Course Resource Cleaner

For the Study Groups & Collaboration part of the Study Buddy Hub, I created a data cleaning table to generate SQL code for the resource table.

This program takes a CSV file of course-related links and

- Creates a clean CSV
- A MySQL insert code to populate the resource table

The script uses

- Pandas
- os
- re
- base64
- urllib.parse

The input dataset expects:

- Query column - the original query search string
- Title column - title or label returned by the search
- Url column - the raw url

In our Github, the use was for Study_Buddy_hub/Project phase 2/ data/course_resources.csv

The script loads the csv with:

df = pd.read_csv(INPUT_CSV, dtype=str, keep_default_na=False)

- dtype=str and keep_default_na=False treats everything as a string.
- Every empty field remains as an empty string

```
for col in ["title", "url"]:
    if col not in df.columns:
        raise ValueError("CSV must have columns: query,title,url")
```

- Makes sure CSV has these required columns
- Script fails if this structure isn't followed

The cleaning process

- Makes URLs usable
- Generates a readable title
- Infers file types for filtering
- strip_weird(s: str)
    - Replaces non-breaking spaces (eg. "\u00a0) and other special characters like ">" with regular spaces
    - Strips any leading or trailing spaces

    - Anything more than a single space is converted to a single space
- URL normalization problems
    - The raw data showed URLs that were wrapped in Bing Redirect links
    - URLs appear as base-64 like instead of http or https
- Problems fixed with **maybe_b64_url(s: str)**
    - A string starting with http or https is returned with no change
        - Otherwise searches for base64 using regex:
          re.finditer(r"[A-Za-z0-9+/_=-]{20,}", t, flags=re.I)
        - "-" converted to "+", "_" converted to "/"
    - Substrings with "aHR0" (base64 for http) used this:
      m = re.search(r"aHR0[0-9A-Za-z+/=_-]*", t, flags=re.I)
        - entire field is base-64 encoded URL
- Problems fixed with **unwrap_bing(u: str)**
    - Parses urls with bing using urlparse
    - If host ends with "bing.com" and starts with "/ck/"
        - Pulls the target from the u query parameter and unquotes
        - Passes it through the maybe_b64_url
- The main part of the cleaning uses **normalize_url(u: str)**
    - Calls strip_weird for whitespace
    - If URL contains bing, it unwraps the bing, else, it calls maybe_b64_url
    - Returns clean URL string
- **Title Problems**
    - Some of the title logins were lower quality such as "home" or "login", or even just domains
- Two functions were used to fix this
    - **title_from_url(u: str)**
        - Takes a base name and strips the extension such as "e623a.html", strips the extension part, and also replaces "-" or "_" with spaces
        - If the path is empty, the code falls back
    - **clean_title(raw_title: str, url: str)**
        - Uses strip_weird to strip the raw title
        - Removes "http://" or "https://"
        - Creates a set of titles we don't want as titles
            - BORING = {"in","login","event","topic","topics","home","index","default","search"}
        - If the title is empty, a domain, shorter than 3 chars, or in the boring set, then it will fall back to "url_fallback"
    - **url_fallback**
        - Uses non_empty path and formats as <host>/ where it is possible
        - Applies .title() and a length limits
        - Some example results of this:
            - Www.Britannica.com / Anthropology

        - Mexico.Internationaltrucks.com / Camiones De Carga
        - Www.Japan-Guide.com / E623A.Html
- **Filetype**
    - The function **filetype_from_url(u: str)**
        - Parses URL path and then finds the extension
        - Uses an extension map of all known extensions
        - The default fallback if none are found in the map is “LINK”

Construction of the data uses this code:

```
cdf = pd.DataFrame({
   "title":       pd.Series(titles, dtype="string"),
   "description": pd.Series([""] * raw_count, dtype="string"),
   "filetype":    pd.Series(sources.map(filetype_from_url).tolist(), dtype="string"),
   "source":      pd.Series(sources, dtype="string")
})
```

- Description is left blank for now, still thinking of a way to do this. I will be leaving it for the future
- Title, filetype, and source are clean

A problem is the problem of meaningless rows. Rows containing empty title and source rows were dropped with:

```
before_drop = len(cdf)
cdf = cdf[~(cdf["title"].str.strip().eq("") & cdf["source"].str.strip().eq(""))].reset_index(drop=True)
after_drop = len(cdf)
```

- Logs raw rows
- Drops empty rows

SQL Building

Code begins with:

```
lines = [
   "USE StudyBuddy;",
   f"SET @uploader_id := {DEFAULT_UPLOADER_ID};",
   "START TRANSACTION;"
]
```

For each cleaned row:

```
INSERT INTO Resource (uploader_id, title, description, filetype, source)
VALUES (@uploader_id, '<title>', '<description>', '<filetype>', '<source>');
```

Values are passed through sql_escape to duplicate any single quotes:

```
def sql_escape(s: str) -> str:
   return str(s).replace("'", "''") if s else ""
```

Seed user and uploader_id are also binded.

- The script has a header block which purpose is to make sure the rows are associated with a system user
    - **FOREIGN KEY (uploader_id) REFERENCES Users(user_id)**
- To handle this automatically, but also correctly, we have a block at the top of the SQL file

```
START TRANSACTION;

-- expose the actual user_id w/ LAST_INSERT_ID
INSERT INTO Users (user_id, email, password_hash, first_name, last_name)
VALUES (1001, 'resources@system.local', 'placeholder', 'System', 'Seeder')
ON DUPLICATE KEY UPDATE

-- Capture the real id (will be 1001 if we created it, or the existing row's id if email already existed)
SET @uploader_id := LAST_INSERT_ID();
```

- A dedicated system user with the id 1001
- Email = resources@system.local (can change later)
- Name: System Seeder

To ensure that the data cleaning was correct, the following checks were put in place:

- Confirmed the resource table in StudyBuddy matched the columns in the code
- Compared raw_count with rows in course_resources_cleaned.csv
- Decoded URLs
- Made clean readable titles
- Ran insert_resources.sql against MySQL schema

Procedures, Indexes, and Query Optimization Analysis

For the Study Groups & Collaboration section of the database, I determined the following as the most important features:

- Discovering public study groups for a course (by activity & size)
- Listing groups a user belongs to
- Viewing pending join requests
- Showing today's sessions and a user's upcoming sessions
- Loading chat history
- Suggested match partners
- Showing recent message requests
- Showing the most recent resources

The original queries I had created were functional but the following areas needed to be optimized:

- Table scans on tables that will most likely include higher traffic (Study_session, Join_request, etc.)
- Aggregation with Count and Max
- Unindexed ORDER BY columns
- Subqueries for match suggestions

I used the EXPLAIN ANALYZE before and after the changes of the queries to verify the improvements using test data. The two main ways to fix efficiency was:

- Target indexes aligned to each query's filter, join, and sort pattern.
- A Group_Summary table with is maintained with triggers to cache group-level aggregates

Every index I created followed the same principle

- Filter Columns -> Join columns -> Sort Columns

The following are the indexes I created:

- Study_Group table
    - **CREATE INDEX idx_group_course_priv ON Study_Group(course_id, is_private, group_id);**
        - Filters by course_id and is_private (public groups for a course)
        - Joins them by their group_id
    - **CREATE INDEX idx_group_course_group ON Study_Group(course_id, group_id);**
        - Uses a shared course logic
        - Creates a quick targeted path from course_id to group_id
- Group_Member table

    - CREATE INDEX idx_gm_user_group ON Group_Member(user_id, group_id);
        - Previous queries often start from the user:
            - “My groups”
            - Upcoming sessions for a user
        - This index helps the process by:
            - Doing a scan by user_id and have group_id for joins
            - Avoids any scans by group_id when we only want what group a user is in
- Study_Session
    - **CREATE INDEX idx_session_date_group_start ON Study_Session(session_date, group_id, start_time);**
        - Filters the session dates for today's date for a view of today's session
        - Joins on group_id
        - Orders by start_time
    - **CREATE INDEX idx_session_group_date_start ON Study_Session(group_id, session_date, start_time);**
        - For a specific user we want to see their upcoming session
        - We start from group_id from group_member
        - Filters session_date as greater than the current date and order it by start_time
- Join_Request
    - **CREATE INDEX idx_jr_status_expire_group ON Join_Request(join_status, expire_date, group_id, user_id, request_date);**
    - **CREATE INDEX idx_jr_status_reqdate_group ON Join_Request(join_status, request_date, group_id, user_id);**
        - The goal is for a dashboard to filter on join_status = ‘pending
            - This will either order by expire_date or request_date
        - By putting join_status first, and then the sort key second, the rows will already be read in the correct order, eliminating the in between separate file sorts
- Chat_Message
    - **CREATE INDEX idx_chat_group_time ON Chat_Message(group_id, sent_time);**
        - Chat views are filtered on group_id and then order by sent_time
        - This index:
            - Gets the latest messages with a reverse index scan
            - Supports keyset pagination with sent_time and message_id
- Message_Request
    - **CREATE INDEX idx_mr_target_created ON Message_Request(target_user_id, created_at, request_id, requester_user_id, course_id, request_status);**
        - The inbox queries filter on target_user_id and then order by created_at
        - Includes the columns and turns them into a covering index
        - Directly from the index page we can run the query

Group_Summary Table
So we don't have to recalculate group level aggregates every request, a group_summary solves the problem.

```
CREATE TABLE Group_Summary (
  group_id     INT NOT NULL,
  member_count INT NOT NULL DEFAULT 0,
  last_session DATE NULL,
  updated_at   DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
                        ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (group_id),
  CONSTRAINT fk_gs_group
    FOREIGN KEY (group_id) REFERENCES Study_Group(group_id)
    ON DELETE CASCADE
);
```

For the backfill of this query:

```
INSERT INTO Group_Summary (group_id, member_count, last_session)
SELECT
  g.group_id,
  COALESCE((
    SELECT COUNT(DISTINCT gm.user_id)
    FROM Group_Member gm
    WHERE gm.group_id = g.group_id
  ), 0) AS member_count,
  (
    SELECT MAX(s.session_date)
    FROM Study_Session s
    WHERE s.group_id = g.group_id
  ) AS last_session
FROM Study_Group g
ON DUPLICATE KEY UPDATE
  member_count = VALUES(member_count),
  last_session = VALUES(last_session);
```

- This is a one time backfill
- This loops through every study_group
- For each g.group_id, it computes the aggregates for that one group
- Select Count counts how many distinct users are in the group
- Uses distinct so the table wont double count in case of dupes
- Coalesce turns group members to 0 if the subquery is null

- The subquery finds the last max session_date for a group, if there are no sessions, the subquery returns null (no sessions
- Group_Summary has a primary key, so inserting a row with the same group_id would cause a dupe key error
    - On DUpliate Key Update solves this problem by updating values instead o causing errors

Triggers

These triggers are to maintain the Group_Summary:

- gm_after_insert

```
CREATE TRIGGER gm_after_insert
AFTER INSERT ON Group_Member
FOR EACH ROW
BEGIN
  INSERT INTO Group_Summary (group_id, member_count)
  VALUES (NEW.group_id, 1)
  ON DUPLICATE KEY UPDATE member_count = member_count + 1;
END//
```

- After insert into group member, this trigger will execute
- New.group_id is the group that the new membership belongs to
- Tries to insert a row, if the row doesnt exist it creates the row and sets member_count to 1
- If there is an existing row, it adds 1 to member_count for a group

- gm_after_delete

```
CREATE TRIGGER gm_after_delete
AFTER DELETE ON Group_Member
FOR EACH ROW
BEGIN
  UPDATE Group_Summary
  SET member_count = GREATEST(member_count - 1, 0)
  WHERE group_id = OLD.group_id;
END//
```

- After a row is deleted from Group_Member this is executed
- OLD.group_id is the group they were in
- For the group, take 1 away from member_count
- Use GREATEST such that a group count can never be negative in case of an accidental error

- session_after_insert

```
CREATE TRIGGER session_after_insert
AFTER INSERT ON Study_Session
FOR EACH ROW
```

```
BEGIN
  INSERT INTO Group_Summary (group_id, last_session)
  VALUES (NEW.group_id, NEW.session_date)
  ON DUPLICATE KEY UPDATE
    last_session = IF(last_session IS NULL OR NEW.session_date > last_session,
                NEW.session_date, last_session);
END//
```

- When a new study_session is created, this is executed
- Tries to insert (group_id, last_session
    - If there is no row:
        - Create a new row and set the last_session as NEW.session_date
- If there is an existing row
    - Uses IF
        - If last_session is null or the new session date is newer, update it
        - Else keep the existing last_session

EXPLAIN ANALYZE Results

Discover Public Groups for a Course **BEFORE**

```
EXPLAIN ANALYZE
SELECT g.group_id, g.group_name, g.max_members,
      COUNT(DISTINCT gm.user_id) AS members,
      MAX(s.session_date) AS last_session
FROM Study_Group g
LEFT JOIN Group_Member gm ON gm.group_id = g.group_id
LEFT JOIN Study_Session s ON s.group_id = g.group_id
WHERE g.course_id = 420
  AND g.is_private = FALSE
GROUP BY g.group_id, g.group_name, g.max_members
ORDER BY (last_session IS NULL) ASC, last_session DESC, members DESC
LIMIT 20;
```

- Used an index lookup on Study_Group g with course_id
- Nested loop left join to Group_Member and Study_Session
- Aggregate uses COUNT and MAX over joined rows
- Ordering by last_session and members

AFTER

```
SELECT g.group_id, g.group_name, g.max_members,
      gs.member_count AS members,
      gs.last_session
FROM Study_Group AS g
LEFT JOIN Group_Summary AS gs
  ON gs.group_id = g.group_id
WHERE g.course_id = 420
  AND g.is_private = FALSE
ORDER BY (gs.last_session IS NULL) ASC,
        gs.last_session DESC,
        gs.member_count DESC
LIMIT 20;
```

- Uses Group_Summary and an index
- Single row index lookup on gs using PRIMARY

- Ordering done over a small result set

Effect:
- Removed the COUNT and MAX from the main query
- Reduced rows processed
- The execution time dropped from .115 ms to .029 ms in the small dataset.
- Scales with the number of groups, not with the size of Group_Member and Study_Session

Pending Join Requests **before** index

```
SELECT g.group_name, jr.user_id, jr.request_date, jr.expire_date
FROM Join_Request jr
JOIN Study_Group g ON g.group_id = jr.group_id
WHERE jr.join_status = 'pending'
ORDER BY jr.expire_date;
```

- Table scans on join_request
- Filters for join_status pending
- Orders by expire_date
    - Less efficient due to scanning all join_request rows

- **AFTER** index on Join_Request
    - The query is the same, but now the index lookup allows only for walking over a subset of rows where join_status = ‘pending’
    - Already ordered by expire_date

Effect:
- EXPLAIN ANALYZE shows MySQL uses a covering index lookup
- Reduces both I/O and CPU as the table grows

Pending Requests + Membership **before** index

```
EXPLAIN ANALYZE
SELECT jr.request_id, jr.group_id, jr.user_id,
      CASE WHEN gm.user_id IS NULL THEN 'NOT_MEMBER' ELSE 'ALREADY_MEMBER'
END AS membership_state
FROM Join_Request jr
LEFT JOIN Group_Member gm
  ON gm.group_id = jr.group_id AND gm.user_id = jr.user_id
WHERE jr.join_status = 'pending'
ORDER BY jr.request_date DESC
LIMIT 50;
```

- Goes through table scan on join_request
- Left join check is not optimized
- Sorts by request_date DESC

After idx_jr_status_reqdate_group

- Only touches pending requests
- PK-based lookup on Group_Member
- Filters and orders

Effect:

- EXPLAIN ANALYZE execution time improved from ~.018 ms to ~.011 ms. Top 50 “pending” stays fast even when the total request table gets large.

Today’s Sessions **Before** index

```
SELECT g.group_name, s.location, s.session_date, s.start_time, s.end_time, s.notes
FROM Study_Session s
JOIN Study_Group g ON g.group_id = s.group_id
WHERE s.session_date = CURRENT_DATE()
ORDER BY g.group_name, s.start_time;
```

- Table scans study_session
- Filters s.session_date = CURRENT_DATE() applied in scan
- JOIN to study_group PK per row
- Sort on g.group_name, s.start_time after join

AFTER idx_session_date_group_start

```
SELECT g.group_name, s.location, s.session_date, s.start_time, s.end_time, s.notes
FROM Study_Session AS s
JOIN Study_Group AS g
  ON g.group_id = s.group_id
WHERE s.session_date = CURRENT_DATE()
ORDER BY g.group_name, s.start_time;
```

- Index makes it such that this is a date first search
- JOIN to Study_Group is still a PK lookup

Effect:

- EXPLAIN ANALYZE shows Execution time is similar from ~.045ms to ~.048ms
- Plan changes from full table scan to index-based range scan on today’s date
- Benefit most likely shows up as the Study_Session table grows.

Upcoming Sessions for a User **Before** index

```
SELECT g.group_name, s.session_date, s.start_time, s.location
FROM Group_Member gm
JOIN Study_Session s ON s.group_id = gm.group_id
JOIN Study_Group g ON g.group_id = gm.group_id
WHERE gm.user_id = 1001
  AND s.session_date >= CURRENT_DATE()
ORDER BY s.session_date, s.start_time
LIMIT 50;
```

- Starts from group_member, joins to study_group, joins to study_session
- Filter s.session_date >= current_date()
- Requires sort on date + time for result window

After idx_gm_user_group + idx_session_group_date_start

```
SELECT g.group_name, s.session_date, s.start_time, s.location
FROM Group_Member AS gm
JOIN Study_Session AS s
  ON s.group_id = gm.group_id
JOIN Study_Group AS g
  ON g.group_id = gm.group_id
WHERE gm.user_id = 1001
  AND s.session_date >= CURRENT_DATE()
ORDER BY s.session_date, s.start_time
LIMIT 50;
```

- Idx_gm_user_group finds user's groups, idx_session_group_date_start is a group first search that date filters applied in the index
- Results come out in session_date, start_time order

Effect:

- Execution time shows a similar time from ~.029 ms to ~.028 ms
- Scales with number of groups that a user is in x their future sessions

Chat History (Paginated vs Keyset) **Before**

```
EXPLAIN ANALYZE
SELECT c.message_id, c.user_id, c.content, c.sent_time
FROM Chat_Message c
WHERE c.group_id = 1
```

```
ORDER BY c.sent_time DESC
LIMIT 50 OFFSET 0;
```

- Order BY sent_time DESC requires sorting for the page
- OFFSET-based pagination would get more expensive as offset grows

After keyset pagination idx_chat_group_time

```
SELECT c.message_id, c.user_id, c.content, c.sent_time
FROM Chat_Message AS c
WHERE c.group_id = 1
ORDER BY c.sent_time DESC, c.message_id DESC
LIMIT 50;
```

- Uses index in reverse order
- Keyset_pattern gives deterministic ordering
- Next pages use a Where clause on sent_time, message_id instead of offset

EFFECT:

- Execution time improved from ~.007 ms to ~.004 ms for the first page in the small dataset
- Prevents pagination cost from growing linearly with page number as chat history gets longer

Match Suggestions **Before**

```
EXPLAIN ANALYZE
SELECT mp2.user_id, mp2.study_style, mp2.meeting_pref
FROM Match_Profile mp1
JOIN Match_Profile mp2
  ON mp2.user_id <> mp1.user_id
WHERE mp1.user_id = 1001
  AND (mp2.meeting_pref = mp1.meeting_pref OR mp2.study_style <> mp1.study_style)
  AND EXISTS (
      SELECT 1
      FROM Group_Member gm1
      JOIN Study_Group g1 ON g1.group_id = gm1.group_id
      WHERE gm1.user_id = mp1.user_id
        AND g1.course_id IN (
          SELECT g2.course_id
          FROM Group_Member gm2
          JOIN Study_Group g2 ON g2.group_id = gm2.group_id
          WHERE gm2.user_id = mp2.user_id
```

```
      )
  )
ORDER BY mp2.user_id
LIMIT 20;
```

- Nested EXISTS subqueries with multiple nested loop joins
- Filter on meeting_pref/study_style applied after scanning candidate mp2 rows
- Sorts on mp2/user_id

AFTER shared_peers & idx_gm_user_group/idx_group_course_group

```
WITH shared_peers AS (
  SELECT DISTINCT gm2.user_id
  FROM Group_Member gm1
  JOIN Study_Group  g1 ON g1.group_id = gm1.group_id
  JOIN Study_Group  g2 ON g2.course_id = g1.course_id
  JOIN Group_Member gm2 ON gm2.group_id = g2.group_id
  WHERE gm1.user_id = 1001 AND gm2.user_id <> 1001
)
SELECT mp2.user_id, mp2.study_style, mp2.meeting_pref
FROM Match_Profile mp1
JOIN Match_Profile mp2
  ON mp1.user_id = 1001
 AND mp2.user_id IN (SELECT user_id FROM shared_peers)
WHERE (mp2.meeting_pref = mp1.meeting_pref OR mp2.study_style <> mp1.study_style)
ORDER BY mp2.user_id
LIMIT 20;
```

- Shared Peers precomputes users who share a course with a targeted user once
- That reduced set of user_ids is all that is touched on match_profile
- Indexes help support the joins

Effect:

- EXPLAIN ANALYZE still shows a nested-loop joins but over a smaller candidate set
- Execution time improved from ~.063ms to ~.049ms

Message-request inbox **Before**

```
EXPLAIN ANALYZE
SELECT
  mr.request_id,
  mr.requester_user_id,
  mr.course_id,
  mr.request_status,
```

```
  mr.created_at
FROM Message_Request AS mr
WHERE mr.target_user_id = 1001
ORDER BY mr.created_at DESC
LIMIT 50;
```

- Index lookup on target_user_id only
- Sort on created_at required for ORDER BY

AFTER idx_mr_target_created

- SQL Code is still the same
- Uses covering index, scanned in reverse created_at order, matching ORDER BY
- All selected columns are in the index, so the table does not need to be touched

Effect:

- EXPLAIN ANALYZE shows “using index” with a reverse range scan
- Execution time improved from ~.017 ms to .007ms
- Scales well when a user has hundreds of message requests

Jacob Craig Phase 2 Contribution

- Designed and implemented all tables related to the Study Group & Collaboration section
- Created Test data for the Study_Groups & Collaboration section
- Built a Python data cleaning script that takes a CSV file for course resources and:
    - Removes duplicates
    - Cleans up whitespace and inconsistent patterns
    - Normalizes title formats
    - Outputs a CSV compatible with MySQL
    - Generates SQL Code for the resource table
        - Looking to improve upon this by eventually automatically inserting from the program into the database
- Created a full query workload with procedures and created indexes for the Study Groups & Collaboration section of the database
- Created automated triggers, procedures, and constraints for the Study Groups & Collaboration section of the database
    - Created a group_summary table to help with optimization
    - Designed three automated triggers to:
        - Increment member count on join
        - Decrement on removal
        - Update the latest session on new session creation
- Recorded working Resource Cleaner Program and working procedures