Creating a Local Usage Collection System: PySUSHI

CHRIS HERGERT, KAREN HARKER, SEPHRA BYRNE

UNT LIBRARIES

Overview

- What are COUNTER and SUSHI?
- What are the core requirements for a locally-built usage collection system?
  - HTTP requesting, error-handling, report parsing, storage
  - Optional: credential storage and querying, automatic request re-sending, and resource matching to holdings
- What's next:
  - SUSHI data collection process
  - Integrating flat files
  - Long-term storage considerations (SQL, NoSQL, schema)

SUSHI Data Collection Process

1. Generate request using credentials
   - Single platform - credentials may be requested from user or retrieved from storage
   - Many platforms – Better to save credentials in a database and retrieve the list of credentials needed for the requests to be made
2. Submit HTTP-GET request to the SUSHI server and collect server response
   - SUSHI server responds with either an XML/JSON load containing either an error message or the requested report
   - This is where errors need to be caught
3. Parse server response into a usable format
4. Save data

1. Generate the HTTP Request

- This is a GET request – easily automated with the Requests library in Python
  - R4 – request parameters are largely contained in encrypted header data
  - R5 – request parameters are appended to the URL address for the SUSHI endpoint, no headers required
- What is the session-scope intent?
  - One platform-report request per session?
  - Full sweeps of many platforms in one session?
- Recommended to store credentials in the same location as retrieved usage data – SQL DB or flat file

2. Send Request and Collect Response

- If a SUSHI server is backed up or cannot complete a request within 120 seconds, it must return a queueing message
  - If request is queued, the SUSHI client should re-attempt this request in two to five minutes.
- Submitted via HTTP-GET in both R4 and R5, but R5 requests can be tested in browser
- Response codes:
  - Standard HTTP status codes, primarily 2xx/4xx

3. Parse Server Response

- Algorithmically parse JSON or XML tree data into a data table
  - In Python, can use Pandas table for R-style dataframes as tables
  - Ignore the top 4-5 layers of the tree (4 for R4, 5 for R5)
- If writing a custom parser:
  1. Create a dataframe with desired column when report load is received
  2. Start at first item node
  3. Generate a blank array with length = table width
  4. Populate blank array with item node attributes, selecting desired attributes by tag.
  5. Add populated array to dataframe
  6. Move to next item node and repeat from step 3 until all item nodes have been parsed

4. Save Data in Long-Term Storage

- Need long-term storage appropriate to data format (SQL vs. NoSQL)
  - NoSQL if data still in tree form, SQL if data has been normalized/tabularized
- After compiling dataframes to be moved into long-term storage:
  - If moving to flat files, iterate through the files and save to a flat file via object methods to save computation
  - If moving to a database:
    1. initialize database connection and cursor (if in Python, PyODBC is recommended)
    2. Insert each dataframe into the appropriate storage table via one of the following:
       - Bulk insert query for each dataframe
       - Iterate through each dataframe and insert by-row into the appropriate table

Example: Handling SUSHI Request Sweep (PySUSHI workflow)

Get credentials from database

pandas

ODBC

- Exported from SQL Server database
- One row per platform-report to collect

Make HTTP-GET request

- Python
- Parse each report from XML/JSON into a DataFrame
- Log SUSHI messages in database

Collate data

pandas

- For each report type (TR, DR, JR1, DB1, etc), assemble all reports collected into a single DataFrame

Import into usage database

- Insert each compiled DataFrame into database table for that report type
- DataFrame schema = table schema

Integrating Flat Files

- Flat files are distinct data files that are generally able to be manipulated at the user level
  - Usually in a common file format like XLSX/CSV/JSON/TXT
  - Different formatting based on COUNTER COP
    - COP4 – 7 info rows above column headers
    - COP5 – 13 infor rows above column headers
- Upon reading file content into long-term storage, should the file be deleted or archived
  - Storage constraints are often the final deciding factor
- Logging is critical because files will pile up
  - Only handle a file (as a delete/archive) after the file's contents have been uploaded and logged
  - Multiple failures to upload a single file can cause a single file's logs to pile up

Long-term Storage: SQL vs NoSQL

- SQL:
  - PRO: more commonly used in academic structures, libraries probably already have MS suite available
  - PRO: Easy interfacing with most programming languages via existing modules, easily converts from existing flat files like CSV/XLSX
  - CON: Requires normalizing all SUSHI data into a tabular structure
- NoSQL:
  - PRO: Maintains the hierarchical structure of SUSHI data
  - CON: Unlikely to be available at most libraries unless Couchbase or MongoDB is in use
  - CON: Querying can be an unfamiliar process, even when SQL querying converters are available

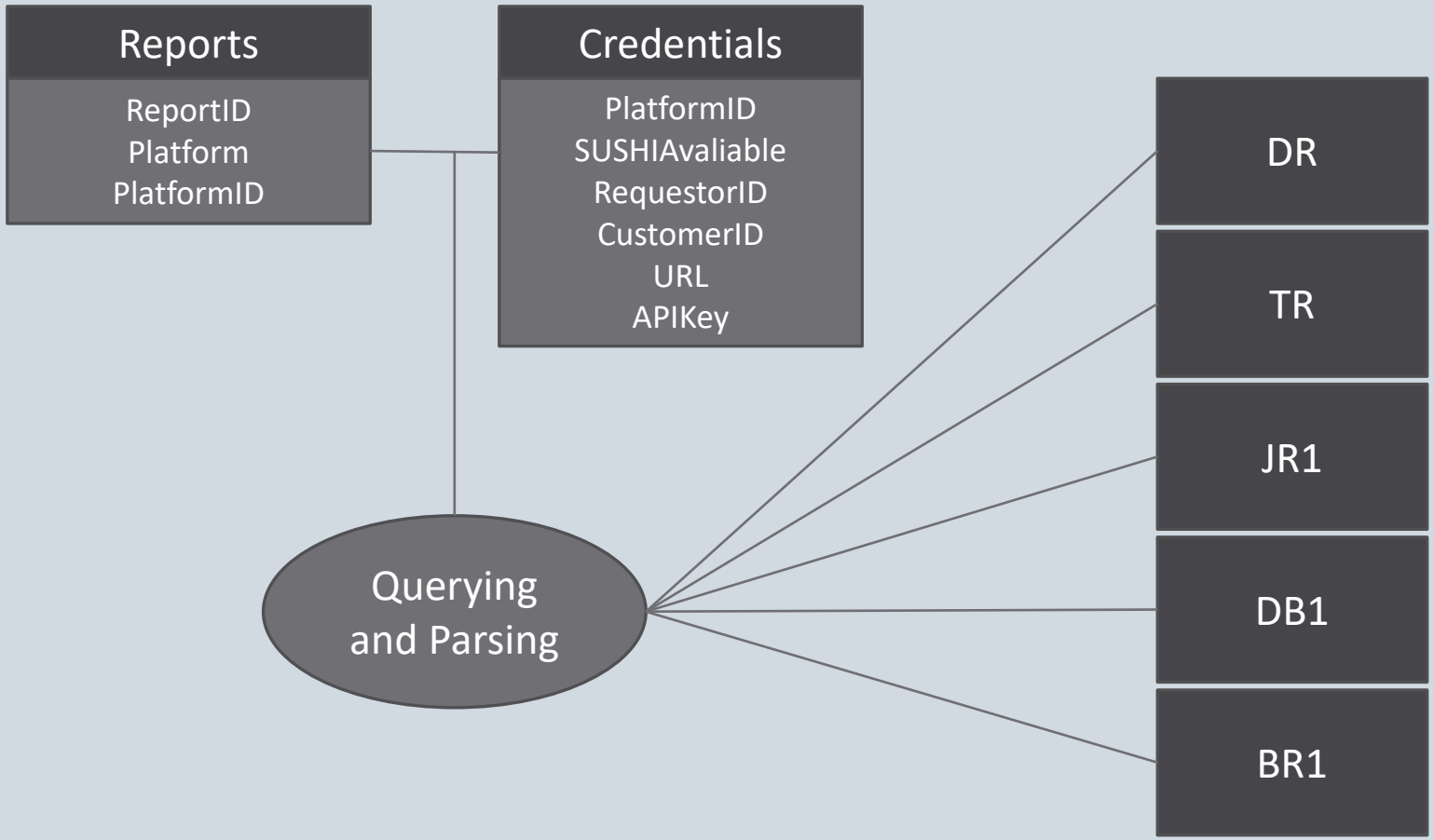
Max Table Size	DB Recommendation
250 rows	SQLite
10,000 rows	MS Access
100,000 rows	MySQL
Beyond 100k	MS SQL Server

Long-term Storage (Cont'd)

Reports
- Match platform identifiers with reports to collect for that platform

Credentials
- Credentials for each platform, with a platform identifier to link to Reports table

DR/TR/JR1/DB1/BR1
- Usage tables, each holding all reports of that type



Questions