Salesforce Data Pipeline for Power BI – Workflow and Requirements

The pipeline is designed to extract large volumes of data from Salesforce using **Bulk API 2.0**, process it in an efficient way, and export the data as a **compressed Parquet file** for consumption in Power BI. It is tailored to run on a **local Windows laptop**, ensuring **stability**, **fault tolerance**, and **memory optimization**. Due to frequent changes in historical data in Salesforce, each pipeline run replaces previous outputs. Incremental updating allowed during the same pipeline run as a fault tolerance measure.

Key Requirements:

- **Data Source**: Salesforce (via Bulk API 2.0 using simple-salesforce).
- **Output Format**: Compressed **Parquet file**.
- **Processing Tools**: **Python**, **Dask**, **SQLite**, **PyArrow**.
- **Logging**: Centralized and verbose logging with **timestamps** and **error handling**.
- **Configuration**: YAML config for object list, SOQL queries, paths, thresholds, and retry strategies.

Pipeline Workflow with Detailed Steps, Retry Mechanisms, and Memory Optimizations

(a) Check Salesforce API Usage

- **Objective**: Check Disk Space and prevent exceeding Salesforce's daily API usage limits before starting data extraction.
- **Steps**:
    1. Check available disk space in the output directory (from YAML). Require at least 20 GB free; halt if insufficient.
    2. Query Salesforce's /limits endpoint using the **simple-salesforce** library to retrieve the current API usage stats.
    3. Display the current usage and calculate percentage (e.g., "18,000 of 20,000 API calls used – 90%").
    4. If usage is at least **80%**, warn and pause with **exponential backoff** (retry after increasing intervals, initial 60s, doubling per retry, max 3 attempts) to allow for retries without overloading Salesforce.
    5. If usage is at least **90%**, **halt** execution immediately to avoid hitting the API limit.
- **Error Handling**:
    1. Retry API calls up to 3 times (from YAML) with **exponential backoff** on rate limits or timeouts.
    - Log errors with timestamps (e.g., "Retrying due to API rate limit. Attempt 2 of 3").
- **Fault Tolerance**:
    - Use a **retry** mechanism for transient errors like timeouts or rate-limiting, ensuring the pipeline doesn't fail immediately.
- **Memory Optimization**:
    - Minimal memory impact, as it only involves a single API request.

(b) Validate SOQL Queries and Field Selection

- **Objective**: Ensure SOQL queries are valid and fields are available and queryable for extraction.
- **Steps**:

1. Load metadata for each object using the provided **build_catalog** function, including field names, types, and visibility.
2. Validate each field in the SOQL query against catalog:
    - **Ensure it's queryable** (e.g., SOQL Compatible is True), **not deprecated**, and **not compound** (i.e., complex fields like formula fields).
    - **Check visibility** and **exclude hidden fields**.
3. If any invalid fields are detected, log and exclude them from the query; raise an exception for critical fields (e.g., Id).
4. **Cache metadata** locally in SQLite database to avoid re-fetching for each run.

- **Error Handling**:
    - If metadata retrieval fails (e.g., timeout), retry the request up to **3 times** with **exponential backoff**.
    - Log invalid fields and missing metadata clearly (e.g., "Field 'customField' is invalid or deprecated").
- **Fault Tolerance**:
    - **Retry** on metadata retrieval errors, ensuring the process doesn't fail due to transient issues.
- **Memory Optimization**:
    - Cache metadata efficiently on disk after the first fetch to avoid re-querying Salesforce each time.

(c) Map Salesforce Field Types to SQLite and Parquet Types

- **Objective**: Create type mappings for SQLite and Parquet to ensure compatibility..
- **Steps**:
    1. Use the validated fields from the SOQL query and the field metadata to map Salesforce types to SQLite (e.g., string to TEXT, boolean to INTEGER, date to TEXT) and Parquet (e.g., string to string, date to timestamp[ns]).
    2. Auto-detect **nullable** fields (nillable in metadata) and assign NOT NULL constraints to non-nullable fields.
    3. Map decimal precision/scale to DECIMAL(.,.) in SQLite and to decimal128 in Parquet if needed.
    4. Handle geolocation (latitude, longitude) and address types by splitting into separate columns.
    5. Log any **unmapped field types** for review and allow manual override via YAML.
    6. Store the **field mapping** in SQLite database for auditing and a JSON schema for Parquet ingestion.
- **Error Handling**:
    - Log unmapped field types and raise a warning or error; halt if critical fields cannot be mapped to SQLite types.
    - Ensure **SQLite compatibility** to avoid data corruption or runtime errors during insertion.
- **Fault Tolerance**:
    - Default to TEXT for unmapped types to prevent failures, but **abort** the pipeline if essential fields are unmapped.
- **Memory Optimization**:

- Use dictionaries for mappings, stored in SQLite, which reduces memory overhead.

(d) Pull Data from Salesforce Using Bulk API 2.0

- **Objective**: Extract data from Salesforce using Bulk API 2.0 with **Primary Key Chunking** for stability and fault tolerance.
- **Steps**:
    1. Use the **Bulk API 2.0** using **simple_salesforce.bulk** to retrieve data in **chunks**, ensuring each chunk doesn't overwhelm memory.
    2. Use **Primary Key Chunking** (chunk size from YAML, e.g., 100,000 records) to break the data into smaller, manageable chunks (e.g., based on record ID ranges).
    3. Process results as CSV streams and ensure **data integrity** by validating each chunk before continuing with the extraction process.
    4. Optional, if feasible based on the requirements and configuration, parallelize extraction for multiple objects using concurrent.futures.ThreadPoolExecutor (max 2 workers to avoid laptop overload). If using ThreadPoolExecutor, guard Bulk calls with a global semaphore to stay under concurrent-API limits.
    5. Implement **exponential backoff** for retries in case of rate limits or connection failures (e.g., "Retrying Bulk API extraction. Attempt 2 of 3").
    6. Log the status of each extraction, including chunk ID, number of records extracted, latest LastModifiedDate, and any errors.
- **Error Handling**:
    - If data extraction fails (e.g., network issues, Salesforce API error), **retry** up to 3 times with **exponential backoff**.
    - Log errors related to rate limiting, connection errors, or incomplete batches (e.g., Chunk 3 for Account failed: timeout).
    - Post-job validation: compare Salesforce's numberRecordsProcessed to what CSV wrote.
- **Fault Tolerance**:
    - Wrap Bulk job creation and polling in a context manager that auto-aborts incomplete jobs on exception.
    - **Chunking** ensures stability by processing data in smaller units, and failed chunks can be retried without affecting other parts of the extraction.
- **Memory Optimization**:
    - **Stream CSV** results to process large batches of data directly without loading everything into memory at once.

(e) Write Data to CSV in Batches (On-Disk Streaming)

- **Objective**: Write the extracted data in **batches** to CSV files to minimize memory usage.
- **Steps**:
    1. Write data to CSV files following a naming convention: salesforce_{object_name}_{chunk_id}_{timestamp}.csv.
    2. **Auto-clean old CSV files** from previous pipeline runs (e.g., remove files older than 24 hours, configurable in YAML).
    3. Validate **schema consistency** (e.g., column names, data types match catalog) for each CSV chunk before writing.

- **Error Handling**:
    - Log failures related to file writes (e.g., insufficient disk space, permission errors) and **retry** the file write operation up to 3 times.
- **Fault Tolerance**:
    - Use **atomic operations** (tempfile module) for file writes to ensure that partial writes don't corrupt data.
- **Memory Optimization**:
    - **On-disk streaming**: Stream writes using pandas.to_csv(chunksize=10000) directly to disk in chunks, avoiding the need to load everything into memory at once.
    - Optional, only if necessary due to processing overhead, use **compression** (e.g., gzip) for CSV files to reduce disk space usage.

(f) Validate Column Schema Across CSV Chunks

- **Objective**: Ensure that all CSV chunks have consistent schemas for further processing.
- **Steps**:
    1. Use **Dask** to lazily read CSV files in chunks and **validate column consistency** across chunks through a building a master column set (e.g., read only the header row of each CSV, such as pd.read_csv(..., nrows=0)).
    2. Pass a Pandas DataFrame schema (meta) into dd.read_csv(..., meta=meta_df) so Dask enforces dtypes/column names before any partition is touched.
    3. If any columns are missing or mismatched, log the issue and **skip inconsistent chunks**. Attempt to fix minor issues (e.g., missing nullable columns).
- **Error Handling**:
    - If schema mismatches are detected, log the error and skip the problematic chunk.
    - Log at DEBUG level: "Chunk file X has columns [A,B,C] vs master [A,B,C,D] – added D as null".
    - Summary at INFO: "Validated 120 chunks: 115 OK, 5 fixed, 0 skipped".
    - **Abort** if more than 30% of chunks are invalid (configurable in YAML).
- **Fault Tolerance**:
    - **Skip inconsistent chunks** without failing the entire process, ensuring the pipeline can continue with valid chunks.
- **Memory Optimization**:
    - Use **Dask's lazy loading** for chunk processing, which ensures that only one chunk is loaded into memory at a time.

(g) Create SQLite Tables Dynamically

- **Objective**: Create SQLite tables based on the validated Salesforce schema.
- **Steps**:
    1. Auto-generate DDL (CREATE TABLE ...) based on the validated schema map, including precision/scale and NOT NULL constraints.
    2. Use CREATE TABLE IF NOT EXISTS to ensure tables are created only once, even if the pipeline is run multiple times.
    3. SQLite Performance PRAGMAs: PRAGMA journal_mode = WAL; PRAGMA synchronous = NORMAL; PRAGMA cache_size = 20000; to speed up bulk loads and reduces lock contention.

4. Implement **batch inserts** (batch size 10,000) using executemany() to avoid memory spikes.
5. Use **transactions** to wrap inserts and ensure data integrity.
6. Log table creation and any errors during insertion.

- **Error Handling**:
  - Log errors related to table creation (e.g., schema conflicts, constraint violations) and **retry** table creation/inserts up to 3 times.
  - Wrap inserts in a retry loop catching sqlite3.OperationalError: database is locked, backing off 1s between up to 3 attempts.
- **Fault Tolerance**:
  - **Transaction-based commits** ensure that data consistency is maintained in case of a failure e.g., Roll back transactions on failure)
- **Memory Optimization**:
  - Use **batch inserts** and **transactions** to reduce memory usage when inserting large datasets.

(h) Create Indexes for SQLite Tables

- **Objective**: Optimize query performance by creating indexes based on object field types.
- **Steps**:
  1. Dynamically create **indexes** for fields marked as **Is Key** or **Filterable** in the catalog.
  2. Log which indexes are created for traceability (e.g., Created index on Account.Id).
  3. **Skip indexing** for deprecated or hidden fields.
  4. Run ANALYZE after index creation so SQLite's query planner uses fresh statistics.
- **Error Handling**:
  - Log index build times (e.g., "Created index IX_Opportunity_1 on (AccountId,LastModifiedDate) in 2.4s").
  - Log any issues related to index creation (e.g., invalid field types) and skip problematic fields.
- **Fault Tolerance**:
  - **Skip index creation** for invalid fields but continue with valid indexes.
- **Memory Optimization**:
  - Creating indexes may require additional memory; therefore, indexes should be created only on essential fields to optimize performance.

(i) Perform SQL Joins Using Dask SQL

- **Objective**: Execute SQL joins on the SQLite tables.
- **Steps**:
  1. Load SQLite tables into **Dask DataFrames**.
  2. Use **Dask SQL** to register tables and perform joins defined in YAML (e.g., Opportunity.AccountId = Account.Id) across them.
  3. Validate and log any **SQL errors**.
- **Error Handling**:
  - Log any SQL errors related to invalid joins or syntax errors and **retry** the join operation up to 3 times on transient errors.
- **Fault Tolerance**:

    - If the join fails, log the error and continue with partial results, ensuring the pipeline does not fail entirely.
- **Memory Optimization**:
    - **Dask DataFrames** allow for distributed, out-of-core processing, which optimizes memory usage.

(j) Write Final Data to Compressed Parquet File

- **Objective**: Save the final joined data to a **Parquet** file with compression.
- **Steps**:
    1. Write the data in **chunks** using **pyarrow** and **snappy compression**.
    2. Ensure that the **schema** is enforced to prevent data inconsistencies.
- **Error Handling**:
    - Log **write errors** and **retry** up to 3 times.
- **Fault Tolerance**:
    - **Chunked writing** allows for partial writing, preventing memory overloads and failures due to large data sizes.
- **Memory Optimization**:
    - Use **on-disk streaming** (dask.dataframe.to_parquet)for writing large files chunk-by-chunk.

(k) Move Parquet File to Network Drive

- **Objective**: Transfer the Parquet file to a shared **on-prem network drive**.
- **Steps**:
    1. Append a **timestamp** to the filename for traceability.
    2. Copy file to network drive (path from YAML).
    3. Log the **transfer status** (e.g., bytes transferred and total duration: “Copied 4.2 GB to \\\\server in 68 s” if success or failure).
- **Error Handling**:
    - Implement **retry logic** (up to 3 times) for file transfer issues (e.g., network timeouts) and log errors (e.g., Network transfer failed: retrying).
- **Fault Tolerance**:
    - Retry **failed file transfers** to ensure the file reaches its destination and log for manual retry.
    - Wrap health-file write in its own try/except so a failure here never blocks the pipeline.
- **Memory Optimization**:
    - No specific memory concerns in this step. Minimal memory usage (file copy operation)

(l) Create Health Tracker File

- **Objective**: Generate a JSON **health tracker file** with key metrics.
- **Steps**:
    1. Track **rows processed**, **duration**, **errors**, **warnings**, **API usage**, **disk space used** and **status** for each run.
    2. Save as health_check_{timestamp}.json in output directory
    3. Store the summary for future review.
- **Error Handling**:

    - Log issues related to the health tracker file creation and **retry** up to 3 times.
- **Fault Tolerance**:
    - Allow pipeline to continue, even if the health tracker file fails.
- **Memory Optimization**:
    - Use efficient **file operations** to store summary data.

Developer Friendly Workflow Description
Salesforce Data Pipeline for Power BI – Workflow and Requirements

High level design decisions (summary)

- **Default staging & joins: SQLite** on local disk (safe for 16 GB RAM).
- **Optional staging & joins (edge case): Parquet + Dask + Dask-SQL** (for Scenario A or if laptop resources permit).
- **Extraction default:** sequential Bulk API 2.0 chunk pulls. PK Chunking attempted, but fallback to sequential if unsupported. Optionally allow up to **2 concurrent** bulk jobs (configurable).
- **Storage:** Avoid storing a full uncompressed CSV (70 GB). Stream results and write compressed Parquet partitions where possible. If CSV chunks are needed temporarily, write compressed CSV and auto-delete after conversion.
- **Schema enforcement:** Use a catalog (from build_catalog) + a YAML-configurable field mapping and meta for Dask/Parquet enforcement.
- **Fault tolerance & resume:** Maintain a job_tracker SQLite table (per-run & per-chunk statuses) so runs are resumable.
- **Logging:** Centralized, verbose, structured logs with run_id and chunk_id.
- **Power BI:** Final Parquet export must contain **all data** for import. If partitioning is used for storage, produce a consolidated single-file Parquet (configurable) or a single-file copy for Power BI.

Preflight & config (one-time per run)

1. **Load config YAML** (objects, SOQLs, chunk_size, pk_chunking, thresholds, temp/output dirs, retry settings, concurrency, parquet options).
2. **Check environment & resources**
    - Confirm Python packages are installed (simple-salesforce, pandas, pyarrow, dask[distributed], dask-sql (optional), tenacity, sqlalchemy).
    - Confirm output_dir and temp_dir exist and are writable.
    - Confirm TMP/TEMP and Dask spill directories point to the fastest local drive (prefer NVMe).
3. **Disk free space check**
    - Calculate minimum required disk: min_free_disk_gb from YAML (default 20 GB) — **override**: because uncompressed CSV ~70 GB, *do not* attempt to produce a full CSV. If free disk < config.min_free_disk_gb, abort.
    - If free disk < safe estimate for your chosen mode, abort and log.

Step (a) — Check Salesforce API usage & preflight estimates

1. **Query /limits** using simple-salesforce to fetch API usage and Bulk limits. Log api_used, api_total, percent used.
2. **Estimate rows & data size**
    - For each object, optionally run a SELECT COUNT() or a small sampling SOQL to estimate rows to extract. Use this to compute estimated output size. Log estimates.
3. **Policy on high API usage**
    - If API usage ≥ max_api_usage_pct_warn (default 80%): log WARNING and **throttle** (reduce concurrency, increase delay), *do not* outright pause unless configured.
    - If API usage ≥ max_api_usage_pct_halt (default 90%): abort run.
4. **Retries & backoff**
    - All API calls guarded with retry/backoff (configurable attempts & base_sleep seconds). Use tenacity or custom exponential backoff. Log each retry with attempt counts.

Step (b) — Validate SOQL & Build Catalog (metadata)

1. **Fetch or load cached catalog** (use build_catalog implementation). Catalog contains: object, field, type, nillable, queryable, filterable, picklist values, precision, length, deprecatedAndHidden, compound info, createable/updateable.
    - Cache catalogs in local SQLite (table metadata_catalog) with last_fetched timestamp and TTL from YAML. If TTL not expired, reuse.
2. **Validate SOQL fields against catalog**
    - For each object and its SOQL, validate that each field exists and queryable==True, not deprecated/hidden, and not a compound field unless expanded explicitly.
    - **Per your instruction:** do NOT notify owners on removed/renamed fields; instead, the build_catalog is integrated so fields are expected stable in a single run.
3. **Output:** a validated field list per object, and a meta dtype mapping (pandas/pyarrow) used downstream.

Step (c) — Map Salesforce types → pandas / Parquet / SQLite types

1. **Load mapping table** if present (configurable path). If absent, **auto-generate** mapping using rules:
    - string/text → pandas object → parquet string → sqlite TEXT
    - boolean → pandas bool (or Int8 with nullable) → parquet bool → sqlite INTEGER
    - date → pandas datetime64[ns] (or object if timezone issues) → parquet timestamp[ns] → sqlite TEXT (ISO)
    - datetime → pandas datetime64[ns, UTC] → parquet timestamp[ns] → sqlite TEXT
    - double/decimal → pandas Decimal or float64 (or object/string if high precision), parquet decimal128 if needed, sqlite NUMERIC or TEXT
    - id → string/TEXT
    - address/geolocation → expand to field__lat, field__lon, etc.
2. **Allow YAML overrides** on a per-field basis (explicit mapping).

3. **Produce a saved JSON schema** and pandas meta DataFrame for Dask read_csv / read_parquet dtype enforcement.

Step (d) — Pull data from Salesforce using Bulk API 2.0
Goals: stable extraction, minimal memory, resumable, safe for Windows laptop.

1. **Attempt PK Chunking**
    - If org supports PK Chunking and pk_chunking: true in YAML, try to start job with PK chunking. If unsupported (API error) -> fallback to sequential.
2. **Default: Sequential chunk pulls**
    - Query creation: Bulk API job (or query batches) that returns CSV/JSON results in chunks.
    - Chunk size from YAML (chunk_size; default 100k). For your machine, prefer 50k–200k depending on object size. Example defaults:
        - Scenario B (likely): chunk_size=200_000
        - Scenario A (edge): chunk_size=100_000 and limit concurrency to 1–2.
3. **Concurrency**
    - Default: sequential (1 worker). Configurable: max_concurrent_jobs up to 2. If concurrency used, apply a global semaphore so total concurrent API usage and job count stay within org limits.
4. **Stream to disk**
    - Stream the Bulk API results directly to disk (compressed CSV or direct Parquet partitions when possible) to avoid holding chunk in RAM.
    - **Naming convention:** salesforce__{object}__chunk_{chunk_id}__{run_id}.csv.gz or .parquet.part
5. **Integrity verification**
    - After each chunk is written: record chunk path, row count, file size, checksum (CRC32 or MD5) in job_tracker sqlite table.
    - If job metadata includes numberRecordsProcessed, compare and log mismatch.
6. **Retries**
    - Retry per-chunk extraction up to attempts with exponential backoff on transient errors.

Step (e) — Write data to CSV/Parquet in batches (on-disk streaming)

1. **Preferred:** if your extraction library can stream rows to Arrow table or write Parquet directly, write to Parquet partitions (fewer IO steps).
2. **Otherwise:** stream to compressed CSV (.csv.gz) to reduce disk usage. Immediately convert each CSV chunk to Parquet (see Step j) and then delete CSV chunk to free disk.
3. **Atomic writes:** write to temp file + atomic move/rename into final chunk filename. Use tempfile for safe partial writes.
4. **Windows specifics:** ensure newline/encoding uniformity (UTF-8). Exclude temp_dir from antivirus scanning if possible to avoid locks and slowdowns.
5. **Logging:** log start/end, rows written, chunk duration, path, and md5.

Step (f) — Validate column schema across chunks & merging strategy (answer to extra Q)

Objective: ensure consistent schema across all chunks before downstream join/write.

1. **Header-only pass**
    - For each chunk file, read header row only (e.g., pd.read_csv(path, nrows=0)) to collect the column set for that chunk.
    - Build master_column_set = union of all columns (or intersection depending on strictness — default: union).
2. **Detect mismatches**
    - If chunk missing columns in master set:
        - If missing columns are nillable per catalog: treat as present with all-null values (automatic fix).
        - If critical fields (e.g., Id) are missing: mark chunk as invalid and retry extraction.
3. **Dask merging (should we merge valid chunks?) — final answer: Yes, allowed — but only as a lazy Dask DataFrame.**
    - Build a meta pandas DataFrame from mapping (column dtypes).
    - Use dd.read_csv(list_of_valid_chunk_paths, dtype=meta_dtypes, assume_missing=True, blocksize=None, sample=0) to create a *lazy* Dask DataFrame that references files as partitions.
    - **Do NOT call .compute() or .persist() on entire dataset** unless machine resources and estimated size permit. Use map_partitions and streaming writes.
    - If a chunk is irreparably invalid, mark status=failed in job_tracker and optionally retry up to config attempts. Skip it if beyond retry limit and log.
4. **Abort policy**
    - If > max_invalid_chunk_pct (YAML default 30%) are invalid → abort run and surface errors.

Step (g) — Create staging (SQLite by default; Parquet + Dask optional)
Default (SQLite staging) — safe for 16 GB RAM and moderate data:

1. **Create dynamic SQLite tables** using mapping; DDL use CREATE TABLE IF NOT EXISTS. Use PRAGMA optimizations:
    - PRAGMA journal_mode = WAL
    - PRAGMA synchronous = NORMAL
    - PRAGMA cache_size = 20000
2. **Batch inserts**
    - Use executemany() with transactions, batch size default 10k rows. Wrap inserts and commit at intervals.
3. **Indexing**
    - Create indexes only on essential join/filter keys (IsKey/Filterable) and only after inserts to speed writes.
4. **Notes**
    - For very large objects (Scenario A), SQLite insertion can be slow; consider using Parquet/Dask (see below).

Optional (Parquet + Dask staging) — used for Scenario A

1. **From validated Dask DataFrame**, write partitioned Parquet directly to parquet_staging/ using to_parquet(..., write_metadata_file=True, engine='pyarrow', compression='snappy', partition_on=...).
2. **Use Dask SQL** to register Parquet tables for joins, or load partitions as Dask DataFrames for joins.
3. **Trade-offs**
    - Faster joins with Dask, out-of-core processing.
    - More complex resource tuning required; ensure Dask LocalCluster has spool-to-disk configured.

Step (h) — Indexes & optimization (SQLite or Dask)

1. **SQLite path**
    - After bulk insert, create indexes on columns used in joins/filters (Id, AccountId, LastModifiedDate).
    - Run ANALYZE to refresh stats.
2. **Dask path**
    - Repartition DataFrames by join-key (e.g., ddf = ddf.shuffle(on='Id')) to improve join performance. Use persist() only for small tables.

Step (i) — Perform SQL joins (two options covered)

Option 1 — SQLite (default):

1. Create read-only connections and perform SELECT ... FROM joined_tables using SQL that was defined in YAML.
2. Export result set in manageable chunks (e.g., chunked SELECT ... LIMIT ... OFFSET ...) into Parquet or to feed into final aggregation.
3. Pros: robust on limited RAM; familiar SQL; easier resume/debug.
4. Cons: may be slower for very large joins (Scenario A).

Option 2 — Dask + Dask-SQL (optional, recommended for Scenario A):

1. Register Dask DataFrames with Dask-SQL catalog and run SQL joins in Dask SQL.
2. Use broadcast joins if one table is small (persist dimension in memory), otherwise use shuffle-based joins with partitioning.
3. Write joined output back to Parquet in streaming partitions.

Step (j) — Write final data to compressed Parquet (and Power BI considerations)

1. **Primary output (recommended):** write the full dataset as a **partitioned Parquet dataset** (partition_on=LastModifiedYear, LastModifiedMonth) using pyarrow/Dask writer and snappy compression. This is efficient for storage and for incremental updates.
    - Example: dask_df.to_parquet(output_path, partition_on=['LastModifiedYear','LastModifiedMonth'], engine='pyarrow', compression='snappy', write_metadata_file=True)
2. **Power BI requirement: include all data**
    - Power BI will import the full dataset. Because some versions of Power BI and on-prem gateways do not always handle Hive-style partition folders the same way,

create an optional consolidated single-file Parquet for Power BI (configurable consolidate_for_powerbi: true).
    - Consolidation approach:
        - If dataset small enough to fit disk/memory, use pyarrow to merge row groups into one Parquet file.
        - Prefer consolidating only the recent N months (configurable), or consolidate the entire dataset if disk allows.
    - **Important:** consolidation is IO & CPU intensive — make it optional and controlled by YAML.
3. **Naming & manifest**
    - Always append run_id and timestamp to outputs. Create _SUCCESS file on completion.
4. **Retries & partial writes**
    - Write in partitions and move into final directory upon success to avoid partial consumption by downstream readers.
5. **Delete temporary CSV chunks** after successful conversion to Parquet to free disk.

Step (k) — Move Parquet (or consolidated Parquet) to network drive

1. **Copy** final Parquet file(s) to UNC path (if needed). For large files use chunked copy with progress and a final atomic rename (write to .tmp then rename).
2. **Retry logic** for network issues (3 attempts default).
3. **Finalize** by writing a health/ready file to the network destination (e.g., final_dataset__{run_id}__READY).

Step (l) — Health tracking & run metadata

1. **job_tracker (SQLite)** — required table columns (example):
    - run_id, object, chunk_id, file_path, rows, md5, status (queued, running, success, failed), attempts, start_ts, end_ts, error_msg
2. **health_check_{run_id}.json** — saved to output dir and contains:
    - run_id, start_ts, end_ts, rows_total, rows_per_object, errors, warnings, api_usage_start/end, disk_free_start/end, dask_config, sqlite_stats, final_files (paths & sizes), git_commit of pipeline code, packages versions.
3. **Logs**
    - Structured logs (JSON or key=value) stored in logs/. Include run_id, object, chunk_id in every message. Rotate logs (e.g., RotatingFileHandler).
4. **Resume semantics**
    - Use job_tracker statuses to resume: skip success chunks, retry failed chunks up to attempts. Developer should implement an idempotent run that can be resumed by run_id or by new run creating a new run_id.

Error handling, retries & failure policy (cross-cutting)

- **Retries**: config-driven attempts for API calls, chunk writes, SQLite inserts, network transfer.
- **Backoff**: exponential backoff with jitter (configurable base & max).

- **Abort thresholds**: abort run if too many chunks fail (> configurable %), or if API usage would exceed the halting threshold.
- **Safe cleanup**: on unrecoverable exceptions, close Bulk jobs, mark job_tracker entries and leave partial Parquet/CSV with clear filenames for manual inspection.

Tuning guidance for your hardware (16 GB RAM, ≥50 GB disk, 2–4 cores)

- **Chunk sizes**:
  - Scenario B (likely): chunk_size 100k–250k. With streaming and immediate conversion to Parquet, disk stays manageable.
  - Scenario A (edge): chunk_size 50k–100k; prefer sequential extraction; set max_concurrent_jobs=1 or 2 only if disk I/O permits.
- **Dask config (if used)**
  - LocalCluster(n_workers=2, threads_per_worker=2, memory_limit='6GB') — but be conservative: allow OS and other processes room.
  - Set temporary_directory & worker.spill to temp_dir.
- **SQLite**
  - Keep client connections single-threaded for inserts. Use PRAGMA tuning as above.
- **Parquet consolidation**
  - Avoid consolidating entire 70GB CSV equivalent dataset on this laptop. Consolidate only if final parquet expected << available disk. Otherwise produce partitioned dataset and separate consolidation step run on a more capable machine if needed.

Developer checklist (what needs to be implemented)

- build_catalog(sf, force_refresh=False) with SQLite caching & TTL (this will be your metadata source).
- Config loader & typed config validation.
- job_tracker SQLite table + helper functions mark_chunk_*.
- Bulk API extraction module:
  - PK chunking attempt + sequential fallback, streaming to disk, per-chunk verification.
- CSV → Parquet conversion module (streaming; atomic writes).
- Schema validation & Dask meta builder.
- Staging module (SQLite writes with batch-inserts + indexing) and optional Parquet/Dask staging.
- Join/transform module supporting both SQLite SQL and Dask SQL paths.
- Parquet write module with optional consolidation for Power BI.
- Health and logging module.