

SOX access evidence in Snowflake, Databricks and Redshift: approvals, leavers and segregation of duties

Approvals, leavers and segregation of duties, checked from each platform's own records with a DuckDB script. Evidence for your auditor to weigh; the auditor decides.

Published 2026-10-31 · Online version and downloads: www.datameaning.com/guides/sox-access-evidence-for-snowflake-databricks-redshift/

A SOX auditor decides what to test; we read the SEC's "access to programs and data" as three checks on financial tables: access approved before it was given, leavers removed on time, and no one both changing the data and giving or approving access to it. Our DuckDB script runs them from platform records plus your approval and HR files.

What the SEC and the statute say about access

Section 404(a) of the Sarbanes-Oxley Act directs the SEC to require each annual report to contain an internal control report that, under (a)(2), must "contain an assessment, as of the end of the most recent fiscal year" of the issuer "of the effectiveness of the internal control structure and procedures of the issuer for financial reporting", and 404(b) says the outside auditor "shall attest to, and report on, the assessment". Section 404(b) does not apply to an emerging growth company, and Section 404(c) removes the attestation for issuers that are neither large accelerated filers nor accelerated filers, so check which you are ([15 U.S.C. 7262 \(SOX Section 404\)](#) (read 2026-10-31), subsections (a)(2), (b) and (c)).

The SEC's interpretive guidance for management names access in its own words: "For purposes of the evaluation of ICFR, management only needs to evaluate those IT general controls that are necessary for the proper and consistent operation of other controls designed to adequately address financial reporting risks. For example, management might consider whether certain aspects of IT general control areas, such as program development, program changes, computer operations, and access to programs and data, apply to its facts and circumstances." ([SEC Release 33-8810](#) (read

2026-10-31), section II.A.1.d, page 20). Footnote 37 adds: “However, the reference to these specific IT general control areas as examples within this guidance does not imply that these areas, either partially or in their entirety, are applicable to all facts and circumstances.”

We looked for PCAOB AS 2201 and could not read it (the PCAOB site answered us with a bot check), so this page does not say what it requires. What follows is our reading of what “access to programs and data” means for financial tables in a data platform: three questions an auditor can put to the records. Which of them your auditor asks, and what they accept as an answer, is the auditor’s call.

1. **Approvals.** For each access granted to a financial table, is there an approval, and is it dated on or before the grant, by someone other than the person who got the access?
2. **Leavers.** For each person HR says left, was access to the financial table removed, how many days after the last day, and did the account use the table after that?
3. **Segregation of duties.** Did anyone both change a financial table and run or approve access to it?

What you will need

- Access to the platforms’ own records: on Snowflake the ACCOUNT_USAGE views, on Databricks the system tables, on Redshift the system views.
- An approval log, an HR leaver file and a login map, as CSV files.
- Python 3 and the duckdb module.

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 twelve files

File	Columns the script reads	Defined in
<code>snowflake_grants_to_users.csv</code>	CREATED_ON, DELETED_ON, ROLE, GRANTEE_NAME, GRANTED_BY	Snowflake: GRANTS_TO_USERS (read 2026-10-31)
<code>snowflake_grants_to_roles.csv</code>	CREATED_ON, PRIVILEGE, GRANTED_ON, NAME, TABLE_CATALOG, TABLE_SCHEMA, GRANTED_TO, GRANTEE_NAME, GRANTED_BY, DELETED_ON	Snowflake: GRANTS_TO_ROLES (read 2026-10-31)

File	Columns the script reads	Defined in
snowflake_query_history.csv	QUERY_ID, QUERY_TEXT, QUERY_TYPE, USER_NAME, ROLE_NAME, START_TIME, EXECUTION_STATUS	Snowflake: QUERY_HISTORY (read 2026-10-31)
snowflake_access_history.csv	QUERY_ID, QUERY_START_TIME, USER_NAME, OBJECTS_MODIFIED	Snowflake: ACCESS_HISTORY (read 2026-10-31)
databricks_audit.csv	event_time, event_date, user_identity, service_name, action_name, request_params, response (user_identity, request_params and response as JSON text)	Databricks: audit log system table (read 2026-10-31)
databricks_table_lineage.csv	target_table_full_name, created_by, event_time, entity_type	Databricks: lineage system tables (read 2026-10-31)
redshift_stl_utilitytext.csv	userid, xid, pid, starttime, endtime, sequence, text	Amazon Redshift: STL_UTILITYTEXT (read 2026-10-31)
redshift_sys_userlog.csv	user_id, user_name, action, record_time	Amazon Redshift: SYS_USERLOG (read 2026-10-31)
redshift_sys_query_history.csv	user_id, username, query_id, query_type, status, start_time, query_text	Amazon Redshift: SYS_QUERY_HISTORY (read 2026-10-31)
approvals.csv	your approval log: one row per approved access, with who it was approved for (email), the table, the approver (email) and the day of approval	yours (the column names are settings)
hr_leavers.csv	your HR file of leavers: work email and last day of work, written YYYY-MM-DD	yours (the column names are settings)
login_map.csv	platform, login, email: one row for every login that is not itself an email address	yours

What each platform keeps, and for how long, decides what you can show. Snowflake's QUERY_HISTORY and ACCESS_HISTORY views each cover the last 365 days, with a latency of up to 45 minutes for QUERY_HISTORY and up to 180 minutes for ACCESS_HISTORY ([Snowflake: QUERY_HISTORY](#) (read 2026-10-31); [Snowflake: ACCESS_HISTORY](#) (read 2026-10-31)), and GRANTS_TO_USERS can lag by up to 120 minutes ([Snowflake: GRANTS_TO_USERS](#) (read 2026-10-31)), so take the export some hours after the last event you need. Databricks' lineage tables keep a rolling window of one year ([Databricks: lineage system tables](#) (read 2026-10-31)). 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 2026-10-31)). STL_UTILITYTEXT, where the script reads grants and revokes, is an STL view, so a quarter of Redshift grants needs a daily export or an unload to S3. We did not find a retention figure for the SYS views the script also reads, so check yours. The script checks every day from PERIOD_START to AS_OF for records in each platform's query or audit history and fails if a day has none. It does not check that the grants files cover the whole period, so confirm that STL_UTILITYTEXT was exported daily.

Step 2. Name the financial tables, and read every timestamp strictly

FINANCIAL_TABLES has no default. Unset, the script stops and says so, because a check with no scope would pass by looking at nothing. Each platform gets its own list of lower-case names (database.schema.table on Snowflake and Databricks, schema.table on Redshift). A name that matches no grant, use or removal in your exports is an input check failure, so a typo cannot send a table to nowhere.

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 from the period 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 eleven 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; seven of the eleven 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; scope names that match nothing; Databricks audit events (a permission change, or a response status) or write lists it could not read; Redshift GRANT or REVOKE statements on a financial table it could not parse; HR rows with no email or no

last day; an HR last day not written YYYY-MM-DD; approval rows with a blank grantee, table, approver or date; approval dates not written YYYY-MM-DD; login map rows with a blank login or email. If any fails, all three lists are printed with `INCOMPLETE` in their heading.

Step 4. Turn each platform's records into the same three events

The platforms record access differently, so the script first reduces each to grants, removals and use of the financial tables, at login level, then matches logins to people.

Snowflake. A person gets access by being granted a role. The script joins each `GRANTS_TO_USERS` row to the privileges that role holds on a financial table in `GRANTS_TO_ROLES` at that moment. `GRANTS_TO_USERS` carries `DELETED_ON`, which Snowflake documents as the time the role was revoked from the user, and which stays empty while the grant is active ([Snowflake: GRANTS_TO_USERS](#) (read 2026-10-31)). The person who ran the grant is not in that view: the script assumes `GRANTED_BY` is the role the grant ran under and looks in `QUERY_HISTORY` for the `GRANT` statement run under that role within 60 seconds that names the role and the grantee. If it finds none, the row shows a dash for the person and counts as follow-up. Writes come from `ACCESS_HISTORY`, whose `OBJECTS_MODIFIED` is a JSON array of the objects associated with a write ([Snowflake: ACCESS_HISTORY](#) (read 2026-10-31)).

```
con.execute(f"""CREATE TABLE sf_acc AS
    SELECT u.ts, u.end_ts, u.login, u.role, u.granted_by, r.obj, string_agg(DISTINCT
        lower(replace(r.privilege, '_', ' ')), ',' ORDER BY lower(replace(r.privilege, '_', ' '))) AS privs
    FROM sf_gu u JOIN sf_gr r ON r.role = u.role AND r.obj IN (SELECT obj FROM scope WHERE
        platform = 'snowflake')
    AND lower(replace(r.privilege, '_', ' ')) IN (SELECT p FROM privs) AND r.ts <=
    u.ts AND (r.end_ts IS NULL OR r.end_ts > u.ts)
    WHERE u.ts IS NOT NULL GROUP BY ALL""")
```

```
con.execute("""CREATE TABLE sf_g AS WITH m AS (
    SELECT a.*, q.login AS grantor, row_number() OVER (PARTITION BY a.ts, a.login,
        a.role, a.obj
            ORDER BY abs(epoch(q.ts) - epoch(a.ts))) AS rn
    FROM sf_acc a LEFT JOIN sf_gh q ON q.query_type = 'GRANT' AND q.role = a.granted_by
        AND abs(epoch(q.ts) - epoch(a.ts)) <= 60 AND contains(q.text, a.role) AND
        contains(q.text, a.login))
    SELECT * FROM m WHERE rn = 1""")
```

Databricks. The audit table records the user, service, action, request parameters and response of each event ([Databricks: audit log system table](#) (read 2026-10-31)). The script reads `updatePermissions` events under `unityCatalog` and takes the table from `securable_full_name` and the principals and privileges added or removed from `changes`. Databricks' audit log reference lists `securable_full_name` and `changes` as the request parameters of `updatePermissions` ([Databricks: audit log reference](#) (read 2026-10-31)), and the system-table page shows `full_name_arg` and a `statusCode` in its examples. It does not document

the layout inside changes, so how the script reads the principals and privileges added or removed is our assumption: open one of your own events and check it. It counts only events whose response status is 200: a permission change with another status is left out and counted on an `[info]` line, a read with another status is a refused attempt and not use, and an event on a financial table whose status cannot be read fails an input check (events on other tables are not looked at). If your export carries no status on some of those events, that check will fail on clean data: open one of your own events and check. Writes come from the lineage table's `target_table_full_name` and `created_by` ([Databricks: lineage system tables](#) (read 2026-10-31)).

Redshift. Grants and revokes are in `STL_UTILITYTEXT`, where a statement longer than 200 characters is stored in pieces ordered by sequence ([Amazon Redshift: STL_UTILITYTEXT](#) (read 2026-10-31)). The script puts them back in order, strips comments, and parses GRANT and REVOKE with one privilege list, one table and one grantee. A GRANT or REVOKE that names a financial table in full but does not fit the one-table, one-grantee pattern fails an input check. The user id becomes a name through `SYS_QUERY_HISTORY`'s `username` ([Amazon Redshift: SYS_QUERY_HISTORY](#) (read 2026-10-31)); a dropped user is a removal in `SYS_USERLOG` ([Amazon Redshift: SYS_USERLOG](#) (read 2026-10-31)).

```
# 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')""")
```

People. A login that is itself an email address is the person; any other login needs a row in `login_map.csv`. Emails are lower-cased and trimmed, so `Ana.Ruiz@Corp.Example` in the approval log and `ANA.RUIZ@CORP.EXAMPLE` in Snowflake are the same person. A login that wrote to a financial table or ran a grant and matches no person is listed, never dropped.

Step 5. List 1: was each access approved, and before it was given?

Every grant the script can see on the tables you name, in the period, is matched to an approval for the same person and table. If more than one approval exists, the closest one dated on or before the grant wins, and only if there is none is the closest later one used. The status is then `NO APPROVAL`, `SELF-APPROVED` (the approver is the person who got the access), `APPROVED AFTER GRANT`, or `approved before grant`. The script checks the approval's date, not who the approver should have been: whether that person was entitled to approve is for you and your auditor.

```

WITH g AS (SELECT row_number() OVER (ORDER BY ts, platform, login, obj, privs) AS gid, *
FROM grants_r
WHERE ts::DATE BETWEEN DATE '{p0}' AND DATE '{p1}'),
cand AS (SELECT g.gid, a.approver, a.approved_on,
row_number() OVER (PARTITION BY g.gid ORDER BY CASE WHEN a.approved_on <=
g.ts::DATE THEN 0 ELSE 1 END,
abs(date_diff('day', a.approved_on, g.ts::DATE))),
a.approver = coalesce(g.email, g.login), a.approver) AS rn
FROM g JOIN approvals a ON a.grantee = coalesce(g.email, g.login) AND a.obj =
g.obj AND a.approved_on IS NOT NULL
AND a.approver <> '')
SELECT g.platform, g.ts::DATE AS granted_on, coalesce(g.email, g.login) AS grantee, g.obj
AS object, g.privs,
coalesce(g.grantor_email, g.grantor_login, '-') AS granted_by_person, c.approver,
c.approved_on,
CASE WHEN c.gid IS NULL THEN 'NO APPROVAL'
WHEN c.approver = coalesce(g.email, g.login) THEN 'SELF-APPROVED'
WHEN c.approved_on > g.ts::DATE THEN 'APPROVED AFTER GRANT'
ELSE 'approved before grant' END AS status
FROM g LEFT JOIN cand c ON c.gid = g.gid AND c.rn = 1 ORDER BY g.ts, g.platform, grantee,
g.obj, g.privs""

```

Step 6. List 2: leavers, each departure on its own

The list starts from the HR file, not from the removal records, so a person who was never removed still appears. A leaver is listed on a platform if your exports show a grant to them there, or if their account used a financial table there after the grace period; anyone else appears only in the [info] count, so read that line. For each leaver and each platform where they held access to a financial table, the script finds the first removal record credited to that departure, counts the days since the last day of work, counts use of a financial table after a grace period of one day (GRACE_DAYS) and before the removal, and counts use after the removal. Use after the removal makes the row ACTIVE AFTER REMOVAL. Each removal is credited to one departure only: the latest HR last day on or before it, or, for a removal before every last day, the earliest last day it precedes by at most 30 days. So someone who left, was removed, came back and left again is two rows, and the first removal never counts for the second departure. For a person who later leaves again, use after the first removal is not counted against the first row, because a rehired person legitimately uses the table in the second stint. A leaver with no grant record on a platform is listed there only if their account used a financial table after the grace period, which is what a grant older than the exports looks like; otherwise the script counts them on an [info] line.

```

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 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
      AND l.next_last_day IS NULL))), -- a rehire's earlier departure needs
a grant record
      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),

```

Step 7. List 3: who both changed the ledger and gave access to it?

The script lists anyone who wrote to a financial table in the period and also ran a grant of access to one, or approved access to one, matching people by email across all platforms. Then it lists every login that matched no person.

```

      WITH w AS (SELECT email, count(*) AS writes, string_agg(DISTINCT platform, ',' ORDER BY
platform) AS wrote_on FROM usage_r
      WHERE is_write AND email IS NOT NULL AND ts::DATE BETWEEN DATE '{p0}' AND DATE
'{p1}' GROUP BY 1),
      g AS (SELECT grantor_email AS email, count(*) AS grants_run FROM grants_r WHERE
grantor_email IS NOT NULL
      AND ts::DATE BETWEEN DATE '{p0}' AND DATE '{p1}' GROUP BY 1),
      a AS (SELECT approver AS email, count(*) AS approvals_given FROM approvals WHERE
approver <> ''
      AND obj IN (SELECT obj FROM scope) AND approved_on BETWEEN DATE '{p0}' AND DATE
'{p1}' GROUP BY 1)
      SELECT w.email, w.writes, w.wrote_on, coalesce(g.grants_run, 0) AS grants_run,
coalesce(a.approvals_given, 0) AS approvals_given,
      'WROTE AND ' || CASE WHEN g.grants_run > 0 AND a.approvals_given > 0 THEN 'GRANTED
AND APPROVED'
      WHEN g.grants_run > 0 THEN 'GRANTED' ELSE 'APPROVED' END AS
status
      FROM w LEFT JOIN g ON g.email = w.email LEFT JOIN a ON a.email = w.email
      WHERE coalesce(g.grants_run, 0) + coalesce(a.approvals_given, 0) > 0 ORDER BY w.email""

```

Each list is fingerprinted with a SHA-256 of its rows, in the order written to its CSV. The fingerprint does not show the rows are right. It lets anyone who re-runs the script on the same files confirm they got the same rows, and shows if a list changed after it was handed over.

Run it on your own data

To run this on your own data, export each platform's records as CSV into one folder, using the file names in the table above and each platform's own column names (upper or lower case does not matter), and put an approval log, an HR leaver file and a login map in the same folder. At the top of the script, set `EXPORT_DIR` to that folder, `PERIOD_START` and `PERIOD_END` to your audit period, `AS_OF` to the day you took the exports, `PLATFORMS_EXPORTED` to the platforms you exported, and `FINANCIAL_TABLES` to your financial tables for each platform, written in lower case as `database.schema.table` (Redshift: `schema.table`). Set `KNOWN_GAPS` to any days with no records that you can explain. If your HR file or approval log names its columns differently, set `HR_EMAIL_COL`, `HR_LAST_DAY_COL`, `APPROVAL GRANTEE_COL`, `APPROVAL_OBJECT_COL`, `APPROVAL_APPROVER_COL` and `APPROVAL_DATE_COL` to your names. Add a row to the login map for every login, on any platform, that is not an email address. Change nothing else, and run it without `--demo`.

The code

The complete download, `sox_access_evidence.py`, exactly as run:

```
#!/usr/bin/env python3
"""SOX access evidence for financial tables in Snowflake, Databricks and Redshift, in DuckDB.

python3 sox_access_evidence.py --demo           # clean generated records
python3 sox_access_evidence.py --demo --plant   # generated records with planted faults
python3 sox_access_evidence.py                 # your exports: set the settings below first
python3 sox_access_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
an
approval log, an HR leaver file and a login map. Prints three lists for the financial tables you
name,
each with its period, filters, run time, row count and a SHA-256 of its rows:
  1. every access grant in the period, matched to an approval (NO APPROVAL, APPROVED AFTER GRANT,
    SELF-APPROVED are flagged)
  2. every HR leaver in the period, per platform: the first removal record credited to that
    departure,
    the days between, and use of a financial table after the removal
  3. anyone who both wrote to a financial table and ran or approved access to one in the period,
    and
    every login that could not be matched to a person
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, never as clean.

The demo records are GENERATED (seed 228): 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/query_history
           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>
 Redshift https://docs.aws.amazon.com/redshift/latest/dg/r_STL_UTILITYTEXT.html
https://docs.aws.amazon.com/redshift/latest/dg/SYS_USERLOG.html
https://docs.aws.amazon.com/redshift/latest/dg/SYS_QUERY_HISTORY.html
 Databricks' audit log reference (<https://docs.databricks.com/aws/en/admin/account-settings/audit-logs>) lists
 securable_full_name and changes as the request parameters of updatePermissions, and the system-
 table page shows
 full_name_arg and statusCode in its examples. The layout inside changes (a JSON list of principal,
 add, remove)
 is our assumption: open one of your own events and check it.
 """"
 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 audit period (UTC, inclusive)
 PERIOD_END = "2026-09-30" # last day of the audit 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
 FINANCIAL_TABLES = None # no default: {"snowflake": ["db.schema.table"], "databricks":
 [...], "redshift": ["schema.table"]}
 KNOWN_GAPS = {} # days you can explain, e.g. {"redshift": ["2026-08-19"]}
 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
 APPROVAL GRANTEE_COL = "grantee" # approval log: who the access was approved for (email)
 APPROVAL_OBJECT_COL = "object" # approval log: the table, written as in FINANCIAL_TABLES
 APPROVAL_APPROVER_COL = "approver" # approval log: who approved (email)
 APPROVAL_DATE_COL = "approved_on" # approval log: the day of approval, 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
 ACCESS_PRIVILEGES = ["select", "insert", "update", "delete", "modify", "ownership", "all", "all
 privileges"]
 # -----

 FILES = {
 "snowflake_grants_to_users": "snowflake_grants_to_users.csv",
 "snowflake_grants_to_roles": "snowflake_grants_to_roles.csv",
 "snowflake_query_history": "snowflake_query_history.csv",
 "snowflake_access_history": "snowflake_access_history.csv",
 "databricks_audit": "databricks_audit.csv",
 "databricks_table_lineage": "databricks_table_lineage.csv",
 "redshift_stl_utilitytext": "redshift_stl_utilitytext.csv",
 "redshift_sys_userlog": "redshift_sys_userlog.csv",
 "redshift_sys_query_history": "redshift_sys_query_history.csv",
 "hr_leavers": "hr_leavers.csv",
 }

```

    "approvals": "approvals.csv",
    "login_map": "login_map.csv",
}
PLATFORM_FILES = {
    "snowflake": ["snowflake_grants_to_users", "snowflake_grants_to_roles",
"snowflake_query_history", "snowflake_access_history"],
    "databricks": ["databricks_audit", "databricks_table_lineage"],
    "redshift": ["redshift_stl_utilitytext", "redshift_sys_userlog", "redshift_sys_query_history"],
}
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",
"granted_by"],
    "snowflake_grants_to_roles": ["created_on", "privilege", "granted_on", "name", "table_catalog",
"table_schema",
                                "granted_to", "grantee_name", "granted_by", "deleted_on"],
    "snowflake_query_history": ["query_id", "query_text", "query_type", "user_name", "role_name",
"start_time", "execution_status"],
    "snowflake_access_history": ["query_id", "query_start_time", "user_name", "objects_modified"],
    "databricks_audit": ["event_time", "event_date", "user_identity", "service_name",
"action_name", "request_params", "response"],
    "databricks_table_lineage": ["target_table_full_name", "created_by", "event_time",
"entity_type"],
    "redshift_stl_utilitytext": ["userid", "xid", "pid", "starttime", "endtime", "sequence",
"text"],
    "redshift_sys_userlog": ["user_id", "user_name", "action", "record_time"],
    "redshift_sys_query_history": ["user_id", "username", "query_id", "query_type", "status",
"start_time", "query_text"],
    "login_map": ["platform", "login", "email"],
}
OUT_DIR = pathlib.Path("evidence-out")
GEN_DIR = pathlib.Path("generated")
DEMO_SCOPE = {"snowflake": ["finance_db.ledger.gl_entries"], "databricks":
["finance.ledger.gl_entries"],
              "redshift": ["ledger.gl_entries"]}
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*$'

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(228)
    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]

    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 = {"f01": 101, "f02": 102, "f03": 103, "f04": 104, "f05": 105, "f06": 106, "f07": 107,
"f08": 108,
        "a01": 201, "a02": 202, "m01": 301, "m02": 302, "etl_svc": 401, "rs_batch2": 402}

    def sf_q(t, user, role, qtype, text):
        qn[0] += 1
        rows["snowflake_query_history"].append([f"sfq{qn[0]:06d}", text, qtype, user.upper(), role,
sfts(t), "SUCCESS"])
        return f"sfq{qn[0]:06d}"

    def sf_use(t, user, write=False):
        text = ("INSERT INTO finance_db.ledger.gl_entries SELECT 1" if write else
        "SELECT * FROM finance_db.ledger.gl_entries LIMIT 10")
        q = sf_q(t, em(user), "FIN_READER", "INSERT" if write else "SELECT", text)
        if write:
            rows["snowflake_access_history"].append(
                [q, sfts(t), em(user).upper(),
                json.dumps({"objectName": "FINANCE_DB.LEDGER.GL_ENTRIES", "objectDomain":
"Table", "columns": []})])

    def dbx(t, actor, action, params, status=200, service="unityCatalog"):
        rows["databricks_audit"].append([plain(t), t.date().isoformat(), json.dumps({"email":
actor}), service, action,
                                json.dumps(params), json.dumps({"statusCode": status})])

    def dbx_perm(t, actor, principal, op, privs, status=200):
        dbx(t, actor, "updatePermissions", {"securable_type": "table", "securable_full_name":
"finance.ledger.gl_entries",
        "changes": json.dumps([{"principal": principal, op: privs})]), status)

    def dbx_use(t, user, write=False):
        if write:
            rows["databricks_table_lineage"].append(["finance.ledger.gl_entries", em(user),
plain(t), "NOTEBOOK"])
        else:
            dbx(t, em(user), "getTable", {"full_name_arg": "finance.ledger.gl_entries"})

```

```

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], qn[0], qtype, status, plain(t),
text])

    # ---- Snowflake roles
    for role, privs in [("FIN_READER", ["SELECT"]), ("FIN_WRITER", ["SELECT", "INSERT",
"UPDATE"])]):
        for p in privs:
            rows["snowflake_grants_to_roles"].append(["2026-01-10 09:00:00.000 +0000", p, "TABLE",
"GL_ENTRIES", "FINANCE_DB",
"LEDGER", "ROLE", role, "SECURITYADMIN", ""])
            rows["snowflake_grants_to_roles"].append(["2026-01-10 09:00:00.000 +0000", "SELECT", "TABLE",
"DEPT_DIRECTORY", "HR_DB",
"PUBLIC", "ROLE", "HR_READER", "SECURITYADMIN", ""])

    # ---- access: (person, platform, role or privilege, grant day, approval day, approver, who
granted)
    plan = [("f01", "snowflake", "FIN_READER", D(2026, 7, 6), D(2026, 7, 9) if plant else D(2026,
7, 3), "m01", "a01"),
("f01", "databricks", "MODIFY", D(2026, 7, 8), D(2026, 7, 7), "m01", "a02"),
("f02", "redshift", "INSERT", D(2026, 7, 15), D(2026, 7, 14), "f02" if plant else
"m02", "a01"),
("f03", "snowflake", "FIN_WRITER", D(2026, 7, 22), D(2026, 7, 20), "m02", "a02"),
("f04", "databricks", "SELECT", D(2026, 8, 3), None if plant else D(2026, 8, 1), "m01",
"a01"),
("f08", "databricks", "SELECT", D(2026, 7, 28), D(2026, 7, 27), "m02", "a02")]
    before = [("f05", "snowflake", "FIN_READER", D(2026, 5, 11), "a01"), ("f05", "databricks",
"SELECT", D(2026, 5, 12), "a02"),
("f06", "redshift", "SELECT", D(2026, 5, 13), "a01"), ("f07", "snowflake",
"FIN_READER", D(2026, 5, 14), "a02"),
("f08", "databricks", "SELECT", D(2026, 5, 15), "a01")]
    sf_grant = {}
    approvals = []
    tk = 100
    allgr = [(p, pl, w, g, None, None, ad) for p, pl, w, g, ad in before] + plan
    for person, plat, what, d, ap_day, approver, admin in allgr:
        t = at(d)
        if plat == "snowflake":
            row = [sfts(t), "", what, em(person).upper(), "SECURITYADMIN"]
            rows["snowflake_grants_to_users"].append(row)
            sf_grant[(person, d)] = row
            sf_q(t, em(admin), "SECURITYADMIN", "GRANT", f'GRANT ROLE {what} TO USER "
{em(person).upper()}')
            obj = "finance_db.ledger.gl_entries"
        elif plat == "databricks":
            dbx_perm(t, em(admin), em(person), "add", [what])
            obj = "finance.ledger.gl_entries"
        else:

```

```

        tkt = f"/* ticket T-{tk} " + "x" * 215 + " */ " if person == "f02" else ""
        rs_dcl(t, admin, f"{tkt}GRANT {what} ON TABLE ledger.gl_entries TO {person};")
        obj = "ledger.gl_entries"
    tk += 1
    if ap_day is not None:
        approvals.append([f"T-{tk}", em(person).title(), obj.upper() if plat == "snowflake"
else obj,
                        em(approver), ap_day.isoformat()])

    # ---- removals
    hr = [[em("f05"), "2026-08-07"], [em("f06"), "2026-08-21"], [em("f07"), "2026-09-04"],
[em("f08"), "2026-07-10"],
        [em("f08"), "2026-09-18"]]
    if not plant:
        sf_grant(["f05", D(2026, 5, 11)])[1] = sfts(at(D(2026, 8, 8)))
        dbx_perm(at(D(2026, 8, 8)), em("a01"), em("f05"), "remove", ["SELECT"])
        rs_dcl(at(D(2026, 8, 30) if plant else D(2026, 8, 22)), "a02", "REVOKE SELECT ON TABLE
ledger.gl_entries FROM f06;")
        sf_grant(["f07", D(2026, 5, 14)])[1] = sfts(at(D(2026, 9, 5)))
        dbx_perm(at(D(2026, 7, 11)), em("a02"), em("f08"), "remove", ["SELECT"])
        if not plant:
            dbx_perm(at(D(2026, 9, 19)), em("a01"), em("f08"), "remove", ["SELECT"])
        rows["redshift_sys_userlog"].append([201, "f06", "alter", plain(at(D(2026, 5, 13)))])
    # ---- use of the financial tables
    def every(a, b, step, fn):
        d = a
        while d <= min(b, as_of):
            fn(at(d), d)
            d += dt.timedelta(days=step)

    every(D(2026, 7, 7), as_of, 6, lambda t, d: sf_use(t, "f01"))
    every(D(2026, 7, 9), as_of, 6, lambda t, d: dbx_use(t, "f01", write=True))
    every(D(2026, 7, 16), as_of, 5, lambda t, d: (rs_q(t, "f02", "INSERT", "INSERT INTO
ledger.gl_entries SELECT 1"),))
    every(D(2026, 7, 23), as_of, 6, lambda t, d: sf_use(t, "f03", write=True))
    every(D(2026, 8, 4), as_of, 9, lambda t, d: dbx_use(t, "f04"))
    every(D(2026, 7, 1), D(2026, 8, 7), 7, lambda t, d: sf_use(t, "f05"))
    every(D(2026, 7, 1), D(2026, 8, 7), 9, lambda t, d: dbx_use(t, "f05"))
    every(D(2026, 7, 1), D(2026, 8, 21), 8, lambda t, d: rs_q(t, "f06", "SELECT", "SELECT * FROM
ledger.gl_entries LIMIT 10"))
    every(D(2026, 7, 1), D(2026, 9, 4), 9, lambda t, d: sf_use(t, "f07"))
    every(D(2026, 7, 1), D(2026, 7, 10), 4, lambda t, d: dbx_use(t, "f08"))
    every(D(2026, 7, 29), D(2026, 9, 18), 8, lambda t, d: dbx_use(t, "f08"))
    every(D(2026, 7, 5), as_of, 10, lambda t, d: rs_q(t, "etl_svc", "INSERT", "INSERT INTO
ledger.gl_entries SELECT 1"))
    # ---- background: every day has some records on every platform
    d = start
    while d <= as_of:
        sf_q(at(d), "bg@example.test", "ANALYST", "SELECT", "SELECT 1")
        dbx(at(d), "bg@example.test", "getTable", {"full_name_arg": "hr.dept.directory"})
        rs_q(at(d), "a01", "SELECT", "SELECT 1")
        rs_q(at(d), "a02", "SELECT", "SELECT 1")
        d += dt.timedelta(days=1)
        dbx_perm(at(D(2026, 8, 12)), em("a01"), em("f09"), "add", ["SELECT"], status=403) # failed
attempt: not an access change
        for u in ["m01", "m02"]:
            rs_q(at(D(2026, 7, 2)), u, "SELECT", "SELECT 1")
    if plant:
        sf_use(at(D(2026, 8, 15)), "f05")

```

leaver f05 still

```

reads after the grace period
    sf_use(at(D(2026, 8, 20)), "f05")
    rs_q(at(D(2026, 8, 25)), "f06", "SELECT", "SELECT * FROM ledger.gl_entries LIMIT 10")  #
f06, before the late removal
    sf_use(at(D(2026, 9, 14)), "f07")  # f07 reads 9 days
after the removal
    dbx_use(at(D(2026, 8, 20)), "a02", write=True)  # an admin who grants
access also writes
    rs_q(at(D(2026, 8, 12)), "rs_batch2", "INSERT", "INSERT INTO ledger.gl_entries SELECT 1")
# a login nobody mapped
    rs_dcl(at(D(2026, 8, 26)), "a01", "GRANT SELECT ON TABLE ledger.gl_entries TO f03, f04;")
# one grantee list
    rows["snowflake_grants_to_users"].append(["07/09/2026 10:00", "", "HR_READER",
"F03@EXAMPLE.TEST", "SECURITYADMIN"])
    rows["snowflake_query_history"].append(list(rows["snowflake_query_history"][40]))
# a repeated query id
    hr.append([em("f09"), "8/5/26"])
    approvals.append(["T-999", em("f02"), "ledger.gl_entries", "", "2026-08-01"])
    rows["redshift_sys_query_history"] = [r for r in rows["redshift_sys_query_history"] if not
r[4].startswith("2026-08-19")]
    rows["redshift_stl_utilitytext"] = [r for r in rows["redshift_stl_utilitytext"] if not
r[3].startswith("2026-08-19")]
# ---- write
for key, cols in COLUMNS.items():
    if key in ("login_map", "redshift_sys_query_history"):
        continue
    name = FILES[key]
    head = [c.upper() for c in cols] if key.startswith("snowflake") else cols
    with open(out / name, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(head)
        w.writerows(rows[key])
rs_names = {rs_ids[k]: k for k in rs_ids}
for r in rows["redshift_sys_query_history"]:
    r.insert(1, rs_names[r[0]])
with open(out / FILES["redshift_sys_query_history"], "w", newline="") as f:
    w = csv.writer(f)
    w.writerow(COLUMNS["redshift_sys_query_history"])
    w.writerows(rows["redshift_sys_query_history"])
with open(out / FILES["hr_leavers"], "w", newline="") as f:
    w = csv.writer(f)
    w.writerow(["email", "last_day"])
    w.writerows(hr)
with open(out / FILES["approvals"], "w", newline="") as f:
    w = csv.writer(f)
    w.writerow(["ticket", "grantee", "object", "approver", "approved_on"])
    w.writerows(approvals)
with open(out / FILES["login_map"], "w", newline="") as f:
    w = csv.writer(f)
    w.writerow(["platform", "login", "email"])
    w.writerows([["redshift", k, em(k)] for k in rs_ids if k not in ("etl_svc", "rs_batch2")]
+ [["redshift", "etl_svc", "data-eng@example.test"]])
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)")
    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 main():
    demo, plant, strict = "--demo" in sys.argv, "--plant" in sys.argv, "--strict" in sys.argv
    folder, scope = EXPORT_DIR, FINANCIAL_TABLES
    if demo:
        folder = generate(plant)
        scope = {k: list(v) for k, v in DEMO_SCOPE.items()}
        if plant:
            scope["redshift"].append("ledger.gl_entires") # planted: a typo in the scope
            print(f"input : GENERATED records (not a company's data), seed 228, plant={plant}")
        else:
            print("input : " + str(folder))
    if scope is None:
        print("STOP: FINANCIAL_TABLES has no default. Name the financial tables for each platform,
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:
        print("STOP: PERIOD_START, PERIOD_END and AS_OF must be YYYY-MM-DD with PERIOD_START <=

```

```

PERIOD_END <= AS_OF.")
    return 2
    p0, p1, asof = PERIOD_START, PERIOD_END, AS_OF
    exported = [p for p in ["snowflake", "databricks", "redshift"] if p in PLATFORMS_EXPORTED]
    not_checked = [p for p in ["snowflake", "databricks", "redshift"] if p not in exported]
    unknown = [p for p in PLATFORMS_EXPORTED if p not in PLATFORM_FILES]
    if unknown 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, "hr_leavers", folder, ["email", "last_day"], {"email":
HR_EMAIL_COL.lower(), "last_day": HR_LAST_DAY_COL.lower()}))
        problems.append(load(con, "approvals", folder, ["grantee", "object", "approver",
"approved_on"],
                        {"grantee": APPROVAL_GRANTEE_COL.lower(), "object":
APPROVAL_OBJECT_COL.lower(),
                        "approver": APPROVAL_APPROVER_COL.lower(), "approved_on":
APPROVAL_DATE_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

    names = [(p, o.strip().lower()) for p in exported for o in scope.get(p, [])]
    con.execute("CREATE TABLE scope (platform VARCHAR, obj VARCHAR)")
    con.executemany("INSERT INTO scope VALUES (?, ?)", names)
    con.execute("CREATE TABLE privs (p VARCHAR)")
    norm = lambda x: x.strip().lower().replace("_", " ") # ALL_PRIVILEGES (Databricks) is "all
privileges"
    con.executemany("INSERT INTO privs VALUES (?)", [(norm(x),) for x in ACCESS_PRIVILEGES])
    priv_list = "[" + ", ".join("'" + norm(x).replace("'", "'") + "'" for x in ACCESS_PRIVILEGES)
+ "]"
    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, '')) <> ''""")
    # a login that is itself an email is the person; anything else needs a row in login_map
    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]))

    # ---- Snowflake
    dup = 0
    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,
            lower(trim(granted_by)) AS granted_by FROM raw_snowflake_grants_to_users""")
        con.execute(f"""CREATE TABLE sf_gr AS SELECT {ts('created_on')} AS ts, {ts('deleted_on')}
        AS end_ts, trim(coalesce(deleted_on, '')) AS end_text,
            upper(trim(privilege)) AS privilege,
            lower(trim(table_catalog) || '.' || trim(table_schema) || '.' || trim(name)) AS obj,
        lower(trim(grantee_name)) AS role
        FROM raw_snowflake_grants_to_roles WHERE upper(trim(granted_on)) = 'TABLE' AND
        upper(trim(granted_to)) = 'ROLE'""")
        con.execute(f"""CREATE TABLE sf_qh AS SELECT trim(query_id) AS id,
        lower(replace(query_text, ',', '')) AS text,
            upper(trim(query_type)) AS query_type, lower(trim(user_name)) AS login,
        lower(trim(role_name)) AS role,
            {ts('start_time')} AS ts, lower(trim(execution_status)) AS status
        FROM raw_snowflake_query_history""")
        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,
            objects_modified 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", "ts")
        count_bad("GRANTS_TO_ROLES.DELETED_ON", "sf_gr", "end_text", "end_ts", False)
        count_bad("QUERY_HISTORY.START_TIME", "sf_qh", "ts", "ts")
        count_bad("ACCESS_HISTORY.QUERY_START_TIME", "sf_ah", "ts", "ts")
        dup += con.execute("SELECT count(*) FROM (SELECT id FROM sf_qh GROUP BY id HAVING count(*)
        > 1)").fetchone()[0]
        con.execute("CREATE TABLE sf_mod (id VARCHAR, ts TIMESTAMP, login VARCHAR, obj VARCHAR)")
        bad_json, mod_rows = 0, []
        for qid, t, login, js in con.execute("SELECT id, ts, login, objects_modified FROM
        sf_ah").fetchall():
            try:
                for o in json.loads(js or "[]"):
                    mod_rows.append((qid, t, login, str(o["objectName"]).strip().lower()))
            except (ValueError, KeyError, TypeError):
                bad_json += 1
        con.executemany("INSERT INTO sf_mod VALUES (?, ?, ?, ?)", mod_rows)
        con.execute(f"""CREATE TABLE sf_acc AS
        SELECT u.ts, u.end_ts, u.login, u.role, u.granted_by, r.obj, string_agg(DISTINCT
        lower(replace(r.privilege, '_', ' ')), ',' ORDER BY lower(replace(r.privilege, '_', ' '))) AS privs
        FROM sf_gu u JOIN sf_gr r ON r.role = u.role AND r.obj IN (SELECT obj FROM scope WHERE
        platform = 'snowflake')
        AND lower(replace(r.privilege, '_', ' ')) IN (SELECT p FROM privs) AND r.ts <=
        u.ts AND (r.end_ts IS NULL OR r.end_ts > u.ts)
        WHERE u.ts IS NOT NULL GROUP BY ALL""")
    else:
        bad_json = 0

```

```

# ---- Databricks
bad_changes = 0
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, '$.email'))) AS login,
lower(trim(service_name)) AS service,
                trim(action_name) AS action, request_params, response,
                coalesce(json_extract_string(response, '$.statusCode'), json_extract_string(response,
'$.status_code')) AS status
                FROM raw_databricks_audit""")
    con.execute(f"""CREATE TABLE dbx_l AS SELECT lower(trim(target_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""")
    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, actor VARCHAR, obj VARCHAR, principal
VARCHAR, op VARCHAR, privilege VARCHAR)")
    chg, failed_changes = [], 0
    dbx_scope = {o for p, o in names if p == "databricks"}
    for t, actor, status, rp in con.execute("""SELECT ts, login, status, request_params FROM
dbx_a
                WHERE service = 'unitycatalog' AND action = 'updatePermissions' AND ts IS NOT
NULL""").fetchall():
        try:
            p = json.loads(rp)
            obj = str(p["securable_full_name"]).strip().lower()
            if obj not in dbx_scope: # a change on another table 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
            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, actor, obj, str(c["principal"]).strip().lower(), op,
str(pr).strip().lower().replace("_", " "))
                        except (ValueError, KeyError, TypeError, AttributeError):
                            bad_changes += 1
            con.executemany("INSERT INTO dbx_chg VALUES (?, ?, ?, ?, ?, ?)", chg)
            bad_changes += con.execute("""SELECT count(*) FROM dbx_a WHERE service = 'unitycatalog' AND
action <> 'updatePermissions'
                AND lower(json_extract_string(request_params, '$.full_name_arg')) IN (SELECT obj FROM
scope WHERE platform = 'databricks')
                AND (status IS NULL OR NOT regexp_full_match(trim(status), '\\d+'))""").fetchone()[0]
        else:
            failed_changes = 0

# ---- Redshift
bad_dcl = 0
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")
    con.execute("CREATE TABLE rs_users AS SELECT uid, any_value(login) AS login FROM rs_qh
GROUP BY uid")
    # 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(regex_replace(stmt, '/\s*.*?\s*/', ' ', 'gs')) AS s FROM st)
    SELECT uid, ts, s, lower(regex_extract(s, '{DCL_RE}', 1)) AS kind, regex_extract(s,
'{DCL_RE}', 2) AS privs,
        lower(replace(regex_extract(s, '{DCL_RE}', 3), '', ' ')) AS obj,
        lower(replace(regex_extract(s, '{DCL_RE}', 4), '', ' ')) AS grantee
    FROM cl WHERE regex_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]

    # ---- events: grants, ends of access, use of the financial tables (login level)
    parts_g, parts_e, parts_u = [], [], []
    if "snowflake" in exported:
        con.execute("""CREATE TABLE sf_g AS WITH m AS (
            SELECT a.*, q.login AS grantor, row_number() OVER (PARTITION BY a.ts, a.login,
a.role, a.obj
                ORDER BY abs(epoch(q.ts) - epoch(a.ts))) AS rn
            FROM sf_acc a LEFT JOIN sf_qh q ON q.query_type = 'GRANT' AND q.role = a.granted_by
            AND abs(epoch(q.ts) - epoch(a.ts)) <= 60 AND contains(q.text, a.role) AND
contains(q.text, a.login))
            SELECT * FROM m WHERE rn = 1""")
        parts_g.append("SELECT 'snowflake' AS platform, ts, login, obj, privs, grantor AS
grantor_login FROM sf_g")
        parts_e.append("SELECT 'snowflake' AS platform, end_ts AS ts, login, obj FROM sf_acc WHERE
end_ts IS NOT NULL")
        parts_u.append("""SELECT 'snowflake' AS platform, q.ts, q.login, s.obj, false AS is_write
FROM sf_qh q JOIN scope s
            ON s.platform = 'snowflake' AND contains(q.text, s.obj) WHERE q.query_type <> 'GRANT'
AND q.ts IS NOT NULL
            AND q.status = 'success' AND NOT EXISTS (SELECT 1 FROM sf_mod m WHERE m.id = q.id AND
m.obj = s.obj)""")
        parts_u.append("SELECT 'snowflake', m.ts, m.login, m.obj, true FROM sf_mod m WHERE m.obj IN
(SELECT obj FROM scope WHERE platform = 'snowflake') AND m.ts IS NOT NULL")
    if "databricks" in exported:

```

```

parts_g.append(f""""SELECT 'databricks' AS platform, ts, principal AS login, obj,
string_agg(DISTINCT privilege, ',' ORDER BY privilege) AS privs,
any_value(actor) AS grantor_login FROM dbx_chg WHERE op = 'add' AND privilege IN
(SELECT p FROM privs)
AND obj IN (SELECT obj FROM scope WHERE platform = 'databricks') GROUP BY ts,
principal, obj""")
parts_e.append(f""""SELECT DISTINCT 'databricks' AS platform, ts, principal AS login, obj
FROM dbx_chg WHERE op = 'remove'
AND privilege IN (SELECT p FROM privs) AND obj IN (SELECT obj FROM scope WHERE
platform = 'databricks')""")
parts_u.append(f""""SELECT 'databricks' AS platform, a.ts, a.login, s.obj, false AS is_write
FROM dbx_a a JOIN scope s
ON s.platform = 'databricks' AND lower(json_extract_string(a.request_params,
'$.full_name_arg')) = s.obj
WHERE a.service = 'unitycatalog' AND a.action <> 'updatePermissions' AND a.ts IS NOT
NULL AND a.status = '200'""")
parts_u.append(f""""SELECT 'databricks', ts, login, obj, true FROM dbx_l WHERE obj IN (SELECT
obj FROM scope WHERE platform = 'databricks') AND ts IS NOT NULL""")
if "redshift" in exported:
parts_g.append(f""""SELECT 'redshift' AS platform, d.ts, d.grantee AS login, d.obj,
lower(trim(d.privs)) AS privs, coalesce(u.login, 'userid ' || d.uid) AS grantor_login
FROM rs_dcl d LEFT JOIN rs_users u ON u.uid = d.uid WHERE d.kind = 'grant' AND d.ts
IS NOT NULL
AND list_has_any(list_transform(string_split(lower(replace(d.privs, '_', ' '))), ','),
x -> trim(x)), {priv_list})
AND d.obj IN (SELECT obj FROM scope WHERE platform = 'redshift')""")
parts_e.append(f""""SELECT 'redshift' AS platform, d.ts, d.grantee AS login, d.obj FROM
rs_dcl d WHERE d.kind = 'revoke' AND d.ts IS NOT NULL
AND list_has_any(list_transform(string_split(lower(replace(d.privs, '_', ' '))), ','),
x -> trim(x)), {priv_list})
AND d.obj IN (SELECT obj FROM scope WHERE platform = 'redshift')""")
parts_u.append(f""""SELECT 'redshift' AS platform, ts, login, NULL AS obj FROM rs_ul WHERE
action = 'drop' AND ts IS NOT NULL""")
# a write is an INSERT, UPDATE, DELETE or COPY whose target is the financial table, not one
that only reads it
parts_u.append(f""""SELECT 'redshift' AS platform, q.ts, q.login, s.obj, q.query_type IN
('INSERT', 'UPDATE', 'DELETE', 'COPY')
AND (q.target = s.obj OR suffix(q.target, '.' || s.obj)) AS is_write
FROM (SELECT *, regexp_extract(trim(regexp_replace(regexp_replace(text, '/\\*.??
\\*/', '', 'gs')), '--[^\n]*', '', 'g')),
'^\\s*
(?:insert\\s+into|update|delete\\s+from|delete|copy)\\s+([^\s(\\s+)]', 1) AS target
FROM rs_qh) q JOIN scope s ON s.platform = 'redshift' AND contains(q.text,
s.obj)
WHERE q.status NOT IN ('failed', 'canceled') AND q.ts IS NOT NULL
AND NOT regexp_matches(q.text, '^\\s*(grant|revoke)\\b')""")
con.execute("CREATE TABLE grants_r AS SELECT platform, ts, login, who(platform, login) AS
email, obj, privs, grantor_login, "
"who(platform, grantor_login) AS grantor_email FROM (" + " UNION ALL
".join(parts_g) + ")")
con.execute("CREATE TABLE ends_r AS SELECT platform, ts, login, who(platform, login) AS email,
obj FROM (" + " UNION ALL " + ".join(parts_e) + ")")
con.execute("CREATE TABLE usage_r AS SELECT platform, ts, login, who(platform, login) AS email,
obj, is_write FROM (" + " UNION ALL " + ".join(parts_u) + ")")

# ---- people and logs 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""")
    con.execute(f""""CREATE TABLE approvals AS SELECT lower(trim(coalesce(grantee, ''))) AS grantee,
lower(trim(coalesce(object, ''))) AS obj,
        lower(trim(coalesce(approver, ''))) AS approver, trim(coalesce(approved_on, '')) AS
date_text,
        {day('approved_on')} AS approved_on FROM raw_approvals""")

# ---- input checks: each says which planted fault trips it
days = {}
for p, sql in [("snowflake", "SELECT DISTINCT ts::DATE FROM sf_qh 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 '{p0}', 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, obj in names:
        seen = con.execute("""SELECT (SELECT count(*) FROM grants_r WHERE platform = ? AND obj = ?)
+ (SELECT count(*) FROM usage_r WHERE platform = ? AND obj = ?)
+ (SELECT count(*) FROM ends_r WHERE platform = ? AND obj = ?)""",
[p, obj, p, obj, p, obj]).fetchone()[0]
        if seen == 0:
            unseen.append(f"{p}:{obj}")
    no_scope = [p for p in exported if not scope.get(p)]
    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]
    ap_bad = con.execute("SELECT count(*) FROM approvals WHERE grantee = '' OR obj = '' OR approver
= '' OR date_text = ''").fetchone()[0]
    ap_date = con.execute("SELECT count(*) FROM approvals WHERE date_text <> '' AND approved_on IS
NULL").fetchone()[0]
    map_bad = con.execute("SELECT count(*) FROM raw_login_map WHERE trim(coalesce(login, '')) = ''
OR trim(coalesce(email, '')) = ''").fetchone()[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 PERIOD_START to AS_OF", sum(len(v) for v in
days.values()), "a day with no Redshift records"),
        ("scope names that match no grant, use or removal in the exports", len(unseen) +
len(no_scope), "a typo in FINANCIAL_TABLES"),
        ("Databricks audit events or Snowflake write lists the script could not read", bad_changes
+ bad_json, "malformed JSON"),
        ("Redshift GRANT or REVOKE on a financial table the script could not parse", bad_dcl, "a
grant to two users in one statement"),
        ("HR leaver rows with no email or no last day", hr_bad, "a NULL in the HR file"),
        ("HR leaver rows whose last day is not written as YYYY-MM-DD", hr_date, "a last day written
8/5/26"),
        ("approval rows with a blank grantee, table, approver or date", ap_bad, "a blank
approver"),
        ("approval rows whose date is not written as YYYY-MM-DD", ap_date, "an approval date
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: {p0} to {asof})")
ok = True
for label, n, fault in checks:
    ok &= n == 0
    print(f"[{'ok' if n == 0 else 'FAIL'}] {label}: {n}" + (" (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(f"the STL views keep seven days (AWS): export daily, or unload the logs to Amazon S3")
    if unseen or no_scope:
        print(f"scope: " + ", ".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}")
    n_unk = con.execute("SELECT count(*) FROM grants_r WHERE ts::DATE BETWEEN DATE '%s' AND DATE '%s' AND grantor_login IS NULL" % (p0, p1)).fetchone()[0]
    print(f"[info] grants in the period where no person who ran them was found: {n_unk}")

# ---- list 1: access approved before it was granted
sql = f"""
    WITH g AS (SELECT row_number() OVER (ORDER BY ts, platform, login, obj, privs) AS gid, *
FROM grants_r
    WHERE ts::DATE BETWEEN DATE '{p0}' AND DATE '{p1}'),
    cand AS (SELECT g.gid, a.approver, a.approved_on,
        row_number() OVER (PARTITION BY g.gid ORDER BY CASE WHEN a.approved_on <=
g.ts::DATE THEN 0 ELSE 1 END,
            abs(date_diff('day', a.approved_on, g.ts::DATE)),
a.approver = coalesce(g.email, g.login), a.approver) AS rn
    FROM g JOIN approvals a ON a.grantee = coalesce(g.email, g.login) AND a.obj =
g.obj AND a.approved_on IS NOT NULL
    AND a.approver <> '')
    SELECT g.platform, g.ts::DATE AS granted_on, coalesce(g.email, g.login) AS grantee, g.obj
AS object, g.privs,
    coalesce(g.grantor_email, g.grantor_login, '-') AS granted_by_person, c.approver,
c.approved_on,
    CASE WHEN c.gid IS NULL THEN 'NO APPROVAL'
    WHEN c.approver = coalesce(g.email, g.login) THEN 'SELF-APPROVED'
    WHEN c.approved_on > g.ts::DATE THEN 'APPROVED AFTER GRANT'
    ELSE 'approved before grant' END AS status
    FROM g LEFT JOIN cand c ON c.gid = g.gid AND c.rn = 1 ORDER BY g.ts, g.platform, grantee,
g.obj, g.privs"""
l1 = population(con, "sox_1_access_grants", "Access approved before it was granted",
    f"grants of {sorted(set(ACCESS_PRIVILEGES))} on the tables in FINANCIAL_TABLES,
granted in the period; one row per person and table",
    sql, ["platform", "granted_on", "grantee", "object", "privileges",
"granted_by_person", "approver", "approved_on", "status"], ok, OUT_DIR)
for r in l1: # every row when there are 30 or fewer, otherwise only the ones to explain
    if len(l1) <= 30 or r[8] != "approved before grant":
        print(" " + " | ".join("--" if v is None else str(v) for v in r))

```

```

# ---- list 2: leavers, per platform, each departure on its own
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 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
                    AND l.next_last_day IS NULL))), -- a rehire's earlier departure needs
a grant record
    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(u.ts) 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(u.ts) 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 uses_after_grace, a.n_removed AS uses_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""""
l2 = population(con, "sox_2_leavers", "Leavers",
                f"HR leavers with last day in the period who held access to a financial table
on the platform, or used 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", "uses_after_grace",

```

```

        "uses_after_removal", "status"], ok, OUT_DIR)

for r in l2:
    print(" " + " | ".join("-" if v is None else str(v) for v in r))
for p in exported:
    n = con.execute(f"""SELECT count(*) FROM leavers l WHERE l.last_day BETWEEN DATE '{p0}' AND
DATE '{p1}' AND l.email <> ''
        AND NOT EXISTS (SELECT 1 FROM grants_r g WHERE g.email = l.email AND g.platform =
'{p}' AND g.ts::DATE <= l.last_day
        AND NOT EXISTS (SELECT 1 FROM leavers l0 WHERE l0.email = l.email AND
l0.last_day < l.last_day
        AND l0.last_day >= g.ts::DATE))
        AND NOT (EXISTS (SELECT 1 FROM usage_r u WHERE u.email = l.email AND u.platform =
'{p}'
        AND u.ts::DATE > l.last_day + INTERVAL {GRACE_DAYS} DAY)
        AND NOT EXISTS (SELECT 1 FROM leavers l2 WHERE l2.email = l.email AND
l2.last_day > l.last_day))""").fetchone()[0]
    print(f"[info] HR leavers in the period with no grant on a financial table and no use after
the grace period seen on {p}, so not listed there: {n}")

# ---- list 3: segregation of duties
sql = f"""
    WITH w AS (SELECT email, count(*) AS writes, string_agg(DISTINCT platform, ',' ORDER BY
platform) AS wrote_on FROM usage_r
        WHERE is_write AND email IS NOT NULL AND ts::DATE BETWEEN DATE '{p0}' AND DATE
'{p1}' GROUP BY 1),
    g AS (SELECT grantor_email AS email, count(*) AS grants_run FROM grants_r WHERE
grantor_email IS NOT NULL
        AND ts::DATE BETWEEN DATE '{p0}' AND DATE '{p1}' GROUP BY 1),
    a AS (SELECT approver AS email, count(*) AS approvals_given FROM approvals WHERE
approver <> ''
        AND obj IN (SELECT obj FROM scope) AND approved_on BETWEEN DATE '{p0}' AND DATE
'{p1}' GROUP BY 1)
    SELECT w.email, w.writes, w.wrote_on, coalesce(g.grants_run, 0) AS grants_run,
coalesce(a.approvals_given, 0) AS approvals_given,
        'WROTE AND ' || CASE WHEN g.grants_run > 0 AND a.approvals_given > 0 THEN 'GRANTED
AND APPROVED'
        WHEN g.grants_run > 0 THEN 'GRANTED' ELSE 'APPROVED' END AS
status
    FROM w LEFT JOIN g ON g.email = w.email LEFT JOIN a ON a.email = w.email
    WHERE coalesce(g.grants_run, 0) + coalesce(a.approvals_given, 0) > 0 ORDER BY w.email"""
l3 = population(con, "sox_3_segregation_of_duties", "Segregation of duties",
    "people (matched by email across platforms) who wrote to a financial table and
also ran a grant of access to one or approved access to one, in the period",
    sql, ["email", "writes", "wrote_on", "grants_run", "approvals_given",
"status"], ok, OUT_DIR)
for r in l3:
    print(" " + " | ".join(str(v) for v in r))
sql = f"""
    SELECT platform, login, count(*) FILTER (WHERE kind = 'write') AS writes, count(*) FILTER
(WHERE kind = 'grant') AS grants_run
    FROM (SELECT platform, login, 'write' AS kind FROM usage_r WHERE is_write AND email IS NULL
AND ts::DATE BETWEEN DATE '{p0}' AND DATE '{p1}'
        UNION ALL SELECT platform, grantor_login, 'grant' FROM grants_r WHERE grantor_email
IS NULL AND grantor_login IS NOT NULL
        AND ts::DATE BETWEEN DATE '{p0}' AND DATE '{p1}') GROUP BY 1, 2 ORDER BY 1, 2"""
unmatched = con.execute(sql).fetchall()
print(f"[info] logins that wrote to a financial table or ran a grant and match no person (add
them to login_map): {len(unmatched)}")

```

```

for r in unmatched:
    print(" UNMATCHED LOGIN | " + " | ".join(str(v) for v in r))

# ---- result
f1 = sum(1 for r in l1 if r[8] != "approved before grant")
f2 = sum(1 for r in l2 if r[7] != "removed in time" or r[5] or r[6])
f3 = len(l3) + len(unmatched) + n_unk
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} grant row(s), {f2} leaver row(s), {f3} segregation or login row(s)
need an explanation")
if not ok or f1 + f2 + f3:
    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

`python3 sox_access_evidence.py --demo --plant`

```

input      : GENERATED records (not a company's data), seed 228, 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 PERIOD_START to AS_OF: 1 (trips on: a day with no
Redshift records)
[FAIL] scope names that match no grant, use or removal in the exports: 1 (trips on: a typo in
FINANCIAL_TABLES)
[ok ] Databricks audit events or Snowflake write lists the script could not read: 0
[FAIL] Redshift GRANT or REVOKE on a financial table the script could not parse: 1 (trips on: a
grant to two users in one statement)
[ok ] HR leaver rows with no email or no last day: 0
[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] approval rows with a blank grantee, table, approver or date: 1 (trips on: a blank
approver)
[ok ] approval rows whose date is not written as YYYY-MM-DD: 0
[ok ] login map rows with a blank login or email: 0
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
scope: redshift:ledger.gl_entires
[info] permission changes that failed (status not 200), left out: 1
[info] grants in the period where no person who ran them was found: 0

== Access approved before it was granted: sox_1_access_grants  ** INCOMPLETE: see input checks **
period      : 2026-07-01 to 2026-09-30 (UTC, inclusive)
filters     : grants of ['all', 'all privileges', 'delete', 'insert', 'modify', 'ownership',
'select', 'update'] on the tables in FINANCIAL_TABLES, granted in the period; one row per person
and table
run at      : 2026-10-31 23:02:55 UTC
rows        : 6
sha256      : 73f5abcf0511706707b275816af19fb10edeb7a3ad3d858e7bae45757d6175ee
  snowflake | 2026-07-06 | f01@example.test | finance_db.ledger.gl_entries | select |
a01@example.test | m01@example.test | 2026-07-09 | APPROVED AFTER GRANT
  databricks | 2026-07-08 | f01@example.test | finance.ledger.gl_entries | modify |
a02@example.test | m01@example.test | 2026-07-07 | approved before grant
  redshift | 2026-07-15 | f02@example.test | ledger.gl_entries | insert | a01@example.test |
f02@example.test | 2026-07-14 | SELF-APPROVED
  snowflake | 2026-07-22 | f03@example.test | finance_db.ledger.gl_entries | insert,select,update |
a02@example.test | m02@example.test | 2026-07-20 | approved before grant
  databricks | 2026-07-28 | f08@example.test | finance.ledger.gl_entries | select |
a02@example.test | m02@example.test | 2026-07-27 | approved before grant
  databricks | 2026-08-03 | f04@example.test | finance.ledger.gl_entries | select |
a01@example.test | - | - | NO APPROVAL

== Leavers: sox_2_leavers  ** INCOMPLETE: see input checks **
period      : 2026-07-01 to 2026-09-30 (UTC, inclusive)
filters     : HR leavers with last day in the period who held access to a financial table on the
platform, or used one after the grace period; removal records credited to one departure; grace 1
day(s)
run at      : 2026-10-31 23:02:55 UTC
rows        : 6
sha256      : c8078fffc669059afbc547aa8b7807cd7971febd68c0d2e3665b67e16d0e02d86
  f08@example.test | 2026-07-10 | databricks | 2026-07-11 | 1 | 0 | 0 | removed in time
  f05@example.test | 2026-08-07 | databricks | 2026-08-08 | 1 | 0 | 0 | removed in time
  f05@example.test | 2026-08-07 | snowflake | - | - | 2 | 0 | NO REMOVAL RECORD
  f06@example.test | 2026-08-21 | redshift | 2026-08-30 | 9 | 1 | 0 | REMOVED LATE
  f07@example.test | 2026-09-04 | snowflake | 2026-09-05 | 1 | 0 | 1 | ACTIVE AFTER REMOVAL
  f08@example.test | 2026-09-18 | databricks | - | - | 0 | 0 | NO REMOVAL RECORD
[info] HR leavers in the period with no grant on a financial table and no use after the grace
period seen on snowflake, so not listed there: 3
[info] HR leavers in the period with no grant on a financial table and no use after the grace
period seen on databricks, so not listed there: 2
[info] HR leavers in the period with no grant on a financial table and no use after the grace
period seen on redshift, so not listed there: 4

== Segregation of duties: sox_3_segregation_of_duties  ** INCOMPLETE: see input checks **
period      : 2026-07-01 to 2026-09-30 (UTC, inclusive)
filters     : people (matched by email across platforms) who wrote to a financial table and also ran
a grant of access to one or approved access to one, in the period
run at      : 2026-10-31 23:02:55 UTC
rows        : 2
sha256      : c4fe6b8a046697a53d984f6d9caa3e24eabd360f4f2d123409ab52347d1fe43
  a02@example.test | 1 | databricks | 3 | 0 | WROTE AND GRANTED
  f02@example.test | 16 | redshift | 0 | 1 | WROTE AND APPROVED
[info] logins that wrote to a financial table or ran a grant and match no person (add them to

```

```
login_map): 1
  UNMATCHED LOGIN | redshift | rs_batch2 | 1 | 0

== Result
input checks : FAILED, the lists above are not complete
not checked  : none
follow-up    : 3 grant row(s), 4 leaver row(s), 3 segregation or login row(s) need an explanation
RESULT: EXCEPTIONS FOUND
```

Run 2: clean records

```
python3 sox_access_evidence.py --demo
```

```
input      : GENERATED records (not a company's data), seed 228, 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 PERIOD_START to AS_OF: 0
[ok ] scope names that match no grant, use or removal in the exports: 0
[ok ] Databricks audit events or Snowflake write lists the script could not read: 0
[ok ] Redshift GRANT or REVOKE on a financial table the script could not parse: 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 ] approval rows with a blank grantee, table, approver or date: 0
[ok ] approval rows whose date 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: 1
[info] grants in the period where no person who ran them was found: 0

== Access approved before it was granted: sox_1_access_grants
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive)
filters   : grants of ['all', 'all privileges', 'delete', 'insert', 'modify', 'ownership',
'select', 'update'] on the tables in FINANCIAL_TABLES, granted in the period; one row per person
and table
run at    : 2026-10-31 23:02:55 UTC
rows      : 6
sha256    : c391d1c2901596ca863849f2ba7f7681bf10389a209f75b4733e9662c6dce484
  snowflake | 2026-07-06 | f01@example.test | finance_db.ledger.gl_entries | select |
a01@example.test | m01@example.test | 2026-07-03 | approved before grant
  databricks | 2026-07-08 | f01@example.test | finance.ledger.gl_entries | modify |
a02@example.test | m01@example.test | 2026-07-07 | approved before grant
  redshift | 2026-07-15 | f02@example.test | ledger.gl_entries | insert | a01@example.test |
m02@example.test | 2026-07-14 | approved before grant
  snowflake | 2026-07-22 | f03@example.test | finance_db.ledger.gl_entries | insert,select,update |
a02@example.test | m02@example.test | 2026-07-20 | approved before grant
  databricks | 2026-07-28 | f08@example.test | finance.ledger.gl_entries | select |
a02@example.test | m02@example.test | 2026-07-27 | approved before grant
  databricks | 2026-08-03 | f04@example.test | finance.ledger.gl_entries | select |
a01@example.test | m01@example.test | 2026-08-01 | approved before grant

== Leavers: sox_2_leavers
period    : 2026-07-01 to 2026-09-30 (UTC, inclusive)
filters   : HR leavers with last day in the period who held access to a financial table on the
platform, or used one after the grace period; removal records credited to one departure; grace 1
```

```

day(s)
run at      : 2026-10-31 23:02:55 UTC
rows       : 6
sha256      : 0ce33859d8fc8b2872d8cbe4f8b341a7d142d6e365ae7d9a10a4757a3fac2345
  f08@example.test | 2026-07-10 | databricks | 2026-07-11 | 1 | 0 | 0 | removed in time
  f05@example.test | 2026-08-07 | databricks | 2026-08-08 | 1 | 0 | 0 | removed in time
  f05@example.test | 2026-08-07 | snowflake  | 2026-08-08 | 1 | 0 | 0 | removed in time
  f06@example.test | 2026-08-21 | redshift   | 2026-08-22 | 1 | 0 | 0 | removed in time
  f07@example.test | 2026-09-04 | snowflake  | 2026-09-05 | 1 | 0 | 0 | removed in time
  f08@example.test | 2026-09-18 | databricks | 2026-09-19 | 1 | 0 | 0 | removed in time
[info] HR leavers in the period with no grant on a financial table and no use after the grace
period seen on snowflake, so not listed there: 3
[info] HR leavers in the period with no grant on a financial table and no use after the grace
period seen on databricks, so not listed there: 2
[info] HR leavers in the period with no grant on a financial table and no use after the grace
period seen on redshift, so not listed there: 4

== Segregation of duties: sox_3_segregation_of_duties
period      : 2026-07-01 to 2026-09-30 (UTC, inclusive)
filters     : people (matched by email across platforms) who wrote to a financial table and also ran
a grant of access to one or approved access to one, in the period
run at      : 2026-10-31 23:02:55 UTC
rows        : 0
sha256      : e3b0c44298fc1c149afb4c8996fb92427ae41e4649b934ca495991b7852b855
[info] logins that wrote to a financial table or ran a grant and match no person (add them to
login_map): 0

== Result
input checks : all passed
not checked  : none
follow-up    : 0 grant row(s), 0 leaver row(s), 0 segregation or login row(s) need an explanation
RESULT: NOTHING TO FOLLOW UP

```

How to read the results

The heading first. A list headed INCOMPLETE was built from input that failed a check. Fix the input before you rely on any row.

List 1. Every row that is not approved before grant needs an answer. NO APPROVAL: find the approval; if it exists, add it to the log with its real date; if it does not, record that access was given without one. APPROVED AFTER GRANT: the approval is dated after the access; keep both dates as they are. SELF-APPROVED means the approver is the person who received the access.

List 2. NO REMOVAL RECORD means no removal was found for that departure on that platform. REMOVED LATE means the first removal record came more than the grace period after the last day. ACTIVE AFTER REMOVAL means the account used a financial table after the removal record. A person who was rehired has one row for each departure; for someone who later leaves again, use after the first removal is not counted against the first row.

List 3. Each person listed both wrote to a financial table and ran or approved access to one in the period. Each unmatched login needs a row in the login map before the list can be trusted.

The result line. NOTHING TO FOLLOW UP says only that the input passed its checks and no row needed an explanation. A platform printed as NOT CHECKED says nothing about that platform. Neither says a control is satisfied; the auditor decides.

Checklist

Done	Item
☐	FINANCIAL_TABLES names each platform's financial tables (no default; the script stops if it is unset)
☐	Each platform's export is complete from PERIOD_START to AS_OF, with no unexplained day without records
☐	Redshift: daily exports or an unload to S3 cover the whole span, not seven days
☐	login_map.csv has a row for every login that is not an email address
☐	All eleven input checks print [ok]; no list is marked INCOMPLETE
☐	List 1: every row that is not 'approved before grant' has an answer on file
☐	List 2: every leaver row that is not 'removed in time' has an answer, and a leaver with no grant on a platform is accounted for
☐	List 3: every person listed, and every unmatched login, has an answer
☐	PLATFORMS_EXPORTED lists every platform you use; any platform printed as NOT CHECKED is checked another way
☐	The three CSVs and their SHA-256 values are kept with the settings used, so a later run can be compared
☐	Your auditor has said which lists they want and what evidence they accept

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, and the layout of the `changes` field in a Databricks permission change is our assumption.
- **These three tests are our reading,** not a list from the SEC, the PCAOB or an auditor. We did not read PCAOB AS 2201. Your auditor decides what to test and what evidence to accept, and a different scope can need different lists.
- **A role that gains a privilege after a person received it** is not seen as a new grant on Snowflake, because the script looks at what the role held when the user was granted it.

- **Use is matched on the table's full name in query text**, with double quotes removed. A query that names the table by a shorter name after a USE statement is not matched.
- **An approval is matched to a person and a table, not to a privilege**. An approval for read access also covers a later grant of more privileges on the same table; check the privileges column of List 1 against what was approved.
- **Access that reaches a financial table by another route is not seen**: a role granted to another role on Snowflake; a grant on a schema or catalog on Databricks; ALL TABLES IN SCHEMA, a schema grant or group membership on Redshift. The script reads a role granted directly to a user, table-level updatePermissions events, and GRANT ... ON a named table.
- **Timestamps with no zone suffix are taken as UTC**. Snowflake exports follow the session time zone, so set it to UTC before you export.
- **Group and role principals** on Databricks and Redshift are treated as logins: they appear in list 1 under the group's name and match no approval unless your log uses the same name.
- **After a first removal, use is not counted for someone who later leaves again**. That use is taken to be the second stint, so a removal that did not take before the rehire is not flagged.
- **The approver's authority is not checked**, only that the approval exists, is dated, and is not the grantee. A grant approved by a person who had no right to approve passes this test.
- **A removal record is not proof of logout**. The script counts later use of the financial table as a second signal; a person can also read the data through a shared account or an extract.
- **Coverage**: only what each platform records, for as long as it keeps it. Redshift keeps seven days in the STL views unless you export them.
- **The empty-day check can fire on a quiet day**, and cannot see a day that has some records but not all.

Sources

- [15 U.S.C. 7262 \(SOX Section 404\)](#) (read 2026-10-31)
- [SEC Release 33-8810](#) (read 2026-10-31)
- [Snowflake: GRANTS_TO_USERS](#) (read 2026-10-31)
- [Snowflake: GRANTS_TO_ROLES](#) (read 2026-10-31)
- [Snowflake: QUERY_HISTORY](#) (read 2026-10-31)
- [Snowflake: ACCESS_HISTORY](#) (read 2026-10-31)
- [Databricks: audit log system table](#) (read 2026-10-31)
- [Databricks: lineage system tables](#) (read 2026-10-31)

- [Amazon Redshift: STL_UTILITYTEXT](#) (read 2026-10-31)
- [Amazon Redshift: SYS_USERLOG](#) (read 2026-10-31)
- [Amazon Redshift: SYS_QUERY_HISTORY](#) (read 2026-10-31)
- [Amazon Redshift: System tables and views reference](#) (read 2026-10-31)
- [Databricks: audit log reference](#) (read 2026-10-31)

See every access change in one place

Data Control Tower deploys into your own Azure subscription from a template your IT team can read before it runs. Every day it reads the records your data platforms already keep, and it never writes to them. It lists every access change and who made it, including access someone gave by hand. Leavers are checked against your HR file, across every platform you connect. No platform passwords stored.

The Governance Evidence Pack, in early access, turns those checks into evidence for SOX or HIPAA. For SOX it flags anyone who does both (changes financial data and approves access to it), using the same access records. We produce the evidence. Your auditor decides.

See what Data Control Tower reads, and why it never writes to your platforms.

[See Data Control Tower](#)