The collection process for 365 schema evolution histories

Panos Vassiliadis, Univ. Ioannina, Greece, pvassil@cs.uoi.gr, March 2020

We detail the experimental method of the compilation of the largest collection of schema evolution histories known to this day, that took place throughout a large part of 2019 and early 2020.

1. Experimental method

Constructing the collection of candidate projects and DDL statements. To obtain data we have relied on publicly available, curated, data collections. Originally, we tried to work with GHTorrent[1] [4], a well-known GitHub mining tool. Still, we were unable to usefully produce a data set with it, and, in order to obtain data, we resorted to one of its querying platforms, Google Cloud BigQuery[2] in order to both obtain and process publicly available data sets. Specifically, among its many datasets[3], BigQuery provides ready (and in relational format, along with SQL facilities) the *Github Activity Data* dataset[4]. The GitHub Activity dataset is a 3TB+ dataset that contains a full snapshot of the content of more than 2.8 million open source GitHub repositories, that, apart from the contents, also include the commits of the monitored projects. The dataset is organized in a relational database format that allows its querying via an SQL interface. We have worked with the snapshot of the GitHub Activity dataset made on 2019-01-09, which was the one available at the time of our search. Specifically, we collected data during 2019-04-24 and 2019-04-25. We processed them in early May 2019.

We queried the contents table for all file descriptions ending to a '.sql' suffix and obtained a collection of SQL file descriptions (to which we refer to as *SQL-Collection*, hereafter) of 5,671,099 file descriptions (ranging from 700 to 833 MB in size, depending on the attributes one chooses to include). After counting, for each repository, the number of sql file descriptions included in it, we found that these 5.6M file descriptions were included in 133,029 repositories. Since the number of resulting files and repositories is extremely high to handle, we attempted to narrow it down, via a principled selection method.

1.1. Identification of Candidate Schema Files to Mine & Data Collection

A path to reduce the overwhelming amount of monitored file descriptions was to combine the SQL-collection that we obtained with another public dataset, available via BigQuery as the Libraries.io[5] dataset. Libraries.io[6] is an opensource community monitoring and gathering metadata for over 2.7M unique open source packages from 3 source code repositories, namely GitHub, Gitlab and BitBucket. The collection we have worked with was the one exported at 2018-12-22. The Libraries.io collection

[1] Georgios Gousios: The GHTorrent dataset and tool suite. MSR 2013, pp. 233-236, 2013
[2] https://cloud.google.com/bigquery/
[3] https://console.cloud.google.com/marketplace/browse?filter=solution-type:dataset
[4] https://console.cloud.google.com/marketplace/details/github/github-repos?filter=solution-type:dataset&id=46ee22ab-2ca4-4750-81a7-3ee0f0150dcb
[5] https://console.cloud.google.com/marketplace/details/libraries-io/librariesio?filter=solution-type:dataset&q=libraries.io&id=603312b3-22e4-40b6-b805-c5df5a8ce106
[6] https://libraries.io/about

offered us several metadata for each project, including whether the project was an original project or a fork, its number of stars, watchers, among several others[7].

Matching Github with Libraries.io on Repo_Name. On 2019-05-08, we combined the two datasets by matching the *repo_name* of projects in the two data sets. At the same time, we applied the extra filters that we look for *original* projects (that were not forks of other projects), with more than 0 zero stars, and, more than one contributor. The result was 6752 SQL file descriptions for 290 projects (again this is due that several projects included more than one SQL file).

The metadata for these SQL files where manually processed. Out of the 290 projects, 13 of them included a dazzling majority of 6108 out of the 6752 files. The individual values ranged from 1860 files to 16, with 3 of them having more than 1600 files. The reasons for this situation were (a) the cases of multi-file schema declaration, and, in particular, cases (b) incremental files and (c) the Cartesian product of variants. Due to the setup of our toolset for the automation of the history extraction, these 13 projects were excluded from our subsequent work.

For the remaining SQL files, we excluded all of those which include the terms 'test' or 'demo' or 'example' in the path. For all the cases where multiple vendors were supported, we chose MySQL as the DBMS to investigate (as the most popular DBMS in our collection). We ended up with:

- 97 projects that were ready for history extraction
- 32 projects with more than one SQL file that required further processing. 11 of these projects had a retrievable history and were added to our candidates.
- Eventually, we ended up with 108 projects out of which 106 were downloaded and two of them failed to produce an output (see History Extraction on the method)
- After post-processing where paths including test', 'sample', etc were removed, as well as a common project with the following attempt to collect data (matching over URL with Libraries.io), *we ended up with 100 projects to check*.

[7] A GitHub project can have (a) stars (i.e., someone considered the project interesting and pressed the 'Star' button), (b) forks (i.e., a user makes a copy of the project in his own 'space' to work independently on it), (c) collaborators (other users contributing to a project owned by someone else) and (c) watchers (users not collaborating, but registering to be notified for changes in the repository).

Matching Github with Libraries.io on URL. On 2019-05-25, a second attempt with the libraries.io was made. Now, instead of matching Libraries.io with GitHub repositories over *repo_name*, we matched them over the URL of the project. Again, we applied the extra filters that we look for original projects, with more than 0 zero stars, and, more than one contributor. The result was 8800 file descriptions. By filtering out the paths including the terms 'example, demo, test, migrat', and, we ended up with just 2670 files (meaning that 6130 (!) files, i.e. 70% of the collection belonged to the filtered-out category), for 437 projects.

We had to divide the vast number of projects in two sets:

- 221 projects that contained a single path for an SQL file. We removed 4 projects with paths including equivalent terms to 'sample', 'demo', etc, Thus, *we ended up with 217 projects*.
- 216 projects with more than one path for their SQL file. We manually inspected the paths and the on-line GitHub repositories of all these projects. We included the ones that could be reduced to a single DDL file with the table creation statements and omitted the ones with no CREATE TABLE statements, incremental upgrade statements, or in general, multi-file schema creation. *We ended up with 48 paths to check*.

Final consolidation. Surprisingly, the outcomes of two different extraction methods had only a single repository in common (which was extracted from the match over repo-name collection). In the rest of our deliberations, we refer to the union of these two sets as the *Lib-io data set*. The *history extraction*, i.e., the local cloning of these 100 + 217 + 48 = 365 repositories to our data collection server took place in 2019-05-26.

Terminology

Path-Filters. We removed the projects whose paths contained the exclusion keywords 'test', 'demo', 'example', 'sample', 'benchmark'.

History Extraction. We locally cloned the git repositories of the selected projects. Within the local git, we extracted the commits made to the DDL file, along with the respective dates. Thus, we obtained the history of the file.

Post-processing. Retention of schema histories with at least 2 versions.

A *post processing phase* took place over the retrieved repositories (2019-07-20), where:

- We removed 14 projects whose history extraction resulted in 0 versions (i.e., their file descriptions in did not match their actual. git that we downloaded).
- We removed the commits with empty files as well as the histories who .sql files did not contain "CREATE TABLE" statements. This involved 24 projects.
- *Out of the remaining 327 repositories, we isolated 132 projects with just one (1) version of the schema file, i.e., projects whose schema never changed. The number is striking: 132 out of 327 is* **a vast 40% of projects without any schema evolution(!)**

Eventually, we ended up *195 repositories* that were used for our subsequent analysis, and to which we refer as the **Schema_Evo_2019 data set**.

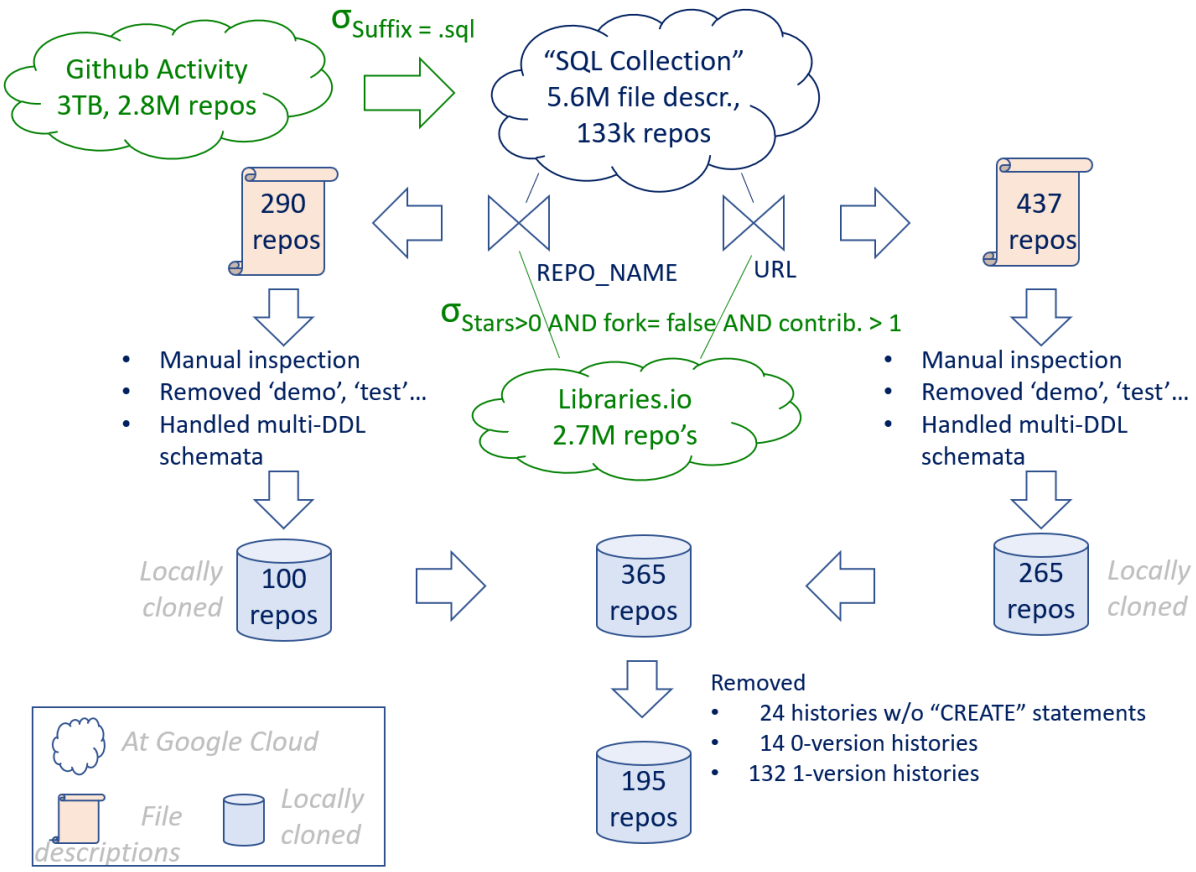

Figure 1. The extraction process for the Schema_evo_2019 data set.

Summary of filtering criteria. We can summarize the filtering criteria and the repositories that qualified of the different phases of the process as follows:

Phase 1 (Google Cloud BigQuery)	-	Selected .sql files and their metadata from the GitHub Activity Dataset
	-	Selected original repositories, with more than 0 stars and more than 1 contributor from the Library-io data set
	-	Selected projects whose name or url matched between the two data sets
Phase 2 (Elicitation of candidate repos)	-	Retention of single-file schema-DDL projects (either directly at the query result or after manual processing) …
	-	Retention of projects without the terms 'example, demo, test, migrat' in their path
	-	Chose MySQL or Postgres (in that order) in the case of more than one supported vendor
Phase 3 (post-processing)	-	Isolation of projects with 0 or 1 versions
	-	Isolation of projects with no CREATE TABLE statements

1.2. Testing

We tested our extraction scripts over the Opencart project, with the results of previous extraction made in 2016. We chose OpenCart as we already had its extracted history available, so, we could compare whether our current scripts were missing something. The testing was successful, as we

obtained a superset of commits against the previous extraction, with only one commit (in fact, the last one) missing out of the 412 commits computed in 2016. We manually tested the histories of the retrieved files against the number of commits reported at GitHub for the respective file, for a random sample of 50 cases. In all cases there was an exact match. There were also 14 cases where we had a failure to retrieve its history. We checked 7 of these projects and indeed, the path reported by the Github Activity dataset was actually not present at the git anymore, which further verified the robustness of our scripts.

1.3. Statistics Extraction

We used out tool, *Hecate*, to extract statistics from the collected schema histories. Hecate compares the sequence of committed versions pairwise, following a strict chronological order and exports the differences between subsequent commits, in terms of insertions and deletions of tables and attributes as well as data type and primary key changes. Hecate reports these measurements at the end of its processing, along with statistics (including version born/killed, the respective schema sizes, the number of updates undergone) for each individual table. Hecate has been independently tested in the past for the correctness and completeness of the reported changes. Statistics extraction was performed on 2019-07-31.

1.4. Project History Extraction

Since all projects had actually been downloaded locally, at our server, we took advantage of their presence to extract their entire history. We applied the git log command to the entire git repository for every one of the 195 projects with schema history. The output was processed to extract the start and end date of commits at the project (and, therefore, their duration), as well as the total amount of commits and files changed.

1.5. On the side: Problems Encountered and Decisions Taken

For the sake of completeness, we make a small detour here to report on problems that we encountered in our extraction process. This will hopefully shed some light to the rationale behind decisions we took in the identification of our target schemata, whose histories we eventually extracted. Some of the problems could be handled:

1. The set of .sql files contains files with only queries, triggers, stored procedures and other material that does not pertain to the creation of the logical schema via CREATE TABLE statements. In this case, these files were ignored (both at the selection and at the post-processing phase)
2. Several projects contain a different DDL file for different vendors, thus studying the essence of the evolution of the schema should be done on just one of them. We picked just one vendor, typically MySQL or Postgres, due to the popularity of these DBMSs in FoSS.
3. There we also several cases, where the SQL scripts included simple test, demo, or exemplary databases, which we omitted from further study, as our goal is to study the evolution of schemata that actually support the operation of the FoSS projects. During our manual inspections, we further excluded paths that included terms like ‘test’, ‘demo’, ‘example’, ‘sample’, ‘benchmark’.

There were also several cases where we could not process the project’s DDL:

1. *Files Void of Schema Declaration*. In several cases, there were no CREATE TABLE statements in the .sql files, but rather, queries triggers or stored procedures and the database creation was hosted in some source code. Retrieving the DDL statements from the source code was not possible with our toolset.

2. *Multi-File Schema Declaration*. One limitation that our method has is that it is very difficult to handle schema histories when the schema creation script is split in many files. Typically, this was done with one of the following patterns:
    a. *File-per-Table*. A popular pattern was that we found, a different file per table, located at the appropriate folder along with the respective source code for this table.
    b. *Incremental Files*. In another vein of multi-file schema creation, and possibly the most serious problem that we encountered, was that, in several cases, the developers had incremental, "upgrade" or "migration" DDL files to track the evolution of the schema.
    c. *Cartesian Product of Variants*. In several cases, the project included a huge number of different schemata. This was mostly cause by the Cartesian product of multiple vendors X different versions of the same schema for different languages (e.g., projects having a different schema for the combination of {English, French, …} by {mysql, postgres, mssql, …})

Why are these cases of multi-file schema creation difficult to handle? The problem with multiple files containing parts of the declaration of the schema is that the files can change independently. Then, for every commit made, we would have to extract or identify the appropriate variant of each of the individual files, artificially reconstruct the schema at this time point in a virtual file, and then, compare these virtual files for differences. For the case of incremental migration, one could possibly think that ordering the files by their first commit, would provide the history of the schema. However, again, we have the same problem of files changing independently; therefore, ordering by commit history is not particularly safe, as we again have to also track changes in all files and not simply just append the new ones to the previous schema. Another problem is that several such DDL files, instead of CREATE TABLE statements, contained 'ALTER TABLE' statements, whose semantics is not supported by our toolset yet. For all these reasons, projects abiding by the aforementioned patterns were omitted from our subsequent history extraction, as *our toolset was developed to handle the evolution of DDL files that are aligned with in the git modus operandi: a single file that evolves, whose versions are all kept (and can be retrieved) from the git*.

Finally, as already mentioned, the sheer number of schemata was too big for us to handle in terms of disk space and processing time, and thus, we focused on selecting a subset of the entire GitHub Collection that we extracted.

2. Threats to Validity

2.1. Scope

The scope of the study is defined as follows. *We are interested in the monitoring of the evolution of the logical-level relational schema for Free Open Source Software projects, hosted in GitHub.*

We want to stress that, in the context of our deliberations, we are not covering or generalizing to proprietary schemata outside the FoSS domain. We do not cover conceptual or physical schemata. We are also restricted in a relational schema and not XML, JSON, or another format.

2.2. External Validity

The external validity refers to the possibility of generalizing the findings of a study to a broader context. So, how representative are our studied projects of a broader population?

The SQL-Collection data set. The SQL-Collection data set includes the locations of schemata that are part of Free Open Source Projects (and not proprietary ones), available via GitHub. We applied the restriction that the respective file ends with a '.sql' suffix. Thus, *the SQL-Collection faithfully represents all the .sql files of GitHub*. Thus, it is (a) not just a representative, but the entire corpus of the respective files in GitHub, and, (b) in our opinion, a very good representative of open source software overall, as GitHub is the main public repository for FoSS software. It is possible that other suffixes, other than .sql, are used by developers. But to the extent that this would be a non-recommended practice, we believe that the projects ending up in our study are the appropriate ones to be monitored as significant projects.

The Lib-io data set. The Lib-io data set is a restricted version of the SQL-collection data set with the schemata whose repository path was monitored by the public Libraries.io data set. We applied the filter of more than one contributor, more than 0 stars and non-forking. We believe this to be a fairly broad scope for *original projects with a degree of recognized significance* (without implying, of course, that other projects are not significant).

Subsequent filtering. We believe that filtering out tests, examples and demos is not decreasing the value or validity of our approach. Although databases of these types have their value, monitoring their evolution, would not say much for the essence of a database supporting the regular operation of a software project.

For the case of multi-vendor support for schemata, we are also confident that our choice to select only one vendor is the appropriate one, especially since we are studying logical-level changes.

The most ambiguous decision -in fact, necessity- was the **restriction to single-file schema** declarations. We understand the design choice of several developers to split the schema of the database in different parts. We also understand the introduction of incremental, complementary DDL files for different versions of the same project. Thus, we would like to be clear that our approach is not covering such cases.

Overall assessment of the Schema_Evo_2019 data set. We believe that our elicited repositories and their extracted history give a fairly representative view of schema evolution in FoSS projects.

2.3. Reliability of Candidate Elicitation, History Extraction and Measurement

The extraction of the SQL-Collection data set is about posing a simple query, with a simply filter on the suffix of the path to the Github collection published via Google's Cloud BigQuery. To the best of our knowledge, due to the inability to pose such queries to Github per se, this is the best way to obtain data for projects hosted in Github. To this extent, the confidence that should be placed to our study is based on the quality of the export that Github has provided to Google Cloud. Similar considerations are held for the Libraries.io data set published in Google's Cloud BigQuery.

Also, we applied two relational joins between the two datasets (the first time over the name and the second time over the URL). Our exploratory study is dependent upon the join of the two involved data sets in Google Cloud BigQuery. It is possible that the join of the two data sets misses important projects, due to the misalignment of repository name or URL in the two joined datasets. However, it is impossible to check the extent to which this happens. In other words, as both datasets as well as their combination were produced via SQL queries in the BigQuery environment, we are certain for the precision of the result, having no false positives, but cannot guarantee the essential recall of the results that could have been compromised due to misrepresentation of a repository in one of the two datasets. In fact, we cannot think of ways that this could be checked.

We have tested both our repository extraction scripts and our diff-and-statistics tool, Hecate. Hecate's parser is not a full-blown SQL parser (thus missing some SQL statements); however, it has passed all the tests with MySQL schemata (manually inspected for completeness and correctness). The versions extractor from the locally obtained git repositories produced consistent results with previous extractions. For every round, we thoroughly checked aggregate statistics for all projects, revisited code and results and isolated "problematic" cases (e.g., the cases of 1-version histories, or histories without CREATE TABLES). Overall, although no one can exclude the possibility of bugs or omissions, we are quite confident with our extraction and measurement process and software tool suite.