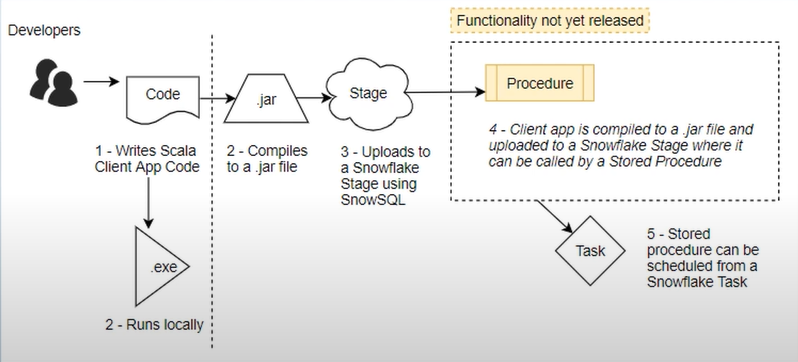
Snowpark API- **Language and tools**

* Scala is the supported language for snowpark with Java, Python
* Developer tools available for API are:

1. VS code
2. IntelliJ
3. Jupyter Notebooks

Snowpark API - **Client Application- stored procedure**

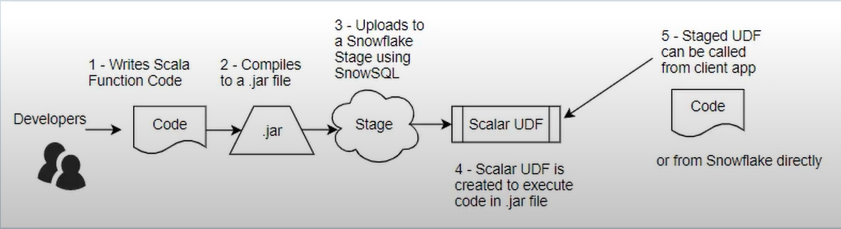
* Developers can use the snowpark library to create client applications
* Client applications can be uploaded and ran from stored procedures



Here we can use other languages like python java or sql in case of scala and while compiling we can compress to other formats like gz, etc refer above documentation.

Snowpark API - **Client Application- UDFs**

* Developers can use the snowspark library to create UDFs
* Here scala or any other language are translated to java in the .jar file before being uploaded to snowflake.
* Snowflake UDFs can be ran from snowflake or called from client applications



Snowflake API- **UDF Restrictions**

Snowflake API does not allow UDFs or client applications to:

* Modify the input value

1. It should remain the same throughout the code
2. **There is no by val or by ref**

* Modify the declared output

1. It can't read file
2. It can't read the console
3. It can't write to a file
4. It can't read or modify a global variable
5. It takes no actions visible outside the code

**Development -UDF**

* Code containing a function library is developed in a local client application
* Code is compiled into a .jar file
* Code is uploaded to a snowflake stage using the PUT command in SnowSQL
* The .jar file in the stage can be called from a client application
* Through the application can contain many functions, only one is registered per UDF in Snowflake

Development- **CLient application-example**

* Snowpark applications can call a snowflake UDF that was uploaded to a snowflake stage
* A dependency has to be declared in the application code to specify the location of the .jar

***snowsession.addDependency(“@jars/walkfactor\_2.12.jar”)***

* The code can then access methods in that UDF
* In this example, there is a DataFrame called df with a column named WALK\_DISTANCE

***Df = df.withColumn(“walk\_factor”, callUDF(“WalkFactor)+col(“WALK\_DISTANCE”))***

* A new column is being added to the existing df and will be named WALK\_FACTOR
* Results from the scalar “WalkFactor” UDF in snowflake, will be added to WALK\_DISTANCE Column value on every row as a new column

Development - **Registering in Snowflake**

* Function in a scalar UDF

1. Specify name of .jar file from snowflake stage as import statement
2. Specify library.class.method as the handler

*create function WalkFactor() returns int*

*language JAVA*

*imports = (‘@jars/walkfactor\_2.12.1.jar’)*

*handle = “WalkFactor.Distance.uphill”;*

* Client application in a stored procedure

1. This is no body to the procedure
2. Specify list of .jar files, including the snowpark .jar in the imports statement

*create or replace procedure WalkingDistanceFactor()*

*returns varchar*

*language JAVA*

*imports = (‘@jars/walkfactor\_2.21.1.jar’, ‘@jars/walkfactor\_2.12.1.jar’)*

*handler = ‘motion.walk,distancefactor’*

*Execute as caller;*

**\*Note:**

When we can schedule queries by using tasks in snowflake, why do we go for third party scheduling tools?

→   
By using tasks we can schedule the tasks and monitor them by TASK\_HISTORY table which is difficult

But third party scheduling tools offer UI based monitoring and it is very easy to control the job flow.

Tools:

Control -M

Airflow

Ansible

TWS

Active Batch

JAMS Scheduler

Q How can I convert my Teradata(oracle, sql) DDL to Snowflake DDL?

→

We can’t convert 100s of table DDL from traditional database to snowflake manually, but we can use below ways:

1. Use roboquery tool which can convert DDL from any database to any other database including SF (https://roboquery.com/app/)
2. You can develop a python script to convert all DDL at a time
3. You can write a procedure in snowflake to convert the DDLs to SF

Q Difference between full load and incremental delta load? How to choose ?

In full load, the complete data set will be loaded every time where we delete /truncate the target table data and load the new dataset

In incremental loads we will fetch only the data that was inserted/ updated after the previous load and load that to the target table by using **UPSERT operations** (update existing record and insert new record). We can pull incremental data from source with help of **LAST\_UPDATE\_TIMESTAMP**  and perform **UPSERT**  by using key fields.

* If there are no keys to identify incremental data from source, we can go for full load.
* If there are no join keys to perform UPSERTs we can go for full load.