Deal with it! Towards Self-Organizing Data Schemas from Semi-Structured Inserts

Benjamin Hättasch
benjamin.haettasch@dfki.de
DFKI & TU Darmstadt
Darmstadt, Germany

Leon Krüger
leon.krueger@stud.tu-darmstadt.de
TU Darmstadt
Darmstadt, Germany

Carsten Binnig
carsten.binnig@cs.tu-darmstadt.de
TU Darmstadt & DFKI
Darmstadt, Germany

Abstract

In scenarios where the structure of information to store is not known upfront or changes over time, neither traditional relational databases nor schemaless approaches are well-suited. Relational databases are great for data analysis and exploration, but require a carefully crafted schema, which causes high manual overhead. Entities not considered during schema design cannot be stored. In contrast, schemaless approaches allow users to store all kinds of data without the need for a schema, but require schema-checking on read to ensure that queries can read certain attributes. Therefore, we present an approach that works on top of existing relational databases and autonomously organizes the schema based on incoming data. We call our system JUSTINE (JUST INsert Engine). Users can store data while omitting table or column names or using the wording of their mental schema, which might deviate from the real one in the database. Our system maps the incoming data to the existing schema and adjusts the schema where needed (i.e., by adding columns or new tables) to handle the insertion of attributes or entities not represented in the schema yet. To ensure a high-quality schema, our system provides operations to periodically fix mistakes and optimize the schema (e.g., by combining tables). We propose a way to evaluate such systems, even though there is no single gold standard for a suitable schema, and use this for an initial evaluation of our approach. Our approach should be seen as a first step towards self-organizing data schemas, therefore, we provide an extensive roadmap for further research.

Keywords

schemas, automatization, semi-structured data, relational databases

ACM Reference Format:
Benjamin Hättasch, Leon Krüger, and Carsten Binnig. 2026. Deal with it! Towards Self-Organizing Data Schemas from Semi-Structured Inserts. In *Nineth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3814940.3815327

1 Introduction

Storing data in a structured format is often crucial to analyzing it later and gaining value from it. How well this data can then be used depends heavily on the semantics of the structure (the schema). Coming up with a useful schema can be very difficult and can cause high overheads. In particular, such a schema will either be very generic (hindering analytics and automated usage of the data) or needs to be adjusted whenever new kinds of information should be stored.

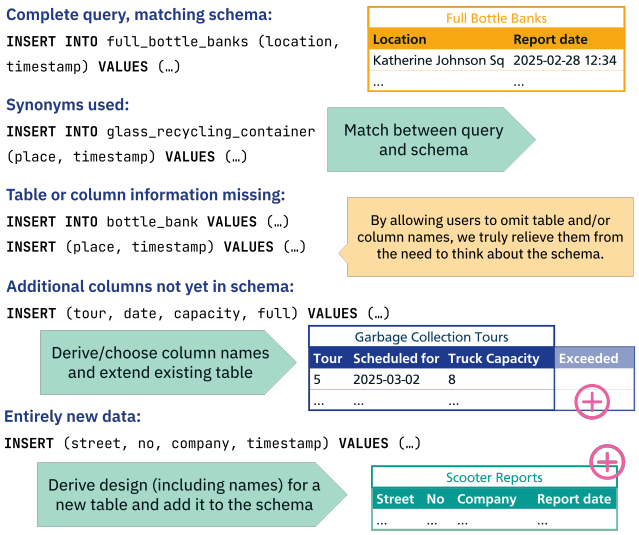

Figure 1: Possible insertion scenarios. The system needs to handle different kinds of inserts, ranging from ones that already match the existing schema to those that contain only partial schematic information (i.e., table or attribute names but not both) and where those names are only synonyms of the ones used in the database. The system needs to map between the input and the existing schema, or extend the schema by adding new attributes or tables. For schema adjustments, suitable names must be chosen.

Supporting Unclear or Changing Inputs. The detailed semantics of data might be unclear when designing the system that handles them, e.g., when dealing with user-generated inputs such as complaints, support tickets, or requests. This can be either due to a high (or even unrestricted) amount of possible attributes or because of situation changes over time. Take, for example, an IT department that wants to improve how they assign and handle support tickets created by their users. Their existing system might only request some basic structured data (e.g., name and email of the user, as well as the ID of the machine the problem is on and maybe a selection between *hardware* or *software* issue), while every other information is entered in a text field. Having more information in a structured format could help to automatically choose who handles the ticket

aiDM '26, Bengaluru, India

ACM ISBN 979-8-4007-2719-1/2026/05
https://doi.org/10.1145/3814940.3815327

or allow the system to retrieve similar previous cases to base the answer on. Manually choosing which fields are required for each issue is unfortunately difficult, since the relevant information can be highly different for each kind of issue (e.g., a URL is crucial for problems with a web app, but completely useless for hardware issues) and might vary over time (e.g., when the software or hardware stacks in the company change). Thus, one needs a system that not only allows the users to enter arbitrary key-value pairs, but one that matches between the wording used by them and the existing structure to store the values in a semantically useful way. To allow for better automatic handling of the tickets, the system should additionally not place all tickets in the same table, but group them in different tables based on the relevant information (e.g., one for the issues of the web app including the URL and browser information, another one for reports of broken printers storing which one is affected). Even after such a specific table was created in the schema, further changes to it might be necessary, e.g., when a shift to working from home suddenly makes information about the private internet provider of the user relevant.

Existing Databases are not Sufficient. Currently, there are two general options for storing and accessing such data:

Relational databases allow efficient querying and easy browsing. This is reached through a strict schema that is usually manually designed upfront, requiring both knowledge about the domain and databases. In the example from above, one would need to decide how the data should be distributed across different tables and which attributes each of them should have. Additionally, the schema must be manually adapted when additional information with deviating attributes should be stored. Storing arbitrary attributes is possible using key-value tables, but without a strict schema, browsing or automatically handling the data gets much harder. Thus, a relational database requires continuous remodeling in scenarios with changing entities and attributes.

Alternatively, schemaless databases such as key-value stores and other non-relational databases allow data to be stored without the necessity to define a schema, but the lack of that schema makes it harder to use it afterwards. It can be unclear what is stored at all and how the information is organized. To really make use of the data, the schema has to be provided on read, either by trying to infer it from the data or again manually by developers or data scientists building on top of it. The effort for schema design will hence often only be postponed.

Self-Organizing Schemas. Neither direction offers the necessary functionality out of the box—having the data exploration and querying options of a relational database combined with the possibility of adding new kinds of information without manual overhead by automatically adjusting the schema. In this paper, we therefore present an approach that works towards this vision and allows users to perform vague inserts without the need to specify a schema upfront. That is relevant since allowing the omission of schema information when setting up the database but then requiring it in the input queries would just shift the place and time of the specification without liberating users from the burden of coming up with a sensible schema. By allowing them to submit vague queries, that responsibility is moved to the data system.

Handling Semi-Structured Inserts. In this paper, we present our approach for this. Our system JUSTINE (JUST INsert Engine) can start with an empty database and derive the complete schema based on the inserts alone, or use a manually created schema as a starting point that is then adjusted during runtime. It needs to support different scenarios (see Figure 1 for examples): In some cases, the inserts might be complete and directly match existing tables. In other cases, they might look complete, but use deviating terms for table and column names (e.g., *timestamp* but the database column is called *report_date*). The system needs to map them to the existing schema. Queries might contain names for table and columns, but without a direct correspondence in the existing schema. In that case, the schema must be updated by adding the missing columns or creating entirely new tables. Finally, when users are unaware of whether a suitable structure already exists and do not know what a good structure would look like, they may create inserts that omit table or column names. Again, the system needs to search for existing suitable structures and adjust the schema if there is no direct correspondence.

This functionality of automatic mapping and adjustments allows the insertion of arbitrary items into the database while providing a relational schema after every step. Future INSERTs of the same type can then be inserted in those adjusted tables, leading to a coherent storage of semantically related concepts. This makes both manual exploration and automated processing easier.

The Right Tables & Columns. The mapping is done with a multi-step approach that first determines suitable candidate tables and then performs an attribute mapping for each of them. The quality of these mappings is then heuristically scored, and the insertion is executed in a way that minimizes the number of schema adjustments. Instead of running the system fully automatically, it is also possible to bring the human into the loop and ask them to choose between these mappings (which still substantially reduces the effort compared to manually coming up with the correct schema directly).

Housekeeping and Optimizing the Schema. With the mapping approach described above, the number of tables and attributes can only grow, but there is no way to correct suboptimal decisions later. Hence, as a second major functionality, such a self-organizing data system needs further means to adjust the schema. In this paper, we describe and evaluate an approach to identify separate tables that should be combined (e.g., when synonyms were not correctly resolved at first). Additionally, we discuss further automatic operations which could be run regularly to optimize the schema.

Contributions & Artifacts. In this paper, we present JUSTINE, a system for self-organizing data schemas. It consists of a multi-step approach for determining target tables and attributes for inserts that may lack table or attribute names, and can use synonyms for all structural information. The approach adds new tables and columns where needed. The second central contribution are operations that try to detect suboptimal aspects of the schema and optimize them, e.g., by splitting or combining tables. Third, we introduce a new metric to measure the quality of the schema with regard to accessing and handling the data afterwards. We publish our code, datasets, and the fine-tuned models as open source at https://link.tuda.systems/JUSTINE.

Outline. The remainder of the paper is structured as follows: We start by giving an overview of the system and its components in Section 2, followed by more details on the architecture and the fine-tuning process in Section 3. In Section 4, we discuss what a good schema should look like and how the quality could be measured, which we then apply in the initial evaluation in Section 5. Afterwards, we discuss a potential research roadmap for further advances in the area in Section 6. An overview of related existing work can be found in Section 7 before we conclude in Section 8.

2 Overview

Our system consists of two components: The first one stores incoming data in relations and updates them based on the insert queries if necessary. To improve the schema, a second component periodically runs operators to uncover suboptimal decisions made during the iterative insertion process and attempts to improve them by moving the affected data or adjusting the schema. In this section, we give a quick overview of the components; the detailed implementation will then be explained in the following section.

Handling Insertions. The insertion is handled iteratively, considering one insertion query at a time. The system needs to decide autonomously in which table to place the values (either determining the correct existing one or proposing the name for a new table) and then to map the values to existing columns or add new columns. We use two Llama 3 models [17] for this semantic bridging between inputs and schema, which we fine-tuned for the tasks of table and column mapping. Additionally, heuristics like fuzzy string matching are used for trivial cases. In addition to the query itself, the models are prompted with a representation of the current schema, including value samples for each relation. The models work across domains and do not need to be fine-tuned for each database. The system then needs to perform necessary adjustments to the schema (adding columns and/or tables) and can execute an adapted version of the insertion query afterwards.

Improving the Schema. Due to the iterative nature of the schema building, the system might take decisions that turn out to be suboptimal in the long run. Hence, the system needs operations to correct these decisions later, e.g., moving records to anther table and (if necessary) deleting attributes that are NULL for all remaining records, renaming tables, or combining multiple tables. The system needs to regularly run a detection method for these issues and then try to fix them. As a first step towards this, we introduce an operation that finds tables with a very similar schema and combines them.

Previous Work. We presented a demo of the basic idea at VLDB'25 [11]. Compared to the demo, we improved the use of the models, in particular, which information is made available to them and which is hidden on purpose to better steer their behavior and reduce inference time. As a result, the system usually has more valid variants for the mappings to choose from. We improved the heuristics for selecting among possible mappings to achieve better results in fully automatic mode. As a completely new component, optimization operations are introduced. Additionally, this paper contains details on the fine-tuning process, provides metrics to evaluate the approach, an initial evaluation, and a research roadmap.

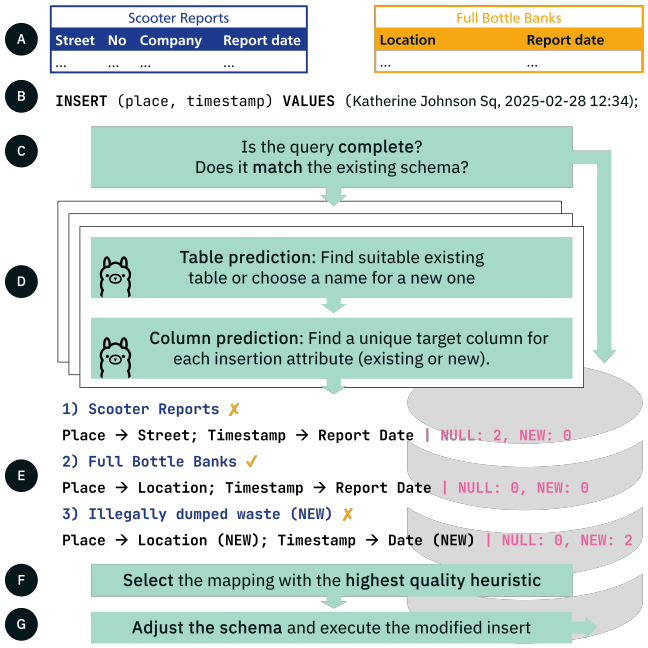

Figure 2: Insertion flow. When a new insertion query (B) is handled, the system first checks whether it matches the existing schema (A) and can directly be inserted (C). If not (i.e., because either table or column names are missing in the query or are not present in the schema) the system runs the mapping stage (D), which first asks the first model to find a suitable table and then uses the second model for column mapping. This stage can be run multiple times to get different possible mappings (E). Afterwards, a heuristic (F) is used to select the mapping that best balances between the fit of the data and necessary changes to the schema. Based on this, the schema is adjusted (if necessary) and the data is inserted (G).

3 Approach

In this section, we describe the insertion workflow, the required models, and the database optimization operations in detail.

3.1 Insertion Workflow

Our approach works on top of a relational database. The insertion flow for a concrete example can be seen in Figure 2. When a new insertion query (B) arrives, the system first tries to insert it as it is (C). But in many cases, table or column names may be missing or do not match the existing schema (A), since the users do not know about the existing schema but use their words and mental schema, or leave it up to the system to make sense of the data. Moreover, the insertion query might contain new entities or attributes that are not yet covered in the schema at all. In this case, the following multi-step approach will be invoked, where the system will try to find suitable tables and columns for the data.

Finding Target Tables and Columns. This is done with a mixture of fine-tuned LLMs and heuristic approaches (e.g., (fuzzy) string

matching) that can augment the LLMs and help to reduce computation load for trivial cases (i.e., where names clearly identify the correct tables and/or columns). This process (D) is run multiple times (e.g., three, the number can be selected to balance between expected quality and higher inference cost) to identify multiple possible mappings and then select the best from them using a heuristic (F). It consists of two stages:

First, the system tries to find suitable tables. One of the candidates is selected using fuzzy string matching, since our experiments shows that using LLMs alone will not only lead to higher inference costs but actually decrease performance in these trivial cases. For the other candidates, the system invokes a model for *table prediction* to find a corresponding existing table. Alternatively, it can give a name for a new table instead. The prompts include the existing tables and their columns. The process is repeated for more candidates, hiding the tables selected already from the schema prompted to the model.

For each candidate, the system can now check whether the insertion query provides column names that match the table's attributes, either exactly or with fuzzy string matching. If not, a second model for *column prediction* is invoked, which iteratively requests a unique target column for each column in the insert (either an existing one or giving the name for a new column). In these prompts, only the details on the selected table are provided (schema and instances for each of the attributes). We tested both prompting for each column mapping individually and requesting all mappings of an insert query at once. In our experiments, the latter often failed to provide a unique mapping, instead mapping multiple input columns to the same target column. Also, the results for queries with more than one to three columns were often invalid, missing some of the mappings or giving invalid values. Hence, we decided to map them one by one, making it easier to enforce correct mappings and reducing inference cost when needing to rerun it.

Selecting the Best Mapping. The system now needs to select one of the mapping candidates, which it does by ranking them. For this ranking, the number of new columns that would be needed when inserting the data into this table, and the number of additional attributes not used by the query can be used as heuristics. Additionally, a certain cost (either static or with an increasing temperature based on the size of the schema already existing) can be associated with creating a new table. This penalty is a hyperparameter that can be selected based on whether the scenario benefits more from a larger number of tables (with only small differences between some of them) or from a smaller schema. As an example, this would steer whether the system tends to place all movies in the same table regardless of their genre, or whether it will create a table for *science fiction movies*, one for *animated movies*, and so on. If wanted, the creation of additional tables or even columns can be turned of at all, preserving a fixed schema and using just the capabilities to handle vague inserts.

The multi-stage approach allows the system to consider both the table and column information before settling on a certain target table, while reducing the amount of context that needs to be provided in each of the prompts and being able to adjust it to the relevant subparts (e.g., by only listing columns of a single table in the column mapping). If we had requested table and column mappings in one prompt, the chances are very high to end with an invalid mapping. Moreover, in an interactive setting, this allows producing a concise list of candidates from a large pool of potential matches from which a user can select, capitalizing on the user's knowledge and the human ability to intuitively find agreeing entries when lists are short enough.

Finally, JUSTINE then performs the schema alterations (if necessary) and inserts the data (G).

Good Table and Column Names. Our system needs to decide which names to use for new tables and columns at multiple places. This is particularly relevant for those insert queries where table or column names are completely missing. However, assuming that those submitting the queries are often people without experience in database schema design, it might not be sensible to use included names as-is, even for those queries that contain column and table names. Thus, JUSTINE prompts the models to propose suitable names when no existing correspondence exists and uses the generated values (which may or may not deviate from the user's inputs).

3.2 Models for Table & Column Mapping

Our system uses two Llama 3 [17] models that are fine-tuned for the tasks of table and column mapping.

Datasets. To fine-tune models for these tasks, we created a dataset out of all 52 databases of the BIRD benchmark [14], 151 databases from the Spider challenge [19], and a sample of 99 tables from WikiDBs [18]. We chose this number of different databases to prevent overfitting. We sampled from the three different datasets to utilize the different naming conventions, ranging from manually adjusted to very generic ones. We keep ten databases from each source for testing purposes. For all other databases, we computed the corresponding insert queries to populate the database to the given state and built two variants of fine-tuning datasets, where we removed the table or column information, respectively, for a uniform sample of 50 % of the queries.

Moreover, we identified three different scenarios for such a query execution: (1) The relevant table/columns names(s) are not in the database. (2) The relevant table/columns names(s) are in the database with the same name. (3) The relevant table/columns names are in the database with an alternative name. For each insert query we created a corresponding database state by selecting (random) subsets of tables and columns or introducing (LLM-generated) synonyms. We balanced these states so that the three scenarios are equally distributed across the dataset.

Fine-Tuning. Both models were fine-tuned using the datasets mentioned before. For efficient fine-tuning of the two models, we opted for the version with eight billion parameters and additionally applied QLoRA [8]. Using a data collator that restricted the fine-tuning to the desired output and not the prompt itself, we were able to further improve the prediction quality.

Prompts. The prompt contains an introduction, the insert query, and the current database state with some example rows for each table, formatted in CSV format. Our experiments show that directly

asking this model to come up with a new table name in case no suitable match is found performs better than generating a placeholder and leaving the name generation task to a second model.

3.3 Optimization Operations

Once more data was inserted, it may turn out that some decisions regarding schema alterations or data placement were not ideal (as explained above). Therefore, the system needs a component that can correct these decisions later. It needs to detect suboptimal parts of the schema or wrongly placed entries and try to improve them. These maintenance/optimization tasks should be run regularly, e.g., when the system, in particular the GPU, is idle. For this paper, we implemented and evaluated one of these operations, which we now describe in detail. Afterwards, we discuss further possible operations.

Combining Tables. Our analysis shows that records of the same type are often placed in different tables (e.g., when the query wording deviates too strongly from the schema). Those tables should be combined into a single one as part of the maintenance operations. Tables can be combined if their schemas are compatible, so candidates can be identified using established schema-matching methods. Schema-based, instance-based, or combination approaches are possible. Intuitively, this requires comparing all pairs of tables and checking whether the similarity exceeds a given threshold. In practice, the candidate pool can first be restricted to table pairs with a similar number of attributes. The remaining candidates must then be compared on a semantic level. As we have shown in a previous paper on schema matching [12], this semantic comparison can again be done in two stages to keep computation cost low, first discarding even more candidates and then finally carefully checking the remaining ones using all available information (i.e., schema and instances already in the database).

Our approach combines schema-based and instance-based matching. Starting with all pairs of tables, the candidate pool is first decreased using a schema-based algorithm. This algorithm uses embeddings of table and column names and keeps the best-matching pairs of tables (the cutoff can be chosen based on available computing capacity at the time of optimization). These remaining pairs are then compared using an instance-based approach that uses the average embedding of the most common values in each column. One option is now to only merge tables if their schemas are homomorphic and differ only in naming, while another option is to accept partial matches, too and mark a pair for merging if the ratio of matching columns is over a defined threshold. Here, a low threshold leads to more tables being merged (and might cause these merged tables to have a bunch of additional columns that are only relevant for parts of the data), while a high threshold leads to a more conservative behavior but might leave tables unmerged even though they differ only slightly. The choice of that hyperparameter again depends on the use case.

The merge is then performed by inserting the rows of the smaller table into the larger one using the column mapping step of the general insertion algorithm. As a variant, moving candidates to the new table could rely entirely on the potential column mappings identified from the instances. Or the optimization step could only be used to find candidates, but the task of moving them could again be handed over to the semantic capabilities of the system's insertion logic by simply rerunning the insertions for these candidates.

Further Optimization Operations. The system might place entries in the wrong table, e.g., due to missing or overly generic table and column names. An example for this can be seen in Figure 3, where the system was not able to differentiate between last names and movie titles, as well as birthdates and release dates, just based on the structure of the values. Later, the system created a table *directors* that better fit the data. A strong sign of such faulty placements is the presence of empty attributes. Another class of optimization operations should therefore try to identify records that do not belong together and move them to different tables. An optimization operation could therefore try to divide the records into two or more groups that each have a list of attributes that the other groups lack (see the wide schema in the figure for an example). Once such candidates are identified, they can either be remapped individually using the existing insertion logic or moved as groups, e.g., using the schema matching approach described above.

Another issue is the presence of tables where the same information is spread across attributes for different rows. Here, approaches to unify those attributes are needed. A starting point can again be comparing semantic representations of multiple instances.

Finally, another interesting class of optimization operations is approaches that do not try to improve the structure, but the names used for tables and columns. A starting point would be to prompt an LLM for new names using instances of a given table (or a sample of them), leading to better names than the predictions in the original insertion logic, which have to find a suitable column name based on a single instance only.

4 What is a good schema?

How can a system decide where to put the data? And how can we measure whether this decision was a good one? Trivial approaches would be storing everything in a very broad table (ignoring table information and using strict string matches for the columns, or enumerating them) or creating a new table for every insert that lacks table information. It seems obvious that the resulting schemas from these approaches will be neither helpful for manual exploration nor for automated analyses. However, in general, deciding whether a schema is reasonable is far from trivial. Even schemas widely considered suboptimal (e.g., storing two concepts in the same table with many NULL values) can still be usable for analysis and discovery. In most cases, however, related items should be stored in the same table, while unrelated items should be stored in different tables. As a result, tables should only contain many NULL values if the inserted data already had them, but not because the columns of different inserts strongly varied. Values with different semantics should not be mapped to the same column, and the table and column names should reflect the contents and be understandable to the system's target audience. These properties should be achieved on a semantic level, even when users use different words for the same concepts in their inserts (or the same word for different concepts in others). Furthermore, expectations regarding quality can vary greatly across use cases, and so can the waiting times and inference costs users are willing to accept.

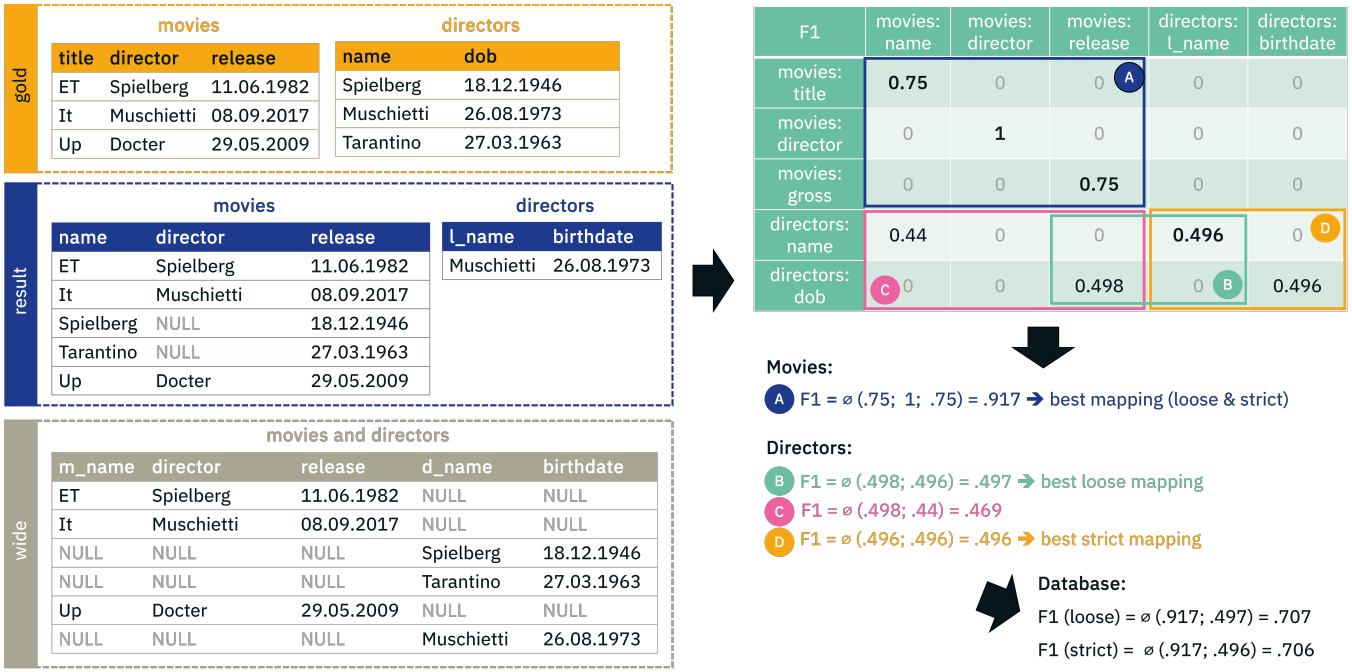

Figure 3: Computation of the database F1 score. For a given gold schema and the result from running the system, the computation is shown. First, the F1 score for all pairs of gold and result columns are computed. Then, the best mappings for each gold table are computed. For *movies*, only correspondences in the result table with the same name *movies* are found (A). The F1 score is hence the average of the F1 scores of the three columns. For *directors*, mappings in *movies* and *directors* are found. The combination of mapping names to *l_name* and *dob* (B) to *release* would yield the highest average F1 score, therefore, this is the loose F1 score. In a strict setting, all mappings need to be in the same result table (only mappings C and D) from which D is selected since it has a slightly higher F1 score. The scores for the tables are then averaged again to get an F1 score for the entire database. Additionally, the figure shows a wide schema variant that contains all data in one single table. That schema would get a perfect F1 score, showing that looking at this score is not sufficient, but one should take the sparsity (number of NULL cells) into account, too, when rating the schema.

4.1 Database F1 Score

There are a lot of schemas for the same use case that are equally valid. First, this covers all homomorphic variants of the same structure with different but semantically plausible names for tables and columns. Second, there are often different ways to distribute the data into different tables. Combined again with semantically exchangable names, this increases the number of possibilities even further. To anyhow measure the quality of the schema based on a given gold schema, we thus propose to check how well the data can be accessed. Two schemas should be considered equivalent in this regard if accessing the data requires the same effort (i.e., values for different attributes in the first schema are not combined into a single attribute in the second, nor are they spread across relations). To quantify the equivalence of two schemas in this regard, we propose a way to transfer the well-known F1 score to entire databases. This can then be used to compare the generated schema with a known gold schema for this setting.

The intuition behind the metric is to first quantify the equivalence between columns in the gold standard database and those in the system-generated database, and then generalize this information across the entire database. Equivalence of two columns can be expressed in terms of recall and precision (measuring the fractions of expected and unexpected values present). Recall and precision can then be used to calculate F1 scores for all possible column combinations and identify the best match. This again can then be used to capture how well distinct concepts were placed in different tables.

The exact computation works as follows. An example can be found in Figure 3.

(1) For a given pair of a gold column c_G and a column in the result c_R compute precision

$$p = \frac{|\forall x \in c_R : x \in c_G|}{|c_R|}$$

recall

$$r = \frac{|\forall x \in c_R : x \in c_G|}{|c_G|}$$

and F1 score

$$F1_{c_G c_R} = \frac{2 \cdot p \cdot r}{p + r}$$

NULL values are ignored.

(2) From all possible combinations $c_G \times c_R$ select the one column c_R with the highest F1 score as correspondence for each of the

columns c_G from the gold database ($m_{c_G c_R}$). This measures how well this column is represented in the resulting schema (independent of its relations).

(3) For all columns c of a given gold table t_G, these values can now be averaged to get a score for each gold table

$$F1_t = avg\left(\forall c \in t_G : F1_{m_c}\right)$$

and averaged again to get one combined score for the whole generated database

$$F1_{loose} = avg\left(\forall t \in G : F1_t\right)$$

Doing this aggregation in two levels ensures all gold tables contribute the same to the overall score, regardless of the number of columns in the relation.

(4) The loose F1 score does not consider whether the assumed correspondences of multiple columns from the same gold table are in the same or different resulting tables. To better capture relations, we can compute a strict version of the F1 score. Here, the mappings between gold and resulting columns of a given gold table must all be in the same result table, still maximizing the resulting average F1 score. These strict table scores are then again averaged across all tables to obtain a strict database score.

4.2 Avoiding Wide Tables & Sparsity

As shown in Figure 3, a wide table that contains the attributes of multiple concepts next to each other can reach a perfect score. This is reasonable, since it still allows for retrieving the same data (by filtering for `NOT NULL` on one of the relevant attributes). Nevertheless, this is often not desired and might hinder browsing the data or lead to inefficient storage.

We therefore suggest considering sparsity as a second metric. It measures the number of additional `NULL` values introduced by the system. Low sparsity usually indicates well-divided tables. However, the system should not solely aim to place the values with the goal of optimizing sparsity. An example where this would lead to the wrong behavior is a *movie* table that already contains a column for *release date*. If there is an insertion for a movie whose release date has not yet been published, the system should still insert the entry into the *movie* table rather than creating a new one with all attributes except the release date. To correctly consider those cases during evaluation, sparsity must always be evaluated with respect to the sparsity of the gold standard tables.

Our experiments show that the trends of the two metrics might indeed diverge, underlining the importance of measuring both covered aspects. Having two different metrics instead of additionally incorporating the NULL cells into the F1 score makes it easier to detect whether a component like an optimization operation trades one aspect (e.g., low fraction of records, indicated by good F1 scores) for another (e.g., very wide tables, indicated by a high sparsity).

5 Initial Evaluation

In this section, we present an initial evaluation of our approach and discuss the directions it indicates for the next research steps, which will be outlined in the following section.

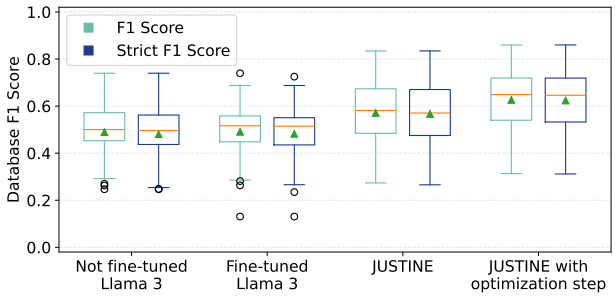

Figure 4: Average F1 scores (loose & strict) for running our system on the evaluation dataset with the modification settings described in Section 5. It can be seen that both fine-tuning the models and using our insertion flow, rather than merely prompting the models, lead to improvements over the base Llama 3 model, and that the optimization step further improves the approach. Values between 0 and 1, higher is better.

Setup & Dataset. We test our system on thirty scenarios based on ten databases from each BIRD [14], the Spider challenge [19], WikiDBs [18], and Spider 2.0[1] [13], for which we computed the insertion queries to produce the database state. We can now modify these queries (i.e., remove table or column information, and/or replace them with synonyms) to test how well the system can create a suitable schema from this information.

The system's exact performance depends on the modifications. We report the results for the following setting as one representative example: 50 % table names removed, 25 % synonyms for table names, 20 % column names removed, and 30 % column synonyms. For the table selection, we set the parameters to use three candidate mappings of which one is always fuzzy string matching and the other two are LLM-based.

Rating the Insertion Logic Quality. The resulting F1 scores can be found in Figure 4. We compare the insertion approach described in this paper to using only fine-tuned models for table and column prediction, without the heuristic components, and to predicting with a Llama 3 model without fine-tuning. The average database F1 score without fine-tuning is 0.490 ± 0.12. Fine-Tuning only marginally boosts the F1 score (0.491 ± 0.13) but decreases the dispersion. A substantial effect is observed when combining the fine-tuned model with our complex insertion logic to form JUSTINE, which reaches 0.571 ± 0.13 and even 0.626 ± 0.12 when the optimization step is applied at the end. The strict F1 scores only differ slightly from the corresponding loose ones. Thus, it can be seen that all aspects (the models, the overall insertion logic, and the optimization step) contribute to creating a better schema and placing the insertions correctly. Another piece of evidence for that can be found when inspecting the internal behavior of JUSTINE. In about two-thirds of the insertions, the system uses a mapping suggested by the LLM; in the remaining cases, the string matching heuristic provides the best mapping. This again underlines the importance of all components.

[1] Spider 2.0 is particularly interesting here, since it is a complex task that was published after the training of the base model and was also not included in our fine-tuning.

Inference Times & Interactivity. If neither table nor column information matches the existing schema exactly, the insertion logic requires one prediction for the table matching and one for each column. To boost the quality, the process has to be repeated for all mapping attempts that do not work purely heuristically. Therefore, we use two eight-billion-parameter models to keep the inference times at an acceptable level. On a NVIDIA DGX system with A100 GPUs (40 GB GPU memory), the whole insertion process usually takes $2 - 4s$ (using a single GPU), but can be a magnitude higher when the database contains very long strings (negatively affecting the token count of each prompt). For larger models, inference time can go up substantially. We tested the system with a more recent Llama 3.3 model and got an average performance of up to 0.2 better than when using the Llama 3 based models, but due to the 70 billion parameters, the inference time increased by a factor of $5 - 10$, rendering it too slow for many applications. Yet, in scenarios where the required compute capacity is available, this can further boost quality.

Typical Issues. We have shown that our approach outperforms solely relying on LLMs for this task. However, it can also be seen that there is still room for improvement. A manual analysis of the resulting schemas shows two primary issues: First, the schemas are often quite wide, since multiple different types that share a few attributes with the same name end up in the same table. This is underlined by an average sparsity of 0.431 ± 0.12 of JUSTINE before the optimization (compared to only 0.066 ± 0.10 in the gold schemas). This can be improved by using different weights for the heuristic to balance between existing and new tables, and by adding maintenance steps that split these tables.

Second, the mapping between generated and user-specified names does not always work. As a result, records of the same type might end in different tables (or, less often, the same attribute might be split into different target attributes). Changing the weight of the mapping selection heuristic affects this, too. Yet, since for one of the issues it is relevant to incentivize more tables and for the other there should be fewer, this is difficult to balance. It therefore seems more promising to tackle these issues using the table combination operation based on schema matching. In the tested scenario, the gold standard contains, 6.2 ± 3.8 tables, while JUSTINE produces 18 ± 9.3 tables, of which 10.7 ± 8.1 remain after the optimization.

6 Research Roadmap

The system presented in this paper is a first step towards self-organizing data schemas. We hope that it fosters discussions on how such systems should work, which scenarios should be supported, and which properties and guarantees are needed—at the workshop and beyond. As a basis for such discussions, we outline the roadmap for the next steps we envision in this section.

Batch Insertion. The amount of information in a single insertion query is limited. This can make it difficult to infer the correct semantics. Take, e.g., the number 42. Even when knowing that it should be inserted in a table about a person, there are different plausible interpretations. Only when there is a second record with a value of 81 for the same attribute, it becomes obvious that this property is probably the age and not an (European) shoe size. Hence, as one future direction, we will investigate ways to group incoming inserts and then jointly map them, giving the insertion logic more information to work with. The quality gain of the optimization through the identification of corresponding tables, which is based on multiple instances at once, suggest that this direction is promising.

Optimizing the Schema. As we saw in the initial evaluation, the maintenance operations will play an important role. As we discussed in Section 3.3, we will add and evaluate additional steps, e.g., to split tables or to rename them. This covers investigating ways to adjust whole groups of records at once (which is connected to the group inserts discussed before).

Protecting the Schema. Currently, we treat all queries the same, allowing them to have the same effects on the schema. It would, however, be interesting, to investigate how queries that are incomplete or otherwise of low quality could automatically be identified. These queries might then be handled differently (e.g., preventing the addition of new tables or columns to match their data).

When applying the system in a real-world use-case that starts with an existing schema, it might furthermore be relevant to allow manually fixating and thus *protecting* some parts of the schema, allowing schema alterations only for other tables. In addition to manual annotations such protection rules could also be derived from user defined functions and stored procedures.

Going Beyond Direct Inserts and 1:1 Mappings. Another very interesting direction to improve the schema quality will be going beyond directly mapping the inserts. This will be relevant since we cannot expect non-experts to know about normalization and why it is important. An extended insertion logic should be able to split the data across multiple tables and introduce suitable primary and foreign keys. Furthermore, it should be able to split values that are currently not normalized (e.g., because the users entered both first and last name in the same field). Finally, adjusting the value representations (e.g., timestamp formatting, categorical encoding, value ranges) will be crucial for high-quality databases when the inputs come from different users or sources.

Leveraging Read Operations. Until now, we have inferred the relevant schema only from the insertions. But for many use cases, it will not only be relevant how the data is stored, but also what schema the users working with the stored data expect. Therefore, we will investigate how a self-organizing schema could adjust based on SELECT queries, too. This might be particularly interesting when combined with the direction mentioned before: Knowing what parts of information are needed and in which combination might give hints on how to distribute the information across different tables, and where records need to be joined. A self-organizing schema could use this to automatically create (materialized) views that allow accessing the data in the way inferred, or even to adjust the schema directly. Our next step will be to incorporate SELECT information into optimization operations.

Improved Usage of LLMs. Which LLMs are used (and how they are prompted) plays an important role for the resulting schema quality, as the initial evaluation shows. Larger models usually generate better predictions, but at a higher inference cost. In the face of the rapid development of large language models, the choice of suitable

models must be re-evaluated regularly. Newer and larger models might, e.g., have higher costs for running a single prediction, but that prediction might be able to suggest multiple mappings at the same time, compared to the one-by-one style currently used. A second factor for the inference cost is the prompt that is handled by the LLM. Here, we want to further investigate how we can make best use of the context, i.e., what should be included and how it is encoded. One specific direction we want to investigate here is the active selection of example values for the attributes in a way that it maximizes the relevance for the LLM for handling a given insertion query. Furthermore, we want to investigate ways to automatically infer the types of the input values and leverage them for the mapping.

Leveraging Domain Knowledge. As a final direction, it will be interesting to try incorporating additional domain knowledge if available. With the amazing zero-shot possibilities of LLMs and work on extending the possible context size far beyond what earlier models could handle [3], it is possible to provide large amounts of background information. This could, e.g., be used to supply the mapping models with technical documentation about the domain, to even better infer the semantics of the given inputs.

Beyond the mentioned research directions, there are lots of other interesting topics, like the extension from textual attributes to multi-modality, or a study on the interactive use of such a system.

7 Related work

In this section, we will analyze related work. To the best of our knowledge, no system provides self-organizing schemas yet.

Cafarella et al. [7] propose `TGen` to create a set of tables from unstructured tuples. Unfortunately, this approach requires the full dataset to be present to discover the schema and cannot iteratively refine it for new inserts. Spoth et al. [16] present an approach called *Adaptive Schema Databases*, which shares the motivation with our approach. Here, the focus is on discovering a useful schema at read time, while the data is stored in an unstructured or semi-structured format. The same holds for many approaches for table discovery in data lakes [1, 9], which require the full data to be present already and do not support continuously updating the schema. Additionally, the focus of these approaches is often not to organize all the data, but rather to discover some relevant information from large amounts of records and present it in a well-defined format [5]. Other approaches like `Constance` by Hai et al. [10] require metadata for schema discovery that may not be available.

The question of whether two schemata are equivalent was already addressed in early database research [4, 6]. Unfortunately, these approaches can only answer whether two schemas are equivalent, but do not provide a means to quantify the degree of overlap. Research on schema matching [2, 15] can be helpful here. Existing work is not applicable to the matching task itself in most scenarios: Schema-based approaches require complete schema information on both sides, but cases where table or column names are missing are challenging scenarios. Instance-based approaches usually require a large amount of instances and can hence not be applied to single insertion queries. Schema matching, however, can be relevant for measuring overlap between schemas. This is relevant for the evaluation of systems for self-organizing schemas, and particularly for schema-optimization operations, since it can help identify tables that should be combined.

8 Conclusion

In this paper, we presented a prototypical system called JUSTINE for self-organizing data schemas, which can be particularly useful for scenarios where the structure of the information to store is not known upfront or changes over time, and can generally relieve developers of the overhead of designing suitable schemas. It works on top of existing relational databases and uses a multi-step approach to determine target tables and attributes for inserts that may lack table or attribute names, and can use synonyms for all structural information. The approach adds new tables and columns where needed. To ensure a high-quality schema, our system provides an optimiziation operation to periodically fix mistakes and optimize the schema. We introduced the metrics *database F1 score* and *sparsity* to measure the quality of the schema with regard to accessing and handling the data afterwards. The approach presented in this paper should be seen as a first step towards self-organizing data schemas, therefore we discussed the next steps in an extensive research roadmap.

Acknowledgments

This research was partially funded by the German Federal Ministry of Research, Technology and Space (BMBFTR) within the "The Future of Value Creation – Research on Production, Services and Work" program (funding number 02L19C150), by the BMFTR and the state of Hesse as part of the NHR program, by the LOEWE Spitzenprofessur of the state of Hesse (III 5-519/05.00.003-(0005)), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC3057/1 "Reasonable Artificial Intelligence", Project No. 533677015). We also thank hessian.AI for their support. The authors are responsible for the content of this publication.

References

[1] Nour Alhammad, Alex Bogatu, and Norman W Paton. 2022. Towards Schema Inference for Data Lakes. arXiv:2206.03881 [cs.DB] https://arxiv.org/abs/2206.03881

[2] Ali A Alwan, Azlin Nordin, Mogahed Alzeber, and Abedallah Zaid Abualkishik. 2017. A survey of schema matching research using database schemas and instances. *International Journal of Advanced Computer Science and Applications* 8, 10 (2017).

[3] Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, Jian-Guang Lou, and Weizhu Chen. 2024. Make Your LLM Fully Utilize the Context. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 62160–62188. doi:10.52202/079017-1986

[4] Catriel Beeri, Alberto O. Mendelzon, Yehoshua Sagiv, and Jeffrey D. Ullman. 1979. Equivalence of relational database schemes. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing* (Atlanta, Georgia, USA) *(STOC '79)*. Association for Computing Machinery, New York, NY, USA, 319–329. doi:10.1145/800135.804424

[5] Alex Bogatu, Alvaro A. A. Fernandes, Norman W. Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 709–720. doi:10.1109/ICDE48307.2020.00067

[6] Sheldon A. Borkin. 1978. Data model equivalence. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4* (West Berlin, Germany) *(VLDB '78)*. VLDB Endowment, 526–534.

[7] Michael J. Cafarella, Dan Suciu, and Oren Etzioni. 2007. Navigating Extracted Data with Schema Discovery. In *International Workshop on the Web and Databases*. https://api.semanticscholar.org/CorpusID:11839029

[8] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. *arXiv preprint arXiv:2305.14314* (2023).

[9] Grace Fan, Jin Wang, Yuliang Li, and Renée J. Miller. 2023. Table Discovery in Data Lakes: State-of-the-art and Future Directions. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 69–75. doi:10.1145/3555041.3589409

[10] Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An Intelligent Data Lake System. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2097–2100. doi:10.1145/2882903.2899389

[11] Benjamin Hättasch, Leon Krüger, and Carsten Binnig. 2025. JUSTINE (JUST-INsert Engine): Demonstrating Self-Organizing Data Schemas. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 5283–5286. doi:10.14778/3750601.3750652

[12] Benjamin Hättasch, Michael Truong-Ngoc, Andreas Schmidt, and Carsten Binnig. 2022. It's AI Match: A Two-Step Approach for Schema Matching Using Embeddings. arXiv:2203.04366 [cs.DB] https://arxiv.org/abs/2203.04366

[13] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763* (2024).

[14] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. arXiv:2305.03111 [cs] doi:10.48550/arXiv.2305.03111

[15] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *The VLDB Journal* 10, 4 (Dec. 2001), 334–350. doi:10.1007/s007780100057

[16] William Spoth, Bahareh Sadat Arab, Eric S Chan, Dieter Gawlick, Adel Ghoneimy, Boris Glavic, Beda Hammerschmidt, Oliver Kennedy, Seokki Lee, Zhen Hua Liu, et al. 2017. Adaptive schema databases. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

[17] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs] doi:10.48550/arXiv.2302.13971

[18] 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

[19] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 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* (Brussels, Belgium, 2018). Association for Computational Linguistics, 3911–3921.