# SCHEMAPILE: A Large Collection of Relational Database Schemas

TILL DÖHMEN, University of Amsterdam, Netherlands
RADU GEACU, University of Amsterdam, Netherlands
MADELON HULSEBOS, UC Berkeley, USA
SEBASTIAN SCHELTER, University of Amsterdam, Netherlands

Access to fine-grained schema information is crucial for understanding how relational databases are designed and used in practice, and for building systems that help users interact with them. Furthermore, such information is required as training data to leverage the potential of large language models (LLMs) for improving data preparation, data integration and natural language querying. Existing single-table corpora such as GitTables provide insights into how tables are structured in-the-wild, but lack detailed schema information about how tables relate to each other, as well as metadata like data types or integrity constraints. On the other hand, existing multi-table (or database schema) datasets are rather small and attribute-poor, leaving it unclear to what extent they actually represent typical real-world database schemas.

In order to address these challenges, we present SCHEMAPILE, a corpus of 221,171 database schemas, extracted from SQL files on GitHub. It contains 1.7 million tables with 10 million column definitions, 700 thousand foreign key relationships, seven million integrity constraints, and data content for more than 340 thousand tables. We conduct an in-depth analysis on the millions of schema metadata properties in our corpus, as well as its highly diverse language and topic distribution. In addition, we showcase the potential of SCHEMAPILE to improve a variety of data management applications, e.g., fine-tuning LLMs for schema-only foreign key detection, improving CSV header detection and evaluating multi-dialect SQL parsers. We publish the code and data for recreating SCHEMAPILE and a permissively licensed subset SCHEMAPILE-PERM.

CCS Concepts: • **Information systems** → *Relational database model*; *Information integration*; *Language models*; • **Computing methodologies** → *Information extraction.*

Additional Key Words and Phrases: relational database schemas; SQL parsing; foreign key detection; CSV parsing; large language models; dataset

**ACM Reference Format:**
Till Döhmen, Radu Geacu, Madelon Hulsebos, and Sebastian Schelter. 2024. SCHEMAPILE: A Large Collection of Relational Database Schemas. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 172 (June 2024), 25 pages. https://doi.org/10.1145/3654975

## 1 INTRODUCTION

Access to fine-grained schema information is crucial for understanding how relational databases are designed and used in practice, for building systems that help users interact with them, and for simplifying and automating data preparation and data integration [5, 11, 33].

Christopher et al. [5] point out the potential of such data for understanding conventions with respect to schema normalisation or table and column naming. Furthermore, the advent of large

Authors' addresses: Till Döhmen, t.r.dohmen@uva.nl, University of Amsterdam, Amsterdam, Netherlands, 1012 WX; Radu Geacu, geacuradu@gmail.com, University of Amsterdam, Amsterdam, Netherlands, 1012 WX; Madelon Hulsebos, madelon@berkeley.edu, UC Berkeley, Berkeley, USA, CA 94720; Sebastian Schelter, s.schelter@uva.nl, University of Amsterdam, Amsterdam, Netherlands, 1012 WX.

ACM 2836-6573/2024/6-ART172
https://doi.org/10.1145/3654975

language models (LLMs) opens up new opportunities for improving data integration and natural language querying techniques [1, 11, 15, 23, 33, 34, 45, 54]. Fernandez et al. [11] point out the promising potential of LLMs trained on large datasets for data enrichment and difficult integration cases in the long tail. Vogel et al. [52] showcase that the scale of existing multi-tabular datasets is not sufficient for representation learning, and that multi-table schemas based on Wikidata [48] improve the accuracy of missing value imputation and table/column name reconstruction tasks.

**Shortcomings of existing table corpora**. Real-world databases, however, are not trivial to obtain. Shah et al. [41] note that "It is almost impossible for researchers to get access to large numbers of truly in-the-wild data from enterprises and other organizations". While valuable insights are shared occasionally [53], the underlying data and schemas remain out of reach for the wider database research community. Large-scale single-table corpora like GitTables [17] or WebTables [3] provide insights into how tables are structured in-the-wild and are a useful resource for entity linking, table search and table QA tasks [7]. However, single table datasets are often derived from CSV files or HTML tables on the web and do neither contain information about how individual tables relate to each other, nor ground truth data type information [49].

On the other hand, existing multi-table (or database schema) datasets [19, 29, 32, 56] are rather small and homogeneous [10, 35]. All schemas in Spider [56] are in English for example and this corpus lacks coverage of important domains such as healthcare (for which it only contains a single schema), which makes it difficult to use for multilingual or domain-specific text-to-SQL training [8, 55, 58]. Apart from a lack of size and diversity, existing multi-table datasets also lack schema properties such as default values, table/column checks or indices that one would typically encounter in real-world databases [10].

**SchemaPile**. We therefore see a clear need for a large-scale, attribute-rich, heterogeneous and accessible dataset of real-world database schemas. Concretely, we establish the following desiderata for such a corpus:

- **Scale** – we need a corpus of real-world relational database schemas that exceeds the 2.5K schemas [5] in the largest existing corpus, at large, to for example aid the training of high-capacity machine learning models for various database-related tasks.
- **Completeness** – the corpus should exhibit fine-grained metadata such as integrity constraints (e.g., with respect to uniqueness, nullability and table checks), indices and default values. Rich metadata is intended to accommodate a wide variety of tasks such as schema matching [26] and foreign key detection [4, 39], database constraint suggestion [40], text-to-SQL [8, 15, 29, 58] data preparation [11, 33, 54], and (database) representation learning [7, 52].
- **Coverage** – the corpus should cover a broad and diverse set of domains and languages to become a suitable resource for studying how relational databases are designed in practice, and to ensure the generalisability of derived machine learning models.
- **Accessibility** – the corpus should be publicly accessible and easy-to-use.

These desiderata guide the design, development, and analysis of our new corpus, SchemaPile. SchemaPile is a dataset of more than 211K database schemas, extracted from data definition language (DDL) statements in SQL files in public GitHub repositories. This corpus contains column definitions, integrity constraints and foreign key relationships. To ensure high-quality ground truth information, we refrain from including synthetic or inferred properties, and apply a conservative extraction process. To the best of our knowledge, SchemaPile is the largest available corpus of its kind, containing more than two orders of magnitude more database schemas than comparable datasets.

**Contributions**. In summary, our contributions are as follows:

- We present SchemaPile, a corpus of 221,171 database schemas and 1.7 million table definitions, extracted from 373,153 SQL files on GitHub. The corpus includes 10 million column definitions, more than 7 million integrity constraints and over 700 thousand foreign key relationships. Furthermore, 29 thousand schemas, 347 thousand tables and 2.2 million columns are populated with data. To the best of our knowledge, SchemaPile is the largest available corpus of its kind, containing two orders of magnitude more table definitions than comparable datasets (Section 2).
- We conduct an in-depth analysis of SchemaPile, where we illustrate its heterogeneity and attribute richness, e.g., that it contains millions of schema metadata properties not present in other datasets (such as integrity constraints, composite foreign key definitions or table checks) as well as a highly diverse language and topic distribution (Section 3).
- We showcase the potential of SchemaPile to improve a variety of data management applications. In particular, we show how to
  - Leverage our corpus as training data of a machine learning model for schema-only foreign key detection, which outperforms a data-based baseline and the commercial large language model GPT-3.5 (Section 4.1).
  - Improve Python's CSV header detection based on column name statistics from SchemaPile (Section 4.2).
  - Evaluate the coverage and correctness of multi-dialect SQL parsers for DDL statements (Section 4.3).
- We publish code and data for recreating SchemaPile, code for our experiments, the foreign key detection models, as well as the permissively licensed dataset SchemaPile-Perm.[1]

## 2 SCHEMAPILE

In this section, we describe the collection process of SchemaPile. We start with a brief overview of the data collection pipeline, followed by a detailed description of the data collection and extraction process (Sections 2.1 & 2.2) and a discussion of the artifacts made available (Section 2.3).

**Overview**. As illustrated in Figure 1, the data collection pipeline consists of two main steps: First, we crawl SQL files from GitHub. Second, we parse the SQL files to extract structured schema metadata and potential table content.

**Design choices**. We intentionally only include ground truth information in our dataset, as found on GitHub, meaning that we do not rely on synthetic data or inferred properties. We also aim for high extraction quality, with a conservative approach of not including content if we are in doubt about the correctness of the extraction process. Moreover, we aim to make SchemaPile easy to consume for researchers and practitioners, by automating the collection and parsing pipeline to generate a single file with a simple, intuitive data structure as output.

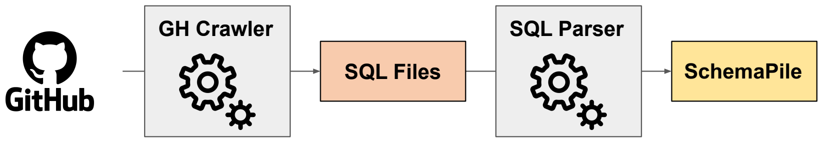


Fig. 1. High-level construction pipeline of SchemaPile.

[1]https://schemapile.github.io

## 2.1 Data Collection

The data source for SchemaPile are SQL files on the software development platform GitHub. We identify and obtain such files with our GitHub SQL crawler based on the GitHub Search API [18]. We search all public GitHub repositories for files that contain SQL code with the expressions `CREATE TABLE` and `FOREIGN KEY`. In particular, the SQL crawler performs the following three steps:

(1) Identify URLs of SQL files via the GitHub search API.
(2) Download the files based on the list of URLs.
(3) Deduplicate the files based on their SHA256 hash.

As of December 2021 (when we ran our crawl), the GitHub search index contains about 7.8M SQL files, of which about 800K match the selected keywords. Out of these, we were able to successfully retrieve approximately 700k files. In order to filter out duplicated files originating from forks and clones, we deduplicate the files via their SHA256 hash code, which reduces the number of files to 373,153 SQL files. Note that the SQL files range in size from 0.1kB to 400kB, with an average file size of 20kB, as GitHub does not index files larger than 400kB for the search API. In total, the files contain 110M lines of code (LOC), including comments and blank lines, with an average of 300 LOC per file.

## 2.2 SQL Parsing

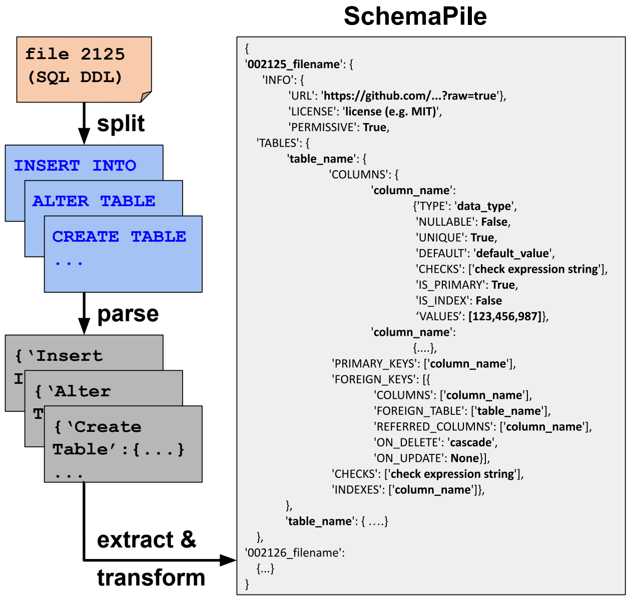
After downloading and deduplicating the files, the next pipeline step is to parse the contained schema data and convert it into a common format that is easy to consume in downstream use cases. During the parsing step, we extract table names, column names, column types, primary keys, foreign keys (including their reference table and their referenced column names), as well as integrity constraints (e.g., uniqueness and nullability constraints and table/column checks), indices and default values for columns.

**Extraction strategy and parsing challenges**. We extract this information from the abstract syntax trees (ASTs) of the `CREATE TABLE` and `ALTER TABLE` statements in the crawled SQL files. A subset of 165.7K SQL scripts contain `INSERT INTO` statements from which we extract data content. Figure 2 shows an overview of the extraction process, which starts with a downloaded SQL script from which an AST is extracted and finally converted to the SchemaPile format.

Through manual exploration, we find that the crawled SQL files have different SQL dialects, contain comments, and are even partially incomplete and/or have syntactical errors. For these reasons, correctly parsing a major part of the downloaded files poses a severe challenge. We conduct an extensive evaluation of the coverage and correctness of different SQL parsers (and parsing strategies) to achieve a high success rate in parsing, while maintaining high quality results at the same time (we refer to Section 4.3 for details on this evaluation).

**Parsing approach**. Based on our findings from the parsing experiments in Section 4.3, we conclude that parsing individual statements has a higher success rate than parsing complete files and that *sqlparser-rs* [44] provides the best trade-off between parsing coverage and extraction correctness. We hence apply the following multi-step approach to extract schema information from each downloaded SQL file:

(1) Detect the character encoding and read the SQL file.
(2) Split up the file contents into individual statements with a non-validating parser.
(3) Parse each statement with a robust multi-dialect parser.
(4) Transform and combine the extracted ASTs into the SchemaPile format.
(5) Deduplicate each extracted schema against the already extracted schemas.



```
{
'002125_filename': {
    'INFO': {
            'URL': 'https://github.com/...?raw=true'},
            'LICENSE': 'license (e.g. MIT)',
            'PERMISSIVE': True,
   'TABLES': {
           'table_name': {
                   'COLUMNS': {
                           'column_name':
                                  {'TYPE': 'data_type',
                                   'NULLABLE': False,
                                   'UNIQUE': True,
                                   'DEFAULT': 'default_value',
                                   'CHECKS': ['check expression string'],
                                   'IS_PRIMARY': True,
                                   'IS_INDEX': False
                                   'VALUES': [123,456,987]},
                           'column_name':
                                  {....},
                   'PRIMARY_KEYS': ['column_name'],
                   'FOREIGN_KEYS': [{
                           'COLUMNS': ['column_name'],
                           'FOREIGN_TABLE': ['table_name'],
                           'REFERRED_COLUMNS': ['column_name'],
                           'ON_DELETE': 'cascade',
                           'ON_UPDATE': None}],
                   'CHECKS': ['check expression string'],
                   'INDEXES': ['column_name']},
           },
           'table_name': { ....}
   },
'002126_filename':
   {...}
}
```

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.

In step (1) we use *chardet*[2] to detect the most likely character encoding and fall back to UTF-8 decoding with automatic error replacement if the read fails. We leverage the non-validating SQL parser *sqlparse* [2] for step (2), which splits the file contents into individual SQL statements.

For parsing each individual statement, we use the multi-dialect SQL parser *sqlparser-rs* [44] via its Python binding *sqloxide* [9] (step (3)). As this parser cannot auto-detect the SQL dialect, we iterate over the supported dialects until we encounter a dialect with which the parsing succeeds. This approach allows us to successfully extract statement-level ASTs from 211K out of 373K SQL files in total.

In step (4), the ASTs of each file are cleaned and transformed into a single schema in our custom format for SchemaPile. In this process, we first iterate over all CREATE TABLE statements from which we extract table names, column names, column types, primary keys, foreign keys (including ON DELETE and ON UPDATE properties), integrity constraints (uniqueness, nullability, column and table checks), indices and default values. Subsequently, we iterate over all ALTER TABLE statements in their natural order of occurrence, extract additional constraints such as primary and foreign keys and update the schema metadata accordingly.

Finally, we collect data values from INSERT INTO statements, including those referring only to a subset of columns. ALTER TABLE and INSERT INTO statements, as well as foreign keys and integrity constraints referring to tables or columns not found in the same file are discarded. In step (5), we deduplicate the schemas based on the extracted metadata (ignoring potential data contents). As a result, each schema in SchemaPile is unique and different from all others.

[2]https://pypi.org/project/chardet/

Table 1. High-level statistics of SchemaPile, and two subsets, SchemaPile-Perm, and SchemaPile-Data in comparison to existing datasets. We list the availability of schemas and data, as well as schema properties like data types, primary and foreign keys (PK/FKs) and integrity constraints (ICs). We further detail the number of databases and tables, as well as the median number of tables per database and the mean number of rows and columns per table.

| **Dataset** | **#DBs** | **Includes schema?** | **Includes data?** | **Includes types?** | **Includes PK/FKs?** | **Includes ICs?** | **#Tables** | **#Tables per DB** | **#Cols p. table** | **#Rows p. table** |
|---|---|---|---|---|---|---|---|---|---|---|
| SQLShare [19] | 64 | ✓ | ✓ | - | - | - | 3.9K | 4 | 18.7 | 11K |
| BIRD [29] | 81 | ✓ | ✓ | ✓ | ✓ | part. | 619 | 5 | 7.2 | 59.4K |
| CTU PRLR [32] | 83 | ✓ | ✓ | ✓ | ✓ | - | 813 | 5 | 6.0 | 4.8K |
| Spider [56] | 166 | ✓ | ✓ | ✓ | ✓ | - | 876 | 4 | 5.1 | 1.8K |
| SchemaDB [5] | 2,500 | ✓ | - | ✓ | ✓ | - | 30.2K | 4 | 6.2 | - |
| WikiDBs [52] | 10,000 | synth. | ✓ | ✓ | ✓ | - | 42.5K | 4 | 17.9 | 46 |
| GitTables [17] | - | - | ✓ | - | - | - | 1M | - | 12.0 | 142 |
| **SchemaPile** | **221,171** | ✓ | **part.** | ✓ | ✓ | ✓ | **1.7M** | **4** | **6.5** | **28** |
| **SchemaPile-Perm** | **22,989** | ✓ | **part.** | ✓ | ✓ | ✓ | **199K** | **4** | **6.7** | **29** |
| **SchemaPile-Data** | **29,076** | ✓ | ✓ | ✓ | ✓ | ✓ | **347K** | **3** | **5.4** | **41** |

**Metadata format**. We store the schema data as a JSON file. We choose JSON as it is easy to consume and well supported across a wide array of programming languages and data processing frameworks. The final resulting 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` attribute which contains the URL of the SQL file from which it was extracted, and license information. The `TABLES` attribute contains a named dictionary of tables in the schema. Each table contains a named dictionary of columns, with type, nullability and uniqueness properties, as well as default values, columns checks, and boolean flags indicating whether the column is part of a primary key or an index.

If data was found, the column also has a `VALUES` attribute, containing the data values. Tables furthermore have primary key, foreign key, table checks and index properties. Foreign key columns and reference columns are lists to be able to represent composite keys. Referenced tables and columns are guaranteed to be present in the same schema.

**Summary of GitHub-related limitations**. We would like to reiterate that our corpus relies on schemas from public GitHub repositories, indexed by the search API (which only indexes files up to 400kB in size), which were available in December 2021. Due to the limited file size, table content recovered from `INSERT INTO` statements may not represent the volume of real-world enterprise data content. The restriction to publicly available data may lead to an underrepresentation (compared to proprietary enterprise databases) of schemas from sensitive domains such as healthcare or finance.

### 2.3 Available Artifacts

We detail the data and code artifacts that we provide for SchemaPile.

**Data collection pipeline**. As part of this work, we release the following artifacts for re-running our data collection pipeline:

- The 695,938 URLs for all identified SQL files from GitHub (together with their corresponding licenses).
- Python scripts to download and recreate the whole SchemaPile dataset as detailed in this section.

Note that this pipeline relies on links to raw files on GitHub that were crawled in December 2021, therefore recreated corpora can slightly differ, based on the availability of the linked-to files. At the time of writing (October 2023), 91,8% of URLs are still available, based on a random probing of 1,000 URLs.

**Preprocessed corpus subset**. In order to make it easier for researchers and practitioners to work with our schema data, we also release a preprocessed version of our data. For legal reasons, we only include data from permissively licensed repositories in this corpus subset, which we call SchemaPile-Perm. We rely on the license information provided by GitHub's repository metadata, and restrict SchemaPile-Perm to the 193 permissive licenses which form the basis for the code dataset *The Stack* [24]. The most commonly occurring permissive licenses in SchemaPile-Perm are MIT (64%), Apache 2 (28%), BSD-3-CLAUSE (4%), and CC0-1.0 (1%). In addition, we maintain an opt-out mechanism for repository owners wishing to remove their data.

Furthermore, we identify personally identifiable information (PII) in the table contents via the *Presidio Analyzer* library [31] and remove the identified values from our released data. We replace removed individual PII-values with a string denoting the `<PIIType>` (such as person, location, etc.) to make it easier for end users to impute them with synthetic data.

SchemaPile-Perm contains about 10% of SchemaPile and is representative of the full dataset. We refer to Section 3.3 for a more detailed comparison. We publish SchemaPile-Perm (in the form of raw SQL files and extracted schemas and data in our JSON format) on *Zenodo* [57][3].

### 2.4 Extensibility

Recently, GitHub limited the ability to search through all public repositories. Our original crawler relied on this, so that it is currently not possible to incorporate recently added GitHub files into our original list of URLs from December 2021. For future extensions of SchemaPile, it may be worthwhile to explore large publicly available datasets such as the aforementioned code dataset *The Stack* [25] or web crawls such as *CommonCrawl* [12] as alternative data sources. Furthermore, enterprises could adapt our data collection and parsing pipeline to create custom internal schema datasets from their proprietary data lakes and code repositories.

## 3 ANALYSIS

In the following, we conduct an in-depth analysis of SchemaPile along the lines of the four desiderata outlined in Section 1.

### 3.1 Scale & Completeness

We investigate the scale and completeness of our corpus by discussing and comparing its high-level statistics, the number and characteristics of the contained schema properties, and by analysing its data contents.

[3]SchemaPile-Perm is available at https://zenodo.org/records/10931803.

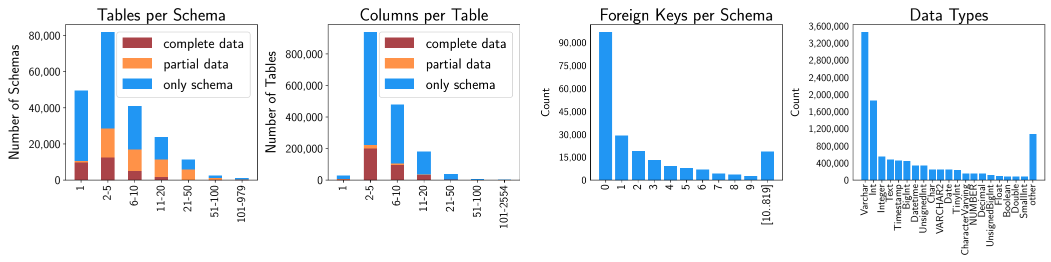


Fig. 3. Distribution of tables, columns, foreign keys and data types in SchemaPile.

**High-level statistics in comparison to existing datasets**. We compare the properties of SchemaPile to seven other datasets (SQLShare [19], BIRD [29], CTU Prague Relational Learning Repository [32], Spider [56], SchemaDB [5], WikiDBs [52] and GitTables [17]) in Table 1, and also detail which schema information the datasets contain. We investigate the number of tables and databases, as well as the median number of tables per database, the mean number of rows and columns per table for SchemaPile. We observe that SchemaPile contains 84.4 times more real-world database schemas than the currently largest schema data set SchemaDB (211K schemas compared to only 2.5K). Furthermore, it is 175x times bigger than the largest non-synthetic dataset with populated database schemas (29.1K databases in SchemaPilevs. 166 databases in Spider). It is also the first dataset of its scale that contains integrity constraints such as unique constraints, default values and column/table checks.

**Number of schema properties**. Next, we detail the number of schema properties in SchemaPile, such as table definitions, integrity constraints and foreign keys in Table 2. Our dataset contains 10.8 million columns and hence also captures 10.8 million column data types. The most commonly used integrity constraint is the `NOT NULL` constraint, followed by the `UNIQUE` constraint, with more than 150 thousand columns which are not primary keys.

Furthermore, our corpus contains 623 thousand default values, more than 21 thousand column checks and more than 16 thousand table checks. The 202 thousand index definitions can be subdivided into 51% single column indices, 30% two column indices and 10% based on three columns, while the remaining 10% of indices are defined on more than 3 columns. In contrast, only 0,9% of primary keys and 0.14% of foreign keys are composites of more than three columns. The vast majority (1.1 million / 84%) of primary keys are based on a single column, followed by (163 thousand / 12%) composite primary keys with two columns. Analogously, the vast majority of foreign keys are single-column keys (97%), followed by composite keys based on two and three columns (1.9% and 0.8%).

**Detailed distributions**. We investigate the detailed distribution of the number of tables per schema, the number of columns per table, the number of foreign keys per schema, and the number of data types in SchemaPile in Figure 3. For the number of tables per schema and the number of columns per table, we additionally detail how the counts differ between databases with full table content, partial table content and schemas only.

**Number of tables per schema**. The vast majority of schemas have 2-5 tables (81,889 / 38.8%), and we also encounter a large number of single table schemas (49,521 / 23.5%), as well as a large number of schemas with with 6-10 tables (40,912 / 19.4%). In general, we find a long tailed distribution of the table counts, including schemas with up to 979 tables. The observed proportions are relatively stable for schemas with partial/full data content.

Table 2. Statistics of schema properties in SchemaPile.

| | **Schema property** | **Count** |
|---|---|---|
| Table Definition | Column Data Types | 10.8M |
| | Primary Keys | 1.3M |
| | Indices | 202K |
| Integrity Constraints | `NOT NULL` Constraint | 6.0M |
| | `UNIQUE` Constraint | 1.3M |
| | Default Values | 623K |
| | Nullable Constraint | 0.6M |
| | Column Checks | 21.9K |
| | Table Checks | 16.7K |
| Foreign Keys | Foreign Keys | 771K |
| | Composite Foreign Key (1 column) | 749K |
| | Composite Foreign Key (2 columns) | 14.6K |
| | Composite Foreign Key (3 columns) | 6.4K |
| | Composite Foreign Key (4-64 columns) | 1,140 |

**Number of columns per table**. The vast majority of tables have 2-5 columns (938,233 / 56.2%), and we also encounter a large number of tables with with 6-10 columns (477,037 / 28.6%). We again observe a long tailed distribution of the number of columns per table, with up to 2,554 columns. The observed proportions are relatively stable for schemas with partial/full data content.

**Number of foreign keys per schema**. The vast majority of database schemas have no foreign keys (96,936 / 45.9%). There is a substantial number of schemas with 1-3 foreign keys (61,356 / 29.1%), and we see diminishing numbers for higher counts of foreign keys. We again encounter a long-tailed distribution of the number of foreign keys per schema, with up to 819 foreign keys.

**Datatypes**. Next, we investigate the distribution of data types in the table columns from SchemaPile. Most of the columns have a string type such as `Varchar` or `Text` (4,330,700 / 39.9%). The second most common type are integers such as `Int` or `Integer` (3,603,277 / 33.2%), followed by fractional numbers, e.g., `Float` or `Double` (1,029,831 / 9.4%) and time-related types (478,553 / 4.4%). We note that our type distribution is similar to the type distribution observed in enterprise databases [53].

**Data content analysis**. Next, we investigate SchemaPile-Data, the subset of the schemas for which we also have table values.

**High-level statistics**. We compute statistics for the subset of schemas in SchemaPile with tables which have at least partially data available. We list the resulting high-level statistics for those schemas in the left column of Table 3 (partial data). We also list the summary statistics of the subset of schemas that contain data values for all tables and columns in the schema (full data).

As already discussed earlier, we find that our dataset contains a sizable number of fully populated schemas (29K), fully populated tables (347K) and 2.2M columns with 58.2M data values in total. We observe that the fully populated schemas have fewer and narrower tables on average (3 tables with 5.4 columns compared to 4 tables with 6.5 columns), but contain more values on average (41.1 values per column compared to only 27.6 in the semi-populated case).

**Distribution of data values**. Next, we take a deeper look into how data values are distributed in Figure 4. On the left, we show a histogram of the number of columns with a given number of rows. We find that most columns have a single row (665,043 / 30.7%) or between 2-5 rows (663,986 / 30.6%), followed by 6-10 rows (304,792 / 14.1%), and we again encounter a long-tailed distribution with columns having up to 30,623 rows. In the middle, we boxplot the distribution of the number of rows in schemas with different table sizes. The overall number of rows is low with a median below 10, and we observe the highest variance for schemas with 51-100 tables. Note that there is a sizable number of outlier schemas with several thousand rows, which the boxplot does not represent. On the right side, we show the number of rows for tables with different column sizes. We again see a low median below 10, and find that the median and variance of the number of rows grows from tables with a single column to tables with 21-50 columns (e.g., such tables seem to have a higher number of rows on average) and drops afterwards for tables with more than 50 rows.

**SQL dialects**. Moreover, we analyse the distribution of SQL dialects in SchemaPile. This is a difficult question due the challenges encountered when parsing our collected files (we refer to Sections 2.2 & 4.3 for details). In order to determine the proportions of SQL dialects in our corpus, we count how often certain parsers identify a file as having a given dialect and report the minimum number per dialect as a conservative estimate. We consider four different SQL dialects (ANSI, MySQL, PostgreSQL, TSQL) and leverage three multi-dialect parsers (SQLGlot [47], sqlparser-rs [44], JSQLParser [22]) as well as two dialect-specific parsers (pglast [36] for Postgres and tidb [46] for MySQL). For the 373,153 files in SchemaPile, we classify 71,112 (19.1%) as being in MySQL

Table 3. Data content statistics of SchemaPile-Data subset, comparing schemas with any data content to complete schemas, where all columns in all tables are filled with data.

| Property | Partial Data | Full Data |
|---|---:|---:|
| #Schemas | 75,565 | 29,076 |
| #Complete Tables | 347.0K | 114.9K |
| #Filled Columns | 2.2M | 615K |
| #Values | 58.2M | 26.2M |
| Median #tables p. schema | 4 | 3 |
| Mean #columns p. schema | 6.5 | 5.4 |
| Mean #values p. column | 27.6 | 41.1 |

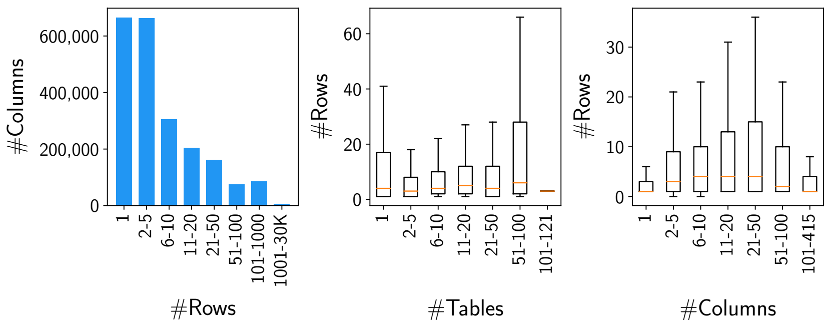


Fig. 4. Histogram of the number of columns with a given row count (left), distribution of the number of rows over different schema sizes (middle), distribution of the number of rows over different column sizes (right).

dialect, 63,937 (17.1%) as being in PostgreSQL dialect, 63,378 (17.0%) as being in TSQL and 59,938 files (16.1%) as being in ANSI SQL. We cannot reliably detect the dialect for the remaining 114,788 (30.8%) files. In summary, we find that MySQL is most common dialect in SchemaPile, followed by ANSI, Postgres and TSQL, which occur in roughly equal proportions.

**Summary**. We find that SchemaPile (211K schemas) and SchemaPile-Perm (23K schemas) far exceed the size of existing corpora (synthetic ones as well as real-world ones) by roughly two orders of magnitude. Furthermore, SchemaPile is the first dataset of its scale to provide comprehensive schema metadata, with attributes such as data types, primary keys, foreign keys, and integrity constraints.

## 3.2 Coverage

We investigate the coverage of languages and domains in our corpus by analysing the semantics of our schemas with respect to table and column naming.

**Naming**. We investigate the names given to tables, columns, primary key columns and foreign key columns in SchemaPile. We plot the most frequent terms and their occurrence counts in Figure 5. As expected, we encounter a long-tailed distribution of names in all cases with 517K unique table names and 1.2M unique column names. Classic database table names from commercial use cases like “users”, “employees”, “customers” are very common. The most common column names are general terms like “id”, “name”, and “description”. The most common primary key and foreign key names (“id”, “id_”, “user_id”, “userid”) refer to general identifiers and users. This indicates that collected schemas and tables indeed refer primarily to real-world concepts that one would expect in an enterprise context.

We also find common column names originating from popular open source projects, which are typically integrated into custom databases. Examples are names like “trigger_name”, “trigger_group” and “sched_name” from the Quartz job scheduler project or the “password_resets” table from the PHP web framework Laravel.

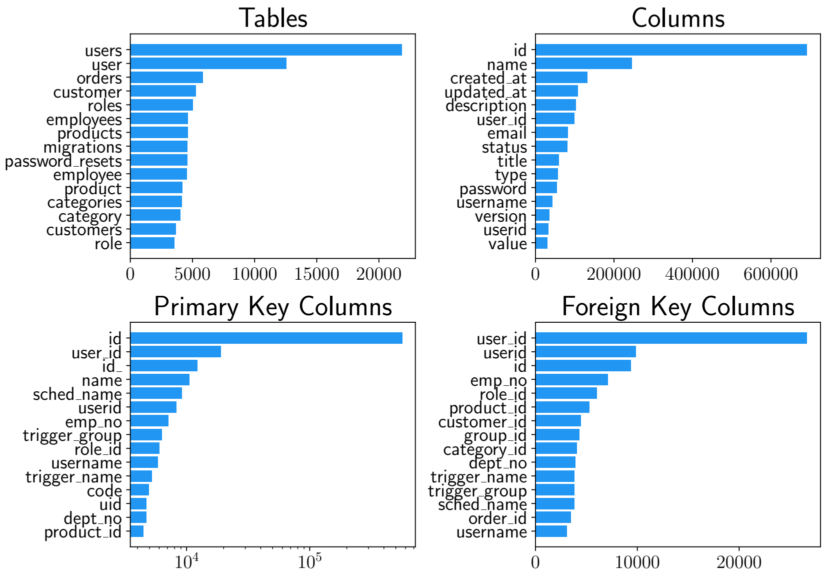


Fig. 5. Most frequent terms in SchemaPile.

We list different random samples from table and column names (as well as checks and default values) in Table 4 to illustrate the high content diversity of SchemaPile. There are, for example, 546 unique table names referring to "order_items" and 44 unique column names referring to "user_access". The column and table names show the variety of naming conventions that occur in the corpus.

Table 4. Samples from SchemaPile.

| | |
|---|---|
| **Table names** (random sample) | physiological_file, eg_chat_conversation_state, MANUFACTURE_T, customExchangeRates, firmware, parts_connection_glues, PokemonTypes, business_rules, AVAILABLE, wy_setting_house,... |
| **Table names** (containing 'order item') | Order_items, CartOrderItem, W6D1_ORDERITEM, dennis_sql_store.order_item_notes, T_SCGORDERITEMS, resorderitem, SITE_DB.project_department_pickupdate_order_items, public.orderitem, T_PLA_CUSTOMER_ORDER_ITEM, GenebankOrderItem,... |
| **Column names** (random sample) | teachercardnumber, level_1_help_include_file_key, employeeschedulehours, lot_specific_id, original_date_posted, vitri_name, t10_10_company_address, allowrecommend, fk_videogame_id, case_status_idcase_status... |
| **Column names** (containing 'user access') | user_accesstoken, DBUserAccess, iduseraccesslog, id_user_access_menu, access_useraccesslevel, user_access_level_name, invited_user_access_level, USER_ACCESS_ID, useraccesslog_result, user_access_token... |
| **Table checks** (random sample) | 'usertype = 'mentee' OR usertype = 'mentor' OR usertype = 'org'', 'age > 0 AND age < 110', 'salary >= ', 'status IN ('not started', 'todo', 'in progress', 'done')', 'price > 0', 'end_date > start_date', 'point IN (3, 2)', 'dow IN (1, 2, 3, 4, 5, 6, 7)', 'art_min_price > 0', 'evaluation BETWEEN 1 AND 5' |
| **Column defaults** (random sample) | '-', 'Y', '0', '0000-00-00 00:00:00', '-1', '0.0000000000', 'false', 'No Payment Method Provided', 'now' |

**Language distribution**. Next we investigate the languages of table schemas covered in SchemaPile. For that, we leverage a language detection model from the *fasttext* [20, 21] library, which we apply to the table names. We restrict our analysis to the tables where the model returns a high-confidence (≥ 75%) prediction and plot the language distribution for the resulting 89,632 tables in the left part of Figure 6.

As expected, English is the most common language (75,404 / 84.1%) used for naming tables. Interestingly however, we also encounter significant proportions of other languages, e.g., Spanish (6,045 / 6.7%), Portuguese (3,682 / 4.1%), French (1,493 / 1.6%), German (444 / 0.5%) or Dutch (200 / 0.2%) and many others. This diversity is in stark contrast to other datasets, where nearly all the table names for which we could determine a language are in English (100% for Spider, 98.6% for BIRD and 95.4% for CTU PRLR).

We further illustrate the language diversity of SchemaPile with a sample of non-English table names in Table 5. These tables originate from a Spanish message board database, a German high-school management database with grades and courses, and a Russian accounting database.

Table 5. Sample of non-English schemas in SchemaPile, showcasing its high language diversity.

| | |
|---|---|
| **#012075, Spanish** | 'usuarios', 'cargo', 'categorias', 'permisos', 'tbl_uploads' |
| **#178998, German** | 'teilnehmer', 'klasse', 'kursleiter', 'module', 'kurse', 'keyuser', 'schulfach', 'pruefung', 'note', 'schulfach_gesamtnote', 'frage', 'rueckmeldung', 'rueckmeldung2frage', 'klasse2Teilnehmer' |
| **#626215, Russian** | 'Гос.Заказ.Челяб.Статус пользователя', 'Гос.Заказ.Челяб.Отделы', 'Гос.Заказ.Челяб.Гос.Бюджет' |

**Topic distribution**. Next, we investigate the semantics of the tables contained in SchemaPile with a topic detection model. For that, we leverage the *Google Natural Language AI* cloud service [14] to classify the topic (`content_categories_version`) per table name, and report the top predicted

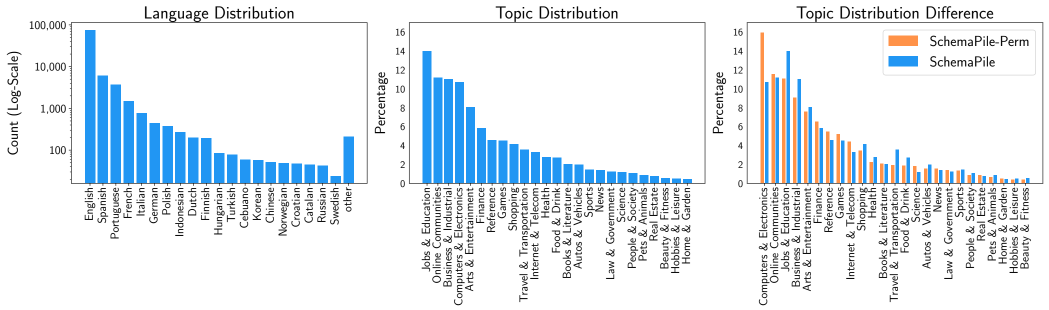


Fig. 6. Distributions of languages (left) and topics (middle) detected from table names. Distributions of topics in the permissively licensed subset of SchemaPile-Perm (orange) and the whole dataset SchemaPile (blue).

high-level topic category per table. We illustrate the results in the right part of Figure 6. We find that tables about jobs and education (14%), online communities (11.2%) and business use cases (11.1%) are most common. However, we also encounter a long tailed topic distribution with diverse content from areas such as as finance (5.8%), gaming (4.5%), sports (1.4%) or pets (0.9%).

**Summary**. Our corpus covers a wide semantic range through more than one million unique column names and more than a half a million unique table names, and contains a long-tailed topic distribution. We would like to especially highlight the high proportion (>15%) of non-English schemas, which stands in stark contrast to existing datasets, which only contain a handful of non-English tables.

## 3.3 Accessibility

Next, we discuss the accessibility of our corpus by analysing its permissively licensed and publicly available subset SchemaPile-Perm. As already detailed in Section 2, while we provide the code and scripts to recreate (crawl, parse, transform) the full SchemaPile corpus, we also provide a subset of SchemaPile called SchemaPile-Perm (originating from permissively licensed repositories) for convenience in already processed form in addition.

We compare the high-level statistics of SchemaPile to SchemaPile-Perm in Table 6. We find that the permissively licensed subset SchemaPile-Perm amounts to roughly one tenth of SchemaPile, in terms of the number of schemas and the number of tables with data, as well as in the total number of data values. The fact that several average statistics (such as the median number of tables per schema, the mean number of columns per schema, and the mean number of values per column) are very close between the datasets, indicates that SchemaPile-Perm forms a representative sample of SchemaPile. We additionally investigate the topic distribution of table names in SchemaPile-Perm (analogous to the previous analysis from Section 3.2) and plot the most common topics in Figure 6, using orange bars for SchemaPile-Perm (orange) and blue bars for SchemaPile. In general, the distributions are close and the long-tailed character is preserved. We observe minor differences in the popularity of certain topics: in SchemaPile-Perm, databases about computers & electronics are most common, and we find a lower fraction of databases about topics such as jobs and education, travel & transportation and food & drink.

**Summary**. SchemaPile-Perm is compliant with licensing and privacy aspects as only the permissively licensed subset of schemas is published and potential PII values are masked. It is hosted on Zenodo, to ensure easy access and persistence. Despite its smaller scale, SchemaPile-Perm closely resembles the properties of the full SchemaPile corpus.

Table 6. Statistics of the permissively licensed subset SchemaPile-Perm of SchemaPile, which provides a representative sample of roughly 10% of the the schemas available in total.

| Dataset | SchemaPile | SchemaPile-Perm |
|---|---:|---:|
| #Schemas | 221,171 | 22,989 |
| #Tables | 1.7M | 199K |
| #Schemas with data | 75.6K | 7.1K |
| #Tables with data | 347.0K | 34.9K |
| #Columns with data | 2.2M | 219.0K |
| #Total data values | 58.2M | 5.9M |
| Median #tables per schema | 4 | 4 |
| Mean #columns per schema | 6.5 | 6.7 |
| Mean #values per column | 27.6 | 28.7 |

## 4 APPLICATIONS

In this section, we detail three example applications for SchemaPile for different data management use cases. Note that our goal here is not to propose novel methods, but to showcase the potential of our corpus to improve and augment existing prediction and evaluation methods.

We train a schema-only foreign key detector based on a large language models with data generated from SchemaPile in Section 4.1, improve Python's CSV header detection based on column name statistics from our corpus in Section 4.2 and evaluate the coverage and correctness of multi-dialect SQL parsers for DDL statements in Section 4.3.

### 4.1 Foreign Key Detection

A long-standing challenge in data discovery is the detection of foreign-key relationships between existing tables [4, 26, 39]. We showcase how the semantics from SchemaPile can improve machine learning models to tackle this challenging task using only schema information.

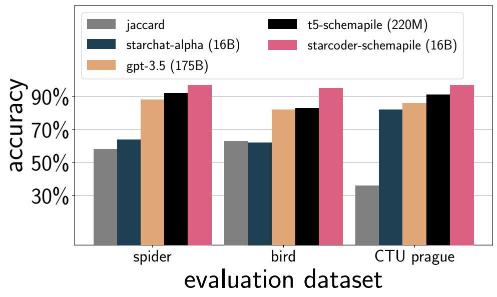
**Evaluation data**. We create three evaluation datasets for our FK detection experiments from the Spider [56], BIRD [29] and CTU Prague Relational Learning Repository (PRLR) [32] database repositories as follows. We first download the `.sqlite` files for the database in each repository (for CTU PRLR, we convert the databases to `.sqlite` files first). For each database, we inspect all pairs of tables and retain pairs of tables with a single foreign-key relation, ordering them such that the first table contains the primary key column, which is referenced by the second table. We filter out pairs of tables with multiple foreign-key relations or composite keys. This leaves us with 555 table pairs for spider, 386 pairs for BIRD and 877 pairs for CTU PRLR.

*4.1.1 The value of SchemaPile for ML-based FK detection.* The goal of our first experiment is to showcase the value of SchemaPile for ML-based FK detection.

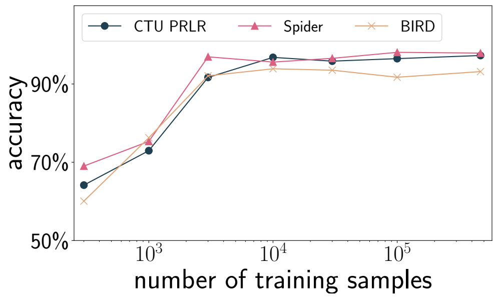
**Experimental setup**. We create an ML-based FK detection model called `starcoder-schemapile` by fine-tuning *starcoder base* [30], a large language model (LLM) for building coding assistants, which is available on Hugging Face and has 16 billion parameters.

**Fine-tuning details for starcoder-schemapile**. We leverage the *jsonformer*[4] library to make our LLM generate structured JSON output. We fine-tune the base model with all FK relations extracted from SchemaPile, following the same extraction protocol as in the evaluation datasets, which results in 468,770 table pairs (about three orders of magnitude more than in the existing

[4]https://github.com/1rgs/jsonformer



(a) Accuracy for FK detection on various repositories. The starcoder and T5 models fine-tuned on SchemaPile, outperform all baselines, including the commercial LLM `gpt-3.5`.



(b) Accuracy improvements for FK detection with growing numbers of training samples (foreign key relations) from SchemaPile. The large number of samples strongly impacts prediction performance.

Fig. 7. Benefits of SchemaPile for ML-based FK detection.

repositories). We ensure that there is no leakage between SchemaPile and the evaluation data, by removing all overlapping PK-FK pairs from the evaluation. This leaves us with 437 pairs for Spider, 242 pairs for BIRD and 558 pairs for CTU PRLR. We apply the following prompting format for the task and train the model to generate a corresponding structured JSON answer:

| | |
|---|---|
| **Prompt:** | You are given the following SQL database tables:<br>address(id, uuid, locality, city, state_id)<br>state(id, uuid, state_name)<br>Output a json string with the following schema {table, column, referencedTable, referencedColumn} that contains the foreign key relationship between the two tables. |
| **Output:** | *{"table": "address", "column": "state_id", "referencedTable": "state", "referencedColumn": "id"}* |

We run fine-tuning for three epochs with four A100 40GB GPUs, which takes about 16 hours. We refer to our repository for details on the exact hyperparameters used.[5].

**Baselines**. We compare `starcoder-schemapile` against a data-based baseline and three more schema-only baselines leveraging LLMs:

- `jaccard` – a data-based method, taken from the *Valentine* benchmark [26], which computes the jaccard distance between the values in all column pairs and returns the column pair which is closest in terms of value overlap as prediction. We directly run the Valentine code on the .sqlite databases containing the table pairs from our evaluation datasets. We stop the prediction for a table pair if the distance computation is not finished after five hours.
- `starchat-alpha` – an instruction-tuned coding assistant model [30] created from starcoder base (with 16 billion parameters as well), for which we apply zero-shot prompting with our previously introduced prompt format.
- `gpt-3.5` – a commercial massive general-purpose LLM from OpenAI (`gpt-3.5-turbo`) with 175 billion parameters, for which we also apply zero-shot prompting with our previously introduced prompt format, and use OpenAI's function API to ensure validated JSON output.

[5]The fine-tuned weights for `starcoder-schemapile` and `t5-schemapile` models are available at https://huggingface.co/tdoehmen/starcoder-schemapile-fk and https://huggingface.co/tdoehmen/t5-schemapile-fk

- `t5-schemapile` – Google's small language model `T5-base` [38] with 220 million parameters, which we fine-tune analogously to `starcoder-schemapile`.[4.1.1]

We evaluate `starcoder-schemapile` along with the baselines discussed above, on the aforementioned evaluation datasets and report the accuracy of their predictions.

**Results and discussion**. We plot the resulting accuracies in Figure 7(a). We find that the schema-only, LLM-based models largely outperform the data-based method `jaccard`. The non-finetuned LLMs `gpt-3.5` and `starchat-alpha` perform already better than `jaccard`, with the exception of one case (`jaccard` is better than `starchat-alpha` on the BIRD corpus). The `jaccard` baseline is also extremely slow, requires several minutes per table pair on average in the Spider and CTU PRLR datasets, and fails to process a large fraction of table pairs within the five hour budget (up to 15% for CTU PRLR). Note that the LLM-based methods only need to run an inference pass, whose runtime is typically in the seconds range.

For the schema-only methods, we observe that `gpt-3.5` always outperforms `starchat-alpha`, which we attribute to the fact that `gpt-3.5` has an order of magnitude more parameters. We encounter a remarkable performance of `starcoder-schemapile`, which outperforms the other models in all settings, and reaches up to 97% accuracy. Despite its small size of 220M parameters, the other fine-tuned model `t5-schemapile` provides the second-best performance and also outperforms the zero-shot prompted `gpt-3.5` baseline. These findings show the value of SchemaPile as training data for LLMs in data integration settings, and that our corpus can be leveraged to make small fine-tuned LLMs (which have many desirable characteristics in terms of deployment options and cost-efficiency) outperform a large commercial LLM.

We also want to highlight the accuracy of `starcoder-schemapile` on typical database benchmark datasets. While the unfiltered CTU PRLR corpus contains TPC-C, TPC-DS, and TPC-E schemas, the majority of TPC-C and TPC-DS FK-relations were removed in the filtering step due to containment in SchemaPile. On 43 unseen FK-relations from TPC-E, `starcoder-schemapile` achieves a 93.3% FK-detection accuracy.

*4.1.2 Benefits of SchemaPile over existing datasets.* Next, we showcase that fine-tuning on SchemaPile is more beneficial than fine-tuning on smaller, existing datasets.

**Experimental setup**. For that, we leverage the 877 FK relations from the second largest database repository CTU PRLR and train a model refered to as `starcoder-ctu` following the same protocol as used for `starcoder-schemapile`. We compare both of these models on the evaluation dataset built from the Spider corpus (note that we skip evaluation on BIRD due to a high table overlap between BIRD and CTU PRLR).

**Results and discussion**. We observe a stark accuracy difference between `starcoder-schemapile`, which achieves 97% accuracy, and `starcoder-ctu`, which only achieves 69% accuracy on the Spider data. This is a strong indication that the huge amount and diversity of the relations from SchemaPile matter (which provides 534 times more training samples). We additionally evaluate another variant `starcoder-ctu-lora`, where we leverage the parameter-efficient fine-tuning (PEFT) method *LoRA* [16], which learns a low-rank decomposition of the weight update for fine-tuning. For this variant, we reuse existing hyperparameters for LoRA-tuning the starcoder model from [27]. However, we find that `starcoder-ctu-lora` provides a slightly lower accuracy of 66% compared to `starcoder-ctu`, which reached 69%.

*4.1.3 Scaling of FK detection accuracy with the number of training samples.* Based on the results from our previous experiment, we take a deeper look into the relationship between the size of the training data (the number of FK-relations to present to the model during fine-tuning) and the resulting accuracy.

**Experimental setup**. We take random samples of increasing size (300, 1,000, 3,000, 10,000, 30,000, 100,000) up to the full size of 468,770 FK relations from SchemaPile, and use them to fine-tune the base starcoder model, analogous to the previous experiments. Next, we evaluate the accuracy of the resulting models on the evaluation data from the Spider, BIRD and CTU PRLR repositories.

**Results and discussion**. We plot the resulting accuracy in relation to the number of training samples (on a logarithmic scale) in Figure 7(b). The results confirm that a larger number of FK relations strongly improves the accuracy of the FK detection model. Going from 300 samples to 3,000 samples improves the accuracy by more than 25% in all cases. The benefit of the additional samples is starting to flatten out between 3,000 and 10,000 examples, after which we only see minor gains of around half a percent of precision. This indicates that the FK detection model benefits from a large size of training samples and that SchemaPile even contains more than sufficient training samples for this task. Also, it is a strong indication for the performance benefits observed in the previous experiment.

*4.1.4 Improving FK detection quality in the Valentine benchmark.* In the following, we showcase the potential of SchemaPile to improve the prediction quality for FK detection in the state-of-the-art data integration benchmark *Valentine* [26].

**Experimental setup**. We experiment with the benchmark dataset and code[6] from Valentine. We filter the benchmark dataset to the 72 *joinable* and *semantically joinable* table pairs with single PK-FK columns, and for example remove unionable tables not relevant for our experiment. We include eight baseline approaches from Valentine and measure performance with the benchmark's proposed mean recall at size of ground truth (MR@SG) metric.

We evaluate the performance of our `starcoder-schemapile` model and an ensemble method called `schemapile-ensemble` in addition. This ensemble first produces a ranked list of FK column candidates based on the `jaccard` baseline method from Valentine (which ranks the FK column candidates by the Jaccard similarity of their values to the values in the PK column). If the column candidate predicted by `starcoder-schemapile` is contained in the list, the ensemble returns this candidate as prediction, otherwise its returns the top column candidate from the `jaccard` method.

Table 7. Mean Recall at size of ground truth (MR@SG) for *joinable* and *semantically joinable* table pairs with a single matching column from the Valentine benchmark [26] (bold: best, underlined: second-best).

| | **Method** | **MR@SG** |
|---|---|---|
| **Valentine baselines** | `semprop` | 0.08 |
| | `embDI` | 0.40 |
| | `similarityflooding` | 0.46 |
| | `coma-s` | 0.50 |
| | `coma-si` | 0.58 |
| | `cupid` | 0.67 |
| | `distribution-based` | 0.71 |
| | `jaccard` | <u>0.83</u> |
| **Ours** | `starcoder-schemapile` | 0.75 |
| | `schemapile-ensemble` | **0.85** |

**Results and discussion**. We list the results in Table 7 and find that our schema-only model `starcoder-schemapile` provides a competitive performance of an MR@SG of 0.75, which outperforms seven out of eight baselines from Valentine and is only beaten by the `jaccard` approach. We

[6]https://github.com/delftdata/valentine/tree/v1.1

note that the high performance of the jaccard method is a result of the synthetically generated join pairs in the Valentine benchmark, inducing a complete value overlap in 50% of the joinable columns, which is not necessarily realistic in practice. Furthermore, as the join pairs are created synthetically based on a single input dataset, both the left and right side table have the same name and are only distinguished by a _source and _target appendix. In contrast to this, our model starcoder-schemapile is trained on real-world FK-pairs, where the table names usually carry part of the signal for the prediction. Since the table names in the benchmark are not meaningful, the model has to primarily base its decision on the column names.

Our ensemble approach schemapile-ensemble, which reranks the jaccard outputs, even further improves performance to an MR@SG of 0.85 and outperforms all single-method approaches. These results confirm the potential of our corpus to improve prediction performance on data integration benchmarks.

## 4.2 CSV Header Detection

A common problem in accessing tabular data in the common CSV format is to detect whether these files contain a header row with column names. This functionality is a standard feature in CSV parsing libaries (e.g., in the Python standard library), and the quality of various header detection approaches is the subject of recent data loading benchmarks such as *Pollock* [50]. We conduct a header detection experiment on a series of CSV datasets to showcase the potential of our corpus to improve this task.

**Experimental setup**. We leverage the "survey sample" dataset from Pollock [50], which is an annotated sample of 100 CSV files, originating from open government data sites. We use the Docker image provided by Pollock to run the experiment on their dataset. Analogous to the experimental design in [50], we measure the F1 score of various header detection methods. We evaluate eight existing baseline approaches from Pollock and additionally integrate the following two approaches leveraging SchemaPile.

- schemapile-lookup – a header detection heuristic based on our corpus: We create a hash table from the schemas in SchemaPile, which contains their column names as key and the number of times a given column name occurred in the corpus as value. At detection time, we first split each row of the CSV file into a list of values (from which we strip quotation characters, etc.). Next, we iterate through the values of each row, interpret each value as column name, look up the corresponding counts in our hash table and sum the resulting counts per row. This yields a list of "name occurrence" counts per row. We apply a simple outlier detection method on these counts by determining if the row with the largest count has at least a 10x higher count than the subsequent row. If we find such a row, we predict it to be the header row.
- schemapile-augmentation – an ensemble approach created by combining the py_csv[7] implementation from Pollock with the previously described schemapile-lookup heuristic. The method py_csv is taken from the csv module in the standard library of Python, and predicts a header if the inferred types of the values in the first row (the header candidate) are inconsistent with the inferred types of the subsequent rows. We observe that this method has a high precision but a low recall. Therefore, our ensemble approach simply uses the prediction from schemapile-lookup in cases where py_csv predicts no header.

**Results and discussion**. We list the resulting F1 scores in Table 8. We find that our simple heuristic schemapile-lookup already outperforms five out of the eight baselines from Pollock with a score of 0.53. Our ensemble schemapile-augmentation significantly improves the performance of the

[7]https://github.com/HPI-Information-Systems/Pollock/blob/main/sut/pycsv/pycsv.py

Table 8. F1 scores for header detection on the "survey sample" dataset from the Pollock benchmark [50] (bold: best, underlined: second-best).

| | Method | F1 score |
|---|---|---|
| **Pollock baselines** | `csv_commons` | 0.26 |
| | `univocity` | 0.40 |
| | `calc` | 0.44 |
| | `dataviz` | 0.48 |
| | `hypoparsr` | 0.51 |
| | `py_csv` | 0.67 |
| | `spreadweb` | 0.68 |
| | `clever_csv` | <u>0.70</u> |
| **Ours** | `schemapile-lookup` | 0.53 |
| | `schemapile-augmentation` | **0.78** |

`py_csv` method, reaches an F1 score of 0.78, and thereby outperforms all existing single-method approaches from the benchmark. These results demonstrate how the semantic information about the column names from SchemaPile helps to further improve header detection techniques. Such semantic information is for example needed to correctly handle the file `10.January_2019.csv` from Pollock, as shown in Figure 8.

| Department Family,Entity,Date,Expense Type,Expense Area,Supplier,Transaction Number,V... |
|---|
| Department of Health, South Warwickshire NHS Foundation Trust, 02/01/2019, Business Rat... |
| Department of Health, South Warwickshire NHS Foundation Trust, 03/01/2019, Serv.Re... |
| Department of Health, South Warwickshire NHS Foundation Trust, 03/01/2019, Pay Co... |

Fig. 8. CSV file from data.gov.uk with an ambiguous header whose types are consistent with the data (and which would therefore not be detected by Python's csv library).

In this CSV file, the inferred data types between the header and content rows are fully consistent, which leads to a false negative prediction from `py_csv`. However, `schemapile-lookup` easily identifies the header row on top via its scores from the co-occurrence counts: the values from the first row appear 27,223 times as column names, while the values from the subsequent rows only occur 45, 20 and 23 times.

## 4.3 Evaluating DDL Parsing

The raw SQL files, to which we provide URLs as part of SchemaPile, can be helpful in more traditional areas of data management. We showcase such a use case by measuring the quality of polyglot (multi-dialect) SQL parsers for DDL statements. Such parsers are important for flexible data loading and transpilation across different dialects. We evaluate the following open source polyglot SQL parsers for our study:

- *SQLGlot 11.4.1* [47], a Python-based parser for 19 different SQL dialects (including Postgres, MySQL, Oracle, and TSQL). It is used by projects such as the dataframe library Ibis[8] or Pinterest's querybook IDE[9].
- *JSQLParser 4.4* [22], a Java-based multi-dialect parser, which handles 7 dialects (Postgres, MySQL, DB2, SQLite, Oracle, MSSQL, H2).

[8]https://ibis-project.org
[9]https://github.com/pinterest/querybook/

- *sqlparser-rs 0.33* [44], a Rust-based multi-dialect parser for 11 dialects (including ANSI-SQL, BigQuery and Snowflake), which is used by query engines from the Apache Arrow ecosystem such as DataFusion[10] and Ballista[11], as well as the GlueSQL[12] library.
- *simple-ddl-parser 0.30* [42], a parser for DDL statements built using Lex and yacc that supports 12 SQL dialects.

*4.3.1 Coverage.* Our first experiment aims to answer what fraction of files/statements from SchemaPile existing parsers can handle without throwing errors.

**Experimental setup**. We pass the raw SQL files from SchemaPile to the parsers in two different forms. First, we pass our collected files in their entirety to a parser, and record how many files can be parsed without throwing an error by each parser (e.g., all statements in the file have to be parsed without errors). Next, we use the `split()` method of the non-validation parser *sql-parse* [2] (whose lexer supports a wide variety of dialects[13]) to extract the individual statements from all our schema files, and pass individual statements to our parsers to evaluate. In total, this experiment included 373,153 files and 24,627,506 individual statements based on the raw SchemaPile SQL files from our URL list (Section 2.3).

SQLGlot and sqlparser-rs require the explicit specification of the SQL dialect to use before parsing statements or files. As this information is not part of our corpus, we loop through all available dialects for these parsers until we find a setting with which the input can be parsed successfully.

Table 9. Coverage of various polygot SQL parsers on SchemaPile in terms of percentage of files and statements that they can parse without throwing errors (bold: best, underlined: second-best).

| Parser | Files | | Statements | |
|---|---|---|---|---|
| SQLGlot | 24.47 | (91,308) | **85.07** | (20,950,924) |
| JSQLParser | **31.59** | (117,862) | 71.14 | (17,519,264) |
| sqlparser-rs | <u>24.97</u> | (93,179) | <u>82.76</u> | (20,380,557) |
| simple-ddl | 17.39 | (64,891) | 64.01 | (15,781,546) |

**Results and discussion**. Table 9 lists how many files and statements per parser are parsed without throwing errors (both in absolute numbers and as fraction of the overall corpus. We encounter a low coverage in terms of parseable files in general, with SQLGlot and sqlparser-rs only being able to parse about a quarter of all files (24.47% and 24.97%) and simple-ddl-parser being able to handle even less files (17.39%).

The best coverage on files is provided by JSQLParser, which manages to parse nearly a third of all files (31.59%). In summary, we find that parsing on the level of whole files is challenging and that roughly one quarter of all files (24.03%) cannot be parsed by any parser.

Fortunately, parsing on the statement level provides a significantly higher coverage, as up to 85% of statements are parsable without error by a single parser. Simple-ddl-parser still provides the worst performance with 64.01% only, however the ranking changes for the other parsers here: JSQLParser has a low performance (71.14%), while SQLGlot manages to parse the highest amount of statements (85.07%), closely followed by sqlparser-rs (82.76%).

[10]https://arrow.apache.org/datafusion/
[11]https://arrow.apache.org/ballista/
[12]https://gluesql.org
[13]https://github.com/andialbrecht/sqlparse/blob/master/sqlparse/keywords.py

The reasons for parsing errors vary, two common causes that we observe are the lack of support for certain data types or custom features. SQLGlot for example does not support the `MEDIUMINT` data type or the `INSERT IGNORE` statement from MySQL, while JSQLParser for example has no support for the `CLUSTERED` keyword. However, we find that only 1.85% of statements cannot be interpreted by any parser.

*4.3.2 Correctness.* One limitation of open-source parsers, which do not originate from the database vendor, is that they are usually contributor-driven projects that are prone to bugs and other issues. As a result, there may be cases where the parser wrongly assumes that it parsed a statement correctly. Therefore, our second experiment aims to answer the question what fraction of statements from SchemaPile these parsers can parse correctly.

**Experimental setup**. In order to evaluate the correctness of the parsing results, we need a source of ground truth. We generate this ground truth by restricting our analysis to files which can be parsed successfully by the single-dialect parser *pglast* [36], which internally uses the *libpg_query* C library built from Postgres' source code. Due to the maturity and wide usage of Postgres (and the usage of the C library in other popular database systems like DuckDB [37]), we assume that the parsing results from this library are reliable enough for us to function as ground truth. This leaves us with 63,937 from the original 373,153 files in SchemaPile.

In order to compare the parsing results, we proceed as follows. We configure SQLGlot and sqlparser-rs (which require the upfront specification of the parsing dialect) to leverage the Postgres dialect. From each AST resulting from a parsed statement, we extract the following: table names with all identifier delimiters stripped, column names found in column definition statements, a list of `DATABASE` names, and the number of `NOT NULL`, `UNIQUE`, `PRIMARY KEY` and `FOREIGN KEY` constraints. In order to measure the correctness of the parsing results, we investigate the overlap of the extracted elements with the information extracted by pglast.

Table 10. Correctness of various polygot SQL parsers on SchemaPile in terms of percentage of outcome matches with pglast (bold: best, underlined: second-best).

| Parser | Tables | Cols | DBs | NotNull | Unique | PK | FK |
|---|---|---|---|---|---|---|---|
| SQLGlot | <u>89.97</u> | **99.83** | **99.98** | 91.32 | 86.89 | 63.20 | 93.51 |
| JSQLParser | 82.26 | 98.16 | <u>99.97</u> | **98.05** | **99.57** | **99.99** | **99.99** |
| sqlparser-rs | **90.70** | 96.13 | 99.94 | **98.05** | <u>99.17</u> | <u>97.43</u> | <u>97.05</u> |
| simple-ddl | 59.93 | <u>99.01</u> | 99.91 | 64.46 | 93.73 | 67.73 | 37.19 |

**Results and discussion**. Table 10 lists the results of our correctness experiment. Both JSQLParser and sqlparser-rs perform very well, with high correctness (>95%) for columns, database names and constraints, and only minor problems with handling some table names (where they only parse 82.26% and 90.70% correctly). SQLGlot comes close to them, however it shows slight problems with `NOT NULL` constraints (where it only parses 91.32% correctly) and severe problems with primary key constraints, where less than two thirds (63.20%) are parsed correctly.

Finally, we encounter severe issues with simple-ddl-parser, which parses only 59.93% of table names correctly, gets only about two thirds of `NOT NULL` and primary key constraints right (64.46% and 67.73%) and parses most of the foreign key constraints wrongly (only 37.19% correct). We notified the contributors of the simple-ddl-parser project about these issues, and this already led to several bugfixes in their code. In summary, these findings indicate that SchemaPile can serve as a valuable testing ground for multi-dialect SQL parsers.

## 5 RELATED WORK

Large-scale datasets of relational tables and databases have been instrumental for developing and evaluating data management systems. Here, we review related datasets with database schemas and entire databases, and discuss a selection of impactful applications that benefit from large-scale datasets such as SchemaPile.

**Datasets with schemas and databases**. Datasets and benchmarks are essential to the development of new database systems. The effectiveness of machine learning for data management applications sparked the development of database collections for tasks such as semantic parsing (e.g. Text-to-SQL), with Spider [56] and BIRD [29], which contain populated databases along with natural language text and query pairs. Despite the value of these datasets for benchmarking, they are relatively small-sized regarding the number of databases and tables (Table 1), making them inadequate for model training. The CTU Prague Relational Learning Repository [32], Public BI Benchmark [13, 53] and the larger WikiDBs [52] datasets have also been introduced to facilitate benchmarking and applications over relational databases, but lack rich metadata, such as integrity constraints. Moreover, these datasets do not provide sufficient real-world and diverse databases to also train and assess high capacity models for generalizability towards real-world schemas and databases. SchemaDB [5] contains 2.5K schemas extracted from GitHub. Despite the identical data source considered for SchemaPile, we find that the search criteria and parsing approach used by SchemaDB only captures a small subset of the schemas and properties present in SchemaPile, while also not being in an easy to use format. Due to its larger size, diversity, and accessibility, SchemaPile has a higher utility for ML model training. Existing corpora of similar scale as SchemaPile, e.g. WebTables [3, 28] and GitTables [17], contain real-world tables but lack rich metadata about relations with other tables, integrity constraints, and exact data types.

### 5.1 Selected applications of DB schema datasets

We briefly discuss related work for our exemplary applications.

**ML for foreign key detection**. Several machine learning methods have been proposed for FK detection in relational databases. As these are typically trained on synthetic or small-scale datasets, Rostin et al. [39] point out that large-scale and diverse datasets are key for robust and accurate training and evaluating ML models. The Valentine benchmark [26] for entity matching revealed the difficulties of existing ML methods such as Aurum [4] for making ML models generalize to new benchmarks. As we show in Section 4.1, SchemaPile exhibits a scale and diversity to train FK-detection methods that generalise well to distinct datasets.

**CSV header detection**. Header detection methods for CSV files [6, 51], typically rely on heuristics such as type-consistency between values in the first row and the following rows. While this works well for tables that contain numeric values and string header names, it is less reliable when header and column values are of the same type (e.g. text). Overall, as identified in the data loader benchmark Pollock [50], none of the existing parsers robustly infers complex headers. In Section 4.2 we illustrate the effectiveness of training models on high-quality schema data available in SchemaPile to infer the headers in potentially messy CSV files with high accuracy.

**Evaluating SQL DDL parsers**. Multi-dialect parsers are important for flexible data loading and transpilation across different dialects. The common DDL parsers SQLGlot [47] and SQLOxide [9], benchmark performance against several multi-dialect SQL parsers, such as sqlparse [43]. However, these testbeds only compare runtime of these parsers with a limited number and diversity of queries, whereas the correctness and coverage are not evaluated. While efficiency is important, DDL parsers are also expected to detect errors in SQL statements while robustly parsing different dialects, which is not straightforward. This is reflected in Section 4.3, where we examine different multi-dialect parsers, which also informed the SQL parsing procedure for SchemaPile itself. Further parsing experiments with SchemaPile have also led to the identification of multiple issues in the `simple-ddl-parser` [42], contributing to its improvement. We believe that SchemaPile is a viable source for structurally testing DDL parsers.

## 6 CONCLUSION

Large-scale collections of real-world databases and schemas are essential to the evaluation and development of data management applications.

In this paper, we present SchemaPile, a heterogeneous, attribute-rich corpus of 211,171 database schemas, and 1.7 million table definitions, extracted from SQL files on GitHub. To the best of our knowledge, SchemaPile is the largest available corpus of its kind, containing almost two orders of magnitude more database schema definitions than comparable datasets, while also containing data content for a sizable subset of tables. We illustrated how SchemaPile satisfies four desiderata regarding scale, completeness, coverage and accessibility through an in-depth analysis on the millions of schema metadata properties provided by our corpus, as well as its highly diverse language and topic distribution. In addition, we showcased its potential to improve a variety of data management applications.

We believe that the utility of SchemaPile stretches far beyond the applications that we explored so far. In future work, we aim to leverage our corpus as training data for schema completion and generation models, which learn to generate or extend schemas, suggest column and table names, foreign key relations, data types, or impute missing values. A particular focus should be put on leveraging the rich set of integrity constraints available in SchemaPile (such as `NOT NULL` constraints, `UNIQUE` constraints and `CHECK`s). These may be valuable to improve the recommendation of validation rules for data validation scenarios, for which current approaches rely on heuristics only [40]. Furthermore, we plan to explore multi-task learning setups, where we finetune a model on a number of tasks covered well in SchemaPile and evaluate its performance on new but related tasks. We also want to explore the potential of our corpus to improve data synthesizing techniques for text-to-SQL [55, 58] models. In addition, we plan to use the data content of SchemaPile to study database schema design in-the-wild (e.g., the level of normalisation in real world databases).

## REFERENCES

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting data errors: Where are we and what needs to be done? *Proceedings of the VLDB Endowment* 9, 12 (2016), 993–1004.

[2] Andi Albrecht. 2023. python-sqlparse – a non-validating SQL parser for Python. https://github.com/andialbrecht/sqlparse

[3] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment* 1, 1 (2008), 538–549.

[4] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[5] Cody James Christopher, Kristen Moore, and David Liebowitz. 2021. SchemaDB: Structures in relational datasets. *arXiv preprint arXiv:2111.12835* (2021).

[6] Till Döhmen, Hannes Mühleisen, and Peter Boncz. 2017. Multi-hypothesis CSV parsing. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. 1–12.

[7] Haoyu Dong, Zhoujun Cheng, Xinyi He, Mengyu Zhou, Anda Zhou, Fan Zhou, Ao Liu, Shi Han, and Dongmei Zhang. 2022. Table pre-training: A survey on model architectures, pre-training objectives, and downstream tasks. *arXiv preprint arXiv:2201.09745* (2022).

[8] Longxu Dou, Yan Gao, Mingyang Pan, and Jian-Guang Lou. 2023. MultiSpider: towards benchmarking multilingual text-to-SQL semantic parsing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 12745–12753.

[9] Will Eaton. 2020. sqloxide. https://github.com/wseaton/sqloxide.

[10] Erki Eessaar. 2023. On the Naming of Database Objects in the SQL Databases of Some Existing Software. In *Computer Science On-line Conference*. Springer, 534–550.

[11] Raul Castro Fernandez, Aaron J Elmore, Michael J Franklin, Sanjay Krishnan, and Chenhao Tan. 2023. How Large Language Models Will Disrupt Data Management. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3302–3309.

[12] Common Crawl Foundation. 2023. Common Crawl – a free, open repository of web crawl data that can be used by anyone. https://commoncrawl.org

[13] Bogdan Ghita, Peter Boncz, and Diego Tomé. 2019. *Public BI benchmark - part 1.* https://doi.org/10.5281/zenodo.6277287

[14] Google. 2023. Google Natural Language AI. https://cloud.google.com/natural-language

[15] Zihui Gu, Ju Fan, Nan Tang, Lei Cao, Bowen Jia, Sam Madden, and Xiaoyong Du. 2023. Few-shot Text-to-SQL Translation using Structure and Content Prompt Learning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–28.

[16] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. *ICLR* (2022).

[17] Madelon Hulsebos, Çagatay Demiralp, and Paul Groth. 2023. GitTables: A large-scale corpus of relational tables. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–17.

[18] GitHub Inc. 2022. The GitHub Search API. https://docs.github.com/en/rest/reference/search.

[19] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. Sqlshare: Results from a multi-year sql-as-a-service experiment. In *Proceedings of the 2016 International Conference on Management of Data.* 281–293.

[20] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. 2016. FastText.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651* (2016).

[21] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of Tricks for Efficient Text Classification. *arXiv preprint arXiv:1607.01759* (2016).

[22] jsqlparse. https://github.com/JSQLParser/JSqlParser

[23] Moe Kayali, Anton Lykov, Ilias Fountalis, Nikolaos Vasiloglou, Dan Olteanu, and Dan Suciu. 2023. CHORUS: Foundation Models for Unified Data Discovery and Exploration. *arXiv preprint arXiv:2306.09610* (2023).

[24] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. arXiv:2211.15533 [cs.CL]

[25] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).

[26] Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, and Asterios Katsifodimos. 2021. Valentine: Evaluating matching techniques for dataset discovery. In *2021 IEEE 37th International Conference on Data Engineering (ICDE).* IEEE, 468–479.

[27] Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M Rush. 2023. Guess & Sketch: Language Model Guided Transpilation. *arXiv preprint arXiv:2309.14396* (2023).

[28] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. 2016. A large public corpus of web tables containing time and context metadata. In *Proceedings of the 25th international conference companion on world wide web.* 75–76.

[29] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint arXiv:2305.03111* (2023).

[30] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

[31] Microsoft. 2023. Presidio Analyzer. https://microsoft.github.io/presidio/analyzer/

[32] Jan Motl and Oliver Schulte. 2015. The CTU Prague Relational Learning Repository. *arXiv preprint arXiv:1511.03086* (2015).

[33] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can Foundation Models Wrangle Your Data? *VLDB* (2022).

[34] Fatemeh Nargesian, Erkang Zhu, Renée J Miller, Ken Q Pu, and Patricia C Arocena. 2019. Data lake management: challenges and opportunities. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1986–1989.

[35] Aggelos Papamichail, Apostolos V Zarras, and Panos Vassiliadis. 2020. Do People Use Naming Conventions in SQL Programming?. In *SOFSEM 2020: Theory and Practice of Computer Science: 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20–24, 2020, Proceedings 46.* Springer, 429–440.

[36] pglast. https://github.com/lelit/pglast.

[37] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data.* 1981–1984.

[38] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[39] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. 2009. A machine learning approach to foreign key discovery.. In *WebDB.*

[40] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1781–1794.

[41] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. 2021. Towards benchmarking feature type inference for automl platforms. In *Proceedings of the 2021 International Conference on Management of Data.* 1584–1596.

[42] simple-ddl parser https://github.com/xnuinside/simple-ddl-parser

[43] sqlparse https://github.com/andialbrecht/sqlparse.

[44] sqlparser rs https://github.com/sqlparser-rs/sqlparser-rs

[45] Michael Stonebraker, Ihab F Ilyas, et al. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41, 2 (2018), 3–9.

[46] tidb. https://github.com/pingcap/tidb

[47] George Sittas Toby Mao. 2022. SQLGlot. https://sqlglot.com/

[48] Theo Van Veen. 2019. Wikidata. *Information technology and libraries* 38, 2 (2019), 72–81.

[49] Petros Venetis, Alon Y Halevy, Jayant Madhavan, Marius Pasca, Warren Shen, Fei Wu, and Gengxin Miao. 2011. Recovering semantics of tables on the web. (2011).

[50] Gerardo Vitagliano, Mazhar Hameed, Lan Jiang, Lucas Reisener, Eugene Wu, and Felix Naumann. 2023. Pollock: A Data Loading Benchmark. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1870–1882.

[51] Gerardo Vitagliano, Lucas Reisener, Lan Jiang, Mazhar Hameed, and Felix Naumann. 2022. Mondrian: Spreadsheet Layout Detection. In *Proceedings of the 2022 International Conference on Management of Data.* 2361–2364.

[52] Liane Vogel and Carsten Binnig. 2023. WikiDBs: A Corpus of Relational Databases From Wikidata. In *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data Bases (VLDB 2023), Vancouver, Canada, August 28 - September 1, 2023 (CEUR Workshop Proceedings, Vol. 3462).* CEUR-WS.org. https://ceur-ws.org/Vol-3462/TADA3.pdf

[53] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get real: How benchmarks fail to represent the real world. In *Proceedings of the Workshop on Testing Database Systems.* 1–6.

[54] David Vos, Till Döhmen, and Sebastian Schelter. 2022. Towards Parameter-Efficient Automation of Data Wrangling Tasks with Prefix-Tuning. In *NeurIPS 2022 First Table Representation Workshop.*

[55] Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. 2020. Grappa: Grammar-augmented pre-training for table semantic parsing. *arXiv preprint arXiv:2009.13845* (2020).

[56] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing.* 3911–3921.

[57] Zenodo. 2023. Zenodo - Research Shared. https://zenodo.org. https://zenodo.org

[58] Yiyun Zhao, Jiarong Jiang, Yiqun Hu, Wuwei Lan, Henry Zhu, Anuj Chauhan, Alexander Li, Lin Pan, Jun Wang, Chung-Wei Hang, et al. 2022. Importance of synthesizing high-quality data for text-to-sql parsing. *arXiv preprint arXiv:2212.08785* (2022).

Received 12 October 2023; revised 15 January 2024; accepted 23 February 2024