DATA CONTROL TOWER · GUIDE

ISO 27001 deletion and masking evidence for Snowflake, Databricks and Redshift

Rows past your retention, deleted data still recoverable, and tagged columns with no mask: three lists from your own policy and the platforms' own records.

Published 2026-11-01 · Online version and downloads: www.datameaning.com/guides/iso-27001-deletion-and-masking-evidence-for-snowflake-databricks-redshift/

The short answer. Deleted does not mean gone. Hand your ISO 27001 auditor three lists from Snowflake, Databricks and Redshift exports: rows past retention, deleted data still recoverable, tagged columns with no mask. The auditor decides what they show. The controls are Annex A 8.10 (Information deletion) and 8.11 (Data masking). A free script prints them; we ran it on made-up records.

The two controls, and what this page does not say

ISO/IEC 27001:2022 and its guidance, ISO/IEC 27002:2022, are paid documents. We have not read them. So this page names two Annex A controls by id and title only: 8.10, Information deletion, and 8.11, Data masking. It does not say what either one requires, and it does not know what your auditor will ask for.

What it can do is answer the question your auditor asks, with code. You write your own policy: how long each table may keep rows, how long deleted data may stay recoverable, and who may see each kind of data in the clear. Your auditor decides whether that policy is enough. The script shows whether the platforms' own records agree with what you wrote.

That needs three lists, because in a warehouse a deleted row is not necessarily gone. What each platform keeps after a delete, from its own documentation:

- **Snowflake.** Time Travel keeps historical data for the table's retention period: for permanent tables on Enterprise Edition and higher, “any value from 0 up to 90 days” (Snowflake: Time Travel (read 2026-11-01)). When it ends, the data moves into Fail-safe, “a (non-configurable) 7-day period during which historical data may be recoverable by Snowflake” (Snowflake: Fail-safe (read 2026-11-01)). “Transient and temporary tables have no Fail-safe period” (Snowflake: temporary and transient tables (read 2026-11-01)).
- **Databricks.** VACUUM removes data files that are no longer in the table's latest state once they pass a retention threshold; “The default retention threshold for vacuum is 7 days” (Databricks: Remove unused data files with VACUUM (read 2026-11-01)). Until a VACUUM runs after that threshold has passed, the old files from a delete stay in storage, and “The ability to query table versions older than the retention

period is lost after running VACUUM" (Databricks: Remove unused data files with VACUUM (read 2026-11-01)). The threshold is the table property `delta.deletedFileRetentionDuration`, which defaults to 7 days (Databricks: VACUUM (SQL) (read 2026-11-01)). On a table with deletion vectors, a DELETE only marks rows: their data stays in the table's current files until `REORG TABLE ... APPLY (PURGE)` rewrites them (OPTIMIZE and auto-compaction can rewrite some of those files, but the page guarantees only REORG rewrites all of them), and a later VACUUM removes the old files (Databricks: deletion vectors (read 2026-11-01)). Whether a table has them on by default "Depends on workspace admin settings and Databricks Runtime version" (Databricks: table properties (read 2026-11-01)).

- **Redshift.** VACUUM DELETE reclaims the disk space of rows marked for deletion, and Redshift runs a DELETE ONLY vacuum automatically in the background (Amazon Redshift: VACUUM (read 2026-11-01)). Snapshots are separate: "By default, manual snapshots are retained indefinitely, even after you delete your cluster", and automated snapshots are deleted at the end of a retention period whose default is one day (Amazon Redshift: snapshots and backups (read 2026-11-01)). A snapshot "contains a copy of the cluster data as of this exact time" (AWS CLI: describe-cluster-snapshots (read 2026-11-01)), so one taken before a delete still holds the rows that were deleted, unless the table was created with BACKUP NO on a cluster that supports it: RG and RA3 clusters and Serverless back up every table (Amazon Redshift: snapshots and backups (read 2026-11-01)).

What you will need

- Your own two lists: a retention register (table, data category, retention days, the date column that ages a row, and the days your policy lets deleted data stay recoverable) and a masking policy (data category, and the roles that may see it in the clear). Table names are written `database.schema.table` on Snowflake and Databricks, and `schema.table` on Redshift.
- Snowflake: `ACCOUNT_USAGE.TABLES` and `ACCOUNT_USAGE.POLICY_REFERENCES` can lag by up to 90 and 120 minutes, so export some hours after your last change (Snowflake: ACCOUNT_USAGE.TABLES (read 2026-11-01); Snowflake: ACCOUNT_USAGE.POLICY_REFERENCES (read 2026-11-01)). POLICY_STATUS is ACTIVE for a mask in force; other values the page lists, such as MULTIPLE_MASKING_POLICY_ASSIGNED_TO_THE_COLUMN, mean it is not.
- Databricks: run `DESCRIBE HISTORY` for each register table and add the table name as a column (Databricks: Table history schema (read 2026-11-01)). History is kept for `logRetentionDuration`, 30 days by default, so export more often than that (Databricks: Work with table history (read 2026-11-01)). `information_schema.column_masks` shows only columns the exporting user can interact with, and the page marks the feature as Public Preview on Unity Catalog (Databricks: COLUMN_MASKS (read 2026-11-01)).
- Redshift: `SVV_ATTACHED_MASKING_POLICY` is visible only to superusers and users with the `sys:secadmin` role, and "Regular users will see 0 rows" (Amazon Redshift: SVV_ATTACHED_MASKING_POLICY (read 2026-11-01)); an empty export fails an input check. `SYS_QUERY_HISTORY` and `SYS_VACUUM_HISTORY` show every row only to a superuser (Amazon Redshift: SYS_QUERY_HISTORY (read 2026-11-01); Amazon Redshift: SYS_VACUUM_HISTORY (read 2026-11-01)), and query text "might be truncated". The pages do not say how far back the system views reach, so check your own: an input check fails if any day of the period has no query history. Snapshots

come from `aws redshift describe-cluster-snapshots` (AWS CLI: describe-cluster-snapshots (read 2026-11-01)).

- Python 3 and the `duckdb` module (we ran it with duckdb 1.5.6 and 2.0.0).

Step 1. Export each platform's records into one folder

The script reads these fourteen files from `EXPORT_DIR`. Column names are the platforms' own; the script lower-cases the header, so `RETENTION_TIME` and `retention_time` both work, and it stops and names the column if one is missing. Names are compared whole and without case or double quotes. Timestamps are read as `YYYY-MM-DD HH:MM:SS` in UTC (a fractional second, a `T` and a `Z`, `+00:00` or `+0000` suffix are accepted; any other offset is not): export from a session set to UTC. A value that cannot be read is counted by an input check, never skipped.

The fourteen files

File	Columns the script reads	Defined in
`retention_register.csv`	platform, table_name, data_category, retention_days, age_column, recoverable_days_allowed	You write it: your register
`row_ages.csv`	platform, table_name, retention_days_used, rows_total, rows_past_retention, rows_no_age, oldest_date, measured_on	The query that --queries prints
`masking_policy.csv`	data_category, roles_in_clear (names separated by semicolons)	You write it: your masking policy
`column_tags.csv`	platform, table_name, column_name, data_category	Your tags (Snowflake TAG_REFERENCES, Databricks column tags, or a list you keep)
`clear_readers.csv`	platform, table_name, column_name, role_name	A list you keep: roles that see the column unmasked
`snowflake_tables.csv`	TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE, IS_TRANSIENT, RETENTION_TIME, DELETED	Snowflake: ACCOUNT_USAGE.TABLES (read 2026-11-01)
`snowflake_policy_references.csv`	POLICY_NAME, POLICY_KIND, REF_DATABASE_NAME, REF_SCHEMA_NAME, REF_ENTITY_NAME, REF_ENTITY_DOMAIN, REF_COLUMN_NAME, POLICY_STATUS	Snowflake: ACCOUNT_USAGE.POLICY_REFERENCES (read 2026-11-01)

File	Columns the script reads	Defined in
`databricks_history.csv`	table_name, version, timestamp, operation (DESCRIBE HISTORY, one table after another)	Databricks: Table history schema (read 2026-11-01)
`databricks_table_properties.csv`	table_name, deleted_file_retention_days (blank if the table does not set it), deletion_vectors (true or false: the table property delta.enableDeletionVectors)	Databricks: VACUUM (SQL) (read 2026-11-01); Databricks: table properties (read 2026-11-01)
`databricks_column_masks.csv`	catalog_name, schema_name, table_name, column_name, mask_catalog, mask_schema, mask_name	Databricks: COLUMN_MASKS (read 2026-11-01)
`redshift_sys_query_history.csv`	user_id, query_id, query_type, status, start_time, query_text	Amazon Redshift: SYS_QUERY_HISTORY (read 2026-11-01)
`redshift_sys_vacuum_history.csv`	schema_name, table_name, vacuum_type, is_automatic, status, start_time, end_time	Amazon Redshift: SYS_VACUUM_HISTORY (read 2026-11-01)
`redshift_snapshots.csv`	SnapshotIdentifier, SnapshotCreateTime, SnapshotType, ManualSnapshotRetentionPeriod	AWS CLI: describe-cluster-snapshots (read 2026-11-01)
`redshift_attached_masking_policy.csv`	policy_name, schema_name, table_name, grantee, grantee_type, output_columns	Amazon Redshift: SVV_ATTACHED_MASKING_POLICY (read 2026-11-01)

Step 2. Write your policy, then measure row ages

No platform view holds the age of a row, so the script prints the query for you. Put your retention register in `retention_register.csv`, then run `python3 deletion_masking_evidence.py --queries`. It prints one query per register table, in the dialect of the table's platform: Snowflake and Redshift use `DATEADD`, Databricks uses `date_sub`. Each query reads the date column of every row in its table, so run it when the load is light. Here is the first one for the demo register:

```
SELECT 'snowflake' AS platform, 'sf_prod.sales.customer_contacts' AS table_name, 730 AS retention_days_used,
       COUNT(*) AS rows_total,
       COALESCE(SUM(CASE WHEN created_at < DATEADD(day, -730, CURRENT_DATE) THEN 1 ELSE 0 END), 0) AS
rows_past_retention,
       COALESCE(SUM(CASE WHEN created_at IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,
       CAST(MIN(created_at) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on
FROM sf_prod.sales.customer_contacts;
```

Run each query on the platform it names and save every answer as one line of `row_ages.csv`. Each line carries the retention days it used, and the script checks that against your register. We printed these queries

but did not run them on a platform: they are plain SQL, and you should read one before you run it. The two lists you keep look like this in the demo (the full register and the full masking policy):

```
retention_register.csv
platform,table_name,data_category,retention_days,age_column,recoverable_days_allowed
snowflake,sf_prod.sales.customer_contacts,contact,730,created_at,14
snowflake,sf_prod.sales.web_events,event,90,event_date,14
snowflake,sf_prod.hr.employee_notes,hr,365,noted_on,14
databricks,main.sales.customer_contacts,contact,730,created_at,14
databricks,main.sales.web_events,event,90,event_date,14
redshift,sales.customer_contacts,contact,730,created_at,14
redshift,sales.web_events,event,90,event_date,14

masking_policy.csv
data_category,roles_in_clear
contact,SUPPORT_LEAD;DATA_STEWARD
hr,HR_ADMIN
network,SECURITY_ANALYST
```

Step 3. How each platform is measured

The second list's "recoverable days" come from the table's settings on Snowflake, and from each delete in the period on Databricks and Redshift. On Redshift they count the time to a completed vacuum or to a snapshot's expiry; they are not a promise that a restore would work.

Snowflake. Recoverable days are the table's retention time, plus the 7-day Fail-safe unless the table is transient. A dropped table with the same name is ignored. Set `SNOWFLAKE_COUNT_FAILSAFE` to False if your policy counts only the days you can query the data yourself, because Fail-safe is recoverable by Snowflake only (Snowflake: Fail-safe (read 2026-11-01)). A temporary table, like a transient one, has no Fail-safe. If your account sets `MIN_DATA_RETENTION_TIME_IN_DAYS`, the effective retention is the larger of it and the table's own (Snowflake: Time Travel (read 2026-11-01)); the TABLES page does not say whether RETENTION_TIME reflects it, so check `SHOW PARAMETERS LIKE 'MIN_DATA_RETENTION_TIME_IN_DAYS' IN ACCOUNT`.

```
hit = con.execute("SELECT rt, transient, ttype FROM sf_t WHERE t = ? AND deleted IS NULL", [t]).fetchall()
if not hit:
    status, basis = "NOT MEASURED", "table not in the ACCOUNT_USAGE.TABLES export"
elif hit[0][0] is None:
    status, basis = "NOT MEASURED", "retention_time is not a whole number"
else:
    rt, transient, ttype = hit[0]
    fs = FAILSAFE_DAYS if (transient == "NO" and "TEMPORARY" not in ttype and SNOWFLAKE_COUNT_FAILSAFE) else
0
    days = rt + fs
    basis = (f"Time Travel {rt} d + Fail-safe {FAILSAFE_DAYS} d" if fs else
             f"Time Travel {rt} d, transient: no Fail-safe" if transient == "YES" else
             f"Time Travel {rt} d, temporary: no Fail-safe" if "TEMPORARY" in ttype else f"Time Travel {rt} d
only")
    status = "RECOVERABLE BEYOND POLICY" if days > allowed else "WITHIN POLICY"
```

Databricks. For each DELETE in the table's history during the period, the data can be removed by the first finished VACUUM that runs at least the table's deleted-file retention after the delete; a VACUUM that ran earlier removes none of that delete's files. On a table with deletion vectors the clock starts at the first REORG after the delete (the setting `DATABRICKS_PURGE_OPS`), because until it rewrites the files the deleted rows are

still in the table's current files; a table that does not say whether deletion vectors are on is NOT MEASURED. If no qualifying VACUUM has run, the delete is open: recoverable at least until the export. Only DELETE operations are read; MERGE, UPDATE, overwrites and TRUNCATE are not. The worst delete is the one reported.

```
for (d,) in dels:
    # with deletion vectors a DELETE only marks rows: a REORG ... APPLY (PURGE) must rewrite the files
    # first, and the VACUUM that removes the old files must come at least the retention after that
    start = d
    if dv == "true":
        hit = con.execute(f"SELECT ts, op FROM db_h WHERE t = ? AND op IN ({', '.join('?' for _ in purge)})
AND ts >= ? ORDER BY ts LIMIT 1",
                          [t, *purge, d]).fetchone()
        start, purge_op = (hit[0], hit[1]) if hit else (None, None)
    v = None if start is None else con.execute("SELECT min(ts) FROM db_h WHERE t = ? AND op = ? AND ts >= ?",
                                               [t, done, start + dt.timedelta(days=keep)]).fetchone()[0]
    if dv not in ("true", "false"):
        val, st, b = None, "unknown", f"delete {d:%Y-%m-%d}; deletion_vectors not stated, so removal cannot
be judged"
    elif v:
        val, st = (v - d).total_seconds() / 86400, None
        b = f"delete {d:%Y-%m-%d}, " + (f"{purge_op} {start:%Y-%m-%d}, " if dv == "true" else "") + f"VACUUM
finished {v:%Y-%m-%d}, files kept {keep} d"
    elif dv == "true" and start is None:
        val, st, b = (as_of_end - d).total_seconds() / 86400, "open", f"delete {d:%Y-%m-%d}, deletion vectors
on and no " + "/".join(purge) + " since: rows still in live files"
    else:
        val, st, b = (as_of_end - d).total_seconds() / 86400, "open", f"delete {d:%Y-%m-%d}, no VACUUM yet
old enough to remove its files (kept {keep} d)"
    if st == "unknown":
        cand = (0, 0.0, "NOT MEASURED", b)
        best = cand if best is None else best
```

Redshift. Deletes come from the query history: a successful statement that starts `DELETE` (FROM is optional in Redshift) once comments and double quotes are removed, and names its table as schema.table. One that names only the table, or sits behind a WITH clause, fails an input check when that name matches a table in your register, because the script cannot tell which schema it meant; one on any other table (a staging table) is left out. Removal is the end of the first completed Delete or Full vacuum that started after the delete. A manual snapshot taken before the delete and kept indefinitely is flagged on its own; a snapshot that expires later pushes the recoverable days out to its expiry. By default a vacuum reclaims space so that at least 95 percent of the remaining rows are not marked for deletion (Amazon Redshift: VACUUM (read 2026-11-01)), so a small share of deleted rows can survive a completed vacuum, and a WITHIN POLICY line on Redshift may be early.

```
indef = [c for c, ty, r in snaps if ty == "manual" and r == -1 and c < d]
exp = [c + dt.timedelta(days=(r if ty == "manual" else auto)) for c, ty, r in snaps
       if c < d and not (ty == "manual" and r == -1) and c + dt.timedelta(days=(r if ty == "manual" else
auto)) > d]
if indef:
    cand = (rank["INDEFINITE SNAPSHOT"], 0.0, "INDEFINITE SNAPSHOT",
            f"delete {d:%Y-%m-%d}; a manual snapshot from {min(indef):%Y-%m-%d} is kept indefinitely")
```

Masking. A column counts as masked if the platform lists a mask on it: a Snowflake `MASKING_POLICY` row with POLICY_STATUS ACTIVE (only Snowflake reports a status), a Databricks `column_masks` row, or a Redshift attached policy whose output columns name the column (compared as a whole word) and whose grantee is PUBLIC. A Redshift policy attached only to a role or user is NOT MASKED FOR PUBLIC, because

PUBLIC “Attaches the masking policy to all users accessing the table” (Amazon Redshift: ATTACH MASKING POLICY (read 2026-11-01)). Databricks lists the attachment, and the script cannot tell from it who the mask applies to. The roles that read the column in the clear come from `clear_readers.csv`, a list you keep, because no platform view names them, and are compared with your policy's list.

```
allowed = {r.strip().lower() for r in roles.split(";") if r.strip()}
clear = sorted({r for (r,) in con.execute("SELECT role FROM clr WHERE platform = ? AND t = ? AND col = ? AND
role IS NOT NULL", [plat, t, col]).fetchall()})
extra = [r for r in clear if r not in allowed]
flags = []
if not active:
    flags.append("MASK NOT ACTIVE (" + ", ".join(sorted(set(inactive))) + ")" if inactive else
                 "NOT MASKED FOR PUBLIC (attached only to " + ", ".join(sorted(set(partial))) + ")" if
partial else "NOT MASKED")
if extra:
    flags.append("CLEAR READER NOT ON POLICY LIST: " + ", ".join(extra))
rows.append((plat, t, col, cat, ", ".join(clear), "; ".join(flags) or "OK"))
```

Step 4. Run it on the clean generated records

`python3 deletion_masking_evidence.py --demo` writes generated exports (seed 231) in each platform's column names and reads them back. In this run every table is inside the retention in its register and every tagged column has a mask, so nothing needs follow-up. That says only that these generated exports agree with this register and policy. The status of the open Databricks deletes is OPEN, WITHIN SO FAR: no VACUUM old enough to remove their files has run, and the days so far are inside the allowance. The 26 input checks all passed (they are shown in step 6).

```
== Rows past retention: iso_1_rows_past_retention
   period 2026-09-16 to 2026-10-15 | filters: register tables on exported platforms | run 2026-11-01
13:48:31Z | rows 7 | sha256 c90b08ff10d0b5bfdb512aab47e8236f675186c6323878ae8342100ab3dd84f9
   platform    table                           category  retention_days  rows    rows_past_retention
oldest_date  status
   ----------  ------------------------------  --------  --------------  ------  -------------------  ------
-----  ------
   databricks  main.sales.customer_contacts    contact   730             207851  0                    2024-
10-21   OK
   databricks  main.sales.web_events           event     90              338638  0                    2026-
07-22   OK
   redshift    sales.customer_contacts         contact   730             161316  0                    2024-
10-22   OK
   redshift    sales.web_events                event     90              147811  0                    2026-
07-19   OK
   snowflake   sf_prod.hr.employee_notes        hr        365             239349  0                    2025-
10-18   OK
   snowflake   sf_prod.sales.customer_contacts  contact   730             51426   0                    2024-
10-19   OK
   snowflake   sf_prod.sales.web_events         event     90              105751  0                    2026-
07-20   OK

== Deleted but still recoverable: iso_2_deleted_but_recoverable
   period 2026-09-16 to 2026-10-15 | filters: register tables; Snowflake from table settings, the others from
deletes in the period | run 2026-11-01 13:48:31Z | rows 7 | sha256
ea257ee2b272b846bd833c976dabb85aec5cd8fd3adf4ab864ab4b356fea82fb
   platform    table                           category  allowed_days  recoverable_days  basis
status
   ----------  ------------------------------  --------  ------------  ----------------  -------------------
```

```
-----------------------------------------------------------  -------------------
   databricks  main.sales.customer_contacts     contact   14           7.6               delete 2026-10-08,
no VACUUM yet old enough to remove its files (kept 7 d)        OPEN, WITHIN SO FAR
   databricks  main.sales.web_events            event     14           7.6               delete 2026-10-08,
no VACUUM yet old enough to remove its files (kept 7 d)        OPEN, WITHIN SO FAR
   redshift    sales.customer_contacts          contact   14           7.9               delete 2026-09-20,
VACUUM DELETE finished 2026-09-21, snapshot kept to 2026-09-28  WITHIN POLICY
   redshift    sales.web_events                 event     14           7.9               delete 2026-09-20,
VACUUM DELETE finished 2026-09-21, snapshot kept to 2026-09-28  WITHIN POLICY
   snowflake   sf_prod.hr.employee_notes        hr        14           8                 Time Travel 1 d +
Fail-safe 7 d                                                   WITHIN POLICY
   snowflake   sf_prod.sales.customer_contacts  contact   14           14                Time Travel 7 d +
Fail-safe 7 d                                                   WITHIN POLICY
   snowflake   sf_prod.sales.web_events         event     14           1                 Time Travel 1 d,
transient: no Fail-safe                                          WITHIN POLICY

== Masking against your policy: iso_3_masking_against_policy
   period 2026-09-16 to 2026-10-15 | filters: columns tagged with a category | run 2026-11-01 13:48:31Z |
rows 10 | sha256 603359d40c940b2c5226048bae8db58eb17234cd6803c638c3302ec44e5d361a
   platform    table                            column      category  roles_reading_in_clear  status
   ----------  -------------------------------  ----------  --------  ----------------------  ------
   databricks  main.sales.customer_contacts     email       contact   data_steward            OK
   databricks  main.sales.customer_contacts     phone       contact                           OK
   databricks  main.sales.web_events            ip_address  network                           OK
   redshift    sales.customer_contacts          email       contact                           OK
   redshift    sales.customer_contacts          phone       contact                           OK
   redshift    sales.web_events                 ip_address  network   security_analyst        OK
   snowflake   sf_prod.hr.employee_notes        note_text   hr        hr_admin                OK
   snowflake   sf_prod.sales.customer_contacts  email       contact   support_lead            OK
   snowflake   sf_prod.sales.customer_contacts  phone       contact                           OK
   snowflake   sf_prod.sales.web_events         ip_address  network                           OK

== Result
   follow-up: none
   input checks: 26 of 26 passed
   RESULT: 0 item(s) need follow-up; 0 input check(s) failed; 0 platform(s) not checked.
```

Step 5. Run it again with findings planted in the records

`python3 deletion_masking_evidence.py --demo --plant` changes the generated records and nothing else: a Snowflake table with a 90-day Time Travel setting, a Databricks table whose deleted-file retention is 30 days, a Redshift manual snapshot kept indefinitely, rows older than their retention in two tables, a table never measured, a tagged column with no mask on Snowflake and on Databricks, and a role that is not on the policy's list reading an email column in the clear.

```
== Rows past retention: iso_1_rows_past_retention
   period 2026-09-16 to 2026-10-15 | filters: register tables on exported platforms | run 2026-11-01
13:48:31Z | rows 7 | sha256 8fc9ade2cbe47539df667646a1ce764749a7dc29dd3a23835461532d1c215ae2
   platform    table                            category  retention_days  rows    rows_past_retention
oldest_date  status
   ----------  -------------------------------  --------  --------------  ------  -------------------  ------
-----  --------------
   databricks  main.sales.customer_contacts     contact   730             207851  0                    2024-
10-21   OK
   databricks  main.sales.web_events            event     90              338638  37                   2026-
07-02   PAST RETENTION
   redshift    sales.customer_contacts          contact   730
NOT MEASURED
   redshift    sales.web_events                 event     90              147811  0                    2026-
```

```
07-19   OK
   snowflake   sf_prod.hr.employee_notes        hr        365             239349  0                   2025-
10-18   OK
   snowflake   sf_prod.sales.customer_contacts  contact   730             51426   0                   2024-
10-19   OK
   snowflake   sf_prod.sales.web_events         event     90              105751  1204                2026-
06-30   PAST RETENTION

== Deleted but still recoverable: iso_2_deleted_but_recoverable
   period 2026-09-16 to 2026-10-15 | filters: register tables; Snowflake from table settings, the others from
deletes in the period | run 2026-11-01 13:48:31Z | rows 7 | sha256
f0ef5eed6dcfa0c11130b0a43bb7f0374c50d0d924f128dd1d6429bcd40530b0
   platform    table                           category  allowed_days  recoverable_days  basis
status
   ----------  ------------------------------  --------  ------------  ----------------  --------------------
-----------------------------------------------------  --------------------------
   databricks  main.sales.customer_contacts     contact   14            25.6              delete 2026-09-20,
no VACUUM yet old enough to remove its files (kept 30 d)  RECOVERABLE BEYOND POLICY
   databricks  main.sales.web_events            event     14            7.6               delete 2026-10-08,
no VACUUM yet old enough to remove its files (kept 7 d)   OPEN, WITHIN SO FAR
   redshift    sales.customer_contacts          contact   14                              delete 2026-09-20;
a manual snapshot from 2026-09-18 is kept indefinitely    INDEFINITE SNAPSHOT
   redshift    sales.web_events                 event     14                              delete 2026-09-20;
a manual snapshot from 2026-09-18 is kept indefinitely    INDEFINITE SNAPSHOT
   snowflake   sf_prod.hr.employee_notes        hr        14            97                Time Travel 90 d +
Fail-safe 7 d                                             RECOVERABLE BEYOND POLICY
   snowflake   sf_prod.sales.customer_contacts  contact   14            14                Time Travel 7 d +
Fail-safe 7 d                                              WITHIN POLICY
   snowflake   sf_prod.sales.web_events         event     14            1                 Time Travel 1 d,
transient: no Fail-safe                                     WITHIN POLICY

== Masking against your policy: iso_3_masking_against_policy
   period 2026-09-16 to 2026-10-15 | filters: columns tagged with a category | run 2026-11-01 13:48:31Z |
rows 10 | sha256 b9385ecd6afa326e48d1c2eacea21c5542ab3406a2b36d6564fbf5ae464a6de6
   platform    table                           column      category  roles_reading_in_clear  status
   ----------  ------------------------------  ----------  --------  ----------------------  ---------------
-----------------------------------
   databricks  main.sales.customer_contacts     email       contact   data_steward            OK
   databricks  main.sales.customer_contacts     phone       contact                           OK
   databricks  main.sales.web_events            ip_address  network                           NOT MASKED
   redshift    sales.customer_contacts          email       contact   marketing_analyst       CLEAR READER
NOT ON POLICY LIST: marketing_analyst
   redshift    sales.customer_contacts          phone       contact                           OK
   redshift    sales.web_events                 ip_address  network   security_analyst        OK
   snowflake   sf_prod.hr.employee_notes        note_text   hr        hr_admin                OK
   snowflake   sf_prod.sales.customer_contacts  email       contact   support_lead            OK
   snowflake   sf_prod.sales.customer_contacts  phone       contact                           NOT MASKED
   snowflake   sf_prod.sales.web_events         ip_address  network                           OK

== Result
   follow-up: 2 past retention, 1 not measured, 2 recoverable beyond policy, 2 indefinite snapshot, 2 not
masked or mask not active, 1 clear reader not on policy list
   input checks: 26 of 26 passed
   RESULT: 10 item(s) need follow-up; 0 input check(s) failed; 0 platform(s) not checked.
```

Ten items need follow-up. The Snowflake table is set to be recoverable for 97 days, 90 of Time Travel and 7 of Fail-safe, which only Snowflake can recover, against 14 allowed. The Databricks table's delete of 2026-09-20 has been recoverable for 25.6 days: a VACUUM finished on 2026-09-28, but the table keeps deleted files for 30 days, so the script counts that VACUUM as removing none of them. The Redshift snapshot is flagged without a number because it has no end date.

Step 6. Run it with faults planted in the inputs

A check is only worth showing if it can fail. `python3 deletion_masking_evidence.py --demo --plant-input` plants one fault for each of the 26 input checks. Every one reports FAIL, and the three lists are marked INCOMPLETE so nobody takes them for a clean result.

```
== Input checks
   [FAIL] every platform name in your lists is snowflake, databricks or redshift: 1 row(s)
   [FAIL] no table appears twice in the retention register: 1 row(s)
   [FAIL] every register line has a category, whole-number days, and a date column that is a plain name: 1
row(s)
   [FAIL] every column tag has a table, a column and a category: 1 row(s)
   [FAIL] every category in the masking policy is on at least one tagged column: 1 row(s)
   [FAIL] every clear reader names a table, a column and a role: 1 row(s)
   [FAIL] every clear reader is on a column that is tagged: 1 row(s)
   [FAIL] every row-age line is for a table in the register: 1 row(s)
   [FAIL] no table has two row-age lines: 1 row(s)
   [FAIL] row-age counts are whole numbers and the oldest date is YYYY-MM-DD (or blank for an empty table): 1
row(s)
   [FAIL] row ages were made with the same retention days as the register: 1 row(s)
   [FAIL] row ages were measured within 3 days before AS_OF, not after it: 1 row(s)
   [FAIL] no register table has rows with no date to age them by: 1 row(s)
   [FAIL] Snowflake: retention_time is a whole number and is_transient is YES or NO: 1 row(s)
   [FAIL] Snowflake: no register table has two live (not dropped) rows: 1 row(s)
   [FAIL] Databricks: every history timestamp is YYYY-MM-DD HH:MM:SS: 1 row(s)
   [FAIL] Databricks: every VACUUM operation in the history is one the settings name: 1 row(s)
   [FAIL] Databricks: each register table's history reaches back to the period's start (or to its first
version): 1 row(s)
   [FAIL] Databricks: a table's deleted-file retention is a whole number of days or blank: 1 row(s)
   [FAIL] Databricks: every register table says whether deletion vectors are on (true or false): 1 row(s)
   [FAIL] Redshift: every query-history, vacuum and snapshot timestamp is YYYY-MM-DD HH:MM:SS: 1 row(s)
   [FAIL] Redshift: every day from the period's start to AS_OF has query history, except KNOWN_GAPS: 1 row(s)
   [FAIL] Redshift: snapshot type is manual or automated, and a manual retention is -1 or whole days from 1:
1 row(s)
   [FAIL] Redshift: REDSHIFT_AUTOMATED_RETENTION_DAYS is set when automated snapshots exist: 1 row(s)
   [FAIL] Redshift: the masking-policy export has rows (a user who is not a superuser or sys:secadmin sees
0): 1 row(s)
   [FAIL] Redshift: every DELETE that could touch a register table starts the statement and names
schema.table: 1 row(s)
```

```
== Result
   follow-up: 2 not measured, 3 not masked or mask not active
   input checks: 0 of 26 passed; the lists above are INCOMPLETE
   RESULT: 5 item(s) need follow-up; 26 input check(s) failed; 0 platform(s) not checked.
```

The fault that trips each input check

#	Check	Planted fault
1	Every platform name in your lists is snowflake, databricks or redshift	a column tag for platform oracle
2	No table appears twice in the retention register	sales.customer_contacts listed twice on Redshift
3	Every register line has a category, whole-number days and a plain date column	recoverable_days_allowed written as 14.5
4	Every column tag has a table, a column and a category	a tag with a blank category

#	Check	Planted fault
5	Every category in the masking policy is on a tagged column	the category financial, which no column carries
6	Every clear reader names a table, a column and a role	a clear-reader row with a blank role
7	Every clear reader is on a tagged column	a clear reader on DOB, which is not tagged
8	Every row-age line is for a register table	a row-age line for sales.old_staging
9	No table has two row-age lines	sf_prod.sales.web_events measured twice
10	Row-age counts are whole numbers, the oldest date is YYYY-MM-DD	rows_total written as many
11	Row ages used the register's retention days	customer_contacts measured with 365 days against 730
12	Row ages were measured shortly before AS_OF	employee_notes measured on 2026-09-01
13	No register table has rows with no date	rows_no_age of 4 on employee_notes
14	Snowflake: retention_time is whole, is_transient is YES or NO	a table with is_transient MAYBE and retention_time seven
15	Snowflake: no register table has two live rows	WEB_EVENTS listed live twice
16	Databricks: every history timestamp is readable	a history timestamp written last tuesday
17	Databricks: every VACUUM operation is one the settings name	an operation called VACUUM FULL
18	Databricks: history reaches back to the period's start	web_events history that starts on 2026-09-20 with no version 0
19	Databricks: deleted-file retention is whole days or blank	deleted_file_retention_days written 7 days
20	Databricks: every register table says whether deletion vectors are on	deletion_vectors written maybe
21	Redshift: every timestamp is readable	a snapshot created yesterday
22	Redshift: every day has query history	no query history on 2026-10-03
23	Redshift: snapshot type and retention are readable	a manual snapshot with retention ten
24	Redshift: the automated retention setting is set	REDSHIFT_AUTOMATED_RETENTION_DAYS left unset
25	Redshift: the masking-policy export has rows	an empty masking-policy export
26	Redshift: every DELETE that could touch a register table starts the statement and names schema.table	a DELETE behind a WITH clause

The code

The complete download, deletion_masking_evidence.py, runnable as written. The same file is on the guide page.

```
#!/usr/bin/env python3
"""Deletion and masking evidence from Snowflake, Databricks and Redshift, in DuckDB.

    python3 deletion_masking_evidence.py --demo               # clean generated records
    python3 deletion_masking_evidence.py --demo --plant       # generated records with planted findings
    python3 deletion_masking_evidence.py --demo --plant-input  # generated records with planted input faults
    python3 deletion_masking_evidence.py --queries            # print the row-age query for every table in
YOUR register
    python3 deletion_masking_evidence.py --demo --queries     # the same for the generated register
    python3 deletion_masking_evidence.py                      # your exports: set the settings below first
    python3 deletion_masking_evidence.py --strict             # exit 1 unless nothing needs follow-up

You keep two lists: a retention register (table, data category, retention days, the date column that ages a
row, and how many days your policy lets deleted data stay recoverable) and a masking policy (data category
and
the roles that may see it in the clear). The script reads each platform's own records, exported as CSV with
the platform's published column names, and prints three lists, each with its period, filters, run time, row
count and a SHA-256 of its rows:
  1. rows past retention: any register table with rows older than its retention (PAST RETENTION), or with no
     row-age export (NOT MEASURED)
  2. deleted but still recoverable: how long deleted data stays recoverable on that platform, against what
     your register allows (RECOVERABLE BEYOND POLICY), and a Redshift manual snapshot kept indefinitely that
     was taken before a delete (INDEFINITE SNAPSHOT, flagged whatever the register says)
  3. masking against your policy: every column tagged with a category your masking policy covers must have a
     mask the platform lists on it (Snowflake also reports a status; the script counts only ACTIVE), and any
role
     you list as reading it in the clear must be on the policy's list (NOT MASKED, NOT MASKED FOR PUBLIC for
a
     Redshift mask attached only to named roles or users, MASK NOT ACTIVE, CLEAR READER NOT ON POLICY LIST)
Before it builds them it runs input checks and marks the lists INCOMPLETE if one fails. A platform you do not
export is printed as NOT CHECKED. The script reads files; it never connects to a platform and never writes to
one.

Input checks: --demo --plant-input plants one fault for each of the 26, and every one reports FAIL, for
example
  a table twice in the register, a masking-policy category ('financial') with no tagged column, a row-age
line for
  a table not in the register ('sales.old_staging'), row ages made with 365 days against the register's 730,
a
  Databricks VACUUM operation the script does not know ('VACUUM FULL'), Databricks history that starts after
the
  period, a Redshift day with no query history (2026-10-03), a Redshift DELETE behind a CTE that cannot be
read,
  an unreadable snapshot, automated snapshots with no retention setting, an empty Redshift masking export,
and a
  Databricks deletion_vectors value that is not true or false.

The demo records are GENERATED (seed 231): not a company's data. Column names come from:
  Snowflake  https://docs.snowflake.com/en/sql-reference/account-usage/tables
             https://docs.snowflake.com/en/sql-reference/account-usage/policy_references
  Databricks https://docs.databricks.com/aws/en/tables/history-schema
             https://docs.databricks.com/aws/en/tables/features/deletion-vectors
             https://docs.databricks.com/aws/en/sql/language-manual/information-schema/column_masks
  Redshift   https://docs.aws.amazon.com/redshift/latest/dg/SYS_QUERY_HISTORY.html
             https://docs.aws.amazon.com/redshift/latest/dg/SYS_VACUUM_HISTORY.html
             https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_ATTACHED_MASKING_POLICY.html
```

```
             https://docs.aws.amazon.com/cli/latest/reference/redshift/describe-cluster-snapshots.html
Our assumptions, to check against one of your own rows: Databricks DESCRIBE HISTORY names a delete DELETE
(the
history-schema page says so), a finished VACUUM VACUUM END, and REORG TABLE ... APPLY (PURGE) is the
operation REORG (the history-schema page names neither
VACUUM's nor REORG's operation names, so both are settings); Snowflake POLICY_KIND is MASKING_POLICY
for a masking policy; a Redshift DELETE is a statement that starts DELETE (FROM is optional) once comments
and
double quotes are removed.
"""
import csv
import datetime as dt
import hashlib
import pathlib
import random
import re
import sys

import duckdb

# ---- CHANGE THESE FOR YOUR OWN DATA -------------------------------------------------------
PERIOD_START = "2026-09-16"          # first day of the review period (UTC, inclusive)
AS_OF = "2026-10-15"                 # the day you took the exports (UTC); records must run to here
EXPORT_DIR = None                    # folder holding the export files named in FILES below
PLATFORMS_EXPORTED = ["snowflake", "databricks", "redshift"]   # a platform not listed is NOT CHECKED
SNOWFLAKE_COUNT_FAILSAFE = True      # judge Snowflake on Time Travel plus the 7-day Fail-safe (False: Time
Travel only)
DATABRICKS_DEFAULT_RETENTION_DAYS = 7        # delta.deletedFileRetentionDuration when the table does not set
it
DATABRICKS_VACUUM_DONE = "VACUUM END"        # the DESCRIBE HISTORY operation that is a finished VACUUM
DATABRICKS_VACUUM_OTHER = ["VACUUM START"]   # other VACUUM operations you know and ignore
DATABRICKS_PURGE_OPS = ["REORG"]             # the operation that rewrites every file with deletion vectors
(REORG ... APPLY (PURGE))
REDSHIFT_AUTOMATED_RETENTION_DAYS = None     # no default: your cluster's automated snapshot retention in
days
ROW_AGES_MAX_STALE_DAYS = 3          # row ages must have been measured within this many days of AS_OF
KNOWN_GAPS = {}                      # days you can explain, e.g. {"redshift": ["2026-10-03"]}
# -------------------------------------------------------------------------------------------

FAILSAFE_DAYS = 7   # Snowflake: "a (non-configurable) 7-day period" (docs.snowflake.com/en/user-guide/data-
failsafe)
FILES = {
    "retention_register": "retention_register.csv",
    "row_ages": "row_ages.csv",
    "masking_policy": "masking_policy.csv",
    "column_tags": "column_tags.csv",
    "clear_readers": "clear_readers.csv",
    "snowflake_tables": "snowflake_tables.csv",
    "snowflake_policy_references": "snowflake_policy_references.csv",
    "databricks_history": "databricks_history.csv",
    "databricks_table_properties": "databricks_table_properties.csv",
    "databricks_column_masks": "databricks_column_masks.csv",
    "redshift_sys_query_history": "redshift_sys_query_history.csv",
    "redshift_sys_vacuum_history": "redshift_sys_vacuum_history.csv",
    "redshift_snapshots": "redshift_snapshots.csv",
    "redshift_attached_masking_policy": "redshift_attached_masking_policy.csv",
}
PLATFORM_FILES = {
    "snowflake": ["snowflake_tables", "snowflake_policy_references"],
    "databricks": ["databricks_history", "databricks_table_properties", "databricks_column_masks"],
    "redshift": ["redshift_sys_query_history", "redshift_sys_vacuum_history", "redshift_snapshots",
                 "redshift_attached_masking_policy"],
}
COLUMNS = {   # the platforms' published column names, lower-cased here; the file's own case does not matter
```

```
    "retention_register": ["platform", "table_name", "data_category", "retention_days", "age_column",
                           "recoverable_days_allowed"],
    "row_ages": ["platform", "table_name", "retention_days_used", "rows_total", "rows_past_retention",
                 "rows_no_age", "oldest_date", "measured_on"],
    "masking_policy": ["data_category", "roles_in_clear"],
    "column_tags": ["platform", "table_name", "column_name", "data_category"],
    "clear_readers": ["platform", "table_name", "column_name", "role_name"],
    "snowflake_tables": ["table_catalog", "table_schema", "table_name", "table_type", "is_transient",
                         "retention_time", "deleted"],
    "snowflake_policy_references": ["policy_name", "policy_kind", "ref_database_name", "ref_schema_name",
                                    "ref_entity_name", "ref_entity_domain", "ref_column_name",
"policy_status"],
    "databricks_history": ["table_name", "version", "timestamp", "operation"],
    "databricks_table_properties": ["table_name", "deleted_file_retention_days", "deletion_vectors"],
    "databricks_column_masks": ["catalog_name", "schema_name", "table_name", "column_name", "mask_catalog",
                                "mask_schema", "mask_name"],
    "redshift_sys_query_history": ["user_id", "query_id", "query_type", "status", "start_time",
"query_text"],
    "redshift_sys_vacuum_history": ["schema_name", "table_name", "vacuum_type", "is_automatic", "status",
                                    "start_time", "end_time"],
    "redshift_snapshots": ["snapshotidentifier", "snapshotcreatetime", "snapshottype",
                           "manualsnapshotretentionperiod"],
    "redshift_attached_masking_policy": ["policy_name", "schema_name", "table_name", "grantee",
"grantee_type",
                                         "output_columns"],
}
PLATFORMS = ["snowflake", "databricks", "redshift"]
OUT_DIR = pathlib.Path("evidence-out")
GEN_DIR = pathlib.Path("generated")
TS_RE = "\\d{4}-\\d{2}-\\d{2}[ T]\\d{2}:\\d{2}:\\d{2}(\\.\\d+)?( ?(Z|\\+00:00|\\+0000))?"
DATE_RE = "\\d{4}-\\d{2}-\\d{2}"
IDENT = re.compile(r"^[A-Za-z_][A-Za-z0-9_$]*$")
NAME = re.compile(r"^[A-Za-z0-9_$]+(\.[A-Za-z0-9_$]+)*$")


def ts(col):
    """A timestamp column read strictly: YYYY-MM-DD HH:MM:SS, taken as UTC (a Z, +00:00 or +0000 suffix is
    accepted, any other offset is NULL and counted). Export from a session set to UTC."""
    return (f"CASE WHEN regexp_full_match({col}, '{TS_RE}') THEN "
            f"try_strptime(substr(replace({col}, 'T', ' '), 1, 19), '%Y-%m-%d %H:%M:%S') END")


def day(col):
    return (f"CASE WHEN regexp_full_match({col}, '{DATE_RE}') THEN try_strptime({col}, '%Y-%m-%d')::DATE
END")


def whole(col):
    """A whole number, or NULL (so '7.5', '1e3' and 'seven' are counted as unreadable, not guessed at)."""
    return f"CASE WHEN regexp_full_match({col}, '-?\\d+') THEN {col}::BIGINT END"


def nm(col):
    """A table or column name compared whole, case-insensitively, without double quotes or backticks."""
    return f"lower(replace(replace({col}, '\"', ''), '`', ''))"


def fingerprint(rows):
    body = "\n".join("|".join("" if v is None else str(v) for v in r) for r in rows)
    return hashlib.sha256(body.encode()).hexdigest()


def die(msg):
    print(f"STOP: {msg}")
```

```
    sys.exit(2)


def row_age_query(plat, table, col, days):
    """The query that measures row ages: no platform view holds them. Run it on each register table and
    save every answer as a row of row_ages.csv."""
    cutoff = {"snowflake": f"DATEADD(day, -{days}, CURRENT_DATE)", "redshift": f"DATEADD(day, -{days},
CURRENT_DATE)",
              "databricks": f"date_sub(current_date(), {days})"}[plat]
    return (f"SELECT '{plat}' AS platform, '{table}' AS table_name, {days} AS retention_days_used,\n"
            f"       COUNT(*) AS rows_total,\n"
            f"       COALESCE(SUM(CASE WHEN {col} < {cutoff} THEN 1 ELSE 0 END), 0) AS
rows_past_retention,\n"
            f"       COALESCE(SUM(CASE WHEN {col} IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,\n"
            f"       CAST(MIN({col}) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on\n"
            f"FROM {table};")


# ---------------------------------------------------------------------------------------------
# Demo records: generated, in each platform's published column names
# ---------------------------------------------------------------------------------------------
def generate(plant, plant_input):
    rnd = random.Random(231)
    out = GEN_DIR / ("planted-input" if plant_input else "planted" if plant else "clean")
    out.mkdir(parents=True, exist_ok=True)
    D, T = dt.date, dt.datetime
    start, as_of = D(2026, 9, 16), D(2026, 10, 15)
    rows = {k: [] for k in FILES}
    stamp = lambda t: t.strftime("%Y-%m-%d %H:%M:%S")
    at = lambda y, m, d, h=0, mi=0, s=0: T(y, m, d, h, mi, s)

    reg = [("snowflake", "sf_prod.sales.customer_contacts", "contact", 730, "created_at", 14),
           ("snowflake", "sf_prod.sales.web_events", "event", 90, "event_date", 14),
           ("snowflake", "sf_prod.hr.employee_notes", "hr", 365, "noted_on", 14),
           ("databricks", "main.sales.customer_contacts", "contact", 730, "created_at", 14),
           ("databricks", "main.sales.web_events", "event", 90, "event_date", 14),
           ("redshift", "sales.customer_contacts", "contact", 730, "created_at", 14),
           ("redshift", "sales.web_events", "event", 90, "event_date", 14)]
    rows["retention_register"] = [list(r) for r in reg]
    # row ages: measured on AS_OF, with the register's own retention
    oldest = {"sf_prod.sales.customer_contacts": "2024-10-19", "sf_prod.sales.web_events": "2026-07-20",
              "sf_prod.hr.employee_notes": "2025-10-18", "main.sales.customer_contacts": "2024-10-21",
              "main.sales.web_events": "2026-07-22", "sales.customer_contacts": "2024-10-22",
              "sales.web_events": "2026-07-19"}
    for p, t, c, ret, col, allowed in reg:
        total = rnd.randrange(40_000, 400_000)
        rows["row_ages"].append([p, t, ret, total, 0, 0, oldest[t], "2026-10-15"])
    if plant:
        for r in rows["row_ages"]:
            if r[1] == "sf_prod.sales.web_events":
                r[4], r[6] = 1204, "2026-06-30"
            if r[1] == "main.sales.web_events":
                r[4], r[6] = 37, "2026-07-02"
        rows["row_ages"] = [r for r in rows["row_ages"] if r[1] != "sales.customer_contacts"]   # never
measured
    rows["masking_policy"] = [["contact", "SUPPORT_LEAD;DATA_STEWARD"], ["hr", "HR_ADMIN"],
                              ["network", "SECURITY_ANALYST"]]
    tags = [("snowflake", "sf_prod.sales.customer_contacts", "EMAIL", "contact"),
            ("snowflake", "sf_prod.sales.customer_contacts", "PHONE", "contact"),
            ("snowflake", "sf_prod.hr.employee_notes", "NOTE_TEXT", "hr"),
            ("snowflake", "sf_prod.sales.web_events", "IP_ADDRESS", "network"),
            ("databricks", "main.sales.customer_contacts", "email", "contact"),
            ("databricks", "main.sales.customer_contacts", "phone", "contact"),
            ("databricks", "main.sales.web_events", "ip_address", "network"),
```

```
            ("redshift", "sales.customer_contacts", "email", "contact"),
            ("redshift", "sales.customer_contacts", "phone", "contact"),
            ("redshift", "sales.web_events", "ip_address", "network")]
    rows["column_tags"] = [list(t) for t in tags]
    rows["clear_readers"] = [["snowflake", "sf_prod.sales.customer_contacts", "EMAIL", "SUPPORT_LEAD"],
                             ["snowflake", "sf_prod.hr.employee_notes", "NOTE_TEXT", "HR_ADMIN"],
                             ["databricks", "main.sales.customer_contacts", "email", "data_steward"],
                             ["redshift", "sales.web_events", "ip_address", "security_analyst"]]
    if plant:
        rows["clear_readers"].append(["redshift", "sales.customer_contacts", "email", "marketing_analyst"])

    # Snowflake: ACCOUNT_USAGE.TABLES and POLICY_REFERENCES
    sf = [["SF_PROD", "SALES", "CUSTOMER_CONTACTS", "BASE TABLE", "NO", 7, ""],
          ["SF_PROD", "SALES", "WEB_EVENTS", "BASE TABLE", "YES", 1, ""],
          ["SF_PROD", "HR", "EMPLOYEE_NOTES", "BASE TABLE", "NO", 90 if plant else 1, ""],
          ["SF_PROD", "SALES", "AUDIT_LOG", "BASE TABLE", "NO", 1, ""],
          ["SF_PROD", "SALES", "WEB_EVENTS", "BASE TABLE", "NO", 90, "2026-08-02 09:00:00.000 +0000"]]  #
dropped copy
    rows["snowflake_tables"] = sf
    pr = [["PII_MASK", "MASKING_POLICY", "SF_PROD", "SALES", "CUSTOMER_CONTACTS", "TABLE", "EMAIL",
"ACTIVE"],
          ["PII_MASK", "MASKING_POLICY", "SF_PROD", "SALES", "CUSTOMER_CONTACTS", "TABLE", "PHONE",
"ACTIVE"],
          ["HR_MASK", "MASKING_POLICY", "SF_PROD", "HR", "EMPLOYEE_NOTES", "TABLE", "NOTE_TEXT", "ACTIVE"],
          ["IP_MASK", "MASKING_POLICY", "SF_PROD", "SALES", "WEB_EVENTS", "TABLE", "IP_ADDRESS", "ACTIVE"],
          ["RAP_REGION", "ROW_ACCESS_POLICY", "SF_PROD", "SALES", "WEB_EVENTS", "TABLE", "", "ACTIVE"]]
    if plant:
        pr = [r for r in pr if r[6] != "PHONE"]
    rows["snowflake_policy_references"] = pr

    # Databricks: DESCRIBE HISTORY per table, table properties, information_schema.column_masks
    props = {"main.sales.customer_contacts": "30" if plant else "7", "main.sales.web_events": ""}
    for t in ("main.sales.customer_contacts", "main.sales.web_events"):
        h = [(0, at(2026, 6, 1, 8), "CREATE TABLE"), (1, at(2026, 9, 1, 8), "WRITE"),
             (2, at(2026, 9, 20, 10), "DELETE"), (3, at(2026, 9, 28, 3), "VACUUM START"),
             (4, at(2026, 9, 28, 3, 4, 10), "VACUUM END"), (5, at(2026, 10, 1, 8), "WRITE"),
             (6, at(2026, 10, 8, 10), "DELETE"), (7, at(2026, 10, 12, 4), "OPTIMIZE")]
        if plant_input and t.endswith("web_events"):
            h = [x for x in h if x[0] >= 2]          # history starts after the period began
            h.append((8, at(2026, 10, 13, 5), "VACUUM FULL"))
        for v, when, op in h:
            rows["databricks_history"].append([t, v, stamp(when) + ".000", op])
    rows["databricks_table_properties"] = [[t, v, "false"] for t, v in props.items()]   #
delta.enableDeletionVectors
    cm = [["main", "sales", "customer_contacts", "email", "main", "gov", "mask_email"],
          ["main", "sales", "customer_contacts", "phone", "main", "gov", "mask_phone"],
          ["main", "sales", "web_events", "ip_address", "main", "gov", "mask_ip"]]
    if plant:
        cm = [r for r in cm if r[3] != "ip_address"]
    rows["databricks_column_masks"] = cm

    # Redshift: SYS_QUERY_HISTORY, SYS_VACUUM_HISTORY, snapshots, SVV_ATTACHED_MASKING_POLICY
    q, qid = [], 1000
    d = start
    while d <= as_of:
        if not (plant_input and d == D(2026, 10, 3)):
            for h in (7, 11, 15):
                qid += 1
                q.append([100 + rnd.randrange(5), qid, "SELECT", "success",
                          stamp(T(d.year, d.month, d.day, h, rnd.randrange(60), rnd.randrange(60))),
                          "select count(*) from sales.web_events"])
        d += dt.timedelta(days=1)
    q.append([101, 2001, "DELETE", "success", "2026-09-20 11:00:00",
              "delete from sales.customer_contacts where created_at < '2024-09-20'"])
```

```
    q.append([101, 2002, "DELETE", "success", "2026-09-20 11:05:00",
              '/* nightly purge */ DELETE FROM "sales"."web_events" WHERE event_date < \'2026-06-20\' -- 90
days'])
    q.append([101, 2003, "DELETE", "failed", "2026-09-22 11:00:00",
              "delete from sales.web_events where event_date < '2026-06-22'"])
    if plant_input:
        q.append([101, 2004, "OTHER", "success", "2026-10-06 11:00:00",
                  "with old as (select 1) delete from sales.web_events where 1 = 1"])
    rows["redshift_sys_query_history"] = q
    rows["redshift_sys_vacuum_history"] = [
        ["sales", "customer_contacts", "Delete", "true", "Complete", "2026-09-21 02:00:00", "2026-09-21
02:20:00"],
        ["sales", "web_events", "Delete", "true", "Complete", "2026-09-21 02:20:00", "2026-09-21 02:41:00"],
        ["sales", "web_events", "Sort", "true", "Complete", "2026-10-02 02:00:00", "2026-10-02 02:30:00"]]
    snaps = [["rs-auto-1014", "2026-10-14 03:00:00", "automated", ""],
             ["rs-auto-1015", "2026-10-15 03:00:00", "automated", ""],
             ["rs-manual-0918", "2026-09-18 09:00:00", "manual", "-1" if plant else "10"]]
    if plant_input:
        snaps.append(["rs-manual-bad", "yesterday", "manual", "ten"])
    rows["redshift_snapshots"] = snaps
    mp = [["pii_mask", "sales", "customer_contacts", "public", "public", "{email}"],
          ["pii_mask", "sales", "customer_contacts", "public", "public", "{phone}"],
          ["ip_mask", "sales", "web_events", "public", "public", "{ip_address}"]]
    rows["redshift_attached_masking_policy"] = [] if plant_input else mp
    if plant_input:
        rows["masking_policy"].append(["financial", "FINANCE_LEAD"])
        rows["column_tags"].append(["snowflake", "sf_prod.sales.customer_contacts", "ssn", ""])
        rows["clear_readers"].append(["snowflake", "sf_prod.sales.customer_contacts", "DOB", "SUPPORT_LEAD"])
        rows["clear_readers"].append(["snowflake", "sf_prod.sales.customer_contacts", "EMAIL", ""])
        rows["row_ages"].append(["redshift", "sales.old_staging", 90, 10, 0, 0, "2026-09-01", "2026-10-15"])
        for r in rows["row_ages"]:
            if r[1] == "sf_prod.sales.customer_contacts":
                r[2] = 365
            if r[1] == "sf_prod.hr.employee_notes":
                r[7], r[5] = "2026-09-01", 4
        rows["retention_register"].append(["redshift", "sales.customer_contacts", "contact", 730,
"created_at", "14.5"])
        rows["column_tags"].append(["oracle", "erp.hr.pay", "salary", "hr"])
        rows["row_ages"].append(["snowflake", "sf_prod.sales.web_events", 90, 105751, 0, 0, "2026-07-20",
"2026-10-15"])
        for r in rows["row_ages"]:
            if r[1] == "main.sales.customer_contacts":
                r[3] = "many"
        rows["snowflake_tables"].append(["SF_PROD", "SALES", "AUDIT_LOG_OLD", "BASE TABLE", "MAYBE", "seven",
""])
        rows["snowflake_tables"].append(["SF_PROD", "SALES", "WEB_EVENTS", "BASE TABLE", "YES", 1, ""])
        rows["databricks_history"].append(["main.sales.customer_contacts", 9, "last tuesday", "WRITE"])
        rows["databricks_table_properties"][1][1] = "7 days"
        rows["databricks_table_properties"][0][2] = "maybe"

    for key, data in rows.items():
        with open(out / FILES[key], "w", newline="") as f:
            w = csv.writer(f)
            w.writerow(COLUMNS[key] if key not in ("snowflake_tables",) else
                       [c.upper() for c in COLUMNS[key]])
            w.writerows(data)
    return out


# ---------------------------------------------------------------------------------------------
# Reading the exports
# ---------------------------------------------------------------------------------------------
def load(con, key, path):
    if not path.is_file():
```

```
        die(f"{path} is missing (set EXPORT_DIR, or change FILES['{key}'])")
    with open(path, newline="", encoding="utf-8-sig") as f:
        header = next(csv.reader(f), [])
    have = {h.strip().lower(): h for h in header}
    missing = [c for c in COLUMNS[key] if c not in have]
    if missing:
        die(f"{path.name} has no column {', '.join(missing)} (it has: {', '.join(header)})")
    sel = ", ".join(f"NULLIF(trim(\"{have[c]}\"), '') AS {c}" for c in COLUMNS[key])
    con.execute(f"CREATE OR REPLACE TABLE {key} AS SELECT {sel} FROM "
                f"read_csv(?, header=true, all_varchar=true, sample_size=-1)", [str(path)])


def build(con, directory):
    for key in ["retention_register", "row_ages", "masking_policy", "column_tags", "clear_readers"] + \
            [k for p in PLATFORMS_EXPORTED for k in PLATFORM_FILES[p]]:
        load(con, key, directory / FILES[key])
    P = ", ".join(f"'{p}'" for p in PLATFORMS_EXPORTED)
    run = lambda sql: con.execute(sql)
    run(f"""CREATE TABLE reg AS SELECT lower(platform) AS platform, {nm('table_name')} AS t, table_name AS
raw_name,
        lower(data_category) AS category, {whole('retention_days')} AS ret,
{whole('recoverable_days_allowed')} AS allowed,
        age_column FROM retention_register WHERE lower(platform) IN ({P})""")
    run(f"""CREATE TABLE ages AS SELECT lower(platform) AS platform, {nm('table_name')} AS t,
        {whole('retention_days_used')} AS ret_used, {whole('rows_total')} AS total,
{whole('rows_past_retention')} AS past,
        {whole('rows_no_age')} AS no_age, {day('oldest_date')} AS oldest, {day('measured_on')} AS measured,
        oldest_date AS oldest_raw FROM row_ages WHERE lower(platform) IN ({P})""")
    # one line per category: roles from every line for the same category are merged
    run("""CREATE TABLE pol AS SELECT lower(data_category) AS category, string_agg(coalesce(roles_in_clear,
''), ';'
        ORDER BY coalesce(roles_in_clear, '')) AS roles FROM masking_policy GROUP BY 1""")
    run(f"""CREATE TABLE tags AS SELECT DISTINCT lower(platform) AS platform, {nm('table_name')} AS t,
{nm('column_name')} AS col,
        lower(data_category) AS category FROM column_tags WHERE lower(platform) IN ({P})""")
    run(f"""CREATE TABLE clr AS SELECT lower(platform) AS platform, {nm('table_name')} AS t,
{nm('column_name')} AS col,
        {nm('role_name')} AS role FROM clear_readers WHERE lower(platform) IN ({P})""")
    if "snowflake" in PLATFORMS_EXPORTED:
        run(f"""CREATE TABLE sf_t AS SELECT {nm("table_catalog || '.' || table_schema || '.' || table_name")}
AS t,
            upper(is_transient) AS transient, upper(coalesce(table_type, '')) AS ttype,
{whole('retention_time')} AS rt,
            deleted FROM snowflake_tables""")
        run(f"""CREATE TABLE sf_pr AS SELECT upper(policy_kind) AS kind, upper(policy_status) AS status,
            {nm("ref_database_name || '.' || ref_schema_name || '.' || ref_entity_name")} AS t,
            {nm('ref_column_name')} AS col FROM snowflake_policy_references""")
    if "databricks" in PLATFORMS_EXPORTED:
        run(f"""CREATE TABLE db_h AS SELECT {nm('table_name')} AS t, {whole('version')} AS version,
{ts('timestamp')} AS ts,
            timestamp AS ts_raw, upper(operation) AS op FROM databricks_history""")
        run(f"""CREATE TABLE db_p AS SELECT {nm('table_name')} AS t, deleted_file_retention_days AS raw,
            {whole('deleted_file_retention_days')} AS days, lower(deletion_vectors) AS dv FROM
databricks_table_properties""")
        run(f"""CREATE TABLE db_m AS SELECT {nm("catalog_name || '.' || schema_name || '.' || table_name")}
AS t,
            {nm('column_name')} AS col FROM databricks_column_masks""")
    if "redshift" in PLATFORMS_EXPORTED:
        run(f"""CREATE TABLE rs_q AS SELECT lower(status) AS status, {ts('start_time')} AS ts, start_time AS
ts_raw,
            query_text FROM redshift_sys_query_history""")
        run(f"""CREATE TABLE rs_v AS SELECT {nm("schema_name || '.' || table_name")} AS t, lower(vacuum_type)
AS vtype,
            lower(status) AS status, {ts('start_time')} AS s_ts, {ts('end_time')} AS e_ts
```

```
            FROM redshift_sys_vacuum_history""")
        run(f"""CREATE TABLE rs_s AS SELECT {ts('snapshotcreatetime')} AS created, lower(snapshottype) AS
stype,
            {whole('manualsnapshotretentionperiod')} AS ret, snapshotcreatetime AS created_raw,
            manualsnapshotretentionperiod AS ret_raw FROM redshift_snapshots""")
        run(f"""CREATE TABLE rs_m AS SELECT {nm("schema_name || '.' || table_name")} AS t, output_columns AS
outc,
            lower(grantee) AS grantee, lower(grantee_type) AS gtype FROM redshift_attached_masking_policy""")


def input_checks(con):
    ds, de = f"DATE '{PERIOD_START}'", f"DATE '{AS_OF}'"
    n = lambda sql: con.execute(sql).fetchone()[0]
    out = []   # (label, number of bad rows)
    add = lambda label, sql: out.append((label, n(sql)))
    known = "('snowflake', 'databricks', 'redshift')"
    add("every platform name in your lists is snowflake, databricks or redshift",
        f"""SELECT count(*) FROM (SELECT lower(platform) p FROM retention_register UNION ALL SELECT
lower(platform) FROM row_ages
            UNION ALL SELECT lower(platform) FROM column_tags UNION ALL SELECT lower(platform) FROM
clear_readers)
            WHERE p IS NULL OR p NOT IN {known}""")
    add("no table appears twice in the retention register",
        "SELECT count(*) FROM (SELECT platform, t FROM reg GROUP BY 1, 2 HAVING count(*) > 1)")
    add("every register line has a category, whole-number days, and a date column that is a plain name",
        """SELECT count(*) FROM reg WHERE category IS NULL OR ret IS NULL OR allowed IS NULL OR ret < 0 OR
allowed < 0
           OR age_column IS NULL OR NOT regexp_full_match(age_column, '[A-Za-z_][A-Za-z0-9_$]*')""")
    add("every column tag has a table, a column and a category",
        "SELECT count(*) FROM tags WHERE t IS NULL OR col IS NULL OR category IS NULL")
    add("every category in the masking policy is on at least one tagged column",
        "SELECT count(*) FROM pol WHERE category NOT IN (SELECT category FROM tags WHERE category IS NOT
NULL)")
    add("every clear reader names a table, a column and a role",
        "SELECT count(*) FROM clr WHERE t IS NULL OR col IS NULL OR role IS NULL")
    add("every clear reader is on a column that is tagged",
        "SELECT count(*) FROM clr WHERE NOT EXISTS (SELECT 1 FROM tags g WHERE g.platform = clr.platform AND
g.t = clr.t AND g.col = clr.col)")
    add("every row-age line is for a table in the register",
        "SELECT count(*) FROM ages WHERE NOT EXISTS (SELECT 1 FROM reg WHERE reg.platform = ages.platform AND
reg.t = ages.t)")
    add("no table has two row-age lines",
        "SELECT count(*) FROM (SELECT platform, t FROM ages GROUP BY 1, 2 HAVING count(*) > 1)")
    add("row-age counts are whole numbers and the oldest date is YYYY-MM-DD (or blank for an empty table)",
        """SELECT count(*) FROM ages WHERE total IS NULL OR past IS NULL OR no_age IS NULL OR measured IS
NULL
           OR (oldest IS NULL AND (oldest_raw IS NOT NULL OR total > 0))""")
    add("row ages were made with the same retention days as the register",
        "SELECT count(*) FROM ages a JOIN reg r ON r.platform = a.platform AND r.t = a.t WHERE a.ret_used IS
DISTINCT FROM r.ret")
    add(f"row ages were measured within {ROW_AGES_MAX_STALE_DAYS} days before AS_OF, not after it",
        f"SELECT count(*) FROM ages WHERE measured IS NOT NULL AND (measured > {de} OR measured < {de} -
{int(ROW_AGES_MAX_STALE_DAYS)})")
    add("no register table has rows with no date to age them by",
        "SELECT count(*) FROM ages WHERE no_age > 0")
    if "snowflake" in PLATFORMS_EXPORTED:
        add("Snowflake: retention_time is a whole number and is_transient is YES or NO",
            "SELECT count(*) FROM sf_t WHERE rt IS NULL OR transient IS NULL OR transient NOT IN ('YES',
'NO')")
        add("Snowflake: no register table has two live (not dropped) rows",
            "SELECT count(*) FROM (SELECT t FROM sf_t WHERE deleted IS NULL AND t IN (SELECT t FROM reg WHERE
platform = 'snowflake') GROUP BY 1 HAVING count(*) > 1)")
    if "databricks" in PLATFORMS_EXPORTED:
        add("Databricks: every history timestamp is YYYY-MM-DD HH:MM:SS",
```

```
            "SELECT count(*) FROM db_h WHERE ts IS NULL OR version IS NULL")
        done, other = DATABRICKS_VACUUM_DONE.upper(), [o.upper() for o in DATABRICKS_VACUUM_OTHER]
        known_ops = ", ".join(f"'{o}'" for o in [done] + other)
        add("Databricks: every VACUUM operation in the history is one the settings name",
            f"SELECT count(*) FROM db_h WHERE op LIKE 'VACUUM%' AND op NOT IN ({known_ops})")
        add("Databricks: each register table's history reaches back to the period's start (or to its first
version)",
            f"""SELECT count(*) FROM (SELECT t, min(ts) mn, min(version) v FROM db_h WHERE t IN
                (SELECT t FROM reg WHERE platform = 'databricks') GROUP BY t) WHERE mn > {ds} AND v <> 0""")
        add("Databricks: a table's deleted-file retention is a whole number of days or blank",
            "SELECT count(*) FROM db_p WHERE raw IS NOT NULL AND (days IS NULL OR days < 0)")
        add("Databricks: every register table says whether deletion vectors are on (true or false)",
            """SELECT count(*) FROM reg r LEFT JOIN db_p p ON p.t = r.t WHERE r.platform = 'databricks'
               AND (p.dv IS NULL OR p.dv NOT IN ('true', 'false'))""")
    if "redshift" in PLATFORMS_EXPORTED:
        add("Redshift: every query-history, vacuum and snapshot timestamp is YYYY-MM-DD HH:MM:SS",
            """SELECT (SELECT count(*) FROM rs_q WHERE ts IS NULL) + (SELECT count(*) FROM rs_v WHERE s_ts IS
NULL
               OR (status = 'complete' AND e_ts IS NULL)) + (SELECT count(*) FROM rs_s WHERE created IS
NULL)""")
        gaps = KNOWN_GAPS.get("redshift", [])
        skip = ", ".join(f"DATE '{g}'" for g in gaps) or "DATE '1900-01-01'"
        add("Redshift: every day from the period's start to AS_OF has query history, except KNOWN_GAPS",
            f"""SELECT count(*) FROM generate_series({ds}, {de}, INTERVAL 1 DAY) g(d) WHERE d::DATE NOT IN
({skip})
                AND NOT EXISTS (SELECT 1 FROM rs_q WHERE ts::DATE = g.d::DATE)""")
        add("Redshift: snapshot type is manual or automated, and a manual retention is -1 or whole days from
1",
            """SELECT count(*) FROM rs_s WHERE stype IS NULL OR stype NOT IN ('manual', 'automated')
               OR (stype = 'manual' AND (ret IS NULL OR (ret <> -1 AND ret < 1)))""")
        add("Redshift: REDSHIFT_AUTOMATED_RETENTION_DAYS is set when automated snapshots exist",
            f"SELECT CASE WHEN {REDSHIFT_AUTOMATED_RETENTION_DAYS is None} AND EXISTS (SELECT 1 FROM rs_s
WHERE stype = 'automated') THEN 1 ELSE 0 END")
        add("Redshift: the masking-policy export has rows (a user who is not a superuser or sys:secadmin sees
0)",
            "SELECT CASE WHEN (SELECT count(*) FROM rs_m) = 0 THEN 1 ELSE 0 END")
    return out


# ---------------------------------------------------------------------------------------------
# The three lists
# ---------------------------------------------------------------------------------------------
def clean_sql(text):
    text = re.sub(r"/\*.*?\*/", " ", text or "", flags=re.S)
    text = re.sub(r"--[^\n]*", " ", text)
    return text.replace('"', "").replace("`", "")


def redshift_deletes(con):
    """Successful DELETE statements by target (schema.table); FROM is optional in Redshift. A DELETE that
does not
    start the statement (a CTE in front), or whose target names no schema, cannot be matched to the register
for
    certain; when its table name is the table part of a Redshift register entry it is counted for the input
check,
    never skipped. One on a table the register cannot hold (a staging table) is left out."""
    found, unreadable = [], 0
    bare = {r[0].split(".")[-1] for r in con.execute("SELECT t FROM reg WHERE platform = 'redshift' AND t IS
NOT NULL").fetchall()}
    for status, when, text in con.execute("SELECT status, ts, query_text FROM rs_q ORDER BY ts,
query_text").fetchall():
        c = re.sub(r"'(?:[^']|'')*'", "''", clean_sql(text))   # string literals cannot name a table
        m = re.match(r"\s*delete\s+(?:from\s+)?(?:only\s+)?([\w$.]+)", c, re.I)
        if status != "success" or when is None:   # an unreadable time is counted by the timestamp check
```

```
            continue
        if m and "." in m.group(1):
            parts = m.group(1).lower().split(".")
            found.append((".".join(parts[-2:]), when))
        else:   # GRANT ... DELETE ON, REVOKE DELETE and the like are not deletes
            w = re.search(r"\bdelete\s+(?:from\s+)?(?:only\s+)?([\w$.]+)", c, re.I) if re.match(r"\s*with\b",
c, re.I) else None
            target = (m or w).group(1).lower() if (m or w) else None
            if target and target.split(".")[-1] in bare:
                unreadable += 1
    return found, unreadable


def list_rows_past(con):
    rows = con.execute("""SELECT r.platform, r.raw_name, r.category, r.ret, a.total, a.past, a.oldest,
        CASE WHEN a.t IS NULL OR a.past IS NULL THEN 'NOT MEASURED' WHEN a.past > 0 THEN 'PAST RETENTION'
ELSE 'OK' END
        FROM reg r LEFT JOIN ages a ON a.platform = r.platform AND a.t = r.t ORDER BY 1, 2""").fetchall()
    return ["platform", "table", "category", "retention_days", "rows", "rows_past_retention", "oldest_date",
"status"], rows


def list_recoverable(con, as_of_end, redshift_dels):
    rows = []
    reg = con.execute("SELECT platform, t, raw_name, category, allowed FROM reg ORDER BY 1, 2").fetchall()
    rank = {"INDEFINITE SNAPSHOT": 4, "RECOVERABLE BEYOND POLICY": 3, "OPEN, WITHIN SO FAR": 2, "WITHIN
POLICY": 1}
    for plat, t, raw, cat, allowed in reg:
        days, basis, status = None, "", ""
        if allowed is None:
            rows.append((plat, raw, cat, allowed, None, "recoverable_days_allowed is not a whole number",
"NOT MEASURED"))
            continue
        if plat == "snowflake":
            hit = con.execute("SELECT rt, transient, ttype FROM sf_t WHERE t = ? AND deleted IS NULL",
[t]).fetchall()
            if not hit:
                status, basis = "NOT MEASURED", "table not in the ACCOUNT_USAGE.TABLES export"
            elif hit[0][0] is None:
                status, basis = "NOT MEASURED", "retention_time is not a whole number"
            else:
                rt, transient, ttype = hit[0]
                fs = FAILSAFE_DAYS if (transient == "NO" and "TEMPORARY" not in ttype and
SNOWFLAKE_COUNT_FAILSAFE) else 0
                days = rt + fs
                basis = (f"Time Travel {rt} d + Fail-safe {FAILSAFE_DAYS} d" if fs else
                         f"Time Travel {rt} d, transient: no Fail-safe" if transient == "YES" else
                         f"Time Travel {rt} d, temporary: no Fail-safe" if "TEMPORARY" in ttype else f"Time
Travel {rt} d only")
                status = "RECOVERABLE BEYOND POLICY" if days > allowed else "WITHIN POLICY"
        elif plat == "databricks":
            if not con.execute("SELECT count(*) FROM db_h WHERE t = ?", [t]).fetchone()[0]:
                status, basis = "NOT MEASURED", "table not in the DESCRIBE HISTORY export"
            else:
                prop = con.execute("SELECT days, dv FROM db_p WHERE t = ?", [t]).fetchone()
                keep = prop[0] if prop and prop[0] is not None else DATABRICKS_DEFAULT_RETENTION_DAYS
                dv = prop[1] if prop else None
                done = DATABRICKS_VACUUM_DONE.upper()
                purge = [o.upper() for o in DATABRICKS_PURGE_OPS]
                dels = con.execute(
                    "SELECT ts FROM db_h WHERE t = ? AND op = 'DELETE' AND ts >= ? AND ts <= ? ORDER BY ts",
                    [t, PERIOD_START + " 00:00:00", as_of_end.strftime("%Y-%m-%d %H:%M:%S")]).fetchall()
                best = None
                for (d,) in dels:
```

```
                    # with deletion vectors a DELETE only marks rows: a REORG ... APPLY (PURGE) must rewrite
the files
                    # first, and the VACUUM that removes the old files must come at least the retention after
that
                    start = d
                    if dv == "true":
                        hit = con.execute(f"SELECT ts, op FROM db_h WHERE t = ? AND op IN ({', '.join('?' for
_ in purge)}) AND ts >= ? ORDER BY ts LIMIT 1",
                                          [t, *purge, d]).fetchone()
                        start, purge_op = (hit[0], hit[1]) if hit else (None, None)
                    v = None if start is None else con.execute("SELECT min(ts) FROM db_h WHERE t = ? AND op =
? AND ts >= ?",
                                                               [t, done, start +
dt.timedelta(days=keep)]).fetchone()[0]
                    if dv not in ("true", "false"):
                        val, st, b = None, "unknown", f"delete {d:%Y-%m-%d}; deletion_vectors not stated, so
removal cannot be judged"
                    elif v:
                        val, st = (v - d).total_seconds() / 86400, None
                        b = f"delete {d:%Y-%m-%d}, " + (f"{purge_op} {start:%Y-%m-%d}, " if dv == "true" else
"") + f"VACUUM finished {v:%Y-%m-%d}, files kept {keep} d"
                    elif dv == "true" and start is None:
                        val, st, b = (as_of_end - d).total_seconds() / 86400, "open", f"delete {d:%Y-%m-%d},
deletion vectors on and no " + "/".join(purge) + " since: rows still in live files"
                    else:
                        val, st, b = (as_of_end - d).total_seconds() / 86400, "open", f"delete {d:%Y-%m-%d},
no VACUUM yet old enough to remove its files (kept {keep} d)"
                    if st == "unknown":
                        cand = (0, 0.0, "NOT MEASURED", b)
                        best = cand if best is None else best
                        continue
                    st = ("RECOVERABLE BEYOND POLICY" if val > allowed else "OPEN, WITHIN SO FAR" if st else
"WITHIN POLICY")
                    cand = (rank[st], val, st, b)
                    best = cand if best is None or cand[:2] > best[:2] else best
                if best is None:
                    status, basis = "NO DELETE IN PERIOD", "no DELETE in the history during the period"
                else:
                    _, days, status, basis = best
            else:
                mine = [d for tt, d in redshift_dels if tt == t and PERIOD_START <= f"{d:%Y-%m-%d}" <= AS_OF]
                if not mine:
                    status, basis = "NO DELETE IN PERIOD", "no successful DELETE in the query history during the
period"
                else:
                    auto = REDSHIFT_AUTOMATED_RETENTION_DAYS or 0
                    # a snapshot whose time or manual retention cannot be read is counted by an input check, not
used here
                    snaps = con.execute("""SELECT created, stype, ret FROM rs_s WHERE created IS NOT NULL
                                           AND NOT (stype = 'manual' AND ret IS NULL)""").fetchall()
                    best = None
                    for d in sorted(mine):
                        v = con.execute("SELECT min(e_ts) FROM rs_v WHERE t = ? AND status = 'complete' AND vtype
IN ('delete', 'full') AND s_ts >= ?",
                                        [t, d]).fetchone()[0]
                        indef = [c for c, ty, r in snaps if ty == "manual" and r == -1 and c < d]
                        exp = [c + dt.timedelta(days=(r if ty == "manual" else auto)) for c, ty, r in snaps
                               if c < d and not (ty == "manual" and r == -1) and c + dt.timedelta(days=(r if ty
== "manual" else auto)) > d]
                        if indef:
                            cand = (rank["INDEFINITE SNAPSHOT"], 0.0, "INDEFINITE SNAPSHOT",
                                    f"delete {d:%Y-%m-%d}; a manual snapshot from {min(indef):%Y-%m-%d} is kept
indefinitely")
                        else:
```

```
                        if v:
                            end = max([v] + exp)
                            val, st = (end - d).total_seconds() / 86400, None
                            b = f"delete {d:%Y-%m-%d}, VACUUM DELETE finished {v:%Y-%m-%d}" + (f", snapshot
kept to {max(exp):%Y-%m-%d}" if exp else "")
                        else:
                            val, st = (max([as_of_end] + exp) - d).total_seconds() / 86400, "open"
                            b = f"delete {d:%Y-%m-%d}, no VACUUM DELETE since"
                        st = ("RECOVERABLE BEYOND POLICY" if val > allowed else "OPEN, WITHIN SO FAR" if st
else "WITHIN POLICY")
                        cand = (rank[st], val, st, b)
                    best = cand if best is None or cand[:2] > best[:2] else best
                _, days, status, basis = best
            if status == "NOT MEASURED":
                days = None
            shown = None if days is None or status == "INDEFINITE SNAPSHOT" else round(days, 1)
            rows.append((plat, raw, cat, allowed, shown, basis, status))
    return ["platform", "table", "category", "allowed_days", "recoverable_days", "basis", "status"], rows


def list_masking(con):
    rows = []
    for plat, t, col, cat, roles in con.execute("""SELECT g.platform, g.t, g.col, g.category, p.roles FROM
tags g
            LEFT JOIN pol p ON p.category = g.category WHERE g.category IS NOT NULL AND g.t IS NOT NULL AND g.col
IS NOT NULL
            ORDER BY 1, 2, 3""").fetchall():
            if roles is None:
                rows.append((plat, t, col, cat, "", "NO POLICY FOR CATEGORY"))
                continue
            active, inactive, partial = False, [], []
            if plat == "snowflake":
                for st, in con.execute("SELECT status FROM sf_pr WHERE kind = 'MASKING_POLICY' AND t = ? AND col
= ?", [t, col]).fetchall():
                    active, inactive = active or st == "ACTIVE", inactive + ([st] if st != "ACTIVE" else [])
            elif plat == "databricks":
                active = con.execute("SELECT count(*) FROM db_m WHERE t = ? AND col = ?", [t, col]).fetchone()[0]
> 0
            else:
                # a policy attached to one role or user masks the column only for that grantee: PUBLIC covers
everyone
                for outc, grantee, gtype in con.execute("SELECT outc, grantee, gtype FROM rs_m WHERE t = ?",
[t]).fetchall():
                    if col in [x.lower() for x in re.findall(r"[A-Za-z0-9_$]+", outc or "")]:
                        if gtype == "public" or grantee == "public":
                            active = True
                        else:
                            partial.append(grantee or "?")
            allowed = {r.strip().lower() for r in roles.split(";") if r.strip()}
            clear = sorted({r for (r,) in con.execute("SELECT role FROM clr WHERE platform = ? AND t = ? AND col
= ? AND role IS NOT NULL", [plat, t, col]).fetchall()})
            extra = [r for r in clear if r not in allowed]
            flags = []
            if not active:
                flags.append("MASK NOT ACTIVE (" + ", ".join(sorted(set(inactive))) + ")" if inactive else
                             "NOT MASKED FOR PUBLIC (attached only to " + ", ".join(sorted(set(partial))) + ")"
if partial else "NOT MASKED")
            if extra:
                flags.append("CLEAR READER NOT ON POLICY LIST: " + ", ".join(extra))
            rows.append((plat, t, col, cat, ", ".join(clear), "; ".join(flags) or "OK"))
    return ["platform", "table", "column", "category", "roles_reading_in_clear", "status"], rows


def show(name, title, cols, rows, filters, incomplete, run_at):
```

```
    rows = sorted(rows, key=lambda r: tuple("" if v is None else str(v) for v in r))
    OUT_DIR.mkdir(exist_ok=True)
    with open(OUT_DIR / f"{name}.csv", "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(cols)
        w.writerows(rows)
    print(f"\n== {title}: {name}" + ("   ** INCOMPLETE: see input checks **" if incomplete else ""))
    print(f"   period {PERIOD_START} to {AS_OF} | {filters} | run {run_at} | rows {len(rows)} | sha256
{fingerprint(rows)}")
    cells = [cols] + [["" if v is None else str(v) for v in r] for r in rows]
    width = [max(len(c[i]) for c in cells) for i in range(len(cols))]
    for k, c in enumerate(cells):
        print("   " + "  ".join(v.ljust(width[i]) for i, v in enumerate(c)).rstrip())
        if k == 0:
            print("   " + "  ".join("-" * w for w in width))


def main():
    global REDSHIFT_AUTOMATED_RETENTION_DAYS
    argv = sys.argv[1:]
    if "--queries" in argv:
        d = generate(False, False) if "--demo" in argv else pathlib.Path(EXPORT_DIR or ".")
        with open(d / FILES["retention_register"], newline="", encoding="utf-8-sig") as f:
            for r in csv.DictReader(f):
                r = {k.strip().lower(): (v or "").strip() for k, v in r.items()}
                p, t, c, n = r["platform"].lower(), r["table_name"], r["age_column"], r["retention_days"]
                if p not in PLATFORMS or not NAME.match(t) or not IDENT.match(c) or not n.isdigit():
                    die(f"cannot write a query for {r}: platform, a plain table name, a plain column name and
whole days are needed")
                print(row_age_query(p, t, c, int(n)), "\n")
        return 0
    plant_input = "--plant-input" in argv
    plant = "--plant" in argv or plant_input
    run_at = dt.datetime.now(dt.timezone.utc).strftime("%Y-%m-%d %H:%M:%SZ")
    if "--demo" in argv:
        directory = generate("--plant" in argv, plant_input)
        REDSHIFT_AUTOMATED_RETENTION_DAYS = None if plant_input else 1
        print(f"Demo: GENERATED records (seed 231) in {directory}/, not a company's data.")
    else:
        if EXPORT_DIR is None:
            die("set EXPORT_DIR (and the other settings) at the top of this file, or run with --demo")
        directory = pathlib.Path(EXPORT_DIR)
    bad_dates = [x for x in (PERIOD_START, AS_OF) if not re.fullmatch(DATE_RE, x)]
    if bad_dates or PERIOD_START > AS_OF:
        die("PERIOD_START and AS_OF must be YYYY-MM-DD, with PERIOD_START not after AS_OF")
    as_of_end = dt.datetime.strptime(AS_OF, "%Y-%m-%d") + dt.timedelta(hours=23, minutes=59, seconds=59)
    con = duckdb.connect()
    build(con, directory)

    print("\n== Input checks")
    checks = input_checks(con)
    redshift_dels, unreadable = [], 0
    if "redshift" in PLATFORMS_EXPORTED:
        redshift_dels, unreadable = redshift_deletes(con)
        checks.append(("Redshift: every DELETE that could touch a register table starts the statement and
names schema.table", unreadable))
    failed = [c for c in checks if c[1]]
    for label, bad in checks:
        print(f"   [{'FAIL' if bad else 'ok'}] {label}" + (f": {bad} row(s)" if bad else ""))
    incomplete = bool(failed)
    not_checked = [p for p in PLATFORMS if p not in PLATFORMS_EXPORTED]
    for p in not_checked:
        print(f"   NOT CHECKED: {p} is not in PLATFORMS_EXPORTED")
```

```
    c1, r1 = list_rows_past(con)
    show("iso_1_rows_past_retention", "Rows past retention", c1, r1,
         "filters: register tables on exported platforms", incomplete, run_at)
    c2, r2 = list_recoverable(con, as_of_end, redshift_dels)
    show("iso_2_deleted_but_recoverable", "Deleted but still recoverable", c2, r2,
         "filters: register tables; Snowflake from table settings, the others from deletes in the period",
incomplete, run_at)
    c3, r3 = list_masking(con)
    show("iso_3_masking_against_policy", "Masking against your policy", c3, r3,
         "filters: columns tagged with a category", incomplete, run_at)

    count = lambda rows, i, s: sum(1 for r in rows if str(r[i]).startswith(s))
    flags = {"past retention": count(r1, 7, "PAST RETENTION"), "not measured": count(r1, 7, "NOT MEASURED") +
count(r2, 6, "NOT MEASURED"),
             "recoverable beyond policy": count(r2, 6, "RECOVERABLE BEYOND"), "indefinite snapshot":
count(r2, 6, "INDEFINITE"),
             "not masked or mask not active": sum(1 for r in r3 if "NOT MASKED" in r[5] or "MASK NOT ACTIVE"
in r[5]),
             "clear reader not on policy list": sum(1 for r in r3 if "CLEAR READER" in r[5]),
             "no policy for category": count(r3, 5, "NO POLICY")}
    total = sum(flags.values())
    print("\n== Result")
    print("   follow-up: " + (", ".join(f"{v} {k}" for k, v in flags.items() if v) or "none"))
    print(f"   input checks: {len(checks) - len(failed)} of {len(checks)} passed" + (
        "; the lists above are INCOMPLETE" if failed else ""))
    if not_checked:
        print("   NOT CHECKED: " + ", ".join(not_checked))
    print(f"   RESULT: {total} item(s) need follow-up; {len(failed)} input check(s) failed; "
          f"{len(not_checked)} platform(s) not checked.")
    return 1 if "--strict" in argv and (total or failed or not_checked) else 0


if __name__ == "__main__":
    sys.exit(main())
```

The runs

Each run is the output the script printed, unedited. The records are generated.

Run 1: the clean generated records (--demo)

```
$ python3 deletion_masking_evidence.py --demo
Demo: GENERATED records (seed 231) in generated/clean/, not a company's data.

== Input checks
   [ok] every platform name in your lists is snowflake, databricks or redshift
   [ok] no table appears twice in the retention register
   [ok] every register line has a category, whole-number days, and a date column that is a plain name
   [ok] every column tag has a table, a column and a category
   [ok] every category in the masking policy is on at least one tagged column
   [ok] every clear reader names a table, a column and a role
   [ok] every clear reader is on a column that is tagged
   [ok] every row-age line is for a table in the register
   [ok] no table has two row-age lines
   [ok] row-age counts are whole numbers and the oldest date is YYYY-MM-DD (or blank for an empty table)
   [ok] row ages were made with the same retention days as the register
   [ok] row ages were measured within 3 days before AS_OF, not after it
   [ok] no register table has rows with no date to age them by
   [ok] Snowflake: retention_time is a whole number and is_transient is YES or NO
   [ok] Snowflake: no register table has two live (not dropped) rows
```

```
   [ok] Databricks: every history timestamp is YYYY-MM-DD HH:MM:SS
   [ok] Databricks: every VACUUM operation in the history is one the settings name
   [ok] Databricks: each register table's history reaches back to the period's start (or to its first
version)
   [ok] Databricks: a table's deleted-file retention is a whole number of days or blank
   [ok] Databricks: every register table says whether deletion vectors are on (true or false)
   [ok] Redshift: every query-history, vacuum and snapshot timestamp is YYYY-MM-DD HH:MM:SS
   [ok] Redshift: every day from the period's start to AS_OF has query history, except KNOWN_GAPS
   [ok] Redshift: snapshot type is manual or automated, and a manual retention is -1 or whole days from 1
   [ok] Redshift: REDSHIFT_AUTOMATED_RETENTION_DAYS is set when automated snapshots exist
   [ok] Redshift: the masking-policy export has rows (a user who is not a superuser or sys:secadmin sees 0)
   [ok] Redshift: every DELETE that could touch a register table starts the statement and names schema.table

== Rows past retention: iso_1_rows_past_retention
   period 2026-09-16 to 2026-10-15 | filters: register tables on exported platforms | run 2026-11-01
13:48:31Z | rows 7 | sha256 c90b08ff10d0b5bfdb512aab47e8236f675186c6323878ae8342100ab3dd84f9
   platform    table                           category  retention_days  rows    rows_past_retention
oldest_date  status
   ----------  ------------------------------  --------  --------------  ------  -------------------  ------
-----  ------
   databricks  main.sales.customer_contacts    contact   730             207851  0                    2024-
10-21   OK
   databricks  main.sales.web_events           event     90              338638  0                    2026-
07-22   OK
   redshift    sales.customer_contacts         contact   730             161316  0                    2024-
10-22   OK
   redshift    sales.web_events                event     90              147811  0                    2026-
07-19   OK
   snowflake   sf_prod.hr.employee_notes       hr        365             239349  0                    2025-
10-18   OK
   snowflake   sf_prod.sales.customer_contacts  contact   730             51426   0                    2024-
10-19   OK
   snowflake   sf_prod.sales.web_events        event     90              105751  0                    2026-
07-20   OK

== Deleted but still recoverable: iso_2_deleted_but_recoverable
   period 2026-09-16 to 2026-10-15 | filters: register tables; Snowflake from table settings, the others from
deletes in the period | run 2026-11-01 13:48:31Z | rows 7 | sha256
ea257ee2b272b846bd833c976dabb85aec5cd8fd3adf4ab864ab4b356fea82fb
   platform    table                           category  allowed_days  recoverable_days  basis
status
   ----------  ------------------------------  --------  ------------  ----------------  --------------------
-------------------------------------------------------------  --------------------
   databricks  main.sales.customer_contacts    contact   14            7.6               delete 2026-10-08,
no VACUUM yet old enough to remove its files (kept 7 d)        OPEN, WITHIN SO FAR
   databricks  main.sales.web_events           event     14            7.6               delete 2026-10-08,
no VACUUM yet old enough to remove its files (kept 7 d)        OPEN, WITHIN SO FAR
   redshift    sales.customer_contacts         contact   14            7.9               delete 2026-09-20,
VACUUM DELETE finished 2026-09-21, snapshot kept to 2026-09-28  WITHIN POLICY
   redshift    sales.web_events                event     14            7.9               delete 2026-09-20,
VACUUM DELETE finished 2026-09-21, snapshot kept to 2026-09-28  WITHIN POLICY
   snowflake   sf_prod.hr.employee_notes       hr        14            8                 Time Travel 1 d +
Fail-safe 7 d                                                   WITHIN POLICY
   snowflake   sf_prod.sales.customer_contacts  contact   14            14                Time Travel 7 d +
Fail-safe 7 d                                                   WITHIN POLICY
   snowflake   sf_prod.sales.web_events        event     14            1                 Time Travel 1 d,
transient: no Fail-safe                                          WITHIN POLICY

== Masking against your policy: iso_3_masking_against_policy
   period 2026-09-16 to 2026-10-15 | filters: columns tagged with a category | run 2026-11-01 13:48:31Z |
rows 10 | sha256 603359d40c940b2c5226048bae8db58eb17234cd6803c638c3302ec44e5d361a
   platform    table                           column      category  roles_reading_in_clear  status
   ----------  ------------------------------  ----------  --------  ----------------------  ------
   databricks  main.sales.customer_contacts    email       contact   data_steward            OK
```

```
   databricks  main.sales.customer_contacts     phone       contact                          OK
   databricks  main.sales.web_events            ip_address  network                          OK
   redshift    sales.customer_contacts          email       contact                          OK
   redshift    sales.customer_contacts          phone       contact                          OK
   redshift    sales.web_events                 ip_address  network   security_analyst       OK
   snowflake   sf_prod.hr.employee_notes        note_text   hr        hr_admin               OK
   snowflake   sf_prod.sales.customer_contacts  email       contact   support_lead           OK
   snowflake   sf_prod.sales.customer_contacts  phone       contact                          OK
   snowflake   sf_prod.sales.web_events         ip_address  network                          OK

== Result
   follow-up: none
   input checks: 26 of 26 passed
   RESULT: 0 item(s) need follow-up; 0 input check(s) failed; 0 platform(s) not checked.
```

Run 2: findings planted in the records (--demo --plant)

```
$ python3 deletion_masking_evidence.py --demo --plant
Demo: GENERATED records (seed 231) in generated/planted/, not a company's data.

== Input checks
   [ok] every platform name in your lists is snowflake, databricks or redshift
   [ok] no table appears twice in the retention register
   [ok] every register line has a category, whole-number days, and a date column that is a plain name
   [ok] every column tag has a table, a column and a category
   [ok] every category in the masking policy is on at least one tagged column
   [ok] every clear reader names a table, a column and a role
   [ok] every clear reader is on a column that is tagged
   [ok] every row-age line is for a table in the register
   [ok] no table has two row-age lines
   [ok] row-age counts are whole numbers and the oldest date is YYYY-MM-DD (or blank for an empty table)
   [ok] row ages were made with the same retention days as the register
   [ok] row ages were measured within 3 days before AS_OF, not after it
   [ok] no register table has rows with no date to age them by
   [ok] Snowflake: retention_time is a whole number and is_transient is YES or NO
   [ok] Snowflake: no register table has two live (not dropped) rows
   [ok] Databricks: every history timestamp is YYYY-MM-DD HH:MM:SS
   [ok] Databricks: every VACUUM operation in the history is one the settings name
   [ok] Databricks: each register table's history reaches back to the period's start (or to its first
version)
   [ok] Databricks: a table's deleted-file retention is a whole number of days or blank
   [ok] Databricks: every register table says whether deletion vectors are on (true or false)
   [ok] Redshift: every query-history, vacuum and snapshot timestamp is YYYY-MM-DD HH:MM:SS
   [ok] Redshift: every day from the period's start to AS_OF has query history, except KNOWN_GAPS
   [ok] Redshift: snapshot type is manual or automated, and a manual retention is -1 or whole days from 1
   [ok] Redshift: REDSHIFT_AUTOMATED_RETENTION_DAYS is set when automated snapshots exist
   [ok] Redshift: the masking-policy export has rows (a user who is not a superuser or sys:secadmin sees 0)
   [ok] Redshift: every DELETE that could touch a register table starts the statement and names schema.table

== Rows past retention: iso_1_rows_past_retention
   period 2026-09-16 to 2026-10-15 | filters: register tables on exported platforms | run 2026-11-01
13:48:31Z | rows 7 | sha256 8fc9ade2cbe47539df667646a1ce764749a7dc29dd3a23835461532d1c215ae2
   platform    table                            category  retention_days  rows    rows_past_retention
oldest_date  status
   ----------  -------------------------------  --------  --------------  ------  -------------------  ------
-----  --------------
   databricks  main.sales.customer_contacts     contact   730             207851  0                    2024-
10-21   OK
   databricks  main.sales.web_events            event     90              338638  37                   2026-
07-02   PAST RETENTION
   redshift    sales.customer_contacts          contact   730
NOT MEASURED
```

```
   redshift    sales.web_events                 event     90              147811  0                   2026-
07-19   OK
   snowflake   sf_prod.hr.employee_notes        hr        365             239349  0                   2025-
10-18   OK
   snowflake   sf_prod.sales.customer_contacts  contact   730             51426   0                   2024-
10-19   OK
   snowflake   sf_prod.sales.web_events         event     90              105751  1204                2026-
06-30   PAST RETENTION

== Deleted but still recoverable: iso_2_deleted_but_recoverable
   period 2026-09-16 to 2026-10-15 | filters: register tables; Snowflake from table settings, the others from
deletes in the period | run 2026-11-01 13:48:31Z | rows 7 | sha256
f0ef5eed6dcfa0c11130b0a43bb7f0374c50d0d924f128dd1d6429bcd40530b0
   platform    table                            category  allowed_days  recoverable_days  basis
status
   ----------  -------------------------------  --------  ------------  ----------------  -------------------
-------------------------------------------------------  -------------------------
   databricks  main.sales.customer_contacts     contact   14            25.6              delete 2026-09-20,
no VACUUM yet old enough to remove its files (kept 30 d)  RECOVERABLE BEYOND POLICY
   databricks  main.sales.web_events            event     14            7.6               delete 2026-10-08,
no VACUUM yet old enough to remove its files (kept 7 d)   OPEN, WITHIN SO FAR
   redshift    sales.customer_contacts          contact   14                              delete 2026-09-20;
a manual snapshot from 2026-09-18 is kept indefinitely    INDEFINITE SNAPSHOT
   redshift    sales.web_events                 event     14                              delete 2026-09-20;
a manual snapshot from 2026-09-18 is kept indefinitely    INDEFINITE SNAPSHOT
   snowflake   sf_prod.hr.employee_notes        hr        14            97                Time Travel 90 d +
Fail-safe 7 d                                             RECOVERABLE BEYOND POLICY
   snowflake   sf_prod.sales.customer_contacts  contact   14            14                Time Travel 7 d +
Fail-safe 7 d                                              WITHIN POLICY
   snowflake   sf_prod.sales.web_events         event     14            1                 Time Travel 1 d,
transient: no Fail-safe                                    WITHIN POLICY

== Masking against your policy: iso_3_masking_against_policy
   period 2026-09-16 to 2026-10-15 | filters: columns tagged with a category | run 2026-11-01 13:48:31Z |
rows 10 | sha256 b9385ecd6afa326e48d1c2eacea21c5542ab3406a2b36d6564fbf5ae464a6de6
   platform    table                            column      category  roles_reading_in_clear  status
   ----------  -------------------------------  ----------  --------  ----------------------  ---------------
-----------------------------------
   databricks  main.sales.customer_contacts     email       contact   data_steward            OK
   databricks  main.sales.customer_contacts     phone       contact                           OK
   databricks  main.sales.web_events            ip_address  network                           NOT MASKED
   redshift    sales.customer_contacts          email       contact   marketing_analyst       CLEAR READER
NOT ON POLICY LIST: marketing_analyst
   redshift    sales.customer_contacts          phone       contact                           OK
   redshift    sales.web_events                 ip_address  network   security_analyst        OK
   snowflake   sf_prod.hr.employee_notes        note_text   hr        hr_admin                OK
   snowflake   sf_prod.sales.customer_contacts  email       contact   support_lead            OK
   snowflake   sf_prod.sales.customer_contacts  phone       contact                           NOT MASKED
   snowflake   sf_prod.sales.web_events         ip_address  network                           OK

== Result
   follow-up: 2 past retention, 1 not measured, 2 recoverable beyond policy, 2 indefinite snapshot, 2 not
masked or mask not active, 1 clear reader not on policy list
   input checks: 26 of 26 passed
   RESULT: 10 item(s) need follow-up; 0 input check(s) failed; 0 platform(s) not checked.
```

Run 3: faults planted in the inputs (--demo --plant-input)

```
$ python3 deletion_masking_evidence.py --demo --plant-input
Demo: GENERATED records (seed 231) in generated/planted-input/, not a company's data.

== Input checks
```

```
   [FAIL] every platform name in your lists is snowflake, databricks or redshift: 1 row(s)
   [FAIL] no table appears twice in the retention register: 1 row(s)
   [FAIL] every register line has a category, whole-number days, and a date column that is a plain name: 1
row(s)
   [FAIL] every column tag has a table, a column and a category: 1 row(s)
   [FAIL] every category in the masking policy is on at least one tagged column: 1 row(s)
   [FAIL] every clear reader names a table, a column and a role: 1 row(s)
   [FAIL] every clear reader is on a column that is tagged: 1 row(s)
   [FAIL] every row-age line is for a table in the register: 1 row(s)
   [FAIL] no table has two row-age lines: 1 row(s)
   [FAIL] row-age counts are whole numbers and the oldest date is YYYY-MM-DD (or blank for an empty table): 1
row(s)
   [FAIL] row ages were made with the same retention days as the register: 1 row(s)
   [FAIL] row ages were measured within 3 days before AS_OF, not after it: 1 row(s)
   [FAIL] no register table has rows with no date to age them by: 1 row(s)
   [FAIL] Snowflake: retention_time is a whole number and is_transient is YES or NO: 1 row(s)
   [FAIL] Snowflake: no register table has two live (not dropped) rows: 1 row(s)
   [FAIL] Databricks: every history timestamp is YYYY-MM-DD HH:MM:SS: 1 row(s)
   [FAIL] Databricks: every VACUUM operation in the history is one the settings name: 1 row(s)
   [FAIL] Databricks: each register table's history reaches back to the period's start (or to its first
version): 1 row(s)
   [FAIL] Databricks: a table's deleted-file retention is a whole number of days or blank: 1 row(s)
   [FAIL] Databricks: every register table says whether deletion vectors are on (true or false): 1 row(s)
   [FAIL] Redshift: every query-history, vacuum and snapshot timestamp is YYYY-MM-DD HH:MM:SS: 1 row(s)
   [FAIL] Redshift: every day from the period's start to AS_OF has query history, except KNOWN_GAPS: 1 row(s)
   [FAIL] Redshift: snapshot type is manual or automated, and a manual retention is -1 or whole days from 1:
1 row(s)
   [FAIL] Redshift: REDSHIFT_AUTOMATED_RETENTION_DAYS is set when automated snapshots exist: 1 row(s)
   [FAIL] Redshift: the masking-policy export has rows (a user who is not a superuser or sys:secadmin sees
0): 1 row(s)
   [FAIL] Redshift: every DELETE that could touch a register table starts the statement and names
schema.table: 1 row(s)

== Rows past retention: iso_1_rows_past_retention   ** INCOMPLETE: see input checks **
   period 2026-09-16 to 2026-10-15 | filters: register tables on exported platforms | run 2026-11-01
13:48:31Z | rows 9 | sha256 bf33e2c308093d9d1494d6ba699bb17c6a1031b35fe1d00a94e063cd80e2813d
   platform    table                           category  retention_days  rows    rows_past_retention
oldest_date  status
   ----------  ------------------------------  --------  --------------  ------  -------------------  ------
-----  ------
   databricks  main.sales.customer_contacts    contact   730                     0                    2024-
10-21   OK
   databricks  main.sales.web_events           event     90              338638  0                    2026-
07-22   OK
   redshift    sales.customer_contacts         contact   730             161316  0                    2024-
10-22   OK
   redshift    sales.customer_contacts         contact   730             161316  0                    2024-
10-22   OK
   redshift    sales.web_events                event     90              147811  0                    2026-
07-19   OK
   snowflake   sf_prod.hr.employee_notes       hr        365             239349  0                    2025-
10-18   OK
   snowflake   sf_prod.sales.customer_contacts  contact   730             51426   0                    2024-
10-19   OK
   snowflake   sf_prod.sales.web_events        event     90              105751  0                    2026-
07-20   OK
   snowflake   sf_prod.sales.web_events        event     90              105751  0                    2026-
07-20   OK

== Deleted but still recoverable: iso_2_deleted_but_recoverable   ** INCOMPLETE: see input checks **
   period 2026-09-16 to 2026-10-15 | filters: register tables; Snowflake from table settings, the others from
deletes in the period | run 2026-11-01 13:48:31Z | rows 8 | sha256
69a812a662c76586a19aff08a27972d39a1cdcc970f84c019afc7afe6298340f
   platform    table                           category  allowed_days  recoverable_days  basis
```

```
status
   ----------  ------------------------------  --------  ------------  -----------------  --------------------
-------------------------------------------------------------  --------------------
   databricks  main.sales.customer_contacts     contact   14                             delete 2026-09-20;
deletion_vectors not stated, so removal cannot be judged       NOT MEASURED
   databricks  main.sales.web_events            event     14            7.6              delete 2026-10-08,
no VACUUM yet old enough to remove its files (kept 7 d)        OPEN, WITHIN SO FAR
   redshift    sales.customer_contacts          contact
recoverable_days_allowed is not a whole number                                    NOT MEASURED
   redshift    sales.customer_contacts          contact   14            7.9              delete 2026-09-20,
VACUUM DELETE finished 2026-09-21, snapshot kept to 2026-09-28  WITHIN POLICY
   redshift    sales.web_events                 event     14            7.9              delete 2026-09-20,
VACUUM DELETE finished 2026-09-21, snapshot kept to 2026-09-28  WITHIN POLICY
   snowflake   sf_prod.hr.employee_notes        hr        14            8                Time Travel 1 d +
Fail-safe 7 d                                                  WITHIN POLICY
   snowflake   sf_prod.sales.customer_contacts  contact   14            14               Time Travel 7 d +
Fail-safe 7 d                                                  WITHIN POLICY
   snowflake   sf_prod.sales.web_events         event     14            1                Time Travel 1 d,
transient: no Fail-safe                                         WITHIN POLICY

== Masking against your policy: iso_3_masking_against_policy   ** INCOMPLETE: see input checks **
   period 2026-09-16 to 2026-10-15 | filters: columns tagged with a category | run 2026-11-01 13:48:31Z |
rows 10 | sha256 4c2d9b912b7f9347dc48a31112ca9ec4c38031b39b90880067411e783f58011b
   platform    table                            column      category  roles_reading_in_clear  status
   ----------  ------------------------------   ----------  --------  ----------------------  ----------
   databricks  main.sales.customer_contacts     email       contact   data_steward            OK
   databricks  main.sales.customer_contacts     phone       contact                           OK
   databricks  main.sales.web_events            ip_address  network                           OK
   redshift    sales.customer_contacts          email       contact                           NOT MASKED
   redshift    sales.customer_contacts          phone       contact                           NOT MASKED
   redshift    sales.web_events                 ip_address  network   security_analyst        NOT MASKED
   snowflake   sf_prod.hr.employee_notes        note_text   hr        hr_admin                OK
   snowflake   sf_prod.sales.customer_contacts  email       contact   support_lead            OK
   snowflake   sf_prod.sales.customer_contacts  phone       contact                           OK
   snowflake   sf_prod.sales.web_events         ip_address  network                           OK

== Result
   follow-up: 2 not measured, 3 not masked or mask not active
   input checks: 0 of 26 passed; the lists above are INCOMPLETE
   RESULT: 5 item(s) need follow-up; 26 input check(s) failed; 0 platform(s) not checked.
```

Run 4: the row-age queries for the demo register (--demo --queries)

```
$ python3 deletion_masking_evidence.py --demo --queries
SELECT 'snowflake' AS platform, 'sf_prod.sales.customer_contacts' AS table_name, 730 AS retention_days_used,
       COUNT(*) AS rows_total,
       COALESCE(SUM(CASE WHEN created_at < DATEADD(day, -730, CURRENT_DATE) THEN 1 ELSE 0 END), 0) AS
rows_past_retention,
       COALESCE(SUM(CASE WHEN created_at IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,
       CAST(MIN(created_at) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on
FROM sf_prod.sales.customer_contacts;

SELECT 'snowflake' AS platform, 'sf_prod.sales.web_events' AS table_name, 90 AS retention_days_used,
       COUNT(*) AS rows_total,
       COALESCE(SUM(CASE WHEN event_date < DATEADD(day, -90, CURRENT_DATE) THEN 1 ELSE 0 END), 0) AS
rows_past_retention,
       COALESCE(SUM(CASE WHEN event_date IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,
       CAST(MIN(event_date) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on
FROM sf_prod.sales.web_events;

SELECT 'snowflake' AS platform, 'sf_prod.hr.employee_notes' AS table_name, 365 AS retention_days_used,
       COUNT(*) AS rows_total,
```

```
       COALESCE(SUM(CASE WHEN noted_on < DATEADD(day, -365, CURRENT_DATE) THEN 1 ELSE 0 END), 0) AS
rows_past_retention,
       COALESCE(SUM(CASE WHEN noted_on IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,
       CAST(MIN(noted_on) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on
FROM sf_prod.hr.employee_notes;

SELECT 'databricks' AS platform, 'main.sales.customer_contacts' AS table_name, 730 AS retention_days_used,
       COUNT(*) AS rows_total,
       COALESCE(SUM(CASE WHEN created_at < date_sub(current_date(), 730) THEN 1 ELSE 0 END), 0) AS
rows_past_retention,
       COALESCE(SUM(CASE WHEN created_at IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,
       CAST(MIN(created_at) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on
FROM main.sales.customer_contacts;

SELECT 'databricks' AS platform, 'main.sales.web_events' AS table_name, 90 AS retention_days_used,
       COUNT(*) AS rows_total,
       COALESCE(SUM(CASE WHEN event_date < date_sub(current_date(), 90) THEN 1 ELSE 0 END), 0) AS
rows_past_retention,
       COALESCE(SUM(CASE WHEN event_date IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,
       CAST(MIN(event_date) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on
FROM main.sales.web_events;

SELECT 'redshift' AS platform, 'sales.customer_contacts' AS table_name, 730 AS retention_days_used,
       COUNT(*) AS rows_total,
       COALESCE(SUM(CASE WHEN created_at < DATEADD(day, -730, CURRENT_DATE) THEN 1 ELSE 0 END), 0) AS
rows_past_retention,
       COALESCE(SUM(CASE WHEN created_at IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,
       CAST(MIN(created_at) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on
FROM sales.customer_contacts;

SELECT 'redshift' AS platform, 'sales.web_events' AS table_name, 90 AS retention_days_used,
       COUNT(*) AS rows_total,
       COALESCE(SUM(CASE WHEN event_date < DATEADD(day, -90, CURRENT_DATE) THEN 1 ELSE 0 END), 0) AS
rows_past_retention,
       COALESCE(SUM(CASE WHEN event_date IS NULL THEN 1 ELSE 0 END), 0) AS rows_no_age,
       CAST(MIN(event_date) AS DATE) AS oldest_date, CURRENT_DATE AS measured_on
FROM sales.web_events;
```

How to read the results

- **INCOMPLETE** on a list means an input check failed, so the list may be missing rows. Fix the export, or explain a quiet Redshift day in `KNOWN_GAPS`, before you show it to anyone.
- **PAST RETENTION** is a table with rows older than your register allows. The count is as of the day the row-age query ran. **NOT MEASURED** means no row-age line or no platform record for that table: the list cannot say it is clean.
- **RECOVERABLE BEYOND POLICY** is a delete whose data stayed recoverable longer than the days your register allows. **OPEN, WITHIN SO FAR** is a delete with no VACUUM yet that is old enough to remove its files (on Redshift, no completed vacuum yet): it is inside the allowance today, and not yet a finding, but it will become one if nothing runs. **NO DELETE IN PERIOD** means the period holds no delete to measure.
- **INDEFINITE SNAPSHOT** is a Redshift manual snapshot with no end date, taken before a delete. It is flagged whatever the register allows, because a snapshot with no end date means the deleted rows it holds have no end date either. Whether to keep it is your decision.

- **NOT MASKED** is a tagged column with no mask in the platform's record. **NOT MASKED FOR PUBLIC** is a Redshift column whose mask is attached only to named roles or users, so everyone else reads it in the clear. **MASK NOT ACTIVE** is a Snowflake mask whose status is not ACTIVE. **CLEAR READER NOT ON POLICY LIST** is a role in your `clear_readers.csv` that your masking policy does not name for that category. **NO POLICY FOR CATEGORY** is a tagged column whose category your masking policy does not mention.
- A clean run on your own exports means only that those exports agree with your register and your policy. OK, WITHIN POLICY and WITHIN SO FAR are the script's labels, not a finding about your controls, and they say nothing about copies outside the three platforms.
- Keep each run's settings, its SHA-256 lines and the CSVs it wrote. Your auditor decides what they show.

Run it on your own data

To run it on your own data, change only the settings block at the top of the download. Set EXPORT_DIR to the folder holding your exports. Set PERIOD_START and AS_OF. Set PLATFORMS_EXPORTED to the platforms you exported. Set REDSHIFT_AUTOMATED_RETENTION_DAYS to your cluster's automated snapshot retention in days if you export Redshift. Write your retention register and your masking policy as the two CSV files named in FILES. Run the script with --queries, run each query it prints on the platform it names, and save the answers as row_ages.csv. Export each platform's own records with the platform's own column names, tag your columns in column_tags.csv, and list the roles that read a tagged column in the clear in clear_readers.csv. If a file has another name, change its entry in FILES. Change nothing else, and run it without --demo.

We tested those steps on a second, separate set of generated exports: other file names, other schema, table and column names, names in mixed case, quoted names in a Redshift delete, a blank role list in the masking policy, a blank policy column and a blank dropped-on date, and Databricks left out of PLATFORMS_EXPORTED. The script reported Databricks as NOT CHECKED, caught the table with rows past its retention, the table never measured, the Snowflake table recoverable for 37 days against 30 allowed, the Redshift snapshot with no end date, the mask that was not active, the column with no mask and the role that was not on the list, and it ran to the end. That test is also generated data.

Download deletion_masking_evidence.py · Download the full guide (PDF)

Method

- **Input:** generated records written by the script as CSV in each platform's published column names (the links in step 1), with a fixed seed (231). The register, the masking policy, the tags and the clear-reader list are generated too. The tables, roles and counts are invented.
- **Period:** 2026-09-16 to 2026-10-15, UTC, inclusive. Deletes are read from that period. Everything is read as of the end of `AS_OF` (2026-10-15). Row ages must have been measured within three days before it.
- **Recoverable days:** Snowflake from the table's settings (retention time, plus 7 days unless transient); Databricks and Redshift from the time between a delete and the event that removes the data, or the time so far if none has happened. The script reads records; it never measures bytes held in storage.

- **Redshift snapshots:** a manual snapshot with retention -1 is retained indefinitely (AWS CLI: describe-cluster-snapshots (read 2026-11-01)); one with a number of days expires that many days after it was created; an automated snapshot expires after the days in `REDSHIFT_AUTOMATED_RETENTION_DAYS`. Only snapshots created before a delete are counted for it.
- **Fingerprints:** each list is sorted on every column before it is hashed, so the same input gives the same SHA-256 on every run. We ran the clean records on duckdb 1.5.6 and on 2.0.0 and the three lists hashed the same.
- **Our assumptions, to check against one of your own rows:** Databricks history calls a delete `DELETE` (the history schema page says so) and a finished VACUUM `VACUUM END` (the history schema page lists VACUUM's metrics but not its operation names, Databricks: Table history schema (read 2026-11-01), so it is a setting, and an unknown operation starting VACUUM fails an input check), and `REORG TABLE ... APPLY (PURGE)` is the operation `REORG` (a setting, `DATABRICKS_PURGE_OPS`; the history schema page does not list it); Snowflake POLICY_KIND is `MASKING_POLICY` for a masking policy; in Redshift's SYS_VACUUM_HISTORY a finished vacuum has the status `Complete` and a vacuum that removes deleted rows has the vacuum_type `Delete` or `Full`, which the page lists (Amazon Redshift: SYS_VACUUM_HISTORY (read 2026-11-01)), with case ignored (check one of your own rows); the platforms' column names are as the linked pages list them on 2026-11-01.

Checklist

Done	Step	Note
☐	Wrote the retention register and the masking policy in your own words	Your policy, not ours
☐	Exported every file for each platform, as a user who sees every row	Redshift: a superuser (sys:secadmin covers only the masking export); Databricks: a user who can see every masked column
☐	Ran the row-age query on every register table and saved the answers	Within 3 days before AS_OF (ROW_AGES_MAX_STALE_DAYS)
☐	Set PERIOD_START, AS_OF, PLATFORMS_EXPORTED and the Redshift automated retention	Settings block only
☐	Ran it, and read the input checks first	Any FAIL marks the lists INCOMPLETE
☐	Looked at every PAST RETENTION and NOT MEASURED line	Measure the missing tables
☐	Looked at every RECOVERABLE BEYOND POLICY, OPEN and INDEFINITE SNAPSHOT line	Decide whether the policy or the setting is wrong
☐	Looked at every NOT MASKED, MASK NOT ACTIVE and CLEAR READER line	Check the clear-reader list is complete
☐	Kept the settings, the SHA-256 lines and the CSVs of the run	Your auditor decides what they use

Limits

- **This is not a client result.** Every record here is generated. We have not run the script on a real Snowflake, Databricks or Redshift account, and we did not run the row-age queries on a platform.
- **The standard was not read.** We name Annex A 8.10 and 8.11 by id and title and say nothing about what they require. Your policy is yours; the script cannot tell you whether it is the right one. The auditor decides what evidence they accept.
- **Row ages** are only as good as the query and the day it ran; a table the query was not run on is NOT MEASURED. Rows with no date to age them by fail an input check.
- **Snowflake:** the days come from table settings, not from bytes actually held. Fail-safe is recoverable by Snowflake only (Snowflake: Fail-safe (read 2026-11-01)); the script counts it unless you set `SNOWFLAKE_COUNT_FAILSAFE` to False.
- **Databricks:** a VACUUM run with a shorter retention than the table's property could remove files earlier than the script assumes, and the history schema page does not say that the history records the retention a VACUUM used. The VACUUM page also warns that with disk caching a cluster may still hold data from deleted files until it restarts (Databricks: Remove unused data files with VACUUM (read 2026-11-01)). Deletes older than the table's history are not seen, which is why an input check requires the history to reach the period's start.
- **Redshift:** the script reads deletes from query text, so a DELETE it cannot place (one behind a WITH clause, or one that names only the table) fails an input check when its table name matches your register, rather than being skipped; one on any other table, such as a staging table, is left out. A successful status is not proof the transaction committed (Amazon Redshift: SYS_QUERY_HISTORY (read 2026-11-01)). VACUUM DELETE reclaims space so that at least 95 percent of the remaining rows are not marked for deletion by default (Amazon Redshift: VACUUM (read 2026-11-01)), and the script treats a completed Delete or Full vacuum as removal. Snapshots can leave out tables created with BACKUP NO on clusters that support it; RG and RA3 clusters and Serverless back up every table (Amazon Redshift: snapshots and backups (read 2026-11-01)). The script cannot see BACKUP NO, so it flags a snapshot against every table. TRUNCATE and DROP are not read. Snapshots come from `describe-cluster-snapshots`, which covers provisioned clusters; Redshift Serverless snapshots are not read. A masking policy attached to one role or user masks the column only for that grantee, so a column counts as masked only when a policy that lists it is attached to PUBLIC (Amazon Redshift: ATTACH MASKING POLICY (read 2026-11-01)).
- **Masking:** "in force" is what the platform's own view says. The roles that read a column in the clear are a list you keep, because no platform view names them; the script trusts it. Databricks `column_masks` lists only columns the exporting user can interact with (Databricks: COLUMN_MASKS (read 2026-11-01)).
- **Evidence, not a verdict.** The script produces evidence. It never says a control is met, and it does not know what any auditor will accept.

Tested on

- **Platform:** DuckDB (the script reads exported CSV files; it never connects to Snowflake, Databricks or Redshift)
- **Edition:** duckdb 1.5.6 and 2.0.0; Python 3
- **Date:** 2026-11-01
- **Dataset:** Generated records (seed 231) in each platform's published column names; not a company's data

When this is not enough

- Copies outside the platform: exports, extracts in BI tools, files in cloud storage, logs and backups made by other systems. The three lists see only what the three platforms record.
- Data a delete never touched. A table can be inside its retention and still hold data your policy says should not be kept at all; the register is only as good as its list of tables and categories.
- A policy that is silent: a category your masking policy does not cover is listed as NO POLICY FOR CATEGORY, but deciding what the policy should say is for you and your auditor.
- Exports taken once. A single run shows one day; a delete or a mask can change the day after. Run it on a schedule and keep every run's lists and hashes.
- Platforms other than the three, and the other Annex A controls.

Sources

- Snowflake: Time Travel: https://docs.snowflake.com/en/user-guide/data-time-travel (read 2026-11-01)
- Snowflake: Fail-safe: https://docs.snowflake.com/en/user-guide/data-failsafe (read 2026-11-01)
- Snowflake: temporary and transient tables: https://docs.snowflake.com/en/user-guide/tables-temp-transient (read 2026-11-01)
- Snowflake: ACCOUNT_USAGE.TABLES: https://docs.snowflake.com/en/sql-reference/account-usage/tables (read 2026-11-01)
- Snowflake: ACCOUNT_USAGE.POLICY_REFERENCES: https://docs.snowflake.com/en/sql-reference/account-usage/policy_references (read 2026-11-01)
- Databricks: Remove unused data files with VACUUM: https://docs.databricks.com/aws/en/delta/vacuum (read 2026-11-01)
- Databricks: VACUUM (SQL): https://docs.databricks.com/aws/en/sql/language-manual/delta-vacuum (read 2026-11-01)
- Databricks: Work with table history: https://docs.databricks.com/aws/en/delta/history (read 2026-11-01)

- Databricks: Table history schema: https://docs.databricks.com/aws/en/tables/history-schema (read 2026-11-01)
- Databricks: COLUMN_MASKS: https://docs.databricks.com/aws/en/sql/language-manual/information-schema/column_masks (read 2026-11-01)
- Amazon Redshift: VACUUM: https://docs.aws.amazon.com/redshift/latest/dg/r_VACUUM_command.html (read 2026-11-01)
- Amazon Redshift: SYS_VACUUM_HISTORY: https://docs.aws.amazon.com/redshift/latest/dg/SYS_VACUUM_HISTORY.html (read 2026-11-01)
- Amazon Redshift: SYS_QUERY_HISTORY: https://docs.aws.amazon.com/redshift/latest/dg/SYS_QUERY_HISTORY.html (read 2026-11-01)
- Amazon Redshift: SVV_ATTACHED_MASKING_POLICY: https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_ATTACHED_MASKING_POLICY.html (read 2026-11-01)
- Amazon Redshift: ATTACH MASKING POLICY: https://docs.aws.amazon.com/redshift/latest/dg/r_ATTACH_MASKING_POLICY.html (read 2026-11-01)
- Amazon Redshift: snapshots and backups: https://docs.aws.amazon.com/redshift/latest/mgmt/working-with-snapshots.html (read 2026-11-01)
- AWS CLI: describe-cluster-snapshots: https://docs.aws.amazon.com/cli/latest/reference/redshift/describe-cluster-snapshots.html (read 2026-11-01)
- Data Control Tower (Data Meaning): https://www.datameaning.com/solutions/data-control-tower.html (read 2026-11-01)

See Data Control Tower on a fictional company's data

The code in this guide is free and needs nothing from us to run. Separately, Data Control Tower deploys into your own Azure subscription from a template your IT team can read before it runs. Every day it reads the records your data platforms already keep, and it never writes to them. It covers Amazon Redshift, Databricks and Snowflake, and no platform passwords are stored.

We'll walk you through it on a fictional company's data, then show what you would share to see your own.

Read about Data Control Tower: https://www.datameaning.com/solutions/data-control-tower.html