snowflake

DYNAMIC DATA MASKING OVERVIEW

Snowflake Professional Services

The need to protect data according to runtime context

Dynamic Data Masking

Policy based

One policy per any number of **columns**. In the future, applied across any number of tables (via tags)

BYO masking algorithm

Object owner **can** be denied access to sensitive data, if desired

Before vs After DDM

```
--PREVIOUSLY, masking as clause per column
CREATE OR REPLACE SECURE VIEW CUSTOMERS_SECV AS
SELECT name, city,
    CASE WHEN CURRENT_ROLE() = 'CLEARANCE_ROLE' THEN socsecno
        ELSE '***MASKED***'
    END AS socsecno
FROM customer_table;

CREATE OR REPLACE SECURE VIEW EMPLOYEES_SECV_HR AS
SELECT name, dept,
    CASE WHEN CURRENT_ROLE() = 'CLEARANCE_ROLE' THEN socsecno
        ELSE '***MASKED***'
    END AS socsecno
FROM employee_table;
```

Governance and more complex scenarios more difficult

```
--NOW, masking as policy as option per object (view, udf etc)
CREATE MASKING POLICY socsecno_mask AS
  (val string) returns string ->
  CASE
    WHEN current_role() IN ('CLEARANCE_ROLE') THEN val
    ELSE  '***MASKED***'
  END;

ALTER TABLE customer_table SET MASKING POLICY = socsecno_mask on column socsoecno;
ALTER TABLE employee_table SET MASKING POLICY = socsecno_mask on column socsoecno;
```

Policy mgmt easy to run part of governance process

Ingestion And Consumption

Dynamically mask protected (PII, PHI) column data at query time

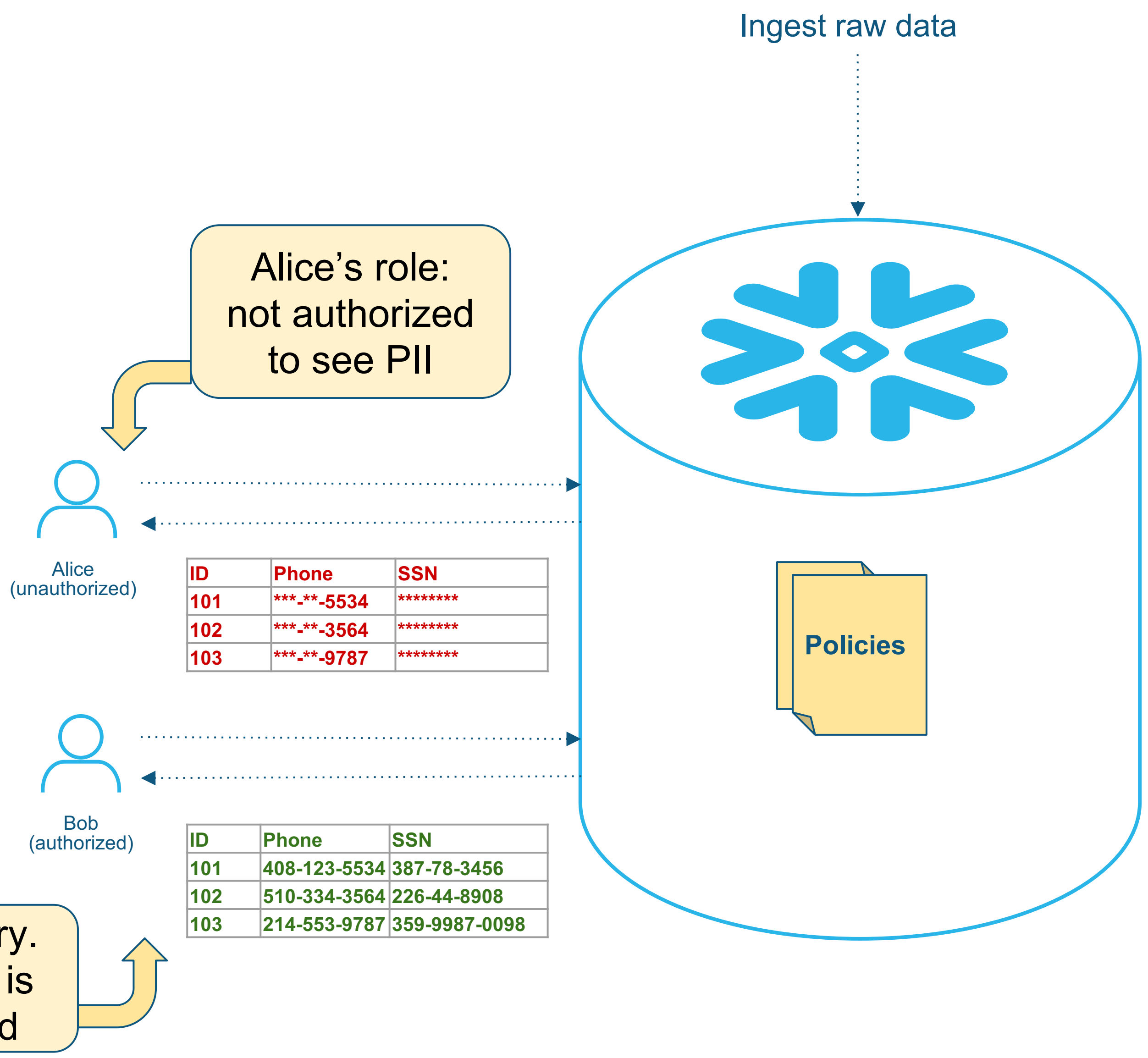
- No change to the stored data
- Mask or partial mask using constant value, hash, and custom functions
- Unmask for authorized users only

Policy based control

- Table/View owners and privileged users (such as ACCOUNTADMIN) unauthorized by default
- **Centralized** policy mgt. Make a change centrally: applied across any number of columns

Ease of Management

- Apply single policy to multiple columns
- Prevent secure view explosion


Ingest raw data
Alice's role: not authorized to see PII
Alice (unauthorized)
Policies
Bob (authorized)
Same query. Bob's role is authorized

ID	Phone	SSN
101	***-**-5534	********
102	***-**-3564	********
103	***-**-9787	********

ID	Phone	SSN
101	408-123-5534	387-78-3456
102	510-334-3564	226-44-8908
103	214-553-9787	359-9987-0098

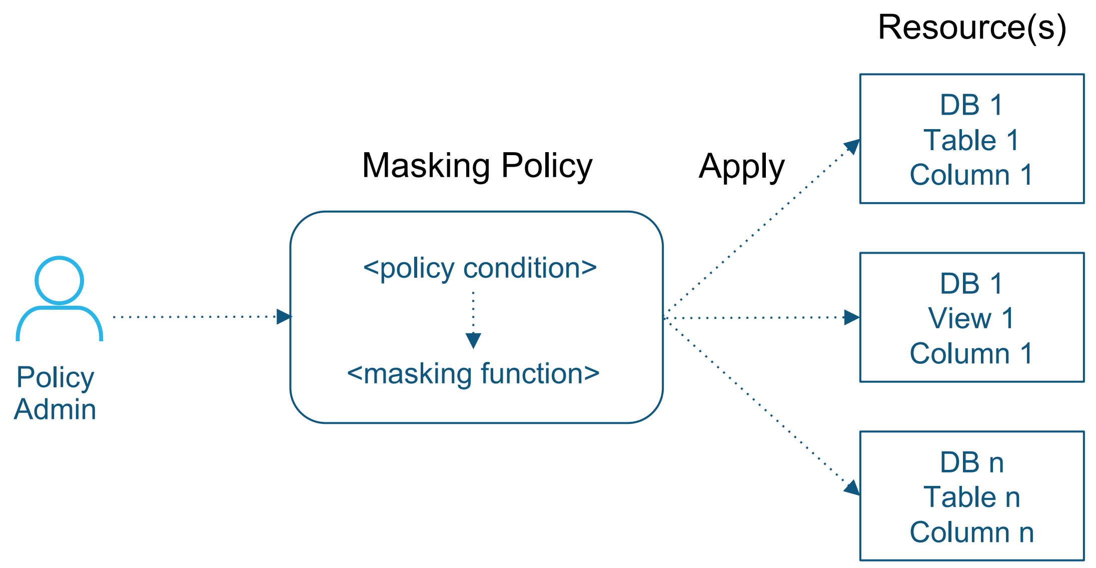
Dynamic Data Masking Policies

Masking Policy

- Policy defines a **masking function** and its required **conditions**
- Policy is applied to one or more table, view, or external table **columns** in an account
- Nested policy execution for views - policy on table **executed before** policy on view(s)

Supports

- All data types, including variant
- Data sharing
- Streams
- Cloning: carries over policy associations



Masking Policy Example

Unmask

Partial mask

Mask

```
CASE
   WHEN invoker_role() IN (‘pii_role’) THEN
val
   WHEN invoker_role() IN (‘support’) THEN
regexp_replace(val,'.+\@','*****@')
   ELSE '********'
END;
```

Create Masking Policy

```
CREATE MASKING POLICY <name> AS
(val <data_type>) returns <data_type> -> (SQL expression on val);
```

Example:

```
CREATE MASKING POLICY email_mask AS

(val string) returns string ->

CASE

    WHEN current_role() IN ('ANALYST') THEN val

    ELSE  '***MASKED***'

END;
```

Masking Policy Examples

Use Case	Policy Example

NULL	`CASE`
	`    WHEN current_role() IN ('ANALYST') THEN val`
	`    ELSE null`
	` END;`

Constant value	`CASE`
	`    WHEN current_role() IN ('ANALYST') THEN val`
	`    ELSE '*********'`
	`END;`

Hash
(useful for join conditions -- hash can act as key)
	`CASE`
	`    WHEN current_role() IN ('ANALYST') THEN val`
	`    ELSE sha2(val)`
	`END;`

Partial mask	`CASE`
	`    WHEN current_role() IN ('ANALYST') THEN`
	`regexp_replace(val,'.+\@','*****@')`
	`    ELSE '*********'`
	`END;`

Masking Policy Examples (cont'd)

Use Case	Policy Example

Using UDF
 	
Useful if masking logic is complex. Entire case statement can be wrapped in UDF as well.
	`CASE`
	`   WHEN current_role() IN ('ANALYST') THEN val`
	`   ELSE mask_udf(val)`
	`END;`

Policy on variant data
 	
OBJECT_INSERT: quickest, if val is at first level.
	`CASE`
	`   WHEN current_role() IN ('ANALYST') THEN val`
	`   OBJECT_INSERT(val, 'USER_IPADDRESS', '****', true)`
	`END;`

Using custom entitlement table	`CASE`
	`   WHEN current_role() IN`
	`       (SELECT role from <db>.<schema>.entitlement`
	`          where mask_method='unmask') THEN val`
	`   ELSE '********'`
	`END;`

Apply Masking Policy To Column(s)

```
ALTER {TABLE | VIEW} <name> MODIFY COLUMN <col_name> [UN]SET MASKING POLICY <name>;
```

Example:

```
ALTER TABLE customer MODIFY COLUMN email SET MASKING POLICY email_mask;

ALTER VIEW customer_v MODIFY COLUMN email SET MASKING POLICY email_mask;
```

Note: policies can also be applied to external tables.

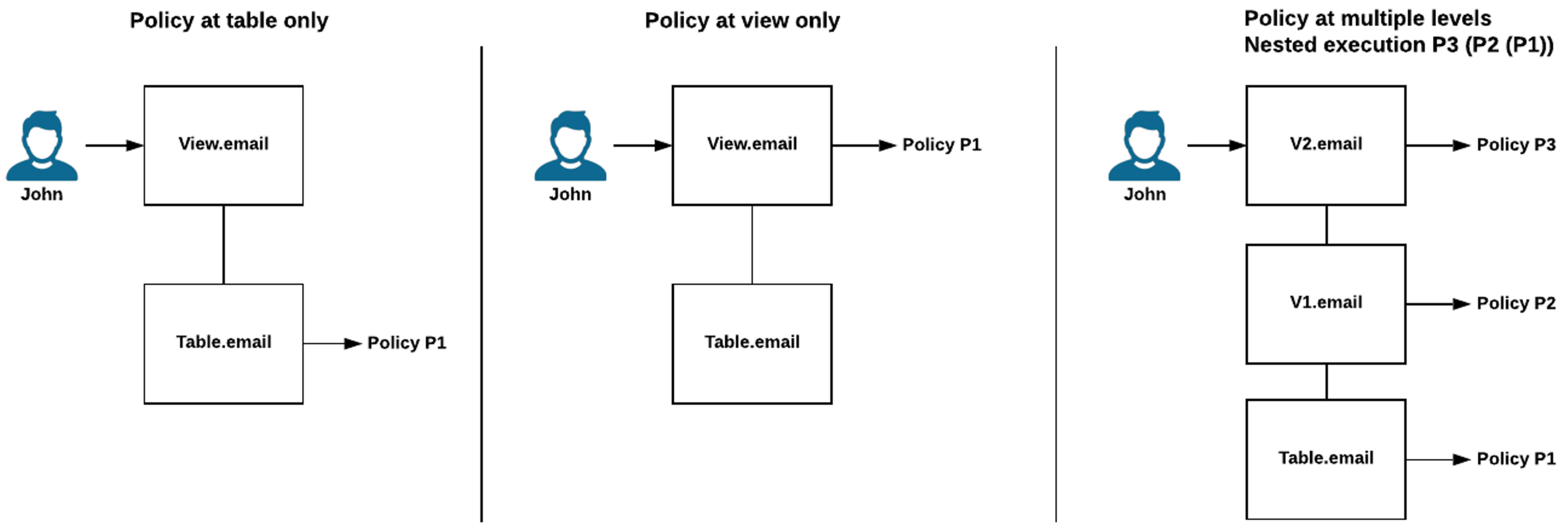
Masking Policy Execution

- Query is rewritten at runtime applying the policy expression on protected column(s) in the query
- The result set cache reuse is disabled for queries with masking columns

Query shape	User submits query	After query rewrite
Simple query	`select name, email from customers;`	`select name, email_mask(email) from customers;`
Query with protected column in the where clause predicate	`select name, email from customers where email = 'bob@acme.com';`	`select name, email_mask(email) from customers where email_mask(email) = 'bob@acme.com';`
Query with protected column in join predicate	`select distinct d.city from emp_basic as b join emp_details as d on b.email = d.email;`	`select distinct d.city from emp_basic as b join emp_details as d on email_mask(b.email) = email_mask(d.email);`

Masking Policy Execution For Views

- Performs nested policy execution; policy not required at every level
- Table level policy (if available) executed first



snowflake®

THANK YOU