JUSTINE (JUST-INsert Engine): Demonstrating Self-organizing Data Schemas

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

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

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

ABSTRACT

Relational databases are great for data analysis and exploration, but require a carefully crafted schema, which causes high manual overhead. Moreover, 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. We therefore advocate for a new class of database systems that organize the data in a schema autonomously when it is inserted schemalessly by users. Such databases should thus be able to store data semantically meaningful but without requiring the user to design a schema, neither upfront during setup nor when an insert is executed. In this demo, we showcase JUSTINE, which is a first implementation of this new class of database systems that can automatically adjust a database schema based on input queries. Our showcase features both (1) an interactive mode where attendees can enter their own data as well as (2) the execution of a full workload where users can see how the database schema evolves during batch execution. The workload can be customized by changing different parameters.

PVLDB Reference Format:
Benjamin Hättasch, Leon Krüger, and Carsten Binnig. JUSTINE (JUST-INsert Engine): Demonstrating Self-organizing Data Schemas. PVLDB, 18(12): 5283 - 5286, 2025.
doi:10.14778/3750601.3750652

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://link.tuda.systems/JUSTINE.

1 INTRODUCTION

The need for structured data. 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. Moreover, it might influence what information can be stored and thus have great effects on the decisions made based on the data later.

Take, for example, the government of a small city that asks its citizens to report any problems they notice, such as broken streetlights, trash in the park, or unfavorable traffic conditions. They want to store these incident reports, but for each kind different fields are relevant. Additionally, they might already have existing structured information (e.g., maintenance logs or the schedule of garbage collections) that they want to link. All these inputs need to be stored differently to capitalize on the specific information they contain. Moreover, the contents and structures of all these inputs might change over time (e.g., suddenly it becomes relevant to report where exactly electric scooters are blocking the sidewalk and who owns them). Currently, there are two general options for storing and accessing such data:

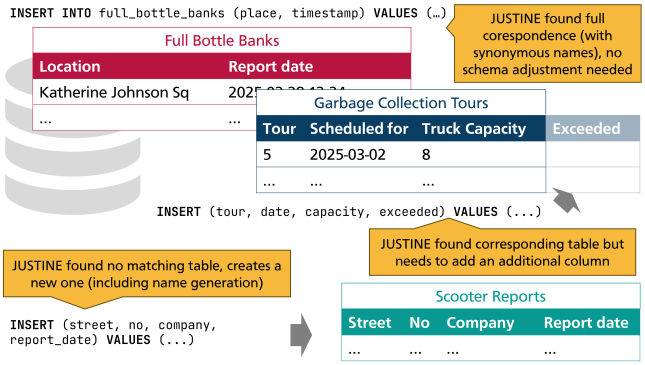

Figure 1: Functionality of JUSTINE. The data is stored in a relational database. When a new insert query is executed, JUSTINE tries to map it to an existing table (which might use different terms for table or column names). If there is no suitable table, JUSTINE creates a new one and then inserts the data. Queries might be incomplete, lacking table or column names. JUSTINE will try to map them anyhow. Moreover, queries might contain columns not considered in the schema yet. JUSTINE will add such columns automatically if needed.

Relational databases provide structure but cause overhead. Traditional relational database systems require the manual design of a suitable database schema upfront. This requires both knowledge about the domain and databases. In the example from above, one would need to decide whether the different kinds of reports should be stored in multiple tables or in a single one and which fields (e.g., location information, report date, comment) are needed. For reports like full bottle banks, it might be sufficient to request a generic location from the citizens, like the square where it is located. In contrast, for broken streetlights, storing a finer location identifier (possibly consisting of the street name, closest house number, and street side) is required. Additionally, the schema must

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750652

be manually adapted when additional information with deviating attributes such as blocked sidewalks (where storing the owning company would allow to automatically request them to remove the electric scooters) should be stored. Everything that is not covered by that schema upfront cannot be stored.[1] Thus, a relational database requires continuous remodeling in scenarios with changing entities and attributes. Their advantage, however, is the possibility of efficiently querying them to make sense of the saved data and gain new information. The strict schema makes it easy and cost-efficient to navigate and browse it in exploratory scenarios.

Are schemaless databases better? 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. In our example, government employees processing the reports might find it difficult to get an overview, group related reports, or concentrate on relevant information. When processing the data automatically, there is no guarantee that all relevant attributes are present, and queries (e.g., to identify which scooter companies should be notified) might fail. Hence, while schemaless databases allow storing data without schema design in the first place, that effort will only be postponed for many use cases.

Towards self-organizing schemas. Therefore, we propose working towards databases with self-organizing data schemas that offer 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. As a first step towards that vision, we demonstrate JUSTINE, which allows users to perform vague inserts without the need to specify a schema upfront. That is relevant since allowing to omit 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 the 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.

Our system needs to support different situations (see Figure 1 for an example): Queries might look complete, but use deviating terms for table and column names (e.g., *timestamp* but the database column is called *report_date*). JUSTINE needs to map them to the existing schema. Queries might contain names for table and columns, but no direct correspondence exists in the existing schema. In that case, the schema has to be updated by adding the missing columns or even entirely new tables. Finally, when users are unaware of whether there is already a suitable structure and neither know what a good structure would look like, they could create inserts that omit table and/or column names. Again, JUSTINE 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 more coherent storage of semantically related concepts. This makes both manual exploration and automated processing easier.

How does JUSTINE work? JUSTINE stores the data in relations and changes these relations based on the insert queries if necessary. The system needs to decide autonomously in which table to place the values (either determining the correct existing one or propose the name for a new table) and then to map the values to existing columns or add new columns. We use two large language models for this semantic bridging between inputs and schema, which we fine-tuned for the tasks of table and column mapping. In addition to the query itself, they 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. JUSTINE then needs to perform necessary adjustments to the schema (adding columns and/or tables) and can execute an adapted version of the insert query afterwards.

JUSTINE can be used to match the incoming queries to an existing schema (by adjusting it if necessary) or can even start with an empty database and completely derive the schema from the inputs. We showcase that approach in an interactive web application where users can enter their own queries and see how the system handles them, or can execute workloads with customized options like the degree of missing information and see how the schema evolves.

Video and Artifacts. A video showing our approach in an interactive demo can be found at https://link.tuda.systems/JUSTINE-video. We publish code, datasets, and the fine-tuned models as open source at https://link.tuda.systems/JUSTINE.

2 SYSTEM DESIGN OF JUSTINE

How should a good resulting schema look like? The central functionality of our system is determining where to put the data. Trivial approaches would be storing everything in a very broad table (ignoring the table information and using strict string matches for the columns or enumerating them) or creating a new table for every insert that lacks the table information. It seems obvious that the resulting schemas from these approaches will be neither helpful for manual exploration nor automated analyses. However, in general, deciding whether a schema is reasonable is far from trivial. Even schemas widely considered suboptimal (e.g., two concepts are stored in the same table with a lot of NULL values) can still be usable for analysis and discovery.

We aim for schemas where related items are stored in the same table while unrelated ones are in different tables. Thus, we try to avoid adding columns that will be NULL for most rows (in case these NULL values were not already contained in the original inserts). On the other hand, values with different semantics should not be mapped to the same column. 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 the terms used in the inputs vary greatly. All that needs to be achieved with reasonable computation effort.

Insertion Workflow. Our approach works on top of a relational database. Some queries can be executed directly. If, however, table or column names are missing or do not match the existing schema in the database, a two-stage mapping approach will be invoked. Both

[1] Except when using generic tables that resemble key-value stores—which then do not offer the advantages of strict schemas.

steps are based on Llama 3 models [3] that we fine-tuned for this task to provide the necessary semantic understanding. Heuristic approaches (e.g., (fuzzy) string matching) can augment them and help to reduce computation load for trivial cases (i.e., where names clearly identify the correct tables and/or columns).

First, JUSTINE invokes a model for *table prediction* to find a corresponding table for the insert query. In case there is no existing suitable match, the model outputs the name for a new table instead. Based on this, the system can augment the query and (if there are column names) try again to execute it. If that fails or the column names are missing, the system invokes another model for *column prediction* that operates analogously to the table prediction. JUSTINE then performs the schema alterations (if necessary) and executes the adjusted query.

To boost the overall quality, the *table prediction* model can be instructed to output multiple (e.g., 3) candidates, from which the best can then be selected by requesting mappings from the *column prediction* model and then ranking them. For this ranking, the number of new columns that would be needed when inserting the data into this table, and whether it was at all possible to create a unique mapping can be used as heuristics. Whether the costs for this additional inference are appropriate depends on the use case and are therefore a setting for the users deploying the approach. In an interactive setting (see Section 3.1), this allows to produce 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 if lists are short enough.

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 come up with suitable names whenever there is no existing correspondence and uses the generated values (which may or may not deviate from the user's inputs) for altering the schema.

Datasets. To fine-tune models for these tasks, we created a dataset out of all 52 databases of the BIRD benchmark [2], 151 databases from the Spider challenge [5] and a sample of 99 tables from WikiDBs [4]. 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. For all these 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 synonyms. We balanced these states in a way that the three scenarios are equally distributed across the dataset.

Fine-Tuning. The *table prediction* model is based on Llama 3 [3], fine-tuned with the before-mentioned dataset. For efficient fine-tuning, we opted for the version with eight billion parameters and additionally applied QLoRA [1]. The prompt contains an introduction, the insert query, and the current database state with some example rows for each table, formatted in CSV format. 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. 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.

The *column prediction* model was fine-tuned analogously. We tested both prompting for each column mapping individually and requesting all mappings of an insert query at once. In our experiments, the latter did not only provide a higher prediction accuracy, but was also much better in preventing mapping multiple values to the same column.

3 DEMONSTRATION SCENARIO

At our demo, attendees can interactively try out how JUSTINE can support a user in adding data to a database interactively. Furthermore, it offers a mode to watch how the schema evolves when a bigger workload of insert queries is performed sequentially. The demo is implemented as a web application. We make this web interface, the underlying insertion logic, the fine-tuned models, and the datasets publicly available at https://link.tuda.systems/JUSTINE.

3.1 Interactive Inserting

A user starts by either selecting an existing database or choosing to work on an empty one. They are then greeted by a page that allows them to input (single or multiple) queries or to upload a file with SQL-like[2] insert queries. Once they submit a query, JUSTINE will check whether the query can be directly executed (in case all information is present and matches the existing database schema). In that case, it will directly execute it. Otherwise, it will use the fine-tuned LLMs to propose a suitable table as well as column matchings. The result will be presented to the user, who will be asked to review it. That review view contains the query, the top three table mappings, and a preview of the insert in the currently selected table. Color codings visualize alternative mappings with lower probability. The user might now either directly confirm it (leading to the system persisting the insert in the database) or can choose another table as target. Furthermore, they might manually adjust the column matching by dragging values to other cells. Once they submit the mapping, JUSTINE performs necessary schema updates, executes the adjusted insert query, and updates the data view the user sees. The user can now continue with the next query. JUSTINE will use the recent manually corrected queries as examples for further prompts to the mapping models.

[2]They are not always syntactically correct SQL queries since the table or column information may be missing.

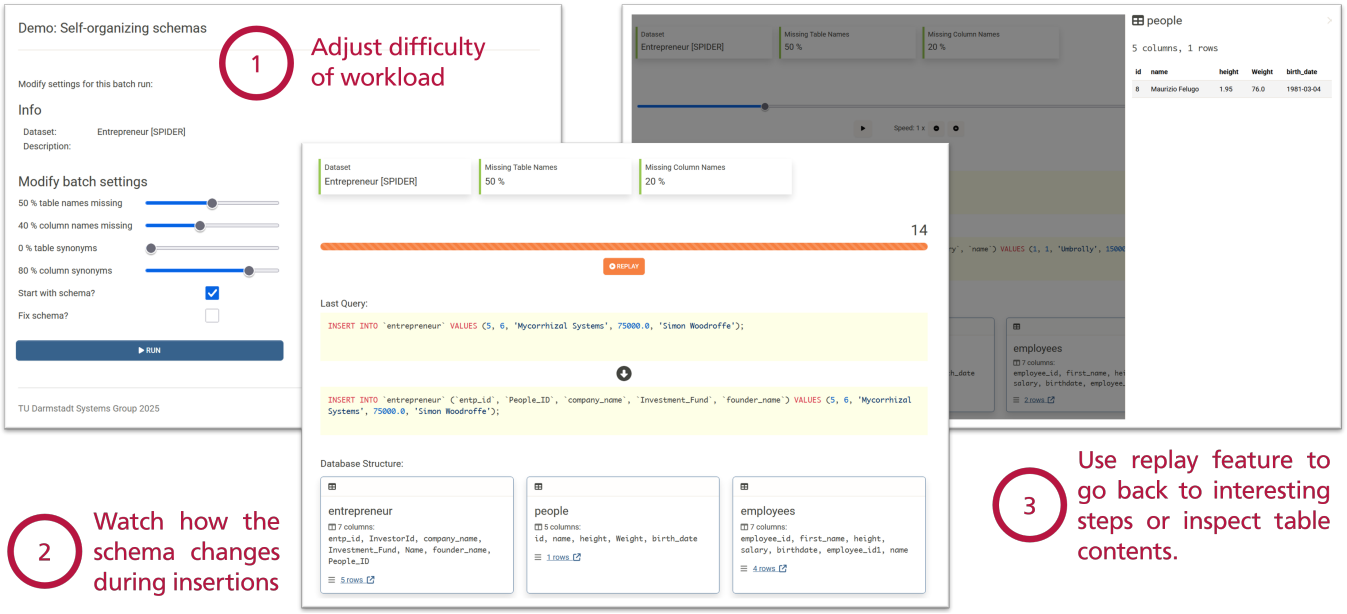

Figure 2: Screenshots of our demo application: Attendees can modify properties of a workflow (e.g., the percentage of inserts without table or column names) and then watch how the schema evolves when these inserts are executed step by step. Afterwards, they can inspect details in a replay mode.

3.2 Batch Mode: Watch Schema evolving

Alternatively, a user can select an existing workload to watch how the database schema evolves over time. Here, they can vary parameters like the number of queries without table or column names, or that synonyms should be used for them. By default, JUSTINE will perform all operations without a predefined schema, but it is possible to start with a database that has the correct schema but is empty, too, to see how well the system can map to that schema depending on the degree of modification of the insert queries. In that case, users can steer whether JUSTINE is allowed to create new tables and columns or whether they should stick to the existing schema. After they select these parameters, the system will adjust the workload accordingly and perform the queries sequentially, showing the database's structure and data after every query (see Figure 2). Finally, users can watch a replay of such a batch insert or inspect the database status after a specific insert.

4 CONCLUSION & FUTURE WORK

In this demo paper, we presented JUSTINE, our approach for handling incomplete insert queries, mapping them to a given database schema, and updating that schema if necessary. Our approach is integrated into an interactive web application. We publish the code and all relevant data as open source.

In the future, we want to extend that system to infer the schema not only from the incoming data but also based on the information needs of those querying it. While our current approach is a first step towards self-organizing schemas and is thus limited to adding new columns and tables, this additional information could be used to also restructure the existing schema (e.g., splitting or merging tables, and renaming them or their attributes) for further optimization.

ACKNOWLEDGMENTS

This research and development project is partially funded by the German Federal Ministry of Education and Research (BMBF) within the "The Future of Value Creation – Research on Production, Services and Work" program (funding number 02L19C150). It has benefited from early-stage funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-3057; funding will begin in 2026. The authors are responsible for the content of this publication.

REFERENCES

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

[2] 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. https://doi.org/10.48550/arXiv.2305.03111 arXiv:2305.03111 [cs]

[3] 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. https://doi.org/10.48550/arXiv.2302.13971 arXiv:2302.13971 [cs]

[4] 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

[5] 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.