DATA CONTROL TOWER · GUIDE

# HIPAA minimum necessary in Snowflake, Databricks and Redshift: show who can read patient data

Show who can read patient data, against your own register, from each platform's own records.

Published 2026-11-01 · Online version and downloads: www.datameaning.com/guides/hipaa-minimum-necessary-evidence-for-snowflake-databricks-redshift/

Identify which staff classes need which categories of patient data (45 CFR 164.514(d)(2)(i)) and keep it as a register. Then compare that register with the records Snowflake, Databricks and Redshift already keep. Show four lists: who holds access beyond the register, which readable columns have no mask attached, access with no read seen, and leavers who held access.

## What the rule asks you to identify

The minimum necessary standard starts in 45 CFR 164.502(b)(1): “When using or disclosing protected health information or when requesting protected health information from another covered entity or business associate, a covered entity or business associate must make reasonable efforts to limit protected health information to the minimum necessary to accomplish the intended purpose of the use, disclosure, or request.” (45 CFR 164.502 (read 2026-11-01), text current as of 2026-10-29). Paragraph (b)(2) lists where the standard does not apply, and its first item is “Disclosures to or requests by a health care provider for treatment;”. That item covers disclosures and requests, not a use inside the organization, so this page does not treat treatment use as exempt from the standard. We do not address the other items on the list.

For uses inside the organization, 164.514(d)(2) says what to identify. It names the covered entity, and 164.502(b) names business associates as well, so check which you are. Paragraph (d)(2)(i): “A covered entity must identify: (A) Those persons or classes of persons, as appropriate, in its workforce who need access to protected health information to carry out their duties; and (B) For each such person or class of persons, the category or categories of protected health information to which access is needed and any conditions appropriate to such access.” Paragraph (d)(2)(ii): “A covered entity must make reasonable efforts to limit the access of such persons or classes identified in paragraph (d)(2)(i)(A) of this section to protected health information consistent with paragraph (d)(2)(i)(B) of this section.” (45 CFR 164.514 (read

2026-11-01)). HHS's guidance on the standard says of uses: "Case-by-case review of each use is not required." (HHS: Minimum Necessary Requirement (read 2026-11-01)).

Two Security Rule texts sit next to it. Access establishment and modification, 164.308(a)(4)(ii)(C), is marked Addressable and asks an entity to "establish, document, review, and modify a user's right of access to a workstation, transaction, program, or process" (45 CFR 164.308 (read 2026-11-01)). For an Addressable specification, 164.306(d)(3) says the entity must (i) "Assess whether each implementation specification is a reasonable and appropriate safeguard in its environment" and then "(A) Implement the implementation specification if reasonable and appropriate; or (B) If implementing the implementation specification is not reasonable and appropriate—" document why, and implement an equivalent alternative measure if reasonable and appropriate (45 CFR 164.306 (read 2026-11-01)). And 164.530(j)(1)(iii) says "If an action, activity, or designation is required by this subpart to be documented, maintain a written or electronic record of such action, activity, or designation." Under (j)(2) that record is kept "for six years from the date of its creation or the date when it last was in effect, whichever is later." (45 CFR 164.530 (read 2026-11-01)). Whether your register falls under (j) is a question for your counsel.

The text quoted here does not mention masking, and none of it sets an interval for reviewing access. We treat both as ways an entity might describe its "reasonable efforts", not as requirements. For the Security Rule's audit-log side, see our guide HIPAA Security Rule evidence from data platform logs; this page is about the Privacy Rule's minimum necessary standard for use of data inside the warehouse.

## What you will need

- Access to the platforms' own records: on Snowflake the ACCOUNT_USAGE views, on Databricks the system tables and information schema, on Redshift the system views. Redshift's masking view needs a superuser or the `sys:secadmin` role, or it returns no rows (Amazon Redshift: SVV_ATTACHED_MASKING_POLICY (read 2026-11-01)). Redshift's SVV_RELATION_PRIVILEGES and SVV_COLUMN_PRIVILEGES show every identity only to a superuser or a user with SYSLOG ACCESS UNRESTRICTED (ACCESS SYSTEM TABLE for column privileges); other users see only identities they have access to or own, and each covers the current database only (Amazon Redshift: SVV_RELATION_PRIVILEGES (read 2026-11-01)). On Databricks, export TABLE_PRIVILEGES as a user who can see every grant: a user with only MANAGE on an object sees only their own grants there (Databricks: TABLE_PRIVILEGES (read 2026-11-01)).
- Your register, your list of PHI columns, an HR file of classes and one of leavers, and a login map, as CSV files.
- 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, once, into one folder

The script reads these files from `EXPORT_DIR`. Column names are the platforms' own; the script lower-cases the header, so `CREATED_ON` and `created_on` both work, and it stops and says which column is missing if one is not there.

The nineteen files

| File | Columns the script reads | Defined in |
|---|---|---|
| `snowflake_grants_to_users.csv` | CREATED_ON, DELETED_ON, ROLE, GRANTEE_NAME | Snowflake: GRANTS_TO_USERS (read 2026-11-01) |
| `snowflake_grants_to_roles.csv` | CREATED_ON, DELETED_ON, PRIVILEGE, GRANTED_ON, NAME, TABLE_CATALOG, TABLE_SCHEMA, GRANTED_TO, GRANTEE_NAME | Snowflake: GRANTS_TO_ROLES (read 2026-11-01) |
| `snowflake_policy_references.csv` | POLICY_KIND, POLICY_STATUS, REF_DATABASE_NAME, REF_SCHEMA_NAME, REF_ENTITY_NAME, REF_COLUMN_NAME | Snowflake: POLICY_REFERENCES (read 2026-11-01) |
| `snowflake_access_history.csv` | QUERY_ID, QUERY_START_TIME, USER_NAME, BASE_OBJECTS_ACCESSED (the JSON array as text) | Snowflake: ACCESS_HISTORY (read 2026-11-01) |
| `databricks_audit.csv` | event_time, user_identity, service_name, action_name, request_params, response (user_identity, request_params and response as JSON text) | Databricks: audit log system table (read 2026-11-01) |
| `databricks_table_lineage.csv` | source_table_full_name, target_table_full_name, created_by, event_time, entity_type | Databricks: lineage system tables (read 2026-11-01) |
| `databricks_column_masks.csv` | catalog_name, schema_name, table_name, column_name, mask_name | Databricks: COLUMN_MASKS (read 2026-11-01) |
| `databricks_table_privileges.csv` | grantee, table_catalog, table_schema, table_name, privilege_type, inherited_from | Databricks: TABLE_PRIVILEGES (read 2026-11-01) |

| File | Columns the script reads | Defined in |
|---|---|---|
| `redshift_stl_utilitytext.csv` | userid, xid, pid, starttime, endtime, sequence, text | Amazon Redshift: STL_UTILITYTEXT (read 2026-11-01) |
| `redshift_svv_relation_privileges.csv` | namespace_name, relation_name, privilege_type, identity_name, identity_type | Amazon Redshift: SVV_RELATION_PRIVILEGES (read 2026-11-01) |
| `redshift_svv_column_privileges.csv` | namespace_name, relation_name, column_name, privilege_type, identity_id, identity_name, identity_type | Amazon Redshift: SVV_COLUMN_PRIVILEGES (read 2026-11-01) |
| `redshift_svv_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) |
| `redshift_sys_query_history.csv` | user_id, username, query_id, query_type, status, start_time, query_text | Amazon Redshift: SYS_QUERY_HISTORY (read 2026-11-01) |
| `redshift_sys_userlog.csv` | user_id, user_name, action, record_time | Amazon Redshift: SYS_USERLOG (read 2026-11-01) |
| `phi_columns.csv` | platform, table, column, category: your list of PHI columns, from your tags where you keep them (Snowflake TAG_REFERENCES, Databricks column_tags) and from a list you keep for Redshift | yours (the column names are settings); Snowflake: TAG_REFERENCES (read 2026-11-01), Databricks: COLUMN_TAGS (read 2026-11-01) |
| `register.csv` | class, category, conditions: your register, one row per class of workforce and PHI category that class needs | yours (the column names are settings) |
| `hr_classes.csv` | work email and class of workforce, one row per person, the class worded as in the register | yours (the column names are settings) |
| `hr_leavers.csv` | work email and last day of work, written YYYY-MM-DD | yours (the column names are settings) |

| File | Columns the script reads | Defined in |
|---|---|---|
| `login_map.csv` | platform, login, email: one row for every login that is not itself an email address, and one row per member of every Databricks group and Redshift group or role | yours |

What each platform keeps decides what you can show. Snowflake's ACCESS_HISTORY, where column-level reads come from, "requires Enterprise Edition (or higher)" and its latency can be up to 180 minutes (Snowflake: ACCESS_HISTORY (read 2026-11-01)); GRANTS_TO_ROLES, GRANTS_TO_USERS and POLICY_REFERENCES can each lag by up to 120 minutes, so take the export some hours after the last event you need. Databricks says its lineage tables "represent a subset of all read/write events, as it is not always possible to capture lineage" (Databricks: lineage system tables (read 2026-11-01)), so a read the table does not show is not proof that no read happened. On Redshift, "STL system views retain seven days of log history" and keeping more means copying them to other tables or unloading them to Amazon S3 (Amazon Redshift: System tables and views reference (read 2026-11-01)); STL_UTILITYTEXT, where the script reads GRANT and REVOKE text, is an STL view. The script checks every day from the start of its span to `AS_OF` for records on each platform and fails if a day has none. Who holds access today comes from views of the current state, Redshift's SVV_RELATION_PRIVILEGES and Databricks' TABLE_PRIVILEGES, so a grant older than the logs is still seen, provided the export is made by a user who can see every grant (see What you will need). STL_UTILITYTEXT and the audit log only date the grants and give the leaver list its removals, and the script does not check that they cover the whole span, so export them daily.

## Step 2. Write the register and the PHI column list, and read every timestamp strictly

The register is your 164.514(d)(2)(i) list: one row per class of workforce and PHI category that class needs, with the conditions in words. The script prints the conditions but does not check them against any record, because no platform record says why someone read a column. The PHI column list names each column the register's categories cover, written as the platform names the table. `NOT_USED_DAYS` has no default: unset, the script stops and says so, because a threshold you never chose would be ours, not yours. `REDSHIFT_HAS_MASKING_POLICIES` starts as `True`, so an empty Redshift masking view fails a check; set it to `False` only if your cluster has no masking policies. The settings you change sit at the top of the script:

```
# ---- CHANGE THESE FOR YOUR OWN DATA ------------------------------------------------------------
PERIOD_START = "2026-07-01"          # first day of the review period (UTC, inclusive)
PERIOD_END = "2026-09-30"            # last 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
NOT_USED_DAYS = None                 # no default: access held this long with no read seen is listed
KNOWN_GAPS = {}                      # days you can explain, e.g. {"redshift": ["2026-08-19"]}
```

```
PHI_PLATFORM_COL = "platform"        # PHI column list: snowflake, databricks or redshift
PHI_TABLE_COL = "table"              # PHI column list: the table, named as the platform names it
PHI_COLUMN_COL = "column"            # PHI column list: the column
PHI_CATEGORY_COL = "category"        # PHI column list: the PHI category, as worded in the register
REGISTER_CLASS_COL = "class"         # role register: the class of workforce
REGISTER_CATEGORY_COL = "category"   # role register: a PHI category that class needs
REGISTER_CONDITIONS_COL = "conditions"   # role register: the conditions on that access (printed, not
checked)
HR_CLASS_EMAIL_COL = "email"         # HR class file: the work email
HR_CLASS_COL = "class"               # HR class file: the class, as worded in the register
HR_EMAIL_COL = "email"               # HR leaver file: the work email
HR_LAST_DAY_COL = "last_day"         # HR leaver file: the last day of work, as YYYY-MM-DD
GRACE_DAYS = 1                       # access must end within this many days of the last day of work
REMOVAL_WINDOW_DAYS = 30             # a removal before every last day counts if it is at most this many
days early
READ_PRIVILEGES = ["select", "ownership", "all", "all privileges"]   # privileges that let a holder read
REDSHIFT_HAS_MASKING_POLICIES = True  # False if your cluster has no masking policies, so an empty view
is expected
```

Timestamps are read by one pattern: `YYYY-MM-DD HH:MM:SS`, with optional fractional seconds and an optional `Z`, `+00:00` or `+0000` suffix. A timestamp with no suffix is taken as UTC, so set your session time zone to UTC when you export; any other offset is counted as unreadable. A lenient date cast can misread a value like `8/5/26` and drop that row without a word, so anything else is counted, listed and fails the input check.

```
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(trim({col}), '{TS_RE}') THEN "
            f"try_strptime(substr(replace(trim({col}), 'T', ' '), 1, 19), '%Y-%m-%d %H:%M:%S') END")


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

## Step 3. Run seventeen input checks before building anything

A list built from incomplete records looks complete, which makes it a risky kind of evidence. The script runs these checks first, and each one names the kind of fault it catches; all seventeen trip in run 1. They are: duplicate Snowflake query ids; timestamps that are not `YYYY-MM-DD HH:MM:SS` in UTC; days with no records on a platform; PHI tables that match nothing in the exports; Databricks audit events, Snowflake access lists or Redshift mask column lists it could not read; Redshift GRANT or REVOKE statements on a PHI table it could not parse; Redshift grants, reads or writes that name a PHI table without its schema; Databricks grants on a catalog or schema that holds a PHI table; Snowflake roles granted to other roles that hold PHI access; Snowflake PHI tables granted to a database role; a Redshift masking view that returned no rows when `REDSHIFT_HAS_MASKING_POLICIES` says you have policies; blanks in the PHI column list, the register, the HR class file, the HR leaver file and the login map; and an HR last day not written YYYY-MM-DD. If any fails, all four lists are printed with `INCOMPLETE` in their heading.

## Step 4. Work out who can read each PHI column today

Each platform grants differently, so the script reduces each to one table, `held`: platform, login, the role or principal the access came through, the table, the column and the date it began. On Snowflake a role with `SELECT` on a table reaches every column of it, and a user reaches the role through GRANTS_TO_USERS. A grant counts if it began before the end of `AS_OF` and had not ended. A role nobody holds still gets one row, with no login, so list 2 can show it.

```
        con.execute(f"""CREATE TABLE sf_hid AS SELECT role AS via, obj, max(ts) AS since FROM sf_gr
              WHERE granted_on = 'TABLE' AND granted_to = 'ROLE' AND obj IN {in_scope('snowflake')} AND
privilege IN (SELECT p FROM privs)
              AND ts < TIMESTAMP '{asof_end}' AND (end_ts IS NULL OR end_ts >= TIMESTAMP '{asof_end}')
GROUP BY role, obj""")
        con.execute(f"""CREATE TABLE sf_mem AS SELECT role, login, ts FROM sf_gu
              WHERE ts < TIMESTAMP '{asof_end}' AND (end_ts IS NULL OR end_ts >= TIMESTAMP
'{asof_end}')""")
        nested = con.execute(f"""SELECT count(*) FROM sf_gr WHERE granted_on = 'ROLE' AND granted_to =
'ROLE'
              AND name IN (SELECT via FROM sf_hid) AND ts < TIMESTAMP '{asof_end}' AND (end_ts IS NULL OR
end_ts >= TIMESTAMP '{asof_end}')""").fetchone()[0]
        db_roles = con.execute(f"""SELECT count(*) FROM sf_gr WHERE granted_on = 'TABLE' AND granted_to =
'DATABASE_ROLE'
              AND obj IN {in_scope('snowflake')} AND privilege IN (SELECT p FROM privs)
              AND ts < TIMESTAMP '{asof_end}' AND (end_ts IS NULL OR end_ts >= TIMESTAMP
'{asof_end}')""").fetchone()[0]
        parts_h.append("""SELECT 'snowflake' AS platform, m.login, h.via, h.obj, p.col, greatest(h.since,
m.ts) AS since
              FROM sf_hid h JOIN phi p ON p.platform = 'snowflake' AND p.obj = h.obj LEFT JOIN sf_mem m
ON m.role = h.via""")
```

On Databricks, who holds access today comes from TABLE_PRIVILEGES, including privileges inherited from a schema or catalog, and on Redshift from SVV_RELATION_PRIVILEGES, plus the column-level grants that SVV_COLUMN_PRIVILEGES shows. Those views give the state on the day you export, however old the grant, when exported by a user who can see every grant. The date access began is the latest grant of a read privilege for that principal and table still in the audit log or in STL_UTILITYTEXT, whose GRANT and REVOKE text the script puts back together from its 200-character pieces (AWS documents the pieces (read 2026-11-01)); with no such record the date is blank. A group or role is expanded through the login map.

```
        # who holds access now comes from table_privileges (the state at the export, inherited grants
included), not
        # from replaying the audit log, which may not reach back to the grant; the latest add in the log
dates it
        con.execute(f"""CREATE TABLE dbx_hid AS SELECT t.via, t.obj, (SELECT max(c.ts) FROM dbx_chg c
WHERE c.principal = t.via AND c.obj = t.obj
              AND c.op = 'add' AND c.privilege IN (SELECT p FROM privs) AND c.ts < TIMESTAMP
'{asof_end}') AS since
              FROM (SELECT DISTINCT via, obj FROM dbx_tp WHERE obj IN {in_scope('databricks')} AND
privilege IN (SELECT p FROM privs)) t""")
```

## Step 5. List 1: does each person's access stay inside their class's register line?

Each person who can read a PHI column is matched to a class through the HR file, and each column to its category. A category the person's class does not list is `ACCESS BEYOND REGISTER`; a person with no class is `NO CLASS`. Logins and groups that match no person are printed, not skipped.

```
    sql = """
        WITH h AS (SELECT platform, email, obj, col, string_agg(DISTINCT via, '+' ORDER BY via) AS vias
FROM held
                   WHERE email IS NOT NULL GROUP BY platform, email, obj, col),
             hp AS (SELECT h.*, p.category FROM h JOIN phi p ON p.platform = h.platform AND p.obj = h.obj
AND p.col = h.col)
        SELECT hp.platform, hp.email,
               coalesce((SELECT string_agg(DISTINCT c.class, '; ' ORDER BY c.class) FROM hr_cls c WHERE
c.email = hp.email), '-') AS class,
               hp.category, count(DISTINCT hp.obj || '.' || hp.col) AS columns, string_agg(DISTINCT
hp.vias, '; ' ORDER BY hp.vias) AS via,
               CASE WHEN NOT EXISTS (SELECT 1 FROM hr_cls c WHERE c.email = hp.email) THEN 'NO CLASS'
                    WHEN NOT EXISTS (SELECT 1 FROM hr_cls c JOIN reg r ON r.class = c.class WHERE c.email
= hp.email AND r.category = hp.category)
                         THEN 'ACCESS BEYOND REGISTER' ELSE 'in register' END AS status
        FROM hp GROUP BY hp.platform, hp.email, hp.category ORDER BY hp.platform, hp.email,
hp.category"""
```

## Step 6. List 2: which roles can read each PHI column, and is a mask attached?

One row per PHI column and role or principal that can read it, with whether a mask is attached and whether any person who holds that role has a class whose register line lists the column's category. A role with no such person is `NO REGISTER LINE`, which covers a role nobody holds. A Snowflake masking policy's own conditions decide what each role sees (Snowflake: column-level security (read 2026-11-01)), and the script does not read that body, so for Snowflake and Databricks a mask attached means the column has one, not that a given role sees masked data. A Snowflake policy counts only while POLICY_REFERENCES reports it `ACTIVE`. A Redshift mask is matched by the name of its output column, compared whole, and only to the identity it is attached to, or to public. The script also prints the mask coverage per platform. Like the other lists, list 2 prints every row when it has thirty or fewer, and otherwise only the rows to explain; the CSV has every row.

```
    sql = """
        WITH x AS (SELECT DISTINCT platform, via, obj, col FROM held)
        SELECT x.platform, x.obj AS object, x.col AS "column", p.category, x.via,
               CASE WHEN EXISTS (SELECT 1 FROM masks m WHERE m.platform = x.platform AND m.obj = x.obj
AND m.col = x.col
                                 AND (m.grantee IS NULL OR m.grantee = x.via OR m.grantee = 'public'))
                    THEN 'mask attached' ELSE 'IN CLEAR' END AS masking,
              (SELECT count(DISTINCT h.email) FROM held h WHERE h.platform = x.platform AND h.via =
x.via AND h.obj = x.obj AND h.col = x.col
                 AND h.email IS NOT NULL) AS people,
               CASE WHEN NOT EXISTS (SELECT 1 FROM held h JOIN hr_cls c ON c.email = h.email JOIN reg r
ON r.class = c.class AND r.category = p.category
                                     WHERE h.platform = x.platform AND h.via = x.via AND h.obj = x.obj
```

```
AND h.col = x.col)
                    THEN 'NO REGISTER LINE' ELSE 'in register' END AS status
        FROM x JOIN phi p ON p.platform = x.platform AND p.obj = x.obj AND p.col = x.col
        ORDER BY x.platform, x.obj, x.col, x.via"""
```

## Step 7. List 3: access held, with no read seen

Access held for at least `NOT_USED_DAYS` with no read seen in that window is listed. It is a prompt for your access review, not proof the access is unneeded: an on-call role may sit unused by design, and Databricks and Redshift show reads at table level only, so those rows say `(table level)`. A grant with no date, such as a Redshift column grant, counts as held long enough.

```
    sql = f"""
        WITH h AS (SELECT platform, email, obj, CASE WHEN platform = 'snowflake' THEN col ELSE '(table
level)' END AS col, min(since) AS since
                   FROM held WHERE email IS NOT NULL GROUP BY ALL),
             r AS (SELECT h.*, (SELECT max(u.ts) FROM usage_r u WHERE u.platform = h.platform AND u.email
= h.email AND u.obj = h.obj
                                AND (h.col = '(table level)' OR u.col = h.col)) AS last_read FROM h)
        SELECT platform, email, obj AS object, col AS "column", since::DATE AS held_since,
last_read::DATE AS last_read_seen,
               CASE WHEN (since IS NULL OR since::DATE <= DATE '{asof}' - INTERVAL {not_used} DAY)
                         AND (last_read IS NULL OR last_read::DATE <= DATE '{asof}' - INTERVAL {not_used}
DAY)
                    THEN 'HELD, NO READ SEEN IN {not_used} DAYS' ELSE 'read within {not_used} days' END
AS status
        FROM r ORDER BY platform, email, obj, col"""
```

## Step 8. List 4: leavers who held PHI access

This is the leaver logic of our SOX access guide on PHI tables: each HR departure is handled on its own, so a rehire is not blurred, the first removal record is credited to the departure it belongs to, and a read after the removal counts. One addition: a leaver who still holds PHI access today is listed even if no grant record falls in the span.

```
        SELECT p.email, p.last_day, p.platform, r.removed_at::DATE AS first_removal_record,
               date_diff('day', p.last_day, r.removed_at::DATE) AS days_to_removal,
               a.n_grace AS reads_after_grace, a.n_removed AS reads_after_removal,
               CASE WHEN r.removed_at IS NULL THEN 'NO REMOVAL RECORD'
                    WHEN a.n_removed > 0 THEN 'ACTIVE AFTER REMOVAL'
                    WHEN date_diff('day', p.last_day, r.removed_at::DATE) > {GRACE_DAYS} THEN 'REMOVED
LATE'
                    ELSE 'removed in time' END AS status
        FROM pairs p LEFT JOIN rem r ON r.email = p.email AND r.platform = p.platform AND r.last_day =
p.last_day
             LEFT JOIN act a ON a.email = p.email AND a.platform = p.platform AND a.last_day = p.last_day
        ORDER BY p.last_day, p.email, p.platform"""
```

## Run it on your own data

Set EXPORT_DIR to the folder holding your export files, named as in the FILES list. Set PLATFORMS_EXPORTED to the platforms you exported. Set NOT_USED_DAYS to the number of days without a read after which held access is listed. Set PERIOD_START, PERIOD_END and AS_OF to your review period and the day you took the exports. If your PHI column list, register or HR files name their columns differently, set the settings ending in _COL to your own headers. Write each PHI table as your platform names it (database.schema.table for Snowflake and Databricks, schema.table for Redshift); case does not matter.

Add a row to the login map for every login, on any platform, that is not an email address, and for every member of a Databricks group or a Redshift group or role that holds access. Change nothing else, and run it without --demo.

We tested those steps on a second, separate set of generated files for a Snowflake-only reader: a different folder, table names written in mixed case, different column names in the PHI column list, the register and both HR files, upper-case emails against lower-case HR emails, a NULL catalog on a warehouse grant, a query with no objects, and Databricks and Redshift left out of `PLATFORMS_EXPORTED`. The script caught the planted user with no class, the planted user holding a role beyond their register line, the role nobody holds, five rows of access held with no read in thirty days, and the leaver with no removal record who read after the last day, and it printed Databricks and Redshift as NOT CHECKED. The Databricks and Redshift paths are exercised by the two runs in this guide, not by that second test.

## The code

The complete download, `hipaa_minimum_necessary_evidence.py`, also at www.datameaning.com/guides/hipaa-minimum-necessary-evidence-for-snowflake-databricks-redshift/hipaa_minimum_necessary_evidence.py.

```
#!/usr/bin/env python3
"""HIPAA minimum necessary evidence for PHI columns in Snowflake, Databricks and Redshift, in DuckDB.

    python3 hipaa_minimum_necessary_evidence.py --demo            # clean generated records
    python3 hipaa_minimum_necessary_evidence.py --demo --plant    # generated records with planted
faults
    python3 hipaa_minimum_necessary_evidence.py                   # your exports: set the settings below
first
    python3 hipaa_minimum_necessary_evidence.py --strict          # exit 1 unless nothing needs follow-
up

Reads each platform's own records, exported as CSV with the platform's published column names, plus your
role register (45 CFR 164.514(d)(2)(i)), a list of your PHI columns, an HR file of classes and one of
leavers, and a login map. Prints four lists for the PHI columns you name, each with its period, filters,
run time, row count and a SHA-256 of its rows:
  1. every person who can read a PHI column, against the categories their class may read in the register
     (ACCESS BEYOND REGISTER, NO CLASS)
  2. every PHI column, which roles or principals can read it, whether a mask is attached, and which of
them
     no register line covers
  3. access held for NOT_USED_DAYS with no read seen (a prompt for the access review, never proof the
```

```
     access is unneeded; Databricks and Redshift records show reads at table level only)
  4. every HR leaver in the period who held PHI access: the first removal record credited to that
     departure, the days between, and any read after the removal
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, and the result line names it.

The demo records are GENERATED (seed 229): not a company's data. Column names come from:
  Snowflake  https://docs.snowflake.com/en/sql-reference/account-usage/grants_to_users
             https://docs.snowflake.com/en/sql-reference/account-usage/grants_to_roles
             https://docs.snowflake.com/en/sql-reference/account-usage/policy_references
             https://docs.snowflake.com/en/sql-reference/account-usage/access_history
  Databricks https://docs.databricks.com/aws/en/admin/system-tables/audit-logs
             https://docs.databricks.com/aws/en/admin/system-tables/lineage
             https://docs.databricks.com/aws/en/sql/language-manual/information-schema/column_masks
             https://docs.databricks.com/aws/en/sql/language-manual/information-schema/table_privileges
  Redshift   https://docs.aws.amazon.com/redshift/latest/dg/r_STL_UTILITYTEXT.html
             https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_RELATION_PRIVILEGES.html
             https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_COLUMN_PRIVILEGES.html
             https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_ATTACHED_MASKING_POLICY.html
             https://docs.aws.amazon.com/redshift/latest/dg/SYS_QUERY_HISTORY.html
             https://docs.aws.amazon.com/redshift/latest/dg/SYS_USERLOG.html
Our assumptions, to check against one of your own rows: the JSON inside Databricks' updatePermissions
request_params (changes: a list of principal, add, remove) and the text in Redshift's output_columns.
"""
import csv
import datetime as dt
import hashlib
import json
import pathlib
import random
import re
import sys

import duckdb

# ---- CHANGE THESE FOR YOUR OWN DATA -------------------------------------------------------
PERIOD_START = "2026-07-01"          # first day of the review period (UTC, inclusive)
PERIOD_END = "2026-09-30"            # last 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
NOT_USED_DAYS = None                 # no default: access held this long with no read seen is listed
KNOWN_GAPS = {}                      # days you can explain, e.g. {"redshift": ["2026-08-19"]}
PHI_PLATFORM_COL = "platform"        # PHI column list: snowflake, databricks or redshift
PHI_TABLE_COL = "table"              # PHI column list: the table, named as the platform names it
PHI_COLUMN_COL = "column"            # PHI column list: the column
PHI_CATEGORY_COL = "category"        # PHI column list: the PHI category, as worded in the register
REGISTER_CLASS_COL = "class"         # role register: the class of workforce
REGISTER_CATEGORY_COL = "category"   # role register: a PHI category that class needs
REGISTER_CONDITIONS_COL = "conditions"   # role register: the conditions on that access (printed, not
checked)
HR_CLASS_EMAIL_COL = "email"         # HR class file: the work email
HR_CLASS_COL = "class"               # HR class file: the class, as worded in the register
HR_EMAIL_COL = "email"               # HR leaver file: the work email
HR_LAST_DAY_COL = "last_day"         # HR leaver file: the last day of work, as YYYY-MM-DD
GRACE_DAYS = 1                       # access must end within this many days of the last day of work
REMOVAL_WINDOW_DAYS = 30             # a removal before every last day counts if it is at most this many
days early
READ_PRIVILEGES = ["select", "ownership", "all", "all privileges"]   # privileges that let a holder read
REDSHIFT_HAS_MASKING_POLICIES = True  # False if your cluster has no masking policies, so an empty view
```

```
is expected
# ------------------------------------------------------------------------------------------

FILES = {
    "snowflake_grants_to_users": "snowflake_grants_to_users.csv",
    "snowflake_grants_to_roles": "snowflake_grants_to_roles.csv",
    "snowflake_policy_references": "snowflake_policy_references.csv",
    "snowflake_access_history": "snowflake_access_history.csv",
    "databricks_audit": "databricks_audit.csv",
    "databricks_table_lineage": "databricks_table_lineage.csv",
    "databricks_column_masks": "databricks_column_masks.csv",
    "databricks_table_privileges": "databricks_table_privileges.csv",
    "redshift_stl_utilitytext": "redshift_stl_utilitytext.csv",
    "redshift_svv_relation_privileges": "redshift_svv_relation_privileges.csv",
    "redshift_svv_column_privileges": "redshift_svv_column_privileges.csv",
    "redshift_svv_attached_masking_policy": "redshift_svv_attached_masking_policy.csv",
    "redshift_sys_query_history": "redshift_sys_query_history.csv",
    "redshift_sys_userlog": "redshift_sys_userlog.csv",
    "phi_columns": "phi_columns.csv",
    "register": "register.csv",
    "hr_classes": "hr_classes.csv",
    "hr_leavers": "hr_leavers.csv",
    "login_map": "login_map.csv",
}
PLATFORM_FILES = {
    "snowflake": ["snowflake_grants_to_users", "snowflake_grants_to_roles",
"snowflake_policy_references",
                  "snowflake_access_history"],
    "databricks": ["databricks_audit", "databricks_table_lineage", "databricks_column_masks",
"databricks_table_privileges"],
    "redshift": ["redshift_stl_utilitytext", "redshift_svv_relation_privileges",
"redshift_svv_column_privileges",
                 "redshift_svv_attached_masking_policy", "redshift_sys_query_history",
"redshift_sys_userlog"],
}
COLUMNS = {   # the platforms' published column names, lower-cased here; the file's own case does not
matter
    "snowflake_grants_to_users": ["created_on", "deleted_on", "role", "grantee_name"],
    "snowflake_grants_to_roles": ["created_on", "deleted_on", "privilege", "granted_on", "name",
"table_catalog",
                                  "table_schema", "granted_to", "grantee_name"],
    "snowflake_policy_references": ["policy_kind", "policy_status", "ref_database_name",
"ref_schema_name",
                                    "ref_entity_name", "ref_column_name"],
    "snowflake_access_history": ["query_id", "query_start_time", "user_name", "base_objects_accessed"],
    "databricks_audit": ["event_time", "user_identity", "service_name", "action_name", "request_params",
"response"],
    "databricks_table_lineage": ["source_table_full_name", "target_table_full_name", "created_by",
"event_time",
                                 "entity_type"],
    "databricks_column_masks": ["catalog_name", "schema_name", "table_name", "column_name",
"mask_name"],
    "databricks_table_privileges": ["grantee", "table_catalog", "table_schema", "table_name",
"privilege_type", "inherited_from"],
    "redshift_stl_utilitytext": ["userid", "xid", "pid", "starttime", "endtime", "sequence", "text"],
    "redshift_svv_relation_privileges": ["namespace_name", "relation_name", "privilege_type",
"identity_name", "identity_type"],
    "redshift_svv_column_privileges": ["namespace_name", "relation_name", "column_name",
"privilege_type",
                                       "identity_id", "identity_name", "identity_type"],
    "redshift_svv_attached_masking_policy": ["policy_name", "schema_name", "table_name", "grantee",
```

```
"grantee_type",
                                            "output_columns"],
    "redshift_sys_query_history": ["user_id", "username", "query_id", "query_type", "status",
"start_time", "query_text"],
    "redshift_sys_userlog": ["user_id", "user_name", "action", "record_time"],
    "login_map": ["platform", "login", "email"],
}
OUT_DIR = pathlib.Path("evidence-out")
GEN_DIR = pathlib.Path("generated")
DEMO_NOT_USED_DAYS = 60
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}"
# Redshift GRANT / REVOKE with one privilege list, one table and one grantee
DCL_RE = '(?i)^\\s*(grant|revoke)\\s+(.+?)\\s+on\\s+(?:table\\s+)?([\\w."$-]+)\\s+(?:to|from)\\s+
(?:group\\s+|role\\s+|user\\s+)?([\\w."$@-]+)\\s*;?\\s*$'
PLATFORMS = ["snowflake", "databricks", "redshift"]


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(trim({col}), '{TS_RE}') THEN "
            f"try_strptime(substr(replace(trim({col}), 'T', ' '), 1, 19), '%Y-%m-%d %H:%M:%S') END")


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


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()


# ---------------------------------------------------------------------------------------------
# Demo records: generated, in each platform's published column names
# ---------------------------------------------------------------------------------------------
def generate(plant):
    rnd = random.Random(229)
    out = GEN_DIR / ("planted" if plant else "clean")
    out.mkdir(parents=True, exist_ok=True)
    D = dt.date
    start, as_of = D(2026, 7, 1), D(2026, 10, 15)
    em = lambda k: f"{k}@example.test"
    rows = {k: [] for k in FILES}
    qn = [1000]
    sf_v, sf_c = "CLINIC_DB.RECORDS.PATIENT_VISITS", "CLINIC_DB.RECORDS.CLAIMS"
    db_v, db_c = "clinic.records.patient_visits", "clinic.records.claims"
    rs_v, rs_c = "records.patient_visits", "records.claims"

    def at(d, hour=None):
        return dt.datetime.combine(d, dt.time(hour if hour is not None else rnd.randrange(8, 18),
rnd.randrange(60), rnd.randrange(60)))

    sfts = lambda t: t.strftime("%Y-%m-%d %H:%M:%S.000 +0000")
    plain = lambda t: t.strftime("%Y-%m-%d %H:%M:%S")
    rs_ids = {"c02": 102, "b01": 103, "b02": 104, "q01": 105, "l01": 106, "a01": 201, "a02": 202, "bg":
301}
```

```
    # ---- the register, the PHI column list, HR
    rows["register"] = [["Clinician", "Demographics", "Patients assigned to them"], ["Clinician",
"Clinical", "Patients assigned to them"],
                        ["Billing staff", "Demographics", "Claims in their queue"], ["Billing staff",
"Billing", "Claims in their queue"],
                        ["Quality analyst", "Clinical", "Quality reviews only"]]
    if plant:
        rows["register"].append(["Billing staff", "", "a blank category"])
    cols = [("patient_visits", "patient_name", "Demographics"), ("patient_visits", "dob",
"Demographics"),
            ("patient_visits", "diagnosis_code", "Clinical"), ("claims", "patient_name",
"Demographics"),
            ("claims", "claim_amount", "Billing")]
    for plat, names in [("snowflake", {"patient_visits": sf_v, "claims": sf_c}), ("databricks",
{"patient_visits": db_v, "claims": db_c}),
                        ("redshift", {"patient_visits": rs_v, "claims": rs_c})]:
        for t, c, cat in cols:
            rows["phi_columns"].append([plat, names[t], c, cat])
    if plant:
        rows["phi_columns"].append(["redshift", "records.patient_vists", "patient_name",
"Demographics"])   # a typo
        rows["phi_columns"].append(["redshift", "records.claims", "", "Billing"])
# a blank column
    rows["hr_classes"] = [[em("c01"), "Clinician"], [em("c02"), "Clinician"], [em("b01"), "Billing
staff"], [em("b02"), "Billing staff"],
                          [em("q01"), "Quality analyst"], [em("l01"), "Billing staff"], [em("l02"),
"Clinician"]]
    rows["hr_leavers"] = [[em("l01"), "2026-08-07"], [em("l02"), "2026-09-04"]]
    if plant:
        rows["hr_leavers"] += [[em("n01"), "8/5/26"], ["", "2026-08-01"]]
        rows["hr_classes"].append([em("n02"), ""])
    rows["login_map"] = [["redshift", k, em(k)] for k in ("c02", "b01", "b02", "q01", "l01")] + \
        [["redshift", "billing_grp", em("b01")], ["redshift", "billing_grp", em("b02")],
         ["databricks", "billing-grp", em("b01")], ["databricks", "billing-grp", em("b02")]]
    if plant:
        rows["login_map"].append(["redshift", "", em("a02")])

    # ---- Snowflake: roles hold the table grants, users hold roles
    old = "2026-01-10 09:00:00.000 +0000"
    for role, tbl in [("CLINICIAN_ROLE", "PATIENT_VISITS"), ("BILLING_ROLE", "CLAIMS")] +
([("LEGACY_ROLE", "CLAIMS")] if plant else []):
        rows["snowflake_grants_to_roles"].append([old, "", "SELECT", "TABLE", tbl, "CLINIC_DB",
"RECORDS", "ROLE", role])
    rows["snowflake_grants_to_roles"].append([old, "", "SELECT", "TABLE", "DEPT_DIRECTORY", "HR_DB",
"PUBLIC", "ROLE", "ANALYST_ROLE"])
    if plant:   # a role handed to another role: its members are not in GRANTS_TO_USERS
        rows["snowflake_grants_to_roles"].append([old, "", "USAGE", "ROLE", "CLINICIAN_ROLE", "", "",
"ROLE", "SF_ENG_ROLE"])
        # a PHI table granted to a database role: who holds that database role is not followed here
        rows["snowflake_grants_to_roles"].append([old, "", "SELECT", "TABLE", "CLAIMS", "CLINIC_DB",
"RECORDS", "DATABASE_ROLE", "CLAIMS_READER"])
    users = [("c01", "CLINICIAN_ROLE", D(2026, 3, 2), None), ("c02", "CLINICIAN_ROLE", D(2026, 3, 3),
None),
             ("b01", "BILLING_ROLE", D(2026, 3, 4), None),
             ("l01", "BILLING_ROLE", D(2026, 5, 11), None if plant else D(2026, 8, 8)),
             ("l02", "CLINICIAN_ROLE", D(2026, 5, 12), D(2026, 9, 5))]
    if plant:
        users.append(("b02", "CLINICIAN_ROLE", D(2026, 8, 20), None))
    for u, role, g, end in users:
        rows["snowflake_grants_to_users"].append([sfts(at(g)), sfts(at(end)) if end else "", role,
em(u).upper()])
```

```
    if plant:
        rows["snowflake_grants_to_users"].append(["07/09/2026 10:00", "", "ANALYST_ROLE",
em("c01").upper()])
    for t, c in [("PATIENT_VISITS", "DOB"), ("CLAIMS", "PATIENT_NAME")]:
        rows["snowflake_policy_references"].append(["MASKING_POLICY", "ACTIVE", "CLINIC_DB", "RECORDS",
t, c])
    if plant:   # a policy Snowflake reports as not in force is not a mask
        rows["snowflake_policy_references"].append(["MASKING_POLICY",
"COLUMN_DATATYPE_MISMATCH_FOR_SECONDARY_ARG", "CLINIC_DB",
                                                    "RECORDS", "CLAIMS", "CLAIM_AMOUNT"])

    def sf_read(t, user, table, cs):
        qn[0] += 1
        js = json.dumps([{"objectName": table, "objectDomain": "Table", "columns": [{"columnName": c}
for c in cs]}])
        rows["snowflake_access_history"].append([f"sfq{qn[0]:06d}", sfts(t), em(user).upper(), js])

    # ---- Databricks: Unity Catalog permission events, lineage for reads
    def dbx(t, actor, action, params, status=200):
        rows["databricks_audit"].append([plain(t), json.dumps({"email": actor}), "unityCatalog", action,
json.dumps(params),
                                         json.dumps({"statusCode": status})])

    def dbx_perm(t, principal, op, table, stype="table"):
        dbx(t, em("a01"), "updatePermissions", {"securable_type": stype, "securable_full_name": table,
            "changes": json.dumps([{"principal": principal, op: ["SELECT"]}])})

    dbx_perm(at(D(2026, 3, 10)), em("c01"), "add", db_v)
    dbx_perm(at(D(2026, 3, 11)), "billing-grp", "add", db_c)
    dbx_perm(at(D(2026, 5, 12)), em("l02"), "add", db_v)
    dbx_perm(at(D(2026, 9, 20) if plant else D(2026, 9, 5)), em("l02"), "remove", db_v)
    if plant:
        dbx_perm(at(D(2026, 8, 25)), em("x01"), "add", db_c)
        dbx_perm(at(D(2026, 8, 26)), em("c02"), "add", "clinic.records", "schema")
    rows["databricks_column_masks"].append(["clinic", "records", "patient_visits", "dob", "mask_dob"])
    # who holds a privilege at the export: the current state, whatever the audit log still covers
    tp = [(em("c01"), "patient_visits", ""), ("billing-grp", "claims", "")]
    if plant:
        tp += [(em("x01"), "claims", ""), (em("c02"), "patient_visits", "clinic.records"), (em("c02"),
"claims", "clinic.records")]
    for who_, tbl, inh in tp:
        rows["databricks_table_privileges"].append([who_, "clinic", "records", tbl, "SELECT", inh])

    def lineage(t, user, table):
        rows["databricks_table_lineage"].append([table, "analytics.tmp.extract", em(user), plain(t),
"NOTEBOOK"])

    # ---- Redshift: GRANT text in STL_UTILITYTEXT, column grants and masks in the SVV views
    xid = [5000]

    def rs_dcl(t, admin, text):
        xid[0] += 1
        pad = text.ljust(((len(text) + 199) // 200) * 200)
        for seq in range(len(pad) // 200):
            rows["redshift_stl_utilitytext"].append([rs_ids[admin], xid[0], 9000 + xid[0], plain(t),
plain(t), seq, pad[seq * 200:(seq + 1) * 200]])

    def rs_q(t, user, qtype, text, status="success"):
        qn[0] += 1
        rows["redshift_sys_query_history"].append([rs_ids[user], user, qn[0], qtype, status, plain(t),
text])
```

```
    long_note = "/* ticket T-4411 " + "x" * 215 + " */ "
    rs_dcl(at(D(2026, 3, 5)), "a01", f"{long_note}GRANT SELECT ON TABLE {rs_v} TO c02;")
    rs_dcl(at(D(2026, 3, 6)), "a01", f"GRANT SELECT ON TABLE {rs_c} TO GROUP billing_grp;")
    rs_dcl(at(D(2026, 5, 13)), "a02", f"GRANT SELECT ON TABLE {rs_c} TO l01;")
    if not plant:
        rs_dcl(at(D(2026, 8, 8)), "a02", f"REVOKE SELECT ON TABLE {rs_c} FROM l01;")
    else:
        rs_dcl(at(D(2026, 8, 26)), "a01", f"GRANT SELECT ON TABLE {rs_c} TO b01, b02;")
    rp = [("patient_visits", "c02", "user"), ("claims", "billing_grp", "group")]
    if plant:
        rp += [("claims", "l01", "user"), ("claims", "b01", "user"), ("claims", "b02", "user")]
    for tbl, who_, typ in rp:
        rows["redshift_svv_relation_privileges"].append(["records", tbl, "SELECT", who_, typ])
    rows["redshift_svv_column_privileges"].append(["records", "patient_visits", "diagnosis_code",
"SELECT", 105, "q01", "user"])
    if not plant:   # a non-superuser sees 0 rows here: the planted run exports an empty view
        rows["redshift_svv_attached_masking_policy"].append(
            ["mask_dob", "records", "patient_visits", "public", "public", json.dumps([{"colname": "dob",
"type": "date"}])])
    rows["redshift_sys_userlog"].append([106, "l01", "alter", plain(at(D(2026, 5, 13)))])

    def every(a, b, step, fn):
        d = a
        while d <= min(b, as_of):
            fn(at(d))
            d += dt.timedelta(days=step)

    # ---- reads of the PHI tables
    vis, clm = ["PATIENT_NAME", "DOB", "DIAGNOSIS_CODE"], ["PATIENT_NAME", "CLAIM_AMOUNT"]
    every(D(2026, 7, 1), as_of, 6, lambda t: sf_read(t, "c01", sf_v, vis))
    every(D(2026, 7, 2), as_of, 6, lambda t: sf_read(t, "c02", sf_v, vis))
    every(D(2026, 7, 1), as_of, 5, lambda t: sf_read(t, "b01", sf_c, clm))
    every(D(2026, 7, 1), D(2026, 8, 7), 7, lambda t: sf_read(t, "l01", sf_c, clm))
    every(D(2026, 7, 1), D(2026, 9, 4), 9, lambda t: sf_read(t, "l02", sf_v, vis))
    every(D(2026, 7, 1), as_of, 7, lambda t: lineage(t, "c01", db_v))
    every(D(2026, 7, 1), as_of, 6, lambda t: lineage(t, "b01", db_c))
    every(D(2026, 7, 2), as_of, 8, lambda t: lineage(t, "b02", db_c))
    every(D(2026, 7, 1), D(2026, 9, 4), 10, lambda t: lineage(t, "l02", db_v))
    every(D(2026, 7, 1), D(2026, 7, 10) if plant else as_of, 5,
          lambda t: rs_q(t, "c02", "SELECT", f"SELECT patient_name FROM {rs_v} LIMIT 10"))
    every(D(2026, 7, 1), as_of, 7, lambda t: rs_q(t, "b01", "SELECT", f"SELECT claim_amount FROM {rs_c}
LIMIT 10"))
    every(D(2026, 7, 2), as_of, 9, lambda t: rs_q(t, "b02", "SELECT", 'SELECT claim_amount FROM
"records"."claims" LIMIT 10'))
    every(D(2026, 7, 3), as_of, 11, lambda t: rs_q(t, "q01", "SELECT", f"SELECT diagnosis_code FROM
{rs_v} LIMIT 10"))
    every(D(2026, 7, 1), D(2026, 8, 7), 8, lambda t: rs_q(t, "l01", "SELECT", f"SELECT claim_amount FROM
{rs_c} LIMIT 10"))
    # ---- background: every day has some records on every platform
    d = start
    while d <= as_of:
        sf_read(at(d), "bg", "HR_DB.PUBLIC.DEPT_DIRECTORY", ["DEPT"])
        dbx(at(d), em("bg"), "getTable", {"full_name_arg": "hr.dept.directory"})
        rs_q(at(d), "bg", "SELECT", "SELECT 1")
        d += dt.timedelta(days=1)
    rs_q(at(D(2026, 8, 12)), "a01", "SELECT", f"SELECT * FROM {rs_v}", status="failed")   # a refused
attempt is not a read
    if plant:
        every(D(2026, 8, 21), as_of, 8, lambda t: sf_read(t, "b02", sf_v, vis))               # b02
uses the clinician role
```

```
        sf_read(at(D(2026, 8, 20)), "l01", sf_c, clm)                                          # l01
reads after the last day
        sf_read(at(D(2026, 8, 25)), "l01", sf_c, clm)
        lineage(at(D(2026, 9, 10)), "l02", db_v)                                               # l02
reads before the late removal
        rows["snowflake_access_history"].append(list(rows["snowflake_access_history"][40]))    # a
repeated query id
        rows["snowflake_access_history"].append(["sfq999998", sfts(at(D(2026, 8, 5))),
em("c01").upper(), "[{not json"])
        rows["redshift_sys_query_history"] = [r for r in rows["redshift_sys_query_history"] if not
r[5].startswith("2026-08-19")]
        rows["redshift_stl_utilitytext"] = [r for r in rows["redshift_stl_utilitytext"] if not
r[3].startswith("2026-08-19")]
        rs_q(at(D(2026, 9, 2)), "b01", "SELECT", "SELECT claim_amount FROM claims LIMIT 10")   # no
schema named
    # ---- write
    for key, cs in COLUMNS.items():
        if key == "login_map":
            continue
        head = [c.upper() for c in cs] if key.startswith("snowflake") else cs
        with open(out / FILES[key], "w", newline="") as f:
            w = csv.writer(f)
            w.writerow(head)
            w.writerows(rows[key])
    for key, head in [("phi_columns", ["platform", "table", "column", "category"]), ("register",
["class", "category", "conditions"]),
                      ("hr_classes", ["email", "class"]), ("hr_leavers", ["email", "last_day"]),
                      ("login_map", ["platform", "login", "email"])]:
        with open(out / FILES[key], "w", newline="") as f:
            w = csv.writer(f)
            w.writerow(head)
            w.writerows(rows[key])
    return str(out)


# ---------------------------------------------------------------------------------------------
def load(con, key, folder, cols, renames=None):
    """Read a CSV as text with its header lower-cased; return a problem string or None."""
    path = pathlib.Path(folder) / FILES[key]
    if not path.is_file():
        return f"{FILES[key]} not found in {folder}"
    names = [r[0] for r in con.execute("DESCRIBE SELECT * FROM read_csv(?, header=true,
all_varchar=true)", [str(path)]).fetchall()]
    low = [n.strip().lower() for n in names]
    renames = renames or {}
    want = [renames.get(c, c) for c in cols]
    missing = [c for c in want if c not in low]
    if missing or len(set(low)) != len(low):
        return f"{FILES[key]} lacks column(s) {missing}" if missing else f"{FILES[key]} has a column
name twice"
    sel = ", ".join(f'"{names[low.index(c)]}" AS "{cols[i]}"' for i, c in enumerate(want))
    con.execute(f"CREATE TABLE raw_{key} AS SELECT {sel} FROM read_csv(?, header=true,
all_varchar=true)", [str(path)])
    return None


def population(con, name, title, filters, sql, header, complete, out_dir):
    rows = con.execute(sql).fetchall()
    out_dir.mkdir(exist_ok=True)
    with open(out_dir / f"{name}.csv", "w", newline="") as f:
        w = csv.writer(f)
```

```
        w.writerow(header)
        w.writerows(rows)
    print(f"\n== {title}: {name}" + ("" if complete else "   ** INCOMPLETE: see input checks **"))
    print(f"period    : {PERIOD_START} to {PERIOD_END} (UTC, inclusive); access as of {AS_OF}")
    print(f"filters   : {filters}")
    print(f"run at    : {dt.datetime.now(dt.timezone.utc):%Y-%m-%d %H:%M:%S} UTC")
    print(f"rows      : {len(rows)}")
    print(f"sha256    : {fingerprint(rows)}")
    return rows


def show(rows, flagged, status_at, limit=30):
    for r in rows:   # every row when there are 30 or fewer, otherwise only the ones to explain
        if len(rows) <= limit or flagged(r[status_at]):
            print("  " + " | ".join("-" if v is None else str(v) for v in r))


def main():
    demo, plant, strict = "--demo" in sys.argv, "--plant" in sys.argv, "--strict" in sys.argv
    folder, not_used = EXPORT_DIR, NOT_USED_DAYS
    if demo:
        folder = generate(plant)
        not_used = DEMO_NOT_USED_DAYS
        print(f"input     : GENERATED records (not a company's data), seed 229, plant={plant}")
    else:
        print("input     : " + str(folder))
    if not_used is None:
        print("STOP: NOT_USED_DAYS has no default. Say how many days without a read makes held access
worth a look, then run again.")
        return 2
    if folder is None:
        print("STOP: EXPORT_DIR is not set. Point it at the folder holding your export files, then run
again.")
        return 2
    bad_settings = [n for n, v in [("PERIOD_START", PERIOD_START), ("PERIOD_END", PERIOD_END), ("AS_OF",
AS_OF)]
                    if not re.fullmatch(DATE_RE, str(v))]
    if bad_settings or not PERIOD_START <= PERIOD_END <= AS_OF or not isinstance(not_used, int) or
not_used < 1:
        print("STOP: PERIOD_START, PERIOD_END and AS_OF must be YYYY-MM-DD with PERIOD_START <=
PERIOD_END <= AS_OF, and NOT_USED_DAYS a whole number of days.")
        return 2
    p0, p1, asof = PERIOD_START, PERIOD_END, AS_OF
    asof_end = (dt.date.fromisoformat(asof) + dt.timedelta(days=1)).isoformat()   # access "as of" the
end of AS_OF
    span0 = min(dt.date.fromisoformat(p0), dt.date.fromisoformat(asof) -
dt.timedelta(days=not_used)).isoformat()
    exported = [p for p in PLATFORMS if p in PLATFORMS_EXPORTED]
    not_checked = [p for p in PLATFORMS if p not in exported]
    if [p for p in PLATFORMS_EXPORTED if p not in PLATFORMS] or not exported:
        print(f"STOP: PLATFORMS_EXPORTED must list snowflake, databricks and/or redshift (got
{PLATFORMS_EXPORTED}).")
        return 2
    print("platforms : checked " + ", ".join(exported) + ("; NOT CHECKED: " + ", ".join(not_checked) if
not_checked else ""))
    con = duckdb.connect()

    # ---- load the files; a missing file or column stops the run: a check that cannot run is a failure
    problems = []
    for p in exported:
        for key in PLATFORM_FILES[p]:
```

```
            problems.append(load(con, key, folder, COLUMNS[key]))
    problems.append(load(con, "login_map", folder, COLUMNS["login_map"]))
    problems.append(load(con, "phi_columns", folder, ["platform", "table", "column", "category"],
                         {"platform": PHI_PLATFORM_COL.lower(), "table": PHI_TABLE_COL.lower(),
                          "column": PHI_COLUMN_COL.lower(), "category": PHI_CATEGORY_COL.lower()}))
    problems.append(load(con, "register", folder, ["class", "category", "conditions"],
                         {"class": REGISTER_CLASS_COL.lower(), "category":
REGISTER_CATEGORY_COL.lower(),
                          "conditions": REGISTER_CONDITIONS_COL.lower()}))
    problems.append(load(con, "hr_classes", folder, ["email", "class"],
                         {"email": HR_CLASS_EMAIL_COL.lower(), "class": HR_CLASS_COL.lower()}))
    problems.append(load(con, "hr_leavers", folder, ["email", "last_day"],
                         {"email": HR_EMAIL_COL.lower(), "last_day": HR_LAST_DAY_COL.lower()}))
    problems = [x for x in problems if x]
    if problems:
        print("\n== Input checks")
        for x in problems:
            print("[FAIL] " + x)
        print("RESULT: STOPPED: the files above could not be read, so no list was built")
        return 2

    norm = lambda x: x.strip().lower().replace("_", " ")   # ALL_PRIVILEGES (Databricks) is "all
privileges"
    con.execute("CREATE TABLE privs (p VARCHAR)")
    con.executemany("INSERT INTO privs VALUES (?)", [(norm(x),) for x in READ_PRIVILEGES])
    priv_list = "[" + ", ".join("'" + norm(x).replace("'", "''") + "'" for x in READ_PRIVILEGES) + "]"
    con.execute("""CREATE TABLE phi AS SELECT DISTINCT lower(trim(platform)) AS platform,
lower(trim(replace("table", '"', ''))) AS obj,
                   lower(trim("column")) AS col, lower(trim(category)) AS category FROM raw_phi_columns
                   WHERE trim(coalesce(platform, '')) <> '' AND trim(coalesce("table", '')) <> '' AND
trim(coalesce("column", '')) <> ''
                   AND trim(coalesce(category, '')) <> '' AND lower(trim(platform)) IN (%s)""" % ",
".join(f"'{p}'" for p in exported))
    con.execute("""CREATE TABLE reg AS SELECT DISTINCT lower(trim(class)) AS class,
lower(trim(category)) AS category FROM raw_register
                   WHERE trim(coalesce(class, '')) <> '' AND trim(coalesce(category, '')) <> ''""")
    con.execute("""CREATE TABLE hr_cls AS SELECT DISTINCT lower(trim(email)) AS email,
lower(trim(class)) AS class FROM raw_hr_classes
                   WHERE trim(coalesce(email, '')) <> '' AND trim(coalesce(class, '')) <> ''""")
    con.execute("""CREATE TABLE ident AS SELECT lower(trim(platform)) AS platform, lower(trim(login)) AS
login, lower(trim(email)) AS email
                   FROM raw_login_map WHERE trim(coalesce(login, '')) <> '' AND trim(coalesce(email,
'')) <> ''""")
    scope = {p: [r[0] for r in con.execute("SELECT DISTINCT obj FROM phi WHERE platform = ? ORDER BY 1",
[p]).fetchall()] for p in exported}
    con.execute("CREATE TABLE scope AS SELECT DISTINCT platform, obj FROM phi")
    phi_cols = {tuple(r) for r in con.execute("SELECT platform, obj, col FROM phi").fetchall()}
    # a login that is itself an email is the person; a group or role needs rows in login_map (one per
member)
    i_join = lambda a: f"LEFT JOIN ident i ON i.platform = {a}.platform AND i.login = {a}.login"
    i_mail = lambda a: f"coalesce(i.email, CASE WHEN {a}.login LIKE '%@%' THEN {a}.login END)"
    con.execute("CREATE MACRO who(p, l) AS coalesce((SELECT any_value(email) FROM ident WHERE
ident.platform = p AND ident.login = l), "
                "CASE WHEN l LIKE '%@%' THEN l END)")
    bad_ts = []     # (label, count) of timestamps that are not YYYY-MM-DD HH:MM:SS in UTC

    def count_bad(label, table, text_col, parsed_col, required=True):
        cond = f"{parsed_col} IS NULL" if required else f"trim(coalesce({text_col}, '')) <> '' AND
{parsed_col} IS NULL"
        bad_ts.append((label, con.execute(f"SELECT count(*) FROM {table} WHERE {cond}").fetchone()[0]))
```

```
    in_scope = lambda p: f"(SELECT obj FROM scope WHERE platform = '{p}')"
    # events: access held (grants_r), ends of access (ends_r), reads (usage_r); login level, with the
person's email
    parts_g, parts_e, parts_u, parts_h, parts_m = [], [], [], [], []
    dup = bad_json = bad_changes = failed_changes = bad_dcl = nested = scope_grants = db_roles =
inactive_masks = unqualified = 0

    # ---- Snowflake
    if "snowflake" in exported:
        con.execute(f"""CREATE TABLE sf_gu AS SELECT {ts('created_on')} AS ts, trim(created_on) AS
ts_text, {ts('deleted_on')} AS end_ts,
              trim(coalesce(deleted_on, '')) AS end_text, lower(trim(role)) AS role,
lower(trim(grantee_name)) AS login FROM raw_snowflake_grants_to_users""")
        con.execute(f"""CREATE TABLE sf_gr AS SELECT {ts('created_on')} AS ts, trim(created_on) AS
ts_text, {ts('deleted_on')} AS end_ts,
              trim(coalesce(deleted_on, '')) AS end_text, lower(replace(trim(privilege), '_', ' ')) AS
privilege,
              upper(trim(granted_on)) AS granted_on, upper(trim(granted_to)) AS granted_to,
              lower(trim(table_catalog) || '.' || trim(table_schema) || '.' || trim(name)) AS obj,
lower(trim(name)) AS name,
              lower(trim(grantee_name)) AS role FROM raw_snowflake_grants_to_roles""")
        # a masking policy counts only while Snowflake reports it ACTIVE
        con.execute("""CREATE TABLE sf_pol AS SELECT DISTINCT lower(trim(ref_database_name) || '.' ||
trim(ref_schema_name) || '.' || trim(ref_entity_name)) AS obj,
              lower(trim(ref_column_name)) AS col FROM raw_snowflake_policy_references
              WHERE upper(policy_kind) LIKE '%MASK%' AND upper(trim(coalesce(policy_status, ''))) =
'ACTIVE'""")
        inactive_masks = con.execute("""SELECT count(*) FROM raw_snowflake_policy_references
              WHERE upper(policy_kind) LIKE '%MASK%' AND upper(trim(coalesce(policy_status, ''))) <>
'ACTIVE'""").fetchone()[0]
        con.execute(f"""CREATE TABLE sf_ah AS SELECT trim(query_id) AS id, {ts('query_start_time')} AS
ts, lower(trim(user_name)) AS login,
              base_objects_accessed AS js FROM raw_snowflake_access_history""")
        count_bad("GRANTS_TO_USERS.CREATED_ON", "sf_gu", "ts_text", "ts")
        count_bad("GRANTS_TO_USERS.DELETED_ON", "sf_gu", "end_text", "end_ts", False)
        count_bad("GRANTS_TO_ROLES.CREATED_ON", "sf_gr", "ts_text", "ts")
        count_bad("GRANTS_TO_ROLES.DELETED_ON", "sf_gr", "end_text", "end_ts", False)
        count_bad("ACCESS_HISTORY.QUERY_START_TIME", "sf_ah", "ts", "ts")
        dup += con.execute("SELECT count(*) FROM (SELECT id FROM sf_ah GROUP BY id HAVING count(*) >
1)").fetchone()[0]
        reads = []
        for qid, t, login, js in con.execute("SELECT id, ts, login, js FROM sf_ah ORDER BY id, ts,
login").fetchall():
            try:
                for o in json.loads(js or "[]"):
                    name = str(o["objectName"]).strip().lower()
                    for c in o.get("columns") or []:
                        col = str(c["columnName"]).strip().lower()
                        if ("snowflake", name, col) in phi_cols and t is not None:
                            reads.append((t, login, name, col, "sf:" + str(qid)))
            except (ValueError, KeyError, TypeError, AttributeError):
                bad_json += 1
        con.execute("CREATE TABLE sf_reads (ts TIMESTAMP, login VARCHAR, obj VARCHAR, col VARCHAR, rid
VARCHAR)")
        if reads:
            con.executemany("INSERT INTO sf_reads VALUES (?, ?, ?, ?, ?)", reads)
        # roles that can read a PHI table now, and the users who hold them now
        con.execute(f"""CREATE TABLE sf_hid AS SELECT role AS via, obj, max(ts) AS since FROM sf_gr
              WHERE granted_on = 'TABLE' AND granted_to = 'ROLE' AND obj IN {in_scope('snowflake')} AND
privilege IN (SELECT p FROM privs)
              AND ts < TIMESTAMP '{asof_end}' AND (end_ts IS NULL OR end_ts >= TIMESTAMP '{asof_end}')
```

```
GROUP BY role, obj""")
        con.execute(f"""CREATE TABLE sf_mem AS SELECT role, login, ts FROM sf_gu
              WHERE ts < TIMESTAMP '{asof_end}' AND (end_ts IS NULL OR end_ts >= TIMESTAMP
'{asof_end}')""")
        nested = con.execute(f"""SELECT count(*) FROM sf_gr WHERE granted_on = 'ROLE' AND granted_to =
'ROLE'
              AND name IN (SELECT via FROM sf_hid) AND ts < TIMESTAMP '{asof_end}' AND (end_ts IS NULL
OR end_ts >= TIMESTAMP '{asof_end}')""").fetchone()[0]
        db_roles = con.execute(f"""SELECT count(*) FROM sf_gr WHERE granted_on = 'TABLE' AND granted_to
= 'DATABASE_ROLE'
              AND obj IN {in_scope('snowflake')} AND privilege IN (SELECT p FROM privs)
              AND ts < TIMESTAMP '{asof_end}' AND (end_ts IS NULL OR end_ts >= TIMESTAMP
'{asof_end}')""").fetchone()[0]
        parts_h.append("""SELECT 'snowflake' AS platform, m.login, h.via, h.obj, p.col,
greatest(h.since, m.ts) AS since
              FROM sf_hid h JOIN phi p ON p.platform = 'snowflake' AND p.obj = h.obj LEFT JOIN sf_mem m
ON m.role = h.via""")
        parts_m.append("SELECT 'snowflake' AS platform, obj, col, CAST(NULL AS VARCHAR) AS grantee FROM
sf_pol")
        # access to a table through a role runs while the user holds the role AND the role holds the
table, in either order
        con.execute(f"""CREATE TABLE sf_acc AS SELECT greatest(u.ts, r.ts) AS ts, least(u.end_ts,
r.end_ts) AS end_ts, u.login, u.role, r.obj
              FROM sf_gu u JOIN sf_gr r ON r.granted_on = 'TABLE' AND r.granted_to = 'ROLE' AND r.role =
u.role AND r.obj IN {in_scope('snowflake')}
              AND r.privilege IN (SELECT p FROM privs) AND r.ts IS NOT NULL
              AND r.ts < coalesce(u.end_ts, TIMESTAMP '9999-12-31') AND u.ts < coalesce(r.end_ts,
TIMESTAMP '9999-12-31')
              WHERE u.ts IS NOT NULL""")
        parts_g.append("SELECT 'snowflake' AS platform, ts, login, obj FROM sf_acc")
        # a removal is when the person's last PHI access on Snowflake ended, across all their logins: an
end while
        # another interval of the same person is open is not one
        parts_e.append("""SELECT 'snowflake' AS platform, e.end_ts AS ts, e.login, e.obj FROM sf_acc e
WHERE e.end_ts IS NOT NULL
              AND NOT EXISTS (SELECT 1 FROM sf_acc o WHERE coalesce(who('snowflake', o.login), o.login)
= coalesce(who('snowflake', e.login), e.login)
                              AND o.ts <= e.end_ts AND (o.end_ts IS NULL OR o.end_ts > e.end_ts))""")
        parts_u.append("SELECT 'snowflake' AS platform, ts, login, obj, col, rid FROM sf_reads")

    # ---- Databricks
    if "databricks" in exported:
        con.execute(f"""CREATE TABLE dbx_a AS SELECT {ts('event_time')} AS ts, trim(event_time) AS
ts_text,
              lower(trim(json_extract_string(user_identity, '$.email'))) AS login,
lower(trim(service_name)) AS service,
              trim(action_name) AS action, request_params,
              coalesce(json_extract_string(response, '$.statusCode'), json_extract_string(response,
'$.status_code')) AS status
              FROM raw_databricks_audit""")
        con.execute(f"""CREATE TABLE dbx_l AS SELECT 'dbx:' || row_number() OVER () AS rid,
lower(trim(source_table_full_name)) AS obj,
              lower(trim(created_by)) AS login, {ts('event_time')} AS ts, trim(event_time) AS ts_text
FROM raw_databricks_table_lineage""")
        con.execute("""CREATE TABLE dbx_tp AS SELECT lower(trim(grantee)) AS via,
              lower(trim(table_catalog) || '.' || trim(table_schema) || '.' || trim(table_name)) AS obj,
              lower(replace(trim(privilege_type), '_', ' ')) AS privilege FROM
raw_databricks_table_privileges""")
        con.execute("""CREATE TABLE dbx_m AS SELECT DISTINCT lower(trim(catalog_name) || '.' ||
trim(schema_name) || '.' || trim(table_name)) AS obj,
              lower(trim(column_name)) AS col FROM raw_databricks_column_masks""")
```

```
        count_bad("system.access.audit event_time", "dbx_a", "ts_text", "ts")
        count_bad("system.access.table_lineage event_time", "dbx_l", "ts_text", "ts")
        con.execute("CREATE TABLE dbx_chg (ts TIMESTAMP, obj VARCHAR, principal VARCHAR, op VARCHAR,
privilege VARCHAR)")
        chg = []
        dbx_scope = set(scope["databricks"])
        for t, status, rp in con.execute("""SELECT ts, status, request_params FROM dbx_a
              WHERE service = 'unitycatalog' AND action = 'updatePermissions' AND ts IS NOT NULL ORDER
BY ts, request_params""").fetchall():
            try:
                p = json.loads(rp)
                obj = str(p["securable_full_name"]).strip().lower()
                stype = str(p.get("securable_type", "")).strip().lower()
                inside = any(o.startswith(obj + ".") for o in dbx_scope)   # a catalog or schema holding
a PHI table
                if obj not in dbx_scope and not inside:   # a change on another object says nothing here
                    continue
                if status is None or not status.strip().isdigit():   # unreadable: the change cannot be
judged
                    bad_changes += 1
                    continue
                if status.strip() != "200":
                    failed_changes += 1
                    continue
                if obj not in dbx_scope or stype != "table":
                    scope_grants += 1
                    continue
                ch = p["changes"]
                for c in (json.loads(ch) if isinstance(ch, str) else ch):
                    for op in ("add", "remove"):
                        for pr in c.get(op) or []:
                            chg.append((t, obj, str(c["principal"]).strip().lower(), op, norm(str(pr))))
            except (ValueError, KeyError, TypeError, AttributeError):
                bad_changes += 1
        if chg:
            con.executemany("INSERT INTO dbx_chg VALUES (?, ?, ?, ?, ?)", chg)
        # who holds access now comes from table_privileges (the state at the export, inherited grants
included), not
        # from replaying the audit log, which may not reach back to the grant; the latest add in the log
dates it
        con.execute(f"""CREATE TABLE dbx_hid AS SELECT t.via, t.obj, (SELECT max(c.ts) FROM dbx_chg c
WHERE c.principal = t.via AND c.obj = t.obj
             AND c.op = 'add' AND c.privilege IN (SELECT p FROM privs) AND c.ts < TIMESTAMP
'{asof_end}') AS since
             FROM (SELECT DISTINCT via, obj FROM dbx_tp WHERE obj IN {in_scope('databricks')} AND
privilege IN (SELECT p FROM privs)) t""")
        parts_h.append("""SELECT 'databricks' AS platform, h.via AS login, h.via, h.obj, p.col, h.since
FROM dbx_hid h
             JOIN phi p ON p.platform = 'databricks' AND p.obj = h.obj""")
        parts_m.append("SELECT 'databricks' AS platform, obj, col, CAST(NULL AS VARCHAR) AS grantee FROM
dbx_m")
        parts_g.append("SELECT 'databricks' AS platform, ts, principal AS login, obj FROM dbx_chg WHERE
op = 'add' AND privilege IN (SELECT p FROM privs)")
        parts_e.append("SELECT 'databricks' AS platform, ts, principal AS login, obj FROM dbx_chg WHERE
op = 'remove' AND privilege IN (SELECT p FROM privs)")
        parts_u.append(f"SELECT 'databricks' AS platform, ts, login, obj, CAST(NULL AS VARCHAR) AS col,
rid FROM dbx_l WHERE obj IN {in_scope('databricks')} AND ts IS NOT NULL")

    # ---- Redshift
    if "redshift" in exported:
        con.execute(f"""CREATE TABLE rs_qh AS SELECT trim(user_id) AS uid, lower(trim(username)) AS
```

```
login, trim(query_id) AS id,
              upper(trim(query_type)) AS query_type, lower(trim(status)) AS status, {ts('start_time')}
AS ts,
              lower(replace(query_text, '"', '')) AS text FROM raw_redshift_sys_query_history""")
        con.execute(f"""CREATE TABLE rs_ut AS SELECT trim(userid) AS uid, trim(xid) AS xid, trim(pid) AS
pid, trim(starttime) AS st_text,
              {ts('starttime')} AS ts, try_cast(sequence AS INTEGER) AS seq, text FROM
raw_redshift_stl_utilitytext""")
        con.execute(f"""CREATE TABLE rs_ul AS SELECT lower(trim(user_name)) AS login,
lower(trim(action)) AS action,
              {ts('record_time')} AS ts, trim(record_time) AS ts_text FROM raw_redshift_sys_userlog""")
        count_bad("SYS_QUERY_HISTORY.start_time", "rs_qh", "ts", "ts")
        count_bad("STL_UTILITYTEXT.starttime", "rs_ut", "st_text", "ts")
        count_bad("SYS_USERLOG.record_time", "rs_ul", "ts_text", "ts")
        # a statement longer than 200 characters is stored in pieces: put them back in order, as AWS
shows
        con.execute(f"""CREATE TABLE rs_dcl AS
            WITH st AS (SELECT uid, xid, pid, st_text, min(ts) AS ts,
                           string_agg(CASE WHEN len(rtrim(text)) = 0 THEN text ELSE rtrim(text) END, ''
ORDER BY seq) AS stmt
                       FROM rs_ut GROUP BY uid, xid, pid, st_text),
                 cl AS (SELECT *, trim(regexp_replace(stmt, '/\\*.*?\\*/', '', 'gs')) AS s FROM st)
            SELECT uid, ts, s, lower(regexp_extract(s, '{DCL_RE}', 1)) AS kind, regexp_extract(s,
'{DCL_RE}', 2) AS privs,
                   lower(replace(regexp_extract(s, '{DCL_RE}', 3), '"', '')) AS obj,
                   lower(replace(regexp_extract(s, '{DCL_RE}', 4), '"', '')) AS grantee
            FROM cl WHERE regexp_matches(s, '(?i)^\\s*(grant|revoke)\\b')""")
        bad_dcl = con.execute("""SELECT count(*) FROM rs_dcl WHERE kind = '' AND EXISTS (
              SELECT 1 FROM scope WHERE platform = 'redshift' AND contains(lower(s),
obj))""").fetchone()[0]
        con.execute(f"""CREATE TABLE rs_rd AS SELECT ts, obj, grantee AS login, kind FROM rs_dcl WHERE
kind IN ('grant', 'revoke') AND ts IS NOT NULL
              AND obj IN {in_scope('redshift')}
              AND list_has_any(list_transform(string_split(lower(replace(privs, '_', ' ')), ','), lambda
x: trim(x)), {priv_list})""")
        # who holds access now comes from SVV_RELATION_PRIVILEGES (the state at the export), not from
replaying
        # STL_UTILITYTEXT, which keeps seven days; the latest GRANT still in the text dates it
        con.execute("""CREATE TABLE rs_rp AS SELECT lower(trim(identity_name)) AS via,
lower(trim(namespace_name) || '.' || trim(relation_name)) AS obj,
              lower(trim(privilege_type)) AS privilege FROM raw_redshift_svv_relation_privileges""")
        con.execute(f"""CREATE TABLE rs_hid AS SELECT t.via, t.obj, (SELECT max(g.ts) FROM rs_rd g WHERE
g.login = t.via AND g.obj = t.obj
              AND g.kind = 'grant' AND g.ts < TIMESTAMP '{asof_end}') AS since
              FROM (SELECT DISTINCT via, obj FROM rs_rp WHERE obj IN {in_scope('redshift')} AND
privilege IN (SELECT p FROM privs)) t""")
        # a grant or read that names a PHI table without its schema cannot be matched to it
        bare = {o.split(".")[-1]: o for o in scope["redshift"]}
        unqualified = con.execute("SELECT count(*) FROM rs_dcl WHERE kind IN ('grant', 'revoke') AND obj
IN (SELECT unnest(?::VARCHAR[]))",
                                  [list(bare)]).fetchone()[0]
        for b in sorted(bare):
            unqualified += con.execute("""SELECT count(*) FROM (SELECT
regexp_replace(regexp_replace(text, '/\\*.*?\\*/', '', 'gs'), '--[^\\n]*', '', 'g') AS t, status
                  FROM rs_qh) WHERE status NOT IN ('failed', 'canceled') AND NOT regexp_matches(t,
'^\\s*(grant|revoke)\\b')
                  AND (regexp_matches(t, '\\b(?:from|join|into|update|using)\\s+' || ? || '(?:$|
[^\\w.])')
                       OR regexp_matches(t, '\\bfrom\\s+(?:[\\w.]+(?:\\s+(?:as\\s+)?\\w+)?\\s*,\\s*)+'
|| ? || '(?:$|[^\\w.])'))""",
                  [re.escape(b), re.escape(b)]).fetchone()[0]
```

```
        con.execute("""CREATE TABLE rs_cp AS SELECT DISTINCT lower(trim(identity_name)) AS via,
              lower(trim(namespace_name) || '.' || trim(relation_name)) AS obj, lower(trim(column_name))
AS col
              FROM raw_redshift_svv_column_privileges WHERE upper(trim(privilege_type)) = 'SELECT'""")
        # the masked column is the policy's output column; each name is compared whole
        mk = []
        for obj, grantee, gtype, oc in con.execute("""SELECT lower(trim(schema_name) || '.' ||
trim(table_name)), lower(trim(grantee)),
              lower(trim(grantee_type)), output_columns FROM raw_redshift_svv_attached_masking_policy
ORDER BY ALL""").fetchall():
            try:
                v = json.loads(oc)
                v = v if isinstance(v, list) else [v]
                names = [str(x.get("colname") or x["name"]) if isinstance(x, dict) else str(x) for x in
v]
            except (ValueError, KeyError, TypeError, AttributeError):
                names = [x.strip(' "()') for x in str(oc or "").split(",")]
            names = [n.strip().strip('"').lower() for n in names if n and n.strip().strip('"')]
            if not names:
                bad_json += 1
            mk += [(obj, grantee, gtype, n) for n in names]
        con.execute("CREATE TABLE rs_mk (obj VARCHAR, grantee VARCHAR, gtype VARCHAR, col VARCHAR)")
        if mk:
            con.executemany("INSERT INTO rs_mk VALUES (?, ?, ?, ?)", mk)
        parts_h.append("""SELECT 'redshift' AS platform, h.via AS login, h.via, h.obj, p.col, h.since
FROM rs_hid h
              JOIN phi p ON p.platform = 'redshift' AND p.obj = h.obj""")
        parts_h.append(f"""SELECT 'redshift' AS platform, c.via AS login, c.via, c.obj, c.col, CAST(NULL
AS TIMESTAMP) AS since FROM rs_cp c
              JOIN phi p ON p.platform = 'redshift' AND p.obj = c.obj AND p.col = c.col""")
        parts_m.append("""SELECT 'redshift' AS platform, k.obj, p.col, CASE WHEN k.gtype = 'public' THEN
'public' ELSE k.grantee END AS grantee
              FROM rs_mk k JOIN phi p ON p.platform = 'redshift' AND p.obj = k.obj AND p.col = k.col""")
        parts_g.append("SELECT 'redshift' AS platform, ts, login, obj FROM rs_rd WHERE kind = 'grant'")
        parts_e.append("SELECT 'redshift' AS platform, ts, login, obj FROM rs_rd WHERE kind = 'revoke'")
        parts_e.append("SELECT 'redshift' AS platform, ts, login, CAST(NULL AS VARCHAR) AS obj FROM
rs_ul WHERE action = 'drop' AND ts IS NOT NULL")
        # a read is a statement that names the table and did not fail or get canceled; GRANT and REVOKE
are not reads
        parts_u.append("""SELECT 'redshift' AS platform, q.ts, q.login, s.obj, CAST(NULL AS VARCHAR) AS
col, 'rs:' || q.id AS rid FROM (
              SELECT *, regexp_replace(regexp_replace(text, '/\\*.*?\\*/', '', 'gs'), '--[^\\n]*', '',
'g') AS t FROM rs_qh) q
              JOIN scope s ON s.platform = 'redshift'
              AND regexp_matches(q.t, '(?:^|[^\\w.])' || regexp_replace(s.obj, '\\.', '\\\\.', 'g') ||
'(?:$|[^\\w.])')
              WHERE q.status NOT IN ('failed', 'canceled') AND q.ts IS NOT NULL AND NOT
regexp_matches(q.t, '^\\s*(grant|revoke)\\b')""")

    # who holds access now (held), identities included: a role nobody holds still has one row with no
login
    con.execute("CREATE TABLE held_raw AS " + " UNION ALL ".join(parts_h))
    con.execute("CREATE TABLE masks AS " + " UNION ALL ".join(parts_m))
    con.execute(f"""CREATE TABLE held AS SELECT x.platform, x.login, x.via, x.obj, x.col, x.since,
{i_mail('x')} AS email
          FROM held_raw x {i_join('x')}""")
    con.execute("CREATE TABLE grants_r AS SELECT x.platform, x.ts, " + i_mail("x") + " AS email, x.obj
FROM (" +
                " UNION ALL ".join(parts_g) + ") x " + i_join("x"))
    con.execute("CREATE TABLE ends_r AS SELECT x.platform, x.ts, " + i_mail("x") + " AS email, x.obj
FROM (" +
```

```
                " UNION ALL ".join(parts_e) + ") x " + i_join("x"))
    con.execute("CREATE TABLE usage_r AS SELECT platform, ts, login, who(platform, login) AS email, obj,
col, rid FROM (" +
                " UNION ALL ".join(parts_u) + ")")

    # ---- people the lists need
    con.execute(f"""CREATE TABLE leavers AS SELECT lower(trim(coalesce(email, ''))) AS email,
trim(coalesce(last_day, '')) AS last_day_text,
          {day('last_day')} AS last_day FROM raw_hr_leavers""")

    # ---- input checks: each says which planted fault trips it
    days = {}
    for p, sql in [("snowflake", "SELECT DISTINCT ts::DATE FROM sf_ah WHERE ts IS NOT NULL"),
                   ("databricks", "SELECT DISTINCT ts::DATE FROM dbx_a WHERE ts IS NOT NULL"),
                   ("redshift", "SELECT DISTINCT ts::DATE FROM rs_qh WHERE ts IS NOT NULL")]:
        if p in exported:
            have = {r[0].isoformat() for r in con.execute(sql).fetchall()}
            allday = [r[0].isoformat() for r in con.execute(
                f"SELECT d::DATE FROM generate_series(DATE '{span0}', DATE '{asof}', INTERVAL 1 DAY)
t(d) ORDER BY 1").fetchall()]
            days[p] = [d for d in allday if d not in have and d not in set(KNOWN_GAPS.get(p, []))]
    unseen = []
    for p in exported:
        for obj in scope[p]:
            seen = con.execute("""SELECT (SELECT count(*) FROM held_raw WHERE platform = ? AND obj = ?)
+ (SELECT count(*) FROM usage_r WHERE platform = ? AND obj = ?)
                                  + (SELECT count(*) FROM grants_r WHERE platform = ? AND obj = ?) +
(SELECT count(*) FROM ends_r WHERE platform = ? AND obj = ?)
                                  + (SELECT count(*) FROM masks WHERE platform = ? AND obj = ?)""", [p,
obj] * 5).fetchone()[0]
            if seen == 0:
                unseen.append(f"{p}:{obj}")
    no_scope = [p for p in exported if not scope[p]]
    blank = lambda t, cond: con.execute(f"SELECT count(*) FROM {t} WHERE {cond}").fetchone()[0]
    nb = lambda c: f"trim(coalesce({c}, '')) = ''"
    phi_blank = blank("raw_phi_columns", " OR ".join(nb(f'"{c}"') for c in ["platform", "table",
"column", "category"]))
    reg_blank = blank("raw_register", " OR ".join(nb(c) for c in ["class", "category"]))
    cls_blank = blank("raw_hr_classes", " OR ".join(nb(c) for c in ["email", "class"]))
    hr_bad = con.execute("SELECT count(*) FROM leavers WHERE email = '' OR last_day_text =
''").fetchone()[0]
    hr_date = con.execute("SELECT count(*) FROM leavers WHERE last_day_text <> '' AND last_day IS
NULL").fetchone()[0]
    map_bad = blank("raw_login_map", " OR ".join(nb(c) for c in ["login", "email"]))
    rs_mask_empty = 1 if "redshift" in exported and REDSHIFT_HAS_MASKING_POLICIES and con.execute(
        "SELECT count(*) FROM raw_redshift_svv_attached_masking_policy").fetchone()[0] == 0 else 0
    checks = [
        ("duplicate Snowflake query ids", dup, "a repeated query id"),
        ("timestamps not written YYYY-MM-DD HH:MM:SS (UTC)", sum(n for _, n in bad_ts), "a timestamp
written 07/09/2026 10:00"),
        ("days with no records, per platform, from the start of the span to AS_OF", sum(len(v) for v in
days.values()), "a day with no Redshift records"),
        ("PHI tables that match no grant, read, removal or mask in the exports", len(unseen) +
len(no_scope), "a typo in the PHI column list"),
        ("Databricks audit events, Snowflake access lists or Redshift mask columns the script could not
read", bad_changes + bad_json, "malformed JSON"),
        ("Redshift GRANT or REVOKE on a PHI table the script could not parse", bad_dcl, "a grant to two
users in one statement"),
        ("Redshift grants, reads or writes that name a PHI table without its schema (not matched here)",
unqualified, "a read FROM claims"),
        ("Databricks grants on a catalog or schema that holds a PHI table (removal not followed in the
```

```
leaver list)", scope_grants, "a schema-level grant"),
        ("Snowflake roles granted to other roles that hold PHI access (members not followed here)",
nested, "a role granted to a role"),
        ("Snowflake PHI tables granted to database roles (holders not followed here)", db_roles, "a
grant to a database role"),
        ("Redshift masking policy view returned 0 rows (it needs a superuser or sys:secadmin; set
REDSHIFT_HAS_MASKING_POLICIES = False if there are none)",
         rs_mask_empty, "an empty SVV_ATTACHED_MASKING_POLICY"),
        ("PHI column list rows with a blank platform, table, column or category", phi_blank, "a blank in
the PHI column list"),
        ("register rows with a blank class or category", reg_blank, "a blank category in the register"),
        ("HR class rows with a blank email or class", cls_blank, "a blank in the HR class file"),
        ("HR leaver rows with no email or no last day", hr_bad, "a NULL in the HR leaver file"),
        ("HR leaver rows whose last day is not written as YYYY-MM-DD", hr_date, "a last day written
8/5/26"),
        ("login map rows with a blank login or email", map_bad, "a blank in the login map"),
    ]
    print(f"\n== Input checks   (span checked for records: {span0} to {asof})")
    ok = True
    for label, n, fault in checks:
        ok &= n == 0
        print(f"[{'ok  ' if n == 0 else 'FAIL'}] {label}: {n}" + ("" if n == 0 else f"   (trips on:
{fault})"))
    for label, n in bad_ts:
        if n:
            print(f"        unreadable: {label}: {n}")
    for p, v in days.items():
        if v:
            print(f"        {p} days with no records: " + ", ".join(v[:12]) + (f" ... ({len(v)} days)"
if len(v) > 12 else ""))
            if p == "redshift":
                print("        the STL views keep seven days (AWS): export daily, or unload the logs to
Amazon S3")
    if unseen or no_scope:
        print("        PHI tables: " + ", ".join(unseen + [f"{p}: no tables named" for p in no_scope]))
    for p in not_checked:
        print(f"[NOT CHECKED] {p}: not in PLATFORMS_EXPORTED, so nothing here says anything about it")
    print(f"[info] permission changes that failed (status not 200), left out: {failed_changes}")
    if "snowflake" in exported:
        print(f"[info] Snowflake masking policies not ACTIVE, not counted as a mask: {inactive_masks}")
    print("[info] conditions in the register are printed with the register, not checked against any
record")

    # ---- list 1: access beyond the register
    sql = """
        WITH h AS (SELECT platform, email, obj, col, string_agg(DISTINCT via, '+' ORDER BY via) AS vias
FROM held
                   WHERE email IS NOT NULL GROUP BY platform, email, obj, col),
             hp AS (SELECT h.*, p.category FROM h JOIN phi p ON p.platform = h.platform AND p.obj =
h.obj AND p.col = h.col)
        SELECT hp.platform, hp.email,
               coalesce((SELECT string_agg(DISTINCT c.class, '; ' ORDER BY c.class) FROM hr_cls c WHERE
c.email = hp.email), '-') AS class,
               hp.category, count(DISTINCT hp.obj || '.' || hp.col) AS columns, string_agg(DISTINCT
hp.vias, '; ' ORDER BY hp.vias) AS via,
               CASE WHEN NOT EXISTS (SELECT 1 FROM hr_cls c WHERE c.email = hp.email) THEN 'NO CLASS'
                    WHEN NOT EXISTS (SELECT 1 FROM hr_cls c JOIN reg r ON r.class = c.class WHERE
c.email = hp.email AND r.category = hp.category)
                         THEN 'ACCESS BEYOND REGISTER' ELSE 'in register' END AS status
        FROM hp GROUP BY hp.platform, hp.email, hp.category ORDER BY hp.platform, hp.email,
hp.category"""
```

```
    l1 = population(con, "hipaa_1_access_beyond_register", "Who can read which PHI category, against the
register",
                    "people holding read access (" + ", ".join(sorted(set(READ_PRIVILEGES))) + ") to a
column in the PHI column list, by grant or by role; one row per person, platform and category",
                    sql, ["platform", "email", "class", "category", "columns", "via", "status"], ok,
OUT_DIR)
    show(l1, lambda s: s != "in register", 6)
    unmatched = con.execute("""SELECT DISTINCT platform, login FROM held WHERE email IS NULL AND login
IS NOT NULL ORDER BY 1, 2""").fetchall()
    print(f"[info] logins or groups with read access that match no person (add them to login_map):
{len(unmatched)}")
    for r in unmatched:
        print("  UNMATCHED LOGIN | " + " | ".join(str(v) for v in r))

    # ---- list 2: unmasked PHI by role
    for p in exported:
        n_col, n_mask = con.execute("""SELECT count(*), count(*) FILTER (WHERE EXISTS (SELECT 1 FROM
masks m WHERE m.platform = p.platform AND m.obj = p.obj AND m.col = p.col))
                                       FROM phi p WHERE p.platform = ?""", [p]).fetchone()
        print(f"[info] mask coverage, {p}: {n_col} PHI columns listed, {n_mask} with a mask attached,
{n_col - n_mask} without")
    sql = """
        WITH x AS (SELECT DISTINCT platform, via, obj, col FROM held)
        SELECT x.platform, x.obj AS object, x.col AS "column", p.category, x.via,
               CASE WHEN EXISTS (SELECT 1 FROM masks m WHERE m.platform = x.platform AND m.obj = x.obj
AND m.col = x.col
                                 AND (m.grantee IS NULL OR m.grantee = x.via OR m.grantee = 'public'))
                    THEN 'mask attached' ELSE 'IN CLEAR' END AS masking,
              (SELECT count(DISTINCT h.email) FROM held h WHERE h.platform = x.platform AND h.via =
x.via AND h.obj = x.obj AND h.col = x.col
                  AND h.email IS NOT NULL) AS people,
               CASE WHEN NOT EXISTS (SELECT 1 FROM held h JOIN hr_cls c ON c.email = h.email JOIN reg r
ON r.class = c.class AND r.category = p.category
                                     WHERE h.platform = x.platform AND h.via = x.via AND h.obj = x.obj
AND h.col = x.col)
                    THEN 'NO REGISTER LINE' ELSE 'in register' END AS status
        FROM x JOIN phi p ON p.platform = x.platform AND p.obj = x.obj AND p.col = x.col
        ORDER BY x.platform, x.obj, x.col, x.via"""
    l2 = population(con, "hipaa_2_unmasked_by_role", "PHI columns, who can read them and whether a mask
is attached",
                    "one row per PHI column and role or principal that can read it; a Snowflake or
Databricks mask names the column, and who sees it in the clear depends on the policy's own body, which
is not read here; Redshift masks match only the identity they are attached to, or public",
                    sql, ["platform", "object", "column", "category", "role_or_principal", "masking",
"people", "status"], ok, OUT_DIR)
    show(l2, lambda s: s != "in register", 7)

    # ---- list 3: held, not used
    sql = f"""
        WITH h AS (SELECT platform, email, obj, CASE WHEN platform = 'snowflake' THEN col ELSE '(table
level)' END AS col, min(since) AS since
                   FROM held WHERE email IS NOT NULL GROUP BY ALL),
             r AS (SELECT h.*, (SELECT max(u.ts) FROM usage_r u WHERE u.platform = h.platform AND
u.email = h.email AND u.obj = h.obj
                                AND (h.col = '(table level)' OR u.col = h.col)) AS last_read FROM h)
        SELECT platform, email, obj AS object, col AS "column", since::DATE AS held_since,
last_read::DATE AS last_read_seen,
               CASE WHEN (since IS NULL OR since::DATE <= DATE '{asof}' - INTERVAL {not_used} DAY)
                         AND (last_read IS NULL OR last_read::DATE <= DATE '{asof}' - INTERVAL
{not_used} DAY)
                    THEN 'HELD, NO READ SEEN IN {not_used} DAYS' ELSE 'read within {not_used} days' END
```

```
AS status
        FROM r ORDER BY platform, email, obj, col"""
    l3 = population(con, "hipaa_3_held_not_used", "PHI access held with no read seen",
                    f"read access held at the end of {asof} and held for at least {not_used} days (a
grant with no date counts), with no read in the {not_used} days to AS_OF; Snowflake per column,
Databricks and Redshift per table; a prompt for the access review, not proof the access is unneeded;
Databricks lineage is a subset of read events",
                    sql, ["platform", "email", "object", "column", "held_since", "last_read_seen",
"status"], ok, OUT_DIR)
    show(l3, lambda s: s.startswith("HELD"), 6)

    # ---- list 4: leavers (the leaver logic of the SOX guide, plus: still holding access now)
    plat_vals = ", ".join(f"('{p}')" for p in exported)
    sql = f"""
        WITH lv AS (SELECT email, last_day, lag(last_day) OVER (PARTITION BY email ORDER BY last_day) AS
prev_last_day,
                           lead(last_day) OVER (PARTITION BY email ORDER BY last_day) AS next_last_day
                    FROM leavers WHERE email <> '' AND last_day IS NOT NULL),
             pairs AS (SELECT l.*, p.platform FROM lv l CROSS JOIN (VALUES {plat_vals}) p(platform)
                       WHERE l.last_day BETWEEN DATE '{p0}' AND DATE '{p1}' AND (EXISTS (
                         SELECT 1 FROM grants_r g WHERE g.email = l.email AND g.platform = p.platform
AND g.ts::DATE <= l.last_day
                           AND (l.prev_last_day IS NULL OR g.ts::DATE > l.prev_last_day)) OR
(l.next_last_day IS NULL AND (EXISTS (
                         SELECT 1 FROM usage_r u WHERE u.email = l.email AND u.platform = p.platform
                           AND u.ts::DATE > l.last_day + INTERVAL {GRACE_DAYS} DAY) OR EXISTS (
                         SELECT 1 FROM held h WHERE h.email = l.email AND h.platform = p.platform))))),
             credited AS (SELECT e.email, e.platform, e.ts, coalesce(
                           (SELECT max(l.last_day) FROM lv l WHERE l.email = e.email AND l.last_day <=
e.ts::DATE),
                           (SELECT min(l.last_day) FROM lv l WHERE l.email = e.email
                             AND l.last_day - INTERVAL {REMOVAL_WINDOW_DAYS} DAY <= e.ts::DATE)) AS
last_day
                          FROM ends_r e WHERE e.email IS NOT NULL),
             rem AS (SELECT email, platform, last_day, min(ts) AS removed_at FROM credited WHERE
last_day IS NOT NULL GROUP BY ALL),
             act AS (SELECT p.email, p.last_day, p.platform,
                       count(DISTINCT u.rid) FILTER (WHERE u.ts::DATE > p.last_day + INTERVAL
{GRACE_DAYS} DAY
                                                    AND (r.removed_at IS NULL OR u.ts < r.removed_at)) AS
n_grace,
                       count(DISTINCT u.rid) FILTER (WHERE r.removed_at IS NOT NULL AND u.ts >
r.removed_at AND p.next_last_day IS NULL) AS n_removed
                     FROM pairs p LEFT JOIN rem r ON r.email = p.email AND r.platform = p.platform AND
r.last_day = p.last_day
                          LEFT JOIN usage_r u ON u.email = p.email AND u.platform = p.platform GROUP BY
ALL)
        SELECT p.email, p.last_day, p.platform, r.removed_at::DATE AS first_removal_record,
               date_diff('day', p.last_day, r.removed_at::DATE) AS days_to_removal,
               a.n_grace AS reads_after_grace, a.n_removed AS reads_after_removal,
               CASE WHEN r.removed_at IS NULL THEN 'NO REMOVAL RECORD'
                    WHEN a.n_removed > 0 THEN 'ACTIVE AFTER REMOVAL'
                    WHEN date_diff('day', p.last_day, r.removed_at::DATE) > {GRACE_DAYS} THEN 'REMOVED
LATE'
                    ELSE 'removed in time' END AS status
        FROM pairs p LEFT JOIN rem r ON r.email = p.email AND r.platform = p.platform AND r.last_day =
p.last_day
             LEFT JOIN act a ON a.email = p.email AND a.platform = p.platform AND a.last_day =
p.last_day
        ORDER BY p.last_day, p.email, p.platform"""
    l4 = population(con, "hipaa_4_leavers", "Leavers who held PHI access",
```

```
                    f"HR leavers with last day in the period who held read access to a PHI table on the
platform, hold it now, or read one after the grace period; removal records credited to one departure;
grace {GRACE_DAYS} day(s)",
                    sql, ["email", "last_day", "platform", "first_removal_record", "days_to_removal",
"reads_after_grace",
                          "reads_after_removal", "status"], ok, OUT_DIR)
    show(l4, lambda s: s != "removed in time", 7)

    # ---- result
    f1 = sum(1 for r in l1 if r[6] != "in register") + len(unmatched)
    f2 = sum(1 for r in l2 if r[7] != "in register")
    f3 = sum(1 for r in l3 if r[6].startswith("HELD"))
    f4 = sum(1 for r in l4 if r[7] != "removed in time" or r[5] or r[6])
    print("\n== Result")
    print(f"input checks : {'all passed' if ok else 'FAILED, the lists above are not complete'}")
    print(f"not checked  : {', '.join(not_checked) if not_checked else 'none'}")
    print(f"follow-up    : {f1} access row(s) beyond the register or unmatched, {f2} role row(s) with no
register line, "
          f"{f3} held-not-used row(s), {f4} leaver row(s) need an explanation")
    if not ok or f1 + f2 + f3 + f4:
        verdict = "EXCEPTIONS FOUND"
    elif not_checked:
        verdict = "NOTHING TO FOLLOW UP ON THE PLATFORMS CHECKED; NOT CHECKED: " + ",
".join(not_checked)
    else:
        verdict = "NOTHING TO FOLLOW UP"
    print(f"RESULT: {verdict}")
    return 1 if strict and verdict != "NOTHING TO FOLLOW UP" else 0


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

## The runs

### Run 1: faults planted (`--demo --plant`)

```
input     : GENERATED records (not a company's data), seed 229, plant=True
platforms : checked snowflake, databricks, redshift

== Input checks   (span checked for records: 2026-07-01 to 2026-10-15)
[FAIL] duplicate Snowflake query ids: 1   (trips on: a repeated query id)
[FAIL] timestamps not written YYYY-MM-DD HH:MM:SS (UTC): 1   (trips on: a timestamp written 07/09/2026
10:00)
[FAIL] days with no records, per platform, from the start of the span to AS_OF: 1   (trips on: a day
with no Redshift records)
[FAIL] PHI tables that match no grant, read, removal or mask in the exports: 1   (trips on: a typo in
the PHI column list)
[FAIL] Databricks audit events, Snowflake access lists or Redshift mask columns the script could not
read: 1   (trips on: malformed JSON)
[FAIL] Redshift GRANT or REVOKE on a PHI table the script could not parse: 1   (trips on: a grant to two
users in one statement)
[FAIL] Redshift grants, reads or writes that name a PHI table without its schema (not matched here): 1
(trips on: a read FROM claims)
[FAIL] Databricks grants on a catalog or schema that holds a PHI table (removal not followed in the
leaver list): 1   (trips on: a schema-level grant)
[FAIL] Snowflake roles granted to other roles that hold PHI access (members not followed here): 1
(trips on: a role granted to a role)
```

```
[FAIL] Snowflake PHI tables granted to database roles (holders not followed here): 1   (trips on: a
grant to a database role)
[FAIL] Redshift masking policy view returned 0 rows (it needs a superuser or sys:secadmin; set
REDSHIFT_HAS_MASKING_POLICIES = False if there are none): 1   (trips on: an empty
SVV_ATTACHED_MASKING_POLICY)
[FAIL] PHI column list rows with a blank platform, table, column or category: 1   (trips on: a blank in
the PHI column list)
[FAIL] register rows with a blank class or category: 1   (trips on: a blank category in the register)
[FAIL] HR class rows with a blank email or class: 1   (trips on: a blank in the HR class file)
[FAIL] HR leaver rows with no email or no last day: 1   (trips on: a NULL in the HR leaver file)
[FAIL] HR leaver rows whose last day is not written as YYYY-MM-DD: 1   (trips on: a last day written
8/5/26)
[FAIL] login map rows with a blank login or email: 1   (trips on: a blank in the login map)
        unreadable: GRANTS_TO_USERS.CREATED_ON: 1
        redshift days with no records: 2026-08-19
        the STL views keep seven days (AWS): export daily, or unload the logs to Amazon S3
        PHI tables: redshift:records.patient_vists
[info] permission changes that failed (status not 200), left out: 0
[info] Snowflake masking policies not ACTIVE, not counted as a mask: 1
[info] conditions in the register are printed with the register, not checked against any record

== Who can read which PHI category, against the register: hipaa_1_access_beyond_register   **
INCOMPLETE: see input checks **
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); access as of 2026-10-15
filters   : people holding read access (all, all privileges, ownership, select) to a column in the PHI
column list, by grant or by role; one row per person, platform and category
run at    : 2026-11-01 09:59:41 UTC
rows      : 30
sha256    : fa9aabe8d93bcd2c6ba07d7624b9d5b99eed1eb531c903131a5495ea08cf82b3
  databricks | b01@example.test | billing staff | billing | 1 | billing-grp | in register
  databricks | b01@example.test | billing staff | demographics | 1 | billing-grp | in register
  databricks | b02@example.test | billing staff | billing | 1 | billing-grp | in register
  databricks | b02@example.test | billing staff | demographics | 1 | billing-grp | in register
  databricks | c01@example.test | clinician | clinical | 1 | c01@example.test | in register
  databricks | c01@example.test | clinician | demographics | 2 | c01@example.test | in register
  databricks | c02@example.test | clinician | billing | 1 | c02@example.test | ACCESS BEYOND REGISTER
  databricks | c02@example.test | clinician | clinical | 1 | c02@example.test | in register
  databricks | c02@example.test | clinician | demographics | 3 | c02@example.test | in register
  databricks | x01@example.test | - | billing | 1 | x01@example.test | NO CLASS
  databricks | x01@example.test | - | demographics | 1 | x01@example.test | NO CLASS
  redshift | b01@example.test | billing staff | billing | 1 | b01+billing_grp | in register
  redshift | b01@example.test | billing staff | demographics | 1 | b01+billing_grp | in register
  redshift | b02@example.test | billing staff | billing | 1 | b02+billing_grp | in register
  redshift | b02@example.test | billing staff | demographics | 1 | b02+billing_grp | in register
  redshift | c02@example.test | clinician | clinical | 1 | c02 | in register
  redshift | c02@example.test | clinician | demographics | 2 | c02 | in register
  redshift | l01@example.test | billing staff | billing | 1 | l01 | in register
  redshift | l01@example.test | billing staff | demographics | 1 | l01 | in register
  redshift | q01@example.test | quality analyst | clinical | 1 | q01 | in register
  snowflake | b01@example.test | billing staff | billing | 1 | billing_role | in register
  snowflake | b01@example.test | billing staff | demographics | 1 | billing_role | in register
  snowflake | b02@example.test | billing staff | clinical | 1 | clinician_role | ACCESS BEYOND REGISTER
  snowflake | b02@example.test | billing staff | demographics | 2 | clinician_role | in register
  snowflake | c01@example.test | clinician | clinical | 1 | clinician_role | in register
  snowflake | c01@example.test | clinician | demographics | 2 | clinician_role | in register
  snowflake | c02@example.test | clinician | clinical | 1 | clinician_role | in register
  snowflake | c02@example.test | clinician | demographics | 2 | clinician_role | in register
  snowflake | l01@example.test | billing staff | billing | 1 | billing_role | in register
  snowflake | l01@example.test | billing staff | demographics | 1 | billing_role | in register
[info] logins or groups with read access that match no person (add them to login_map): 0
[info] mask coverage, snowflake: 5 PHI columns listed, 2 with a mask attached, 3 without
```

```
[info] mask coverage, databricks: 5 PHI columns listed, 1 with a mask attached, 4 without
[info] mask coverage, redshift: 6 PHI columns listed, 0 with a mask attached, 6 without

== PHI columns, who can read them and whether a mask is attached: hipaa_2_unmasked_by_role   **
INCOMPLETE: see input checks **
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); access as of 2026-10-15
filters   : one row per PHI column and role or principal that can read it; a Snowflake or Databricks
mask names the column, and who sees it in the clear depends on the policy's own body, which is not read
here; Redshift masks match only the identity they are attached to, or public
run at    : 2026-11-01 09:59:41 UTC
rows      : 31
sha256    : 192826c9c4ac1a23a199342896a444d96222c9fe30d204be83c01081f5ecae7d
  databricks | clinic.records.claims | claim_amount | billing | c02@example.test | IN CLEAR | 1 | NO
REGISTER LINE
  databricks | clinic.records.claims | claim_amount | billing | x01@example.test | IN CLEAR | 1 | NO
REGISTER LINE
  databricks | clinic.records.claims | patient_name | demographics | x01@example.test | IN CLEAR | 1 |
NO REGISTER LINE
  snowflake | clinic_db.records.claims | claim_amount | billing | legacy_role | IN CLEAR | 0 | NO
REGISTER LINE
  snowflake | clinic_db.records.claims | patient_name | demographics | legacy_role | mask attached | 0 |
NO REGISTER LINE

== PHI access held with no read seen: hipaa_3_held_not_used   ** INCOMPLETE: see input checks **
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); access as of 2026-10-15
filters   : read access held at the end of 2026-10-15 and held for at least 60 days (a grant with no
date counts), with no read in the 60 days to AS_OF; Snowflake per column, Databricks and Redshift per
table; a prompt for the access review, not proof the access is unneeded; Databricks lineage is a subset
of read events
run at    : 2026-11-01 09:59:41 UTC
rows      : 24
sha256    : 250ca3169f3ed7ca9ebdc3edc11413c26e4837297abb0dd7bb776eab33907c9d
  databricks | b01@example.test | clinic.records.claims | (table level) | 2026-03-11 | 2026-10-11 | read
within 60 days
  databricks | b02@example.test | clinic.records.claims | (table level) | 2026-03-11 | 2026-10-14 | read
within 60 days
  databricks | c01@example.test | clinic.records.patient_visits | (table level) | 2026-03-10 | 2026-10-
14 | read within 60 days
  databricks | c02@example.test | clinic.records.claims | (table level) | - | - | HELD, NO READ SEEN IN
60 DAYS
  databricks | c02@example.test | clinic.records.patient_visits | (table level) | - | - | HELD, NO READ
SEEN IN 60 DAYS
  databricks | x01@example.test | clinic.records.claims | (table level) | 2026-08-25 | - | read within
60 days
  redshift | b01@example.test | records.claims | (table level) | 2026-03-06 | 2026-10-14 | read within
60 days
  redshift | b02@example.test | records.claims | (table level) | 2026-03-06 | 2026-10-09 | read within
60 days
  redshift | c02@example.test | records.patient_visits | (table level) | 2026-03-05 | 2026-07-06 | HELD,
NO READ SEEN IN 60 DAYS
  redshift | l01@example.test | records.claims | (table level) | 2026-05-13 | 2026-08-02 | HELD, NO READ
SEEN IN 60 DAYS
  redshift | q01@example.test | records.patient_visits | (table level) | - | 2026-10-10 | read within 60
days
  snowflake | b01@example.test | clinic_db.records.claims | claim_amount | 2026-03-04 | 2026-10-14 |
read within 60 days
  snowflake | b01@example.test | clinic_db.records.claims | patient_name | 2026-03-04 | 2026-10-14 |
read within 60 days
  snowflake | b02@example.test | clinic_db.records.patient_visits | diagnosis_code | 2026-08-20 | 2026-
10-08 | read within 60 days
  snowflake | b02@example.test | clinic_db.records.patient_visits | dob | 2026-08-20 | 2026-10-08 | read
```

```
within 60 days
  snowflake | b02@example.test | clinic_db.records.patient_visits | patient_name | 2026-08-20 | 2026-10-
08 | read within 60 days
  snowflake | c01@example.test | clinic_db.records.patient_visits | diagnosis_code | 2026-03-02 | 2026-
10-11 | read within 60 days
  snowflake | c01@example.test | clinic_db.records.patient_visits | dob | 2026-03-02 | 2026-10-11 | read
within 60 days
  snowflake | c01@example.test | clinic_db.records.patient_visits | patient_name | 2026-03-02 | 2026-10-
11 | read within 60 days
  snowflake | c02@example.test | clinic_db.records.patient_visits | diagnosis_code | 2026-03-03 | 2026-
10-12 | read within 60 days
  snowflake | c02@example.test | clinic_db.records.patient_visits | dob | 2026-03-03 | 2026-10-12 | read
within 60 days
  snowflake | c02@example.test | clinic_db.records.patient_visits | patient_name | 2026-03-03 | 2026-10-
12 | read within 60 days
  snowflake | l01@example.test | clinic_db.records.claims | claim_amount | 2026-05-11 | 2026-08-25 |
read within 60 days
  snowflake | l01@example.test | clinic_db.records.claims | patient_name | 2026-05-11 | 2026-08-25 |
read within 60 days

== Leavers who held PHI access: hipaa_4_leavers   ** INCOMPLETE: see input checks **
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); access as of 2026-10-15
filters   : HR leavers with last day in the period who held read access to a PHI table on the platform,
hold it now, or read one after the grace period; removal records credited to one departure; grace 1
day(s)
run at    : 2026-11-01 09:59:41 UTC
rows      : 4
sha256    : 04d79537a22c02808fb8e9a4d030801399572749a9c64cd6929463f7d79027d6
  l01@example.test | 2026-08-07 | redshift | - | - | 0 | 0 | NO REMOVAL RECORD
  l01@example.test | 2026-08-07 | snowflake | - | - | 2 | 0 | NO REMOVAL RECORD
  l02@example.test | 2026-09-04 | databricks | 2026-09-20 | 16 | 1 | 0 | REMOVED LATE
  l02@example.test | 2026-09-04 | snowflake | 2026-09-05 | 1 | 0 | 0 | removed in time

== Result
input checks : FAILED, the lists above are not complete
not checked  : none
follow-up    : 4 access row(s) beyond the register or unmatched, 5 role row(s) with no register line, 4
held-not-used row(s), 3 leaver row(s) need an explanation
RESULT: EXCEPTIONS FOUND
```

## Run 2: clean records (`--demo`)

```
input     : GENERATED records (not a company's data), seed 229, plant=False
platforms : checked snowflake, databricks, redshift

== Input checks   (span checked for records: 2026-07-01 to 2026-10-15)
[ok  ] duplicate Snowflake query ids: 0
[ok  ] timestamps not written YYYY-MM-DD HH:MM:SS (UTC): 0
[ok  ] days with no records, per platform, from the start of the span to AS_OF: 0
[ok  ] PHI tables that match no grant, read, removal or mask in the exports: 0
[ok  ] Databricks audit events, Snowflake access lists or Redshift mask columns the script could not
read: 0
[ok  ] Redshift GRANT or REVOKE on a PHI table the script could not parse: 0
[ok  ] Redshift grants, reads or writes that name a PHI table without its schema (not matched here): 0
[ok  ] Databricks grants on a catalog or schema that holds a PHI table (removal not followed in the
leaver list): 0
[ok  ] Snowflake roles granted to other roles that hold PHI access (members not followed here): 0
[ok  ] Snowflake PHI tables granted to database roles (holders not followed here): 0
[ok  ] Redshift masking policy view returned 0 rows (it needs a superuser or sys:secadmin; set
REDSHIFT_HAS_MASKING_POLICIES = False if there are none): 0
```

```
[ok  ] PHI column list rows with a blank platform, table, column or category: 0
[ok  ] register rows with a blank class or category: 0
[ok  ] HR class rows with a blank email or class: 0
[ok  ] HR leaver rows with no email or no last day: 0
[ok  ] HR leaver rows whose last day is not written as YYYY-MM-DD: 0
[ok  ] login map rows with a blank login or email: 0
[info] permission changes that failed (status not 200), left out: 0
[info] Snowflake masking policies not ACTIVE, not counted as a mask: 0
[info] conditions in the register are printed with the register, not checked against any record

== Who can read which PHI category, against the register: hipaa_1_access_beyond_register
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); access as of 2026-10-15
filters   : people holding read access (all, all privileges, ownership, select) to a column in the PHI
column list, by grant or by role; one row per person, platform and category
run at    : 2026-11-01 09:59:40 UTC
rows      : 19
sha256    : d623296791976fac7a54e0d77d1f29cdfd739b52ff68fe7d8b09b064420010e5
  databricks | b01@example.test | billing staff | billing | 1 | billing-grp | in register
  databricks | b01@example.test | billing staff | demographics | 1 | billing-grp | in register
  databricks | b02@example.test | billing staff | billing | 1 | billing-grp | in register
  databricks | b02@example.test | billing staff | demographics | 1 | billing-grp | in register
  databricks | c01@example.test | clinician | clinical | 1 | c01@example.test | in register
  databricks | c01@example.test | clinician | demographics | 2 | c01@example.test | in register
  redshift | b01@example.test | billing staff | billing | 1 | billing_grp | in register
  redshift | b01@example.test | billing staff | demographics | 1 | billing_grp | in register
  redshift | b02@example.test | billing staff | billing | 1 | billing_grp | in register
  redshift | b02@example.test | billing staff | demographics | 1 | billing_grp | in register
  redshift | c02@example.test | clinician | clinical | 1 | c02 | in register
  redshift | c02@example.test | clinician | demographics | 2 | c02 | in register
  redshift | q01@example.test | quality analyst | clinical | 1 | q01 | in register
  snowflake | b01@example.test | billing staff | billing | 1 | billing_role | in register
  snowflake | b01@example.test | billing staff | demographics | 1 | billing_role | in register
  snowflake | c01@example.test | clinician | clinical | 1 | clinician_role | in register
  snowflake | c01@example.test | clinician | demographics | 2 | clinician_role | in register
  snowflake | c02@example.test | clinician | clinical | 1 | clinician_role | in register
  snowflake | c02@example.test | clinician | demographics | 2 | clinician_role | in register
[info] logins or groups with read access that match no person (add them to login_map): 0
[info] mask coverage, snowflake: 5 PHI columns listed, 2 with a mask attached, 3 without
[info] mask coverage, databricks: 5 PHI columns listed, 1 with a mask attached, 4 without
[info] mask coverage, redshift: 5 PHI columns listed, 1 with a mask attached, 4 without

== PHI columns, who can read them and whether a mask is attached: hipaa_2_unmasked_by_role
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); access as of 2026-10-15
filters   : one row per PHI column and role or principal that can read it; a Snowflake or Databricks
mask names the column, and who sees it in the clear depends on the policy's own body, which is not read
here; Redshift masks match only the identity they are attached to, or public
run at    : 2026-11-01 09:59:40 UTC
rows      : 16
sha256    : d05002f02ae6709f2427556e8de5d49cce33e633a27d6f021f059d6bc3edfbb3
  databricks | clinic.records.claims | claim_amount | billing | billing-grp | IN CLEAR | 2 | in register
  databricks | clinic.records.claims | patient_name | demographics | billing-grp | IN CLEAR | 2 | in
register
  databricks | clinic.records.patient_visits | diagnosis_code | clinical | c01@example.test | IN CLEAR |
1 | in register
  databricks | clinic.records.patient_visits | dob | demographics | c01@example.test | mask attached | 1
| in register
  databricks | clinic.records.patient_visits | patient_name | demographics | c01@example.test | IN CLEAR
| 1 | in register
  redshift | records.claims | claim_amount | billing | billing_grp | IN CLEAR | 2 | in register
  redshift | records.claims | patient_name | demographics | billing_grp | IN CLEAR | 2 | in register
  redshift | records.patient_visits | diagnosis_code | clinical | c02 | IN CLEAR | 1 | in register
```

```
  redshift | records.patient_visits | diagnosis_code | clinical | q01 | IN CLEAR | 1 | in register
  redshift | records.patient_visits | dob | demographics | c02 | mask attached | 1 | in register
  redshift | records.patient_visits | patient_name | demographics | c02 | IN CLEAR | 1 | in register
  snowflake | clinic_db.records.claims | claim_amount | billing | billing_role | IN CLEAR | 1 | in
register
  snowflake | clinic_db.records.claims | patient_name | demographics | billing_role | mask attached | 1
| in register
  snowflake | clinic_db.records.patient_visits | diagnosis_code | clinical | clinician_role | IN CLEAR |
2 | in register
  snowflake | clinic_db.records.patient_visits | dob | demographics | clinician_role | mask attached | 2
| in register
  snowflake | clinic_db.records.patient_visits | patient_name | demographics | clinician_role | IN CLEAR
| 2 | in register

== PHI access held with no read seen: hipaa_3_held_not_used
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); access as of 2026-10-15
filters   : read access held at the end of 2026-10-15 and held for at least 60 days (a grant with no
date counts), with no read in the 60 days to AS_OF; Snowflake per column, Databricks and Redshift per
table; a prompt for the access review, not proof the access is unneeded; Databricks lineage is a subset
of read events
run at    : 2026-11-01 09:59:40 UTC
rows      : 15
sha256    : e817fb3f2a9066988a44660f96106d218ff2cbee5a9fb69ecf8749628e0e5d2c
  databricks | b01@example.test | clinic.records.claims | (table level) | 2026-03-11 | 2026-10-11 | read
within 60 days
  databricks | b02@example.test | clinic.records.claims | (table level) | 2026-03-11 | 2026-10-14 | read
within 60 days
  databricks | c01@example.test | clinic.records.patient_visits | (table level) | 2026-03-10 | 2026-10-
14 | read within 60 days
  redshift | b01@example.test | records.claims | (table level) | 2026-03-06 | 2026-10-14 | read within
60 days
  redshift | b02@example.test | records.claims | (table level) | 2026-03-06 | 2026-10-09 | read within
60 days
  redshift | c02@example.test | records.patient_visits | (table level) | 2026-03-05 | 2026-10-14 | read
within 60 days
  redshift | q01@example.test | records.patient_visits | (table level) | - | 2026-10-10 | read within 60
days
  snowflake | b01@example.test | clinic_db.records.claims | claim_amount | 2026-03-04 | 2026-10-14 |
read within 60 days
  snowflake | b01@example.test | clinic_db.records.claims | patient_name | 2026-03-04 | 2026-10-14 |
read within 60 days
  snowflake | c01@example.test | clinic_db.records.patient_visits | diagnosis_code | 2026-03-02 | 2026-
10-11 | read within 60 days
  snowflake | c01@example.test | clinic_db.records.patient_visits | dob | 2026-03-02 | 2026-10-11 | read
within 60 days
  snowflake | c01@example.test | clinic_db.records.patient_visits | patient_name | 2026-03-02 | 2026-10-
11 | read within 60 days
  snowflake | c02@example.test | clinic_db.records.patient_visits | diagnosis_code | 2026-03-03 | 2026-
10-12 | read within 60 days
  snowflake | c02@example.test | clinic_db.records.patient_visits | dob | 2026-03-03 | 2026-10-12 | read
within 60 days
  snowflake | c02@example.test | clinic_db.records.patient_visits | patient_name | 2026-03-03 | 2026-10-
12 | read within 60 days

== Leavers who held PHI access: hipaa_4_leavers
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive); access as of 2026-10-15
filters   : HR leavers with last day in the period who held read access to a PHI table on the platform,
hold it now, or read one after the grace period; removal records credited to one departure; grace 1
day(s)
run at    : 2026-11-01 09:59:40 UTC
rows      : 4
```

```
sha256    : 2c77edda0c374aa9cb478fbe97b91733920462d9391dfef9b5fc66fb6d8182d4
  l01@example.test | 2026-08-07 | redshift | 2026-08-08 | 1 | 0 | 0 | removed in time
  l01@example.test | 2026-08-07 | snowflake | 2026-08-08 | 1 | 0 | 0 | removed in time
  l02@example.test | 2026-09-04 | databricks | 2026-09-05 | 1 | 0 | 0 | removed in time
  l02@example.test | 2026-09-04 | snowflake | 2026-09-05 | 1 | 0 | 0 | removed in time

== Result
input checks : all passed
not checked  : none
follow-up    : 0 access row(s) beyond the register or unmatched, 0 role row(s) with no register line, 0
held-not-used row(s), 0 leaver row(s) need an explanation
RESULT: NOTHING TO FOLLOW UP
```

## How to read the results

Read the input checks first. A line that reads `[FAIL]` says which kind of fault tripped it, and every list printed after it is marked `INCOMPLETE`: fix the export, then run again.

List 1: `in register` means the person's class lists that category. `ACCESS BEYOND REGISTER` and `NO CLASS` are rows to explain or fix: change the access, or change the register and say why. List 2: `NO REGISTER LINE` means no person who holds that role has a class that lists the column's category. `mask attached` says a mask exists on the column, not that a given role sees masked data. List 3: a held-and-unused row is a prompt for your access review, not proof the access is unneeded. List 4: `NO REMOVAL RECORD`, `REMOVED LATE` and `ACTIVE AFTER REMOVAL` are rows to explain. The result line counts the rows that need an explanation; it is not a finding that anything is or is not in line with the standard.

## Checklist

| | Step | Where |
|---|---|---|
| ☐ | Write the register: each class of workforce, each PHI category it needs, the conditions | Your own list under 164.514(d)(2)(i) |
| ☐ | List the PHI columns, from tags where you keep them | phi_columns.csv |
| ☐ | Export each platform's records, in UTC, once, into one folder | Step 1; Redshift STL views keep seven days |
| ☐ | Map HR classes, leavers and logins; add group members to the login map | hr_classes.csv, hr_leavers.csv, login_map.csv |
| ☐ | Set NOT_USED_DAYS, the period and AS_OF | Settings at the top of the script |
| ☐ | Run the script; every input check reads [ok] | A failed check marks every list INCOMPLETE |
| ☐ | List 1: explain each ACCESS BEYOND REGISTER and NO CLASS row, or change the register or the access | hipaa_1_access_beyond_register.csv |
| ☐ | List 2: give each NO REGISTER LINE role a register line or review it; decide which IN CLEAR columns, if any, to mask | hipaa_2_unmasked_by_role.csv; HIPAA does not require masking |

| | Step | Where |
|---|---|---|
| ☐ | List 3: take each held-and-unused row to the access review | hipaa_3_held_not_used.csv; a prompt, not proof |
| ☐ | List 4: explain each leaver row | hipaa_4_leavers.csv |
| ☐ | Keep the settings, the four lists and their SHA-256 values for each pull | Your counsel decides the retention period |

## Limits

- **This is not a client result.** The records are generated. The runs show that the script finds planted faults. They say nothing about any real company or how often such faults occur.
- **We have not run it on a real account.** Real exports can differ from the documentation. The layout of the `changes` field in a Databricks permission change and the text of Redshift's `output_columns` are our assumptions: open one of your own rows and check.
- **A register line is not a legal conclusion.** The script compares records with your own list. Whether the list is the right one, and whether your organization is a covered entity or a business associate, is for you and your counsel.
- **Conditions are not checked.** A register line can say "patients assigned to them"; no platform record shows whether a read was for an assigned patient.
- **Masks are only partly read.** The script does not read a Snowflake or Databricks masking function's body, so it cannot say which role sees clear data. Redshift masks match only the identity they are attached to, or public, and a mask attached to a role matches only when the access is held through that role.
- **Reads are partial.** Databricks lineage is a subset of read events, and Redshift and Databricks show reads at table level, so a held-and-unused row is a prompt for review, not proof. Snowflake ACCESS_HISTORY needs Enterprise Edition or higher.
- **Access that reaches a PHI table by another route is not followed:** on Snowflake a role granted to another role, and a PHI table granted to a database role, fail an input check instead, and future grants are not read. A Databricks grant on a schema or catalog shows in today's access through TABLE_PRIVILEGES, but its removal is not followed in the leaver list, so it fails an input check too. On Redshift a grant, read or write that names a PHI table without its schema fails an input check, and `GRANT ... ON ALL TABLES IN SCHEMA` and role membership are not read for the leaver list. Group and role membership on Databricks and Redshift comes from the login map you export, as of that day, so a membership that changed earlier is not seen.
- **The privilege views are only as complete as the export.** An export of SVV_RELATION_PRIVILEGES, SVV_COLUMN_PRIVILEGES or TABLE_PRIVILEGES made by a user who cannot see every grant leaves rows out, and the script cannot tell. These views list grants, so a table owner's own access is not listed; check owners separately.

- **On Snowflake, access through a role** runs from the later of the two grants (the role to the user, the table to the role) to the earlier of the two removals. In the leaver list, a removal on Snowflake is the moment the person's last PHI access there ended, so a table leaving one role is not a removal while other access stays open. The current-access lists use the end of `AS_OF`.
- **Timestamps with no zone suffix are taken as UTC.** Snowflake exports follow the session time zone, so set it to UTC before you export.
- **The empty-day check can fire on a quiet day,** and cannot see a day that has some records but not all.
- **Treatment and other uses:** the script does not decide which uses fall under the standard. It lists access against your register.

## Sources


- https://docs.aws.amazon.com/redshift/latest/dg/SYS_QUERY_HISTORY.html (read 2026-11-01)
- https://docs.aws.amazon.com/redshift/latest/dg/SYS_USERLOG.html (read 2026-11-01)
- https://docs.aws.amazon.com/redshift/latest/dg/cm_chap_system-tables.html (read 2026-11-01)
- https://docs.aws.amazon.com/redshift/latest/dg/r_STL_UTILITYTEXT.html (read 2026-11-01)
- https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_ATTACHED_MASKING_POLICY.html (read 2026-11-01)
- https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_COLUMN_PRIVILEGES.html (read 2026-11-01)
- https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_RELATION_PRIVILEGES.html (read 2026-11-01)
- https://docs.databricks.com/aws/en/admin/system-tables/audit-logs (read 2026-11-01)
- https://docs.databricks.com/aws/en/admin/system-tables/lineage (read 2026-11-01)
- https://docs.databricks.com/aws/en/sql/language-manual/information-schema/column_masks (read 2026-11-01)
- https://docs.databricks.com/aws/en/sql/language-manual/information-schema/column_tags (read 2026-11-01)
- https://docs.databricks.com/aws/en/sql/language-manual/information-schema/table_privileges (read 2026-11-01)
- https://docs.snowflake.com/en/sql-reference/account-usage/access_history (read 2026-11-01)
- https://docs.snowflake.com/en/sql-reference/account-usage/grants_to_roles (read 2026-11-01)
- https://docs.snowflake.com/en/sql-reference/account-usage/grants_to_users (read 2026-11-01)
- https://docs.snowflake.com/en/sql-reference/account-usage/policy_references (read 2026-11-01)
- https://docs.snowflake.com/en/sql-reference/account-usage/tag_references (read 2026-11-01)
- https://docs.snowflake.com/en/user-guide/security-column-intro (read 2026-11-01)
- https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-C/part-164/subpart-C/section-164.306 (read 2026-11-01)
- https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-C/part-164/subpart-C/section-164.308 (read 2026-11-01)
- https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-C/part-164/subpart-E/section-164.502 (read 2026-11-01)
- https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-C/part-164/subpart-E/section-164.514 (read 2026-11-01)
- https://www.ecfr.gov/current/title-45/subtitle-A/subchapter-C/part-164/subpart-E/section-164.530 (read 2026-11-01)

- https://www.hhs.gov/hipaa/for-professionals/privacy/guidance/minimum-necessary-requirement/index.html (read 2026-11-01)

**Want your platforms' access records read every day?**

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.

The Governance Evidence Pack, in early access, checks every night that sensitive data has an owner, is deleted when your policy says, and is reached only through your approval process, and turns those checks into evidence for SOX or HIPAA. It also covers how many sensitive columns are masked. We produce the evidence. Your auditor decides.

See Data Control Tower