Home Exercise

Preface

1. The goal of this exercise is to understand how you think, work and solve problems
2. You can use pseudo-code to explain the design or python code
3. You can use AI tools but document your (successful …) prompts and be ready to explain the AI response
4. If something is not clear - send me a message/email
5. Please use shared folder (GDrive, git, OneDrive) to share the results (document and code)

Exercise

The project include 3 phases:

1. Scraping a public web site
2. Inserting the downloaded data to a DB
3. Building a BI site to extract meaningful insights from the data

The project environment:

1. Local (on prem) server (laptop)
2. Ubuntu 24.04
3. Python 3.12.3
4. Postgres DB

Phase 1 - Web Scrapping

1. The site includes ~5000 indexes of IDs and data for all months from 2010
2. For each ID, we can download a ZIP file with 1-10 CSV files
3. The CSV file name includes the ID, month, data table name
4. Each data table has different column names and types (e.g. name, cost, code)
5. We have total of 25 unique data table name
    - What are some of the potential issues for scrapping this web site and organising the data?
    - Suggest methods to overcome them? Explain and write a code sample

Phase 2 - Inserting the downloaded data to a DB

1. Once the CSV files are downloaded, the data needs to be inserted to the DB
    - Suggest a method for building and inserting the data
    - Note one major design issue in the process so far and suggest how to overcome it (hint: schema). Explain and write a code sample
2. We want to be able to have an audit trail of the data - from the web source all the way to the BI analysis. This is required for any journalistic and legal actions.
    - Suggest a method for building and maintaining such an audit trail. Explain and write a code sample
3. Some of the tables in the DB can be very large (10's of millions of rows). This may cause several issues with the DB management and the BI
    - What are the potential issues in this case?

    - Can you suggest methods to overcome the issues?
4. DB maintenance is critical for this project - data will continue to download and more BI tools created
    - Suggest the list of maintenance tools and routines for the project. For each suggestions, explain why and how to achieve it with as much automation as possible (hint: the project python code is handling all the DB operations)

Phase 3 - BI

1. The project uses metabase OSS BI tool
2. The BI site has multiple users with several functions - admin, creator, viewer
    - For metabase, suggest a method to ensure clear permissions for the entities in the BI for each of the functions
    - What are the limitations in metabase, if any, for achieving the required controls?
3. For better visibility and maintenance, we want to add server and DB **performance** dashboard to the BI
    - Suggest performance parameters for the server and DB
    - Suggest and show code sample for creating a performance dashboard

My ansowers

Phase 1
What are some of the potential issues for scrapping this web site and organising the data?

1. **Scale & orchestration** – 5000 IDs × 2010→today → huge task set; retries/backoff; resume without redoing work; strict idempotency.
2. **Discovery & availability** – inconsistent URL patterns, JS-rendered links, “no data for this month” pages, missing months.
3. **Network robustness** – timeouts, partial/truncated responses, 200-OK with error HTML; must detect and handle.
4. **ZIP safety** – corruption/CRC errors, zip-slip paths, nested dirs, unknown file counts, extreme expansion (zip bombs).
5. **Filename/metadata parsing** – drift (extra underscores/spaces), month formats (2010-1 vs 2010-01), table name collisions.
6. **CSV/encoding heterogeneity** – UTF-8 vs Windows-1255, BOM, ,/;/\t delimiters, quoting quirks, embedded newlines, RTL control chars.
7. **Schema drift across 25 tables** – columns/types/units change over years, duplicate/missing headers, inconsistent date/currency formats.
8. **Identity & duplicates** – true keys may be composite; reissued months; identical/near-identical files with tiny diffs.
9. **Storage & performance (laptop)** – disk footprint, inode pressure, directory hot-spots, OS file-descriptor limits.
10. **Data quality & anomalies** – out-of-range values, repeated header rows, mixed locale numerics, free-text contamination.
11. **Observability & provenance (technical)** – without a manifest/hashes/progress tracking, failures go silent and runs aren’t reproducible.

Suggest methods to overcome them? Explain and write a code sample

Scale & orchestration

- Generate tasks lazily (for each (id, yyyy-mm)); keep a **manifest** (SQLite) recording status (pending/ok/failed/skipped), attempts, error.
- **Idempotency**: skip tasks already ok with same hash; write downloads atomically (.part → rename).
- **Retries**: exponential backoff with jitter; cap attempts (e.g., 3).

Discovery & availability

- Build URLs deterministically from a template or read from an index page per ID/month.
- Do a **light probe** (HEAD or small GET) to detect “no data” pages and avoid full downloads.

Network robustness

- Timeouts, truncated bodies → stream to disk, verify size if Content-Length exists; verify magic bytes after write.
- Treat 200 OK HTML error pages as failures (magic bytes ≠ PK\x03\x04).

ZIP safety

- **CRC check** (ZipFile.testzip()), protect against **zip-slip** (no ../absolute paths), limit expansion size.
- Handle unknown internal layout (nested dirs, extra files).

Filename/metadata parsing

- Parse (id, month, table) from Content-Disposition or filename with strict regex; quarantine if ambiguous.

CSV/encoding heterogeneity

- Normalize CSVs to UTF-8 + , delimiter; sniff dialect; handle BOM; keep a raw copy.

Schema drift across 25 tables

- Use a simple **Schema Registry** (YAML/JSON per table) to map raw→canonical names; store schema_version with artifacts (even if Phase-2 uses it later).

Storage & laptop performance

- Folder sharding: raw/{table}/{id}/{yyyy}/{mm}/source.zip to avoid hot-spots.
- Limit parallelism (e.g., max_workers=4–8), cap file descriptors.

Observability & provenance

- Structured logs; manifest rows per stage (raw/extracted/normalized) with sha256, size, timestamps, error message.
- Small daily summary (counts) from the manifest.

Code :
In file phase1_scraping.py
How it works:

1. **Creates data/manifest.sqlite3 and folders: data/raw/ (original ZIPs) and data/normalized/ (UTF-8, comma CSVs).**
2. **Generates tasks for every id and month from Jan-2010 to today.**
3. **For each task, consults the manifest to skip completed items and cap retries.**
4. **Downloads the ZIP (or fabricates one in DRY_RUN) to .part, then renames atomically; on failure, retries with backoff.**
5. **Validates ZIP magic + CRC, computes sha256, and records sha256/path/status/attempts/error in the manifest.**
6. **Extracts CSVs in memory, sniffs delimiter, normalizes encoding (utf-8-sig → cp1255 → latin-1 → replace), and writes to data/normalized/{table}/{id}/{yyyy}/{mm}/….**
7. **Marks task ok (or failed with the error saved), logs a one-line summary, and continues with bounded concurrency.**
8. **To scrape the real site, set DRY_RUN=False and provide URL_TEMPLATE (e.g., https://host/path?id={id}&year={year}&month={month:02d}).**

Phase 2

Once the CSV files are downloaded, what method do we use to build and insert the data into Postgres?

Use a two-tier, fully generic pipeline that handles all (up to and beyond) 25 table types without hard-coding any columns:

- **Staging (schema-on-read):** Load every CSV row into staging.rows as **JSONB** with lineage (file_id, row_num, entity_id, yyyy, mm, table_name).

- **Canonical (typed, evolving):** For each discovered table_xx, keep a typed public.table_xx. On each load, **discover new columns** from staging, **ALTER TABLE** to add them, and record the choice in a **schema registry**.
- **Idempotency:** Register each CSV by **sha256** in lineage.files. If already seen → **skip** re-load.
- **Bulk ingest:** Use batched inserts (execute_values) for speed.
- **Audit-ready:** Canonical rows retain src_file_id, src_row_num, batch_id to trace back to the raw file and row.

Note one major design issue (hint: schema) and how to overcome it; include a code sample.

- **Issue: Schema drift** — columns appear/disappear/rename or change types across months/years.
- **Fix:** Keep **JSONB staging** (accept anything) and evolve **canonical** tables automatically using a **schema registry** that records chosen data types and versions.
  Code :
  In file phase2_load.py

We want to be able to have an audit trail of the data - from the web source all the way to the BI analysis. This is required for any journalistic and legal actions.

Suggest a method for building and maintaining such an audit trail. Explain and write a code sample

- **How:** The loader creates:
  - lineage.files (one per CSV) with file_sha256, rel_path, table_name, entity_id, yyyy, mm.
  - staging.rows (one per CSV row) with JSONB rec + FK to file.
  - Canonical public.table_xx rows with src_file_id, src_row_num, batch_id. This chain lets you trace any BI record to the exact CSV and ZIP (you can extend to store ZIP hash from Phase 1).

Sql query:

```
-- Find original CSV (and sha) for canonical records
SELECT c.*, f.rel_path, f.file_sha256, f.first_seen
FROM public.table_01 c
JOIN lineage.files f ON f.file_id = c.src_file_id
WHERE c.entity_id = '00042' AND c.yyyy = 2024 AND c.mm = 6
LIMIT 50;

-- Jump to the raw JSON record as it appeared on disk
SELECT r.row_num, r.rec
FROM public.table_01 c
JOIN staging.rows r ON (r.file_id = c.src_file_id AND r.row_num = c.src_row_num)
WHERE c.src_file_id = 123 AND c.src_row_num = 77;

-- See schema evolution history
SELECT * FROM lineage.schema_registry
```

```
WHERE table_name = 'table_01'
ORDER BY added_at;
```

Very large tables (10s of millions of rows) — what issues may appear and how do we overcome them?

- **Issues:** slow queries, big/bloated indexes, vacuum lag, heavy upserts, backup size/time, BI timeouts.
- **Mitigations:**
  - **Partitioning** by yyyy*100+mm (predictable from the folder path).
  - **Right indexes** (on (entity_id, yyyy, mm) + query-specific columns); drop unused ones.
  - **Materialized views** / **rollups** for BI; incremental refresh by month.
  - **Bulk ops** (COPY, batched inserts); **work_mem** tuning for ETL session.
  - Regular **VACUUM/ANALYZE/REINDEX**, periodic **CLUSTER** on access patterns.
  - **Retention** on staging.rows (e.g., keep 6–12 months), archive old partitions.

DB maintenance is critical — what tools/routines should we automate in Python and why?

- **Vacuum & Analyze:** keep planner stats fresh, reclaim dead tuples.
- VACUUM (VERBOSE, ANALYZE) public.table_01;
- **Partition orchestration:** create next-month partitions ahead of time; detach/archive old ones.
- -- One-time: convert to partitioned table
- ALTER TABLE public.table_01 PARTITION BY RANGE ((yyyy*100 + mm));
-
- -- Monthly (automate):
- CREATE TABLE IF NOT EXISTS public.table_01_2024_10
- PARTITION OF public.table_01
- FOR VALUES FROM (202410) TO (202411);
- **Bloat/Index health reports:** query pg_stat_user_tables, pg_stat_user_indexes and store results.
- **Backups:** nightly base backup or logical dump of lineage + public schemas.
- **DQ checks:** store assertions (e.g., non-negative numeric fields) and log pass/fail per batch.
- **Retry & Alerts:** track lineage.batches; notify on failed files; quarantine bad CSVs.

How do we expose BI-friendly views on top of Phase 2?

- Create **stable views or materialized views** per table_xx with the columns analysts use most; keep lineage columns for traceability.

Code sql query:

```
-- Example: a BI view with a few selected columns + lineage
CREATE OR REPLACE VIEW bi.table_01_latest AS
SELECT
  entity_id, yyyy, mm,
  col_a::text AS col_a,
```

```
  col_b::bigint AS col_b,
  src_file_id, src_row_num, batch_id
FROM public.table_01;

-- Example: monthly aggregate as a materialized view
CREATE SCHEMA IF NOT EXISTS bi;

CREATE MATERIALIZED VIEW IF NOT EXISTS bi.table_01_monthly AS
SELECT
  yyyy, mm,
  COUNT(*) AS rows_cnt
FROM public.table_01
GROUP BY yyyy, mm;

-- Refresh strategy (incremental by partition month is recommended)
REFRESH MATERIALIZED VIEW CONCURRENTLY bi.table_01_monthly;
```

Phase 3:

For Metabase, suggest a method to ensure clear permissions for the entities in the BI for each of the functions (admin, creator, viewer).
Use a layered model: Postgres roles → Metabase groups → Collection permissions.

- **Postgres: create a read-only DB user exposing only curated schemas/views; (optionally) a second ops-read user for audit/ops schemas.**
- **Metabase groups:**
  - **Admins: full admin, all connections, native SQL allowed, curate all Collections.**
  - **Creators: read-only connection, native SQL allowed, curate "Workbench/Curated" Collections.**
  - **Viewers: read-only connection, native SQL disabled, view curated Collections only; consider disabling downloads.**
- **Collections:**
  - **01 – Curated Dashboards → View (Viewers), Curate (Creators/Admins).**
  - **02 – Workbench → Curate (Creators/Admins), no access (Viewers).**
  - **03 – Ops & Lineage → Admins (and selected Creators).**
    This keeps raw/audit data away from Viewers, lets Creators build safely, and gives Admins full control.

What are the limitations in Metabase, if any, for achieving the required controls?

- **No row-level security in OSS.** *Workaround:* enforce **Postgres RLS** or expose restricted **views**, and bind Viewers to that connection.
- **No field-level masking.** *Workaround:* masked views (hash/null PII) and only expose those.
- **Limited versioning/approval workflow.** *Workaround:* store critical SQL in Git; adopt a **Workbench → Curated** promotion policy.
- **Downloads are group-level, not per-Collection.** *Workaround:* conservative defaults for Viewers and separate "public" Collections with stricter data.

- **Fine-grained data sandboxing & SSO features** require Enterprise. *Workaround:* handle in Postgres (roles/RLS) and keep Metabase permissions simple.

For better visibility and maintenance, suggest performance parameters for the server and DB.

- **Server (host):** CPU %, load (1/5/15m), memory used %, swap %, disk usage % per volume, disk IO/throughput, network rx/tx, process health (Metabase/DB up).
- **Postgres:** active/idle connections vs max_connections, long-running queries, blocking/blocked sessions, deadlocks, buffer cache hit %, temp files/bytes, WAL rate, replication lag (if any), index usage %, table/index scans, autovacuum/analyze activity, table/partition growth, approximate bloat.

Suggest and show code sample for creating a performance dashboard.

1. **Data sources:** Create Saved Questions on Postgres system views (pg_stat_activity, pg_stat_database, pg_stat_user_tables, pg_statio_*, autovacuum logs via views). For server metrics, ingest lightweight host stats periodically into a small bi_perf.host_metrics table (or import summaries from your existing monitoring).
2. **Curation:** Keep helper views and the host metrics table in a dedicated schema (e.g., bi_perf) exposed to the BI read-only connection.
3. **Questions:** KPIs (cache-hit %, active connections, long-running count), time-series (CPU, memory, IO, network, WAL), diagnostics (blocking pairs, top temp users, index usage, largest/fastest-growing tables).
4. **Dashboard:** "**Ops: Server & DB Health**" with KPI tiles on top, time-series charts in the middle, diagnostic tables below; add dashboard filters (time range, host).
5. **Access & alerts:** Place in **Ops & Lineage** Collection (Admins/selected Creators). Configure Metabase email/Slack **alerts** for thresholds (e.g., cache-hit <95%, connections >80% of max, long-running >N).