GitSchemas: A Dataset for Automating Relational Data Preparation Tasks

Till Döhmen
Fraunhofer FIT
till.doehmen@fit.fraunhofer.de

Madelon Hulsebos
University of Amsterdam
m.hulsebos@uva.nl

Christian Beecks
Fraunhofer FIT & University of Hagen
christian.beecks@fit.fraunhofer.de

Sebastian Schelter
University of Amsterdam
s.schelter@uva.nl

***Abstract*—The preparation of relational data for machine learning (ML) has largely remained a manual, labor-intensive process, while automated machine learning has made great strides in recent years. Long-standing challenges, such as reliable foreign key detection still pose a major hurdle towards more automation of data integration and preparation tasks. We created a new dataset aimed at increasing the level of automation of data preparation tasks for relational data. The dataset, called GITSCHEMAS, consists of schema metadata for almost 50k real-world databases, collected from public GitHub repositories. To our knowledge, this is the largest dataset of such kind, containing approximately 300k table names, 2M column names including data types, and 100k real (not semantically inferred) foreign key relationships. In this paper, we describe how GITSCHEMAS was created, and provide key insights into the dataset. Furthermore, we show how GITSCHEMAS can be used to find relevant tables for data augmentation in an AutoML setting.**

***Index Terms*—database schemas, relational data, data preparation, machine learning**

I. INTRODUCTION

Data preparation and feature engineering are insufficiently automated, time-consuming, and knowledge-intensive tasks in the data science workflow. Data scientists or data engineers must first identify useful data sets for a given prediction problem and then transform the raw input data into a set of numerical features that have high predictive power for the task at hand (s. Fig. 1).

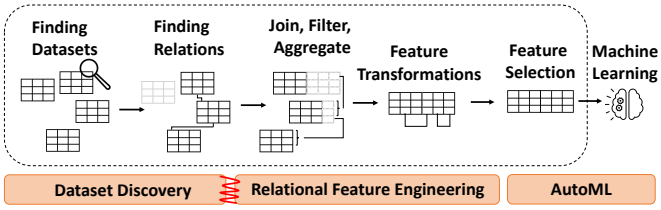

Fig. 1: A typical machine learning (ML) data preparation workflow of (1) finding relevant data; (2) identifying relationships between tables (primary and foreign keys); (3) joining, filtering, and aggregating data into a single feature table; (4) applying feature transformations; (5) and running feature selection. The process spans multiple problem areas (in orange). GITSCHEMAS supports work towards establishing a better integration between dataset discovery and relational feature engineering (red zig-zag).

Towards automating data preparation for ML. Modern AutoML systems such as AWS Auto-Gluon [1] and Google AutoML Tables [2] allow users to create quite powerful machine learning models with a single button click or a few lines of code, thereby simplifying the model creation process and making it more accessible to less technical users. These AutoML systems, however, usually expect a *single table as input* and do not assist users with integrating and preparing data from multiple sources.

Relational feature engineering frameworks like featuretools [3] automatically extract predictive features from multiple tables by joining and aggregating (resolving 1-n and n-n relationships) data based on predefined relationships. However, discovering relevant tables and their relationships in schema-less data stores can in itself be a tedious task prone to errors and oversights. Hence, in many practical scenarios, e.g., when relevant data is spread across different tables in an enterprise data lake, or different CSV files in an open data portal, or different feature groups in a feature store, manual data discovery is still a major hurdle to one-button-click relational data preparation for ML.

Data discovery systems such as *Aurum* [4] or *D3L* [5] are built to navigate schema-less data and find related tables, but are geared towards human-guided usage where incorrectly related tables can easily be weaved out by the user. Integrating dataset discovery with automated relational feature engineering solutions, to automatically augment training data with information from other tables, is particularly challenging. Relations between tables that are falsely assumed to be correct by the dataset discovery system have a direct negative impact on the resulting ML model's quality. As we will show in section IV, the GITSCHEMAS corpus is useful for making the results of such an end-to-end system more reliable.

The case for GITSCHEMAS. To improve the results of our data augmentation pipeline (s. section IV) the core idea is to learn from real foreign key relationships in relational database schemas that contain foreign key relationships between columns and tables. The largest public relational database corpus we could find was the CTU Prague Relational Learning Repository [6], which includes 83 database schemas. The second largest was the Public BI Benchmark Repository [7] with 43 datasets, inspired by Vogelsgesang et al.'s [8] paper on the need for real-world benchmarks. Both

appear too small for our purposes.

Other data sources are also not very suitable. Tabular data collections such as WebTables [9], GitTables [10] or Kaggle [11] are rich sources for individual tables, but do not contain original information about the relationships between tables. Large knowledge graphs such as DBPedia [12] contain curated relationships between entities. However, these are mostly natural language terms that do not commonly appear in database schemas. For example, the relationship "Person"→"Place" would more likely be modeled as "person_id"→"place_id" in a database schema.

As noted in Shah et al.'s work [13] towards automatic feature type detection, "It is almost impossible for researchers to get access to large numbers of truly "in-the-wild" data from enterprises and other organizations". However, do we necessarily need the data content to "learn" something from real data to make automated data preparation more efficient and reliable? Inspired by the seminal work of Hulsebos et al. on GitTables [10], and Yan and He's work on Auto-Suggest [14], we utilize public code repositories to extract a large amount of relational schema metadata from `CREATE TABLE` statements in *SQL* scripts, resulting in GitSchemas.

Broader utility of GitSchemas. We believe GitSchemas can facilitate use-cases beyond relational data integration as well. For example, Shah et al.'s [13] work on feature type detection aims at separating, e.g., "id" columns from numeric columns to inform automated feature engineering pipelines. The benefit of GitSchemas being created from SQL schema definitions is that it distinguishes between, e.g., `serial` and `integer` types (s. Figure 2) for the "id" column. GitSchemas is rich of such more granular data types which can be used to inform automated feature engineering pipelines.

A related example is the data validation system Deequ, which uses machine learning to infer from the column name which quality constraints should apply to the column (e.g. "id" columns should have constraints *isUnique*) [15]. We find that schemas in SQL files frequently contain such constraints.

Another use case could be header detection in CSV files. Since the schema in CSV files is underdefined, correctly identifying header rows in heterogeneous CSV files from, e.g. Open Government Data Portals, can be challenging [16]. With a corpus like GitSchemas, which contains a large number of known column names used in the wild, one could in the simplest case identify the header row based on whether it has above average hits in the corpus.

In this paper, we first describe the collection process of GitSchemas in Section II. Then, we provide an analysis of the dataset, as well as information about licenses and availability in Section III. Finally, we demonstrate how GitSchemas can be leveraged for automated relational data augmentation in Section IV. In summary, we make the following contributions:

- A process for extracting schema metadata from SQL files.
- A schema metadata dataset and analysis thereof.
- An experiment showing how GitSchemas improves data augmentation methods in ML pipelines.

II. Data Collection

In this section, we describe the collection process of GitSchemas. We extract raw SQL files from GitHub, and parse them in order to extract structured schema metadata. Although some of the SQL scripts contain `INSERT INTO` statements (i.e. potential table contents), we focus on extracting metadata such as table names, column names, and foreign key column names. Figure 2 shows an overview of the extraction process, which starts with a crawled SQL script (see GitHub crawler paragraph) that is transformed with a SQL parser into an Abstract Syntax Tree (AST) from which the schema metadata can be easily extracted (see SQL parser paragraph). The source code of our data collection pipeline is publicly available at https://github.com/tdoehmen/gitschemas.

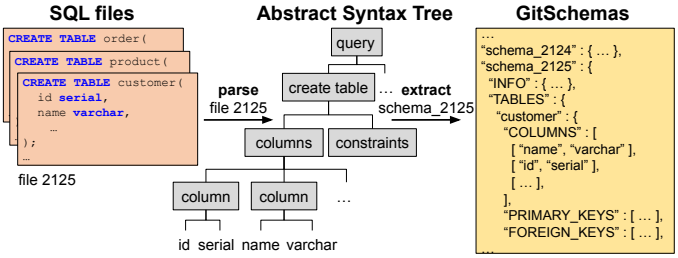

Fig. 2: The data extraction process. We extract SQL files indicating table schemas from GitHub, parse them into an Abstract Syntax Tree, and extract relevant schema metadata.

GitHub crawler. The GitHub SQL crawler is built based on the GitHub Search API [17]. It uses the API to search for all public GitHub repositories that contain SQL code with the expressions `CREATE TABLE` and `FOREIGN KEY`. With this, we want to reduce the set of all SQL scripts as much as possible to those that are used to create a reasonably complex schema (with at least one foreign key reference). The SQL crawler performs the following three steps:

1) Crawl a list of URLs via the GitHub API.
2) Download files based on the list of URLs.
3) Deduplicate the files based on their SHA256 hash.

As of December 2021, the GitHub search index contains about 7.8M SQL files, of which about 800k contain the selected keywords. Of these, approximately 700k files could be successfully retrieved. To filter out duplicated files from forks and clones, the files were deduplicated using a file hash, which reduced the number to approx. 370k.

SQL files range in size from 0.1kB to 400kB, with an average file size of 20kB. GitHub does not index files larger than 400kB for the search API. The files contain a total of 110M lines of code (LOC), including comments and blank lines, with an average 300 LOC per file.

SQL parser. To facilitate downstream use cases, we aimed at extracting structured schema information from the crawled SQL files. This includes table names, column names, primary keys, and foreign keys including their reference table and their reference column names. We deemed this information to be

most easily and reliably extractable from an abstract syntax tree (AST) of the `CREATE TABLE` queries inside of the crawled SQL files. Unsurprisingly, a manual sampling showed that the crawled SQL files have different SQL dialects, contain comments, and are even partially incomplete and/or have syntactical errors. We therefore tested different SQL parsers on a random subset of the data, including *mysqlparse*. [18], *pglast* [19], and *sqlparse* [20]. *mysqlparse* could only parse <5% of SQL files successfully, whereas *pglast* could successfully parse and create an abstract syntax tree (AST) for approx. 16% of the files. This eventually allowed us to extract 61k ASTs from the 373k SQL files in total. Given the ASTs, we identified `CREATE TABLE` statements and extracted all appropriate metadata. By choosing a production-grade SQL parser, the extraction process is very reliable and leads to high-quality data.

We think that the retrieval rate can still be improved though. The non-validating SQL parser *sqlparse* was able to successfully read, tokenize and lexicalize >95% of all SQL files. But with this tool, the creation of the AST is entirely up to the user, and the subtle differences in SQL syntax between different databases (e.g. the way comments are marked, or the way control sequences are escaped) make it difficult to create a correct AST without knowing the type and version of the target database, which is the case for our crawled SQL files. Reliably extracting schema information from heterogeneous SQL files appears to be an interesting and challenging problem that we could not exhaustively explore, yet. A framework such as *Apache Calcite* [21], which contains approx. 35 different SQL dialects, could be a promising starting point for future work on the parsing problem.

In summary, we established a process for retrieving schema-defining SQL files from GitHub and accurately extracting schema metadata from them.

III. Dataset

This section describes GitSchemas. We explain the schema of the dataset and present an analysis illustrating the scale and coverage of the dataset. Finally, we discuss accessibility and license restrictions.

Schema metadata. The schema data extracted as described in Section II is stored as a JSON file. This representation is suitable because the schema data is considerably hierarchical. The final JSON file consists of a list of numbered and named database schemas as depicted on the right side of Figure 2. Each schema has an info section that contains the URL of the SQL file, the filename, repository name, file size, and the license. Each schema also has a named list of tables contained in the schema. Each table has a name attribute and contains a list of columns (column name and data type), a list of primary keys, and a list of foreign keys consisting of a foreign key column, reference table name, and reference columns.

Analysis. GitSchemas currently contains a total of 49k database schemas. Of the 61k schemas originally extracted from the SQL files, approximately 20% were identified as exact duplicates and removed from the dataset. Each database schema has on average 6.6 tables and each table has on average 6.3 columns. The exact count of all entities is shown in Table I. This table contains an additional column showing the number of entities extracted from code published under licenses that permit redistribution.

Entity	Full Dataset	Permissive Licenses
schemas	49,146	6,642
tables	323,953 (114,926 unique)	51,594 (21,255 unique)
columns	2,054,026 (303,443 unique)	363,420 (55,157 unique)
primary keys	248,187 (31,599 unique)	36,620 (5,053 unique)
foreign keys	142,421 (31,041 unique)	24,380 (5,772 unique)

TABLE I: Entity counts in GitSchemas.

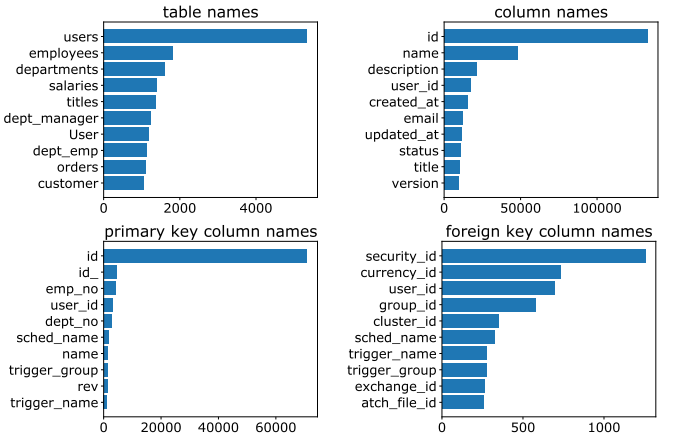

Fig. 3: Most frequent terms in GitSchemas (Full Dataset) illustrating its resemblance to enterprise database schemas.

Figure 3 shows the most frequently appearing terms in different attributes. One can see that classic database tables like "users", "employees", "customers" are very common. Similarly for the most common column names like "id", "name", and "description". Not surprisingly, "id" is the most common primary key, which is the same for foreign keys. The distribution of names in all cases has a very long tail (a high number of unique/low-frequency terms). Table II illustrates the depth of GitSchemas by the high diversity in column names starting with the prefix "user_". The diversity of column names in a random excerpt illustrates the breadth of domains covered.

beginning with 'user_'	user_address_id, user_agent_fk, user_agent_id, user_answer_id, user_app_id, user_associated, user_attendance_id, user_auth_id, user_authentication
random sample	id_title, role_ref_id, reg_no, partnerid, rent_book_no, teamId, n_orden_pag, id_apartment, history_id

TABLE II: Two samples of foreign key columns occurring in GitSchemas (Full Datset). The one beginning with "user_" shows that the dataset is rich in nuances, and the random sample gives an impression of the breadth of domains it covers.

60% of all tables have a primary key, of which 77% are simple keys and the rest are composite primary keys. 27% of all tables have a foreign key relationship to at least one

other table, 44% of them to more than one table. Foreign key relationships are established via a simple (non-compound) key in 95% of the cases.

Download and Licenses. 80% of the files we crawled from GitHub were published without license information. Overall, we are only able to ensure for about 13% of the files that they were published under a permissive MIT or Apache 2.0 license. This is the subset of files that we make publicly available. The extent to which derivatives of the other sources can be made publically available has yet to be determined. However, we make the full dataset available for reproduction upon request, and furthermore, anyone is free to reproduce a full dataset locally using the scripts provided in our official repository (see section II).

IV. Exemplar use-case

In this section, we demonstrate how GitSchemas supports data augmentation based on foreign-key detection to improve machine learning (ML) pipelines. We note that the utility of GitSchemas is not limited to this use-case as it benefits a plethora of data management tasks as discussed in Section I.

Automated data augmentation for ML pipelines. When the performance of an ML pipeline is unsatisfactory, data practitioners often attempt to augment their data in order to improve performance [22]. To identify complementary tables and join them, one may use schema-matching methods to find database tables with schemas similar to the table at hand, based on attribute, value, and semantic overlap, and data types, distribution, and embeddings [23]. One bottleneck is that these methods often yield false positives with the consequence that the integrated tables do not provide a relevant signal for an ML pipeline or, even worse, deteriorate its performance. Database schemas in GitSchemas provide rich and accurate metadata regarding the joinability of database tables based on foreign-key relations to improve this step.

Experimental setup. To illustrate this, we consider the *Stats* database from the CTU Prague Relational Learning Repository [6], which is not contained in the GitSchemas corpus, and train a regression model from auto-sklearn to predict the "Reputation" attribute from the "users" table. We measure the performance of the AutoML pipeline with the R^2 metric. We compare the model performance when 1) we do not augment the initial "users" table, 2) we augment it based on a schema-matching method, 3) we augment the table using foreign-key lookups in GitSchemas.

Results. Without joining the "users" table with any other table, the model yields an R^2 of 0.72 as shown in Table III. To improve this model, we augment the "users" table by first deploying the Cupid schema-matching method from the Valentine library [23]. We augment the "users" table by matching its schema to overlapping schemas. This results in joining the "Id" column from the "badges" table, to the "AccountId" column from the "users" table. We retrain the ML pipeline on this augmented dataset and find a decreased R^2 value of 0.69. In contrast, if we re-rank the suggestions from Cupid by the string-distance to the closest match in GitSchemas (s. Figure 4), the model performance increases to an R^2 of 0.85. This demonstrates the bottleneck of current schema-matching methods and shows how the accurate schema relations in GitSchemas can be leveraged to improve data augmentation for ML pipelines, even with a relatively simple lookup method.

Data augmentation method	AutoML accuracy (R^2)
No Joins	0.72
Joins by Cupid schema-matching	0.69
Joins by lookup in GitSchemas	**0.85**

TABLE III: Accuracy (R^2 value) of an automated machine learning pipeline with (1) no data augmentation, (2) data augmentation by schema-matching using Cupid, and (3) data augmentation by re-ranking Cupid results with GitSchemas.

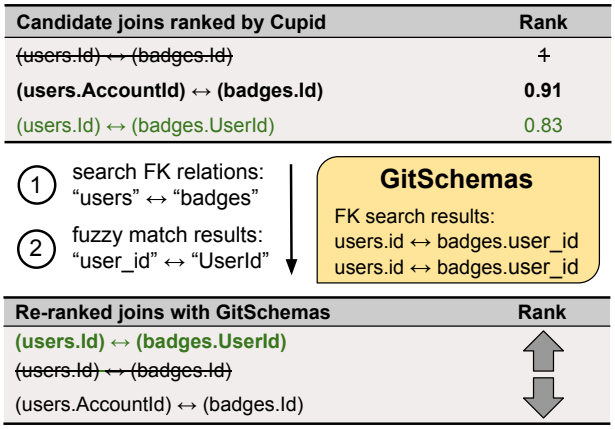

Fig. 4: Initial join candidates for augmenting the "users" table as ranked by Cupid (upper table), and re-ranked based on Foreign Key (FK) search in GitSchemas (lower table). Green indicates the correct matches, bold the predicted matches, and strikethrough invalid ones.

V. Conclusion

In this paper, we present GitSchemas: a relational schema metadata dataset corresponding to 50k databases and 300k tables comprising table names, and column names, types, and relations. We outline the data extraction process which parses SQL files from GitHub through Abstract Syntaxt Trees into structured metadata, and provide an analysis of the dataset illustrating its resemblance to typical enterprise database schemas. In addition, we demonstrate how GitSchemas improves existing methods for augmenting tables in automated data preparation pipelines.

We have high expectations about the utility of GitSchemas for various data management use-cases, such as data integration, feature type detection and learning data validation rules, and are ourselves in an early stage of experimentation. We share this early version of the dataset with the community to spark more use-cases and support ongoing research efforts in the data management and machine learning communities.

REFERENCES

[1] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505*, 2020.

[2] Y. Lu, "An end-to-end automl solution for tabular data at kaggledays," 2019. [Online]. Available: http://ai.googleblog.com/2019/05/anend-to-end-automl-solution-for.html

[3] J. M. Kanter and K. Veeramachaneni, "Deep feature synthesis: Towards automating data science endeavors," in *2015 IEEE international conference on data science and advanced analytics (DSAA)*. IEEE, 2015, pp. 1–10.

[4] R. C. Fernandez, Z. Abedjan, F. Koko, G. Yuan, S. Madden, and M. Stonebraker, "Aurum: A data discovery system," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1001–1012.

[5] A. Bogatu, A. A. Fernandes, N. W. Paton, and N. Konstantinou, "Dataset discovery in data lakes," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 709–720.

[6] J. Motl and O. Schulte, "The ctu prague relational learning repository," *arXiv preprint arXiv:1511.03086*, 2015.

[7] [Online]. Available: https://github.com/cwida/public_bi_benchmark

[8] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then, "Get real: How benchmarks fail to represent the real world," in *Proceedings of the Workshop on Testing Database Systems*, 2018, pp. 1–6.

[9] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang, "Webtables: exploring the power of tables on the web," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 538–549, 2008.

[10] M. Hulsebos, Ç. Demiralp, and P. Groth, "Gittables: A large-scale corpus of relational tables," *arXiv preprint arXiv:2106.07258*, 2021.

[11] [Online]. Available: https://www.kaggle.com/

[12] P. N. Mendes, M. Jakob, and C. Bizer, "Dbpedia: A multilingual cross-domain knowledge base," 2012.

[13] V. Shah, J. Lacanlale, P. Kumar, K. Yang, and A. Kumar, "Towards benchmarking feature type inference for automl platforms," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1584–1596.

[14] C. Yan and Y. He, "Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1539–1554.

[15] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, "Automating large-scale data quality verification," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1781–1794, 2018.

[16] T. Döhmen, H. Mühleisen, and P. Boncz, "Multi-hypothesis csv parsing," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, 2017, pp. 1–12.

[17] G. Inc., "The GitHub Search API," https://docs.github.com/en/rest/reference/search, 2022.

[18] [Online]. Available: https://github.com/seporaitis/mysqlparse

[19] [Online]. Available: https://github.com/lelit/pglast

[20] [Online]. Available: https://github.com/andialbrecht/sqlparse

[21] [Online]. Available: https://github.com/apache/calcite/tree/master/core/src/main/java/org/apache/calcite/sql/dialect

[22] S. Castelo, R. Rampin, A. Santos, A. Bessa, F. Chirigati, and J. Freire, "Auctus: a dataset search engine for data discovery and augmentation," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 2791–2794, 2021.

[23] C. Koutras, G. Siachamis, A. Ionescu, K. Psarakis, J. Brons, M. Fragkoulis, C. Lofi, A. Bonifati, and A. Katsifodimos, "Valentine: Evaluating matching techniques for dataset discovery," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 468–479.