TailorSQL: An NL2SQL System Tailored to Your Query Workload

Kapil Vaidya[1*], Jialin Ding[2], Sebastian Kosak[3*], David Kernert[4*], Chuan Lei[2], Xiao Qin[2], Abhinav Tripathy[2], Ramesh Balan[2], Balakrishnan Narayanaswamy[2], Tim Kraska[2]
[1]Parallel Web Systems [2]Amazon Web Services [3]Technical University of Munich [4]STACKIT

ABSTRACT

NL2SQL (natural language to SQL) translates natural language questions into SQL queries, thereby making structured data accessible to non-technical users, serving as the foundation for intelligent data applications. State-of-the-art NL2SQL techniques typically perform translation by retrieving database-specific information, such as the database schema, and invoking a pre-trained large language model (LLM) using the question and retrieved information to generate the SQL query.

However, existing NL2SQL techniques miss a key opportunity which is present in real-world settings: NL2SQL is typically applied on existing databases which have already served many SQL queries in the past. The past query workload implicitly contains information which is helpful for accurate NL2SQL translation and is not apparent from the database schema alone, such as common join paths and the semantics of obscurely-named tables and columns. We introduce TailorSQL, a NL2SQL system that takes advantage of information in the past query workload to improve both the accuracy and latency of translating natural language questions into SQL. By specializing to a given workload, TailorSQL achieves up to 2× improvement in execution accuracy on standardized benchmarks.

VLDB Workshop Reference Format:
Kapil Vaidya, Jialin Ding, Sebastian Kosak, David Kernert, Chuan Lei, Xiao Qin, Abhinav Tripathy, Ramesh Balan, Balakrishnan Narayanaswamy, Tim Kraska. TailorSQL: An NL2SQL System Tailored to Your Query Workload. VLDB 2025 Workshop: Applied AI for Database Systems and Applications (AIDB 2025).

1 INTRODUCTION

NL2SQL translates natural language questions into SQL queries, making data analysis accessible to non-technical users and serving as a foundation for intelligent data applications like smart dashboards and visualizations. For instance, a business owner can simply ask, "What were last month's total sales by product?" and the data application can use the NL2SQL system to generate the corresponding SQL, retrieve the relevant data, and summarize the results. Such possibilities make NL2SQL an important tool to explore.

NL2SQL tools are increasingly being integrated into widely-used commercial database systems [1–3]. Usage of NL2SQL tools typically exhibit two behaviors. First, NL2SQL-generated queries are only a small fraction of the overall query workload that is executed on a database; most SQL queries are still either hand-written or machine-generated according to application logic. Second, users of NL2SQL tools care not only about the accuracy of the output SQL query, but also the latency of invoking the NL2SQL tool.

Recent improvements in NL2SQL technology has been propelled by the advent of large language models (LLMs). What sets LLMs apart from the previous language models is their capacity to generalize to unseen tasks. LLMs achieve this through in-context learning, where examples of the task completion and information relevant to the task are provided within their context. Using this context, the LLM generates the desired output. Capitalizing on these capabilities, LLMs consistently excel in various tasks, including NL2SQL, where they currently lead on two well-known NL2SQL benchmarks: Spider [37] and BIRD [13].

Since NL2SQL aims to output SQL queries that can execute on the user's database, one key step in the NL2SQL pipeline is acquiring database-specific information (e.g., the names of tables which the SQL query should reference) to provide in the LLM's context. Some commercial NL2SQL tools ask the user to manually specify certain information, such as the tables and views to be used in the SQL query [1] or the literals to use in filters [3]. On the other hand, NL2SQL techniques proposed in the research literature typically aim to automatically retrieve the relevant database-specific information without any human intervention [7, 20, 32].

However, one source of database-specific information is consistently ignored or underutilized by existing NL2SQL tools: the logs of past SQL queries which have executed on the database. These logs typically contain queries originating from various applications, including machine learning pipelines, ETL processes, and reporting tools, rather than being limited to those generated by NL2SQL interfaces. Databases often have repetitive workloads with similar queries in historical logs due to overlapping analyses driven by consistent business goals and standard reports. This leads to recurrences of similar patterns across queries, such as frequently-used join paths. As a result, SQL queries generated by NL2SQL systems often match or share components with logged queries.

An Illustrative Example. We illustrate the potential benefit of using past query logs for NL2SQL with an example which is derived from the BIRD benchmark [13]. Fig. 1A depicts a chemistry database containing tables named `atoms`, `cnt`, and `bond`, among others. Notably, the cryptically-named `cnt` table is meant to "connect" the `atoms` and `bond` tables via their respective `id` fields. Assume that a number of SQL queries have already been run on this chemistry database, not necessarily generated from the NL2SQL interface. Given the user's natural language question, an accurate NL2SQL system should produce the desired output SQL query, which joins `atoms` and `bond` via the `cnt` table.

A typical NL2SQL pipeline (Fig. 1B) will invoke an LLM with a prompt that includes the user's question and schema information

Proceedings of the VLDB Endowment. ISSN 2150-8097.

*Work performed while employed at AWS.

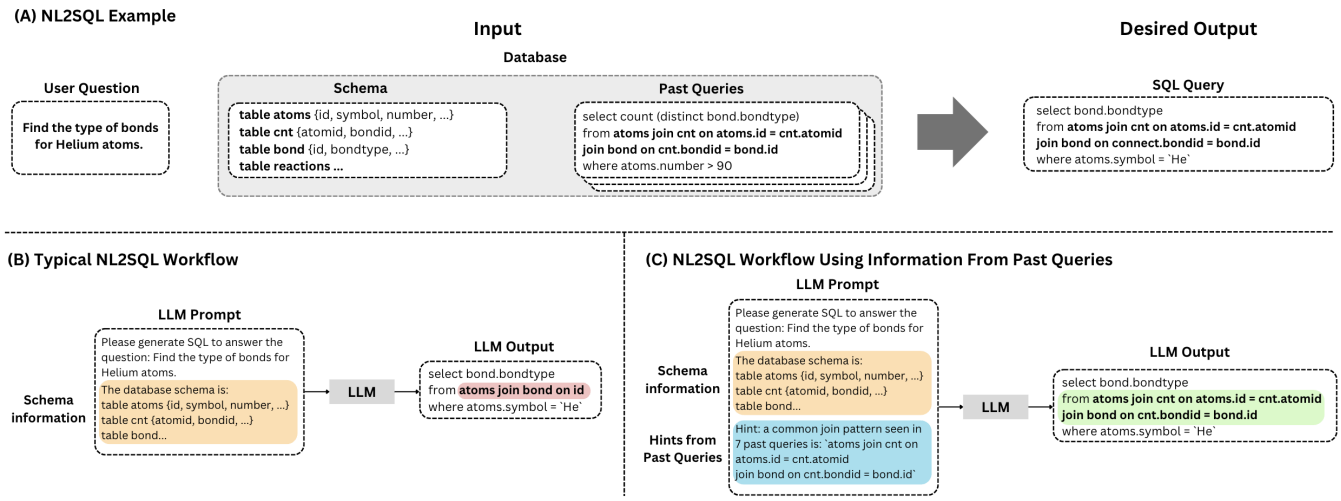

Figure 1: (A) The input to NL2SQL is a user question over a database which has a certain schema and past query workload, and the desired output is a SQL query. (A) The typical NL2SQL workflow will invoke the LLM with a prompt that includes the user question and the database's schema, which may not include enough information to produce an accurate SQL query. (C) Hints extracted from past queries help the LLM produce more accurate queries.

from the database. However, due to the cryptic name of the `cnt` table, the LLM might erroneously choose an incorrect join path: in this example, it joins the `atoms` and `bond` tables on their respective `id` columns. Figuring out the correct join path based solely on table schemas is challenging for the LLM, especially if there is no join path information and tables have no conclusive names.

To mitigate this lack of information, we can take advantage of past queries to augment the information provided in the prompt (Fig. 1C): suppose that the join path between the `atoms`, `cnt`, and `bonds` tables has been observed in multiple past queries, such as the query shown in Fig. 1A. We incorporate this common join path as a *hint* in the LLM prompt, which guides the LLM to generate the correct SQL query. In this example, past queries provide information about common query patterns, which implicitly indicate the appropriate usage of schema items which may otherwise be unclear from inspecting the database schema alone. Note that hints are not directives, i.e., they do not force the LLM to act a certain way, but rather provide the LLM with useful information which the LLM is allowed to ignore.

Following this intuition, we introduce TailorSQL, a NL2SQL system that tailors itself to a given query workload by harnessing the information that is implicitly stored in past queries. TailorSQL improves both the accuracy and latency of NL2SQL with two core ideas.

First, TailorSQL performs an offline analysis of the past query workload and extracts *hints* from past queries (e.g., the hint about the common join path in Fig. 1). Hints provide useful information for accurate NL2SQL translation which is missing from schema information alone. Individual hints are stored as text-based *documents* and can be retrieved into the LLM context if relevant for a given question. TailorSQL also stores schema information in documents.

Second, TailorSQL introduces a workload-specialized framework for retrieving the relevant documents for a given user question, while ignoring irrelevant documents. Retrieval frameworks (see Section 2) are often necessary for real-world NL2SQL pipelines because LLMs are subject to a context limit, which is the maximum amount of input text the LLM can process. Database schemas often comprise thousands of tables and including all of them in the limited prompt context becomes infeasible. Even in cases where the entire schema fits into the prompt context, it is desirable to avoid adding irrelevant information to an LLM's prompt, since doing so will incur higher latency and cost when invoking the LLM. A typical NL2SQL retrieval framework maps user questions and documents into an embedding space, ensuring the proximity of questions to relevant documents; then for a given question, the framework retrieves documents in order of decreasing embedding similarity until the LLM's context is filled. TailorSQL's retrieval framework improves upon the typical framework by using the past query workload to create fine-tuned document embeddings, which improves its ability to retrieve relevant documents while avoiding irrelevant documents. Furthermore, instead of using a single context size limit for all documents, TailorSQL performs an analysis over past queries to allocate a separate context limit for each class of document (i.e., schema vs. hint documents) in order to mitigate class imbalances when performing document retrieval.

Finally, given that TailorSQL specializes to a specific workload, it may perform poorly if the workload distribution changes in the future. To mitigate such performance regressions, we introduce an *abstention policy*, which TailorSQL uses to dynamically decide whether to abstain from using query hints in the NL2SQL pipeline (i.e., whether to fall back to a generic NL2SQL pipeline that does not use TailorSQL's proposed specializations). The abstention policy is based on a multi-armed bandit framework and incorporates user feedback into the decision-making process.

We evaluate TailorSQL across multiple NL2SQL benchmarks, demonstrating up to 2× improvement in execution accuracy compared to baselines that do not take advantage of past queries. Additionally, for the same execution accuracy, TailorSQL improves SQL generation latency by 2-4× compared to the baselines. Furthermore,

TailorSQL does not overfit to historical queries and exhibits robustness against shifts in query distribution. In summary, we make the following key contributions:

- We present TailorSQL, a NL2SQL system that tailors itself to a given database by utilizing database-specific hints extracted from past queries to improve accuracy.
- We demonstrate how TailorSQL tailors its embedding and retrieval procedures to a given database, which improves SQL generation latency without sacrificing accuracy.
- We ensure that TailorSQL is robust to workload distribution shifts using a bandit-based policy to decide when to abstain from utilizing query hints.
- We present an evaluation of TailorSQL's overall performance in terms of accuracy and latency as well as microbenchmarks on its individual components.

2 PRELIMINARIES

In this section we give background on retrieval models and retrieval-augmented generation, which is useful for understanding TailorSQL.

Dense Retrieval Models and Document Retrieval: Dense retrieval models are a type of information retrieval method that efficiently matches text-based queries[1] to text-based documents by encoding queries and documents into dense vectors (commonly referred to as *embeddings*) in a continuous vector space, then retrieving documents whose embeddings are *similar* to a given query embedding. The similarity between a query embedding E_q and a document vector E_d in the embedding space can be measured using various similarity metrics. One popular metric, which we use throughout this paper, is *cosine similarity*, defined as:

$$\text{CosSim}(E_q, E_d) = \frac{E_q \cdot E_d}{\|E_q\| \cdot \|E_d\|} \tag{1}$$

where the numerator denotes the dot product between the query and document embeddings, and the denominator represent their respective Euclidean norms. Cosine similarity varies between -1 (denoting perfect dissimilarity) and 1 (denoting perfect similarity).

Semantically-related queries and documents should ideally have similar embeddings. A common approach for generating embeddings is to use a pretrained sentence transformer such as Sentence-BERT (SBERT) [22], which has been trained on large text corpora specifically for the purpose of generating semantically meaningful embeddings of sentences such that embeddings can be compared using cosine similarity. Here, "sentence" refers to arbitrary pieces of text that can contain multiple English sentences. Therefore, to avoid ambiguity, for the remainder of this paper we use the term *document* instead of "sentence."

Given a query, one can retrieve the top-K most similar documents by computing the similarity between the query embedding and all document embeddings. To enable efficient retrieval at query time, it is common to precompute document embeddings offline.

Retrieval-Augmented Generation (RAG) is an approach that merges retrieval-based and generative models to improve text generation tasks. RAG comprises three main elements: a document store, retrieval model, and generative model. The document store contains relevant text-based documents, typically along with each document's precomputed embedding. The retrieval model efficiently selects the most relevant documents based on a given query, employing techniques described above. Typically, the generative model is a pre-trained LLM, which produces contextually-relevant text based on the query and retrieved documents.

When RAG is applied to NL2SQL, the document store typically contains documents with database-specific information, such as schema information (e.g., a document describing the column names and data types of a particular table) or database documentation (e.g., a document describing the supported SQL syntax for the database's particular SQL dialect).

[1]Here, "queries" is a general term that does not necessarily refer to SQL queries.

3 TAILORSQL SYSTEM OVERVIEW

TailorSQL is a NL2SQL system which accepts a natural language question as input from the user and produces a corresponding SQL query as output. Like other recent NL2SQL systems [7, 20, 32], TailorSQL employs an LLM by creating a prompt based on the input question, invoking the LLM using the prompt, and extracting the SQL output from the LLM's response. TailorSQL's core novelty compared to prior NL2SQL systems is its ability to specialize the NL2SQL pipeline to the user's workload by taking advantage of historical query logs.

We first describe the end-to-end workflow of TailorSQL's NL2SQL pipeline, with a particular emphasis on how TailorSQL specializes each part of this pipeline using historical query logs. We then provide some further illustrative examples that intuitively showcase how historical query logs can improve NL2SQL accuracy.

3.1 End-to-End Workflow

TailorSQL's NL2SQL system (shown in Fig. 2) is composed of an offline pipeline and an runtime pipeline. The offline pipeline preprocesses the information in the database and historical query logs so that the information can be retrieved for user questions. The offline pipeline must run before TailorSQL can start serving user questions, and it can be retriggered periodically (e.g., every night) so that TailorSQL is up to date with changes in the database schema or query workload patterns. The offline pipeline has two parts:

- **Document generation:** we compile information from the database which might be useful to include in the LLM prompt into individual pieces of text called *documents*. For example, prior NL2SQL techniques often generate one document for each table in the database, containing information such as the table's name and the name and data type of its columns. TailorSQL specializes to the workload by additionally generating documents containing information about common patterns in the historical query workload. Section 4 describes the process in further detail.
- **Document embedding generation:** each document is identified by a fixed-length vector embedding. The embedding is meant to capture semantic information about the contents of the document, so that the embedding of a document is similar (in terms of cosine similarity; see Eq. (1)) to the embeddings of user questions which make use of the document and dissimilar to the embeddings of unrelated user questions. A common approach for generating document embeddings is to feed the content of the document through a pretrained embedding model, such as SBERT [22]. TailorSQL specializes to the workload by generating document embeddings that are a function of not only the document's contents, but also

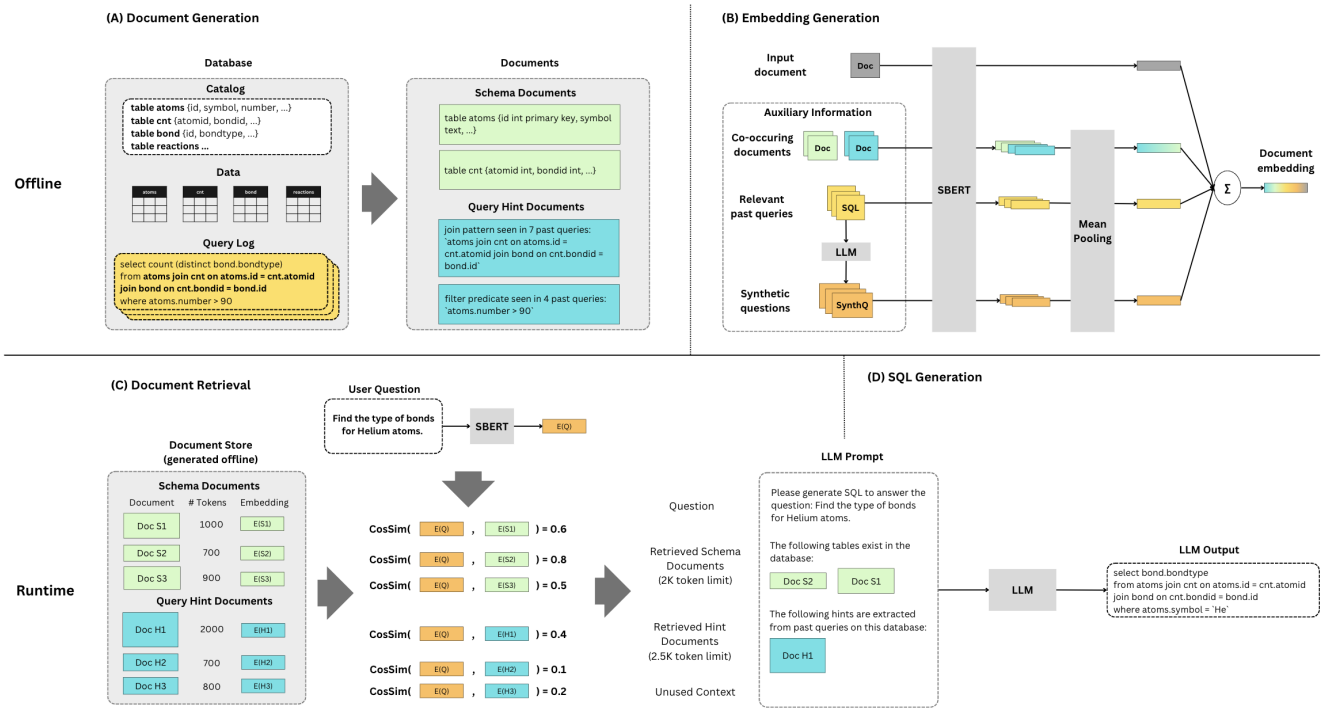

Figure 2: In an offline process, TailorSQL (A) generates schema and query hint documents based on information in the database, then (B) generates an embedding for each document using a combination of the document itself and several pieces of auxiliary information related to the document. (C) At runtime, we compute similarity between the user question's embedding and document embeddings in order to retrieve the most relevant documents, up to a per-document-class token limit. (D) The prompt, which includes the question and information in retrieved documents, is used to invoke an LLM to generate the SQL query.

of other related documents and related queries from the historical logs, which helps capture semantic information that is not present in the document's contents alone. Section 5 describes the process in further detail.

The runtime pipeline is triggered whenever the user asks a natural language question, and its output is a SQL query. The runtime pipeline has two components:

- **Document retrieval:** we compute the embedding of the user question, then compute the cosine similarity between the question embedding and the document embeddings which were generated offline. Prior NL2SQL techniques would then retrieve documents in decreasing order of similarity, until the prompt limit is saturated (i.e., until the total number of tokens in all retrieved documents surpasses the token limit of the LLM prompt). However, prior techniques do not distinguish between different types of documents. TailorSQL introduces a *context allocator* which performs an offline analysis to determine how much of the context to allocate for documents of each class in order to balance the precision and recall of document retrieval on the historical query workload. Section 6 describes the process in further detail.
- **SQL generation:** we assemble the retrieved documents into a prompt template. The same prompt template is used across all user questions. We invoke the LLM with the constructed prompt and parse the LLM response to yield the SQL query. TailorSQL does not perform any workload specialization for the SQL generation step. Indeed, prior NL2SQL systems make use of complex reasoning pipelines to perform SQL generation, e.g., by taking advantage of chain-of-thought and least-to-most prompting [20]. These improvements to SQL generation are orthogonal and complementary to TailorSQL's use of information from past queries. Specialized NL2SQL pipelines perform well when future questions have similar patterns as historical query logs, but can perform poorly otherwise. To avoid performance regressions, TailorSQL uses an abstention policy (Section 7) to dynamically decide when to fall back to using a non-specialized NL2SQL pipeline.

3.2 Illustrative Examples

The example from Section 1 showed how past queries can help disambiguate cryptically-named tables by providing information about common join paths. We now present two further examples where past queries provide information beyond that which is already provided through the database schema.

Filter Expressions: Filter expressions in past queries provide insights into the format of various columns, including literals within those columns, aiding the LLM in determining the appropriate filter for a query. In Fig. 3, the `user` table contains user information such as ids and birth date. Notably, `birthdate` is stored as an 8-digit integer in a distinctive manner. Past queries use filter expressions of a specific form to accurately extract the year from `birthdate` values. Incorporating this query hint into the prompt assists the LLM in understanding the format of the `birthdate` column.

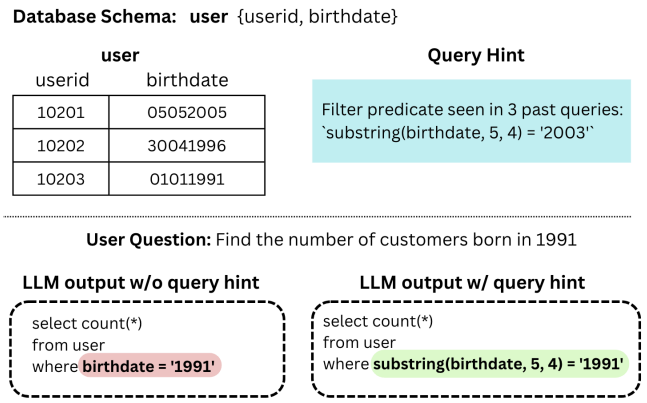

Figure 3: Past filter predicates help interpret column values.

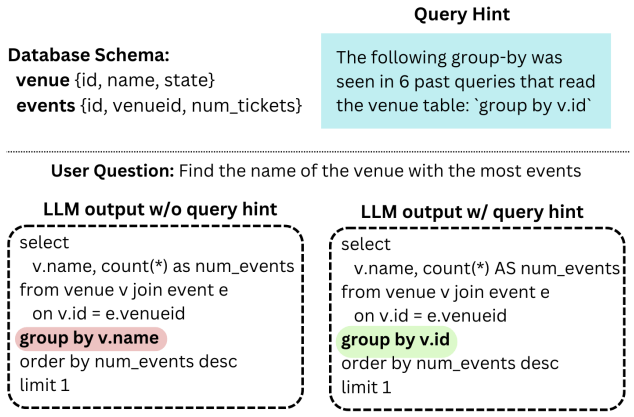

Figure 4: Past group-by clauses help indicate an implicit primary key. Note that the SQL query on the right only executes for SQL dialects that allow bare columns in aggregation queries, like SQLite.

Group-By Clauses: In Fig. 4, the `id` column serves as the primary key for the `venue` table, not the `name` column, since multiple venues may share the same name. Past aggregation queries tend to group by the `id` column, signifying that `id` refers to unique venues. If a user requests information about venues that requires aggregation, a query hint based on past group-by clauses will guide the LLM to aggregate over the `id` column instead of the `name` column.

4 DOCUMENT STORE

TailorSQL uses information about the database schema and past query workload to help generate accurate SQL queries for user questions. TailorSQL stores this information in two broad classes of documents: (1) schema documents, which capture information about the database schema, and (2) query hint documents, which capture information from the past query workload.

4.1 Schema Documents

Schema documents capture information about the database schema. Similar to prior NL2SQL techniques [20, 32], TailorSQL generates a document for each table in the database. By default, the schema document will contain the `CREATE TABLE` SQL statement which is used to create the table, which typically captures the following information: (1) the table's name, (2), the name, data type, default value, nullability, and other properties for each column, and (3) table properties such as the primary key and any foreign keys. All of this information helps the LLM reason about the correct SQL for a user question.

Note that the exact contents of the schema document will vary depending on the database management system. For example, some multi-node database systems define distribution keys for each table, which do not exist for single-node systems. Indeed, we found that certain additional information beyond the `CREATE TABLE` SQL statement are useful for TailorSQL, which we describe further in Section 8.2. In general, the exact contents of the schema document are orthogonal to the core idea of TailorSQL, and TailorSQL's procedure will work with whatever schema documents are generated for a given DBMS.

4.2 Query Hint Documents

Unlike schema documents, which are standard in prior NL2SQL techniques, query hint documents are a novel class of documents introduced by TailorSQL. Query hint documents capture information about common query structures in the past workload. Since queries often repeat in analytic workloads, including information about past query structures should help the LLM generate accurate SQL for future user questions. There are many options for the structure of query hint documents, which vary on a tradeoff space between information content and information density:

- Create a hint document for each query observed in the past workload, containing the SQL text of the query. This provides the maximum amount of information to the LLM. However, there are several disadvantages to this approach: (1) SQL queries can have very long text [30, 31], which quickly exhausts the context space. (2) SQL queries often have repetitive text, which means that hint documents would contain redundant information. For exact repeats of the same SQL query text, it is simple to deduplicate documents, but queries often do not repeat exactly but rather have repeating subcomponents (e.g., repeating filter predicates [30]). (3) SQL query text often contains boilerplate text which does not provide any useful information.
- To alleviate the redundancy described above, an alternative is to create a hint document for each past query *template*, where a template is created by removing literals from the SQL text, and then to store the literals separately. This approach might help reduce information redundancy for dashboarding and reporting workloads where the same queries are issued repeatedly with differing literals, but it does not work as well for ad-hoc query workloads where distinct templates are not as prevalent.
- Break down each past SQL query into subcomponents, where each subcomponent roughly corresponds to a different SQL clause, and generate a hint for each distinct subcomponent. The intuition is that queries in the past workload, despite being distinct overall, share significant subcomponents like join paths and filter predicates. Although this approach decreases information redundancy in documents, its disadvantage is that by extracting only a subset of information in each query, we lose some information content.

TailorSQL uses the approach of breaking down past queries into subcomponents, and in Section 8.2 we describe the specific content of hint documents used in our evaluation. However, we believe the

workload-specialization techniques described in this paper are helpful regardless of the exact format of hint documents. Deciding on the content of hint documents, similar to deciding on the content of schema documents and LLM prompt engineering more generally, is more an art than a science, and further optimization to the hint document format is left to future research.

5 DOCUMENT EMBEDDING GENERATION

In this section, we describe how TailorSQL generates an embedding for each of the documents in its document store. One simple workload-agnostic approach is to feed the contents of a document through an embedding model such as SBERT [22], which produces an embedding that captures the semantics of the document contents. For the remainder of this section, we refer to a document's embedding produced by feeding its contents through an embedding model as a *raw* document embedding. However, TailorSQL generates *tailored* embeddings in a manner that makes use of the past query workload, which we show in Section 8 to perform better than workload-agnostic raw SBERT embeddings.

For a given user question, we define a *relevant* document as one which is helpful for answering the question. A schema document is relevant to a question if the table described in the document is used in the SQL query that answers the question. Similarly, a query hint document is relevant to a question if any content in the hint document is used in the SQL query that answers the question.

TailorSQL's embedding generation procedure takes advantage of the following intuition: the ideal document embedding is one that maximizes the similarity between the document embedding and the embeddings of future relevant user questions (and also minimizes its similarity to the embeddings of future irrelevant questions). We first describe how TailorSQL formalizes this intuition into an optimization objective which can be used to quantify the goodness of a document embedding. We then describe TailorSQL's procedure for generating document embeddings to optimize that objective.

5.1 Optimization Objective

We define embedding similarity as the cosine similarity (see Eq. (1)) between two embeddings. For a given pair of question embedding $\mathbf{E}_{\text{Q}}$ and document embedding $\mathbf{E}_{\text{doc}}$, we define cosine loss as:

$$\text{loss}(\mathbf{E}_{\text{Q}},\mathbf{E}_{\text{doc}}) = \begin{cases} 1-\cos(\mathbf{E}_{\text{Q}},\mathbf{E}_{\text{doc}}) & \text{if doc relevant for Q} \\ \max(0,\cos(\mathbf{E}_{\text{Q}},\mathbf{E}_{\text{doc}}) & \text{if doc irrelevant for Q} \end{cases}$$

Loss is high when either (1) the document is relevant to the question but the embeddings are not similar, or (2) the document is not relevant to the question but the embeddings are similar.

One challenge is that we do not know the exact questions that the user will ask in the future, so it is difficult to directly optimize for $\text{loss}(\mathbf{E}_{\text{Q}},\mathbf{E}_{\text{doc}})$. TailorSQL tackles this challenge by taking advantage of the observation that in a stable workload, past SQL queries are likely to be similar to the queries that answer future user questions. Therefore, TailorSQL generates a synthetic question workload using the past SQL queries: for each past query, TailorSQL generates a synthetic question which, when given to a NL2SQL system, would yield that particular past query as the answer. TailorSQL employs an LLM model to generate these synthetic questions. The model is prompted with the SQL query and the schema of the tables involved in the query, and is tasked with generating a potential user question.

Given this synthetic question workload, our objective is to find an embedding for every document in order to minimize the total cosine loss for all documents in the document store $\mathbf{D}$ over all queries in the synthetic query workload $\mathbf{Q}$:

$$\min \sum_{\text{doc}\in\mathbf{D}} \sum_{\text{synthQ}\in\mathbf{Q}} \text{loss}(\mathbf{E}_{\text{synthQ}},\mathbf{E}_{\text{doc}}) \tag{2}$$

5.2 Optimization Procedure

Given this optimization objective, one possible strategy for determining document embeddings is to independently create an embedding for each document that minimizes cosine loss over the synthetic question workload. However, we found that this leads to overfitting. For example, in a degenerate case where a document is only relevant for a single synthetic question, then we would set the document embedding to be the same as the synthetic question embedding; clearly, this would not generalize to future questions which make use of the document, but which are not the same as the synthetic question.

Therefore, instead of allowing arbitrary embeddings for each document, in TailorSQL we impose a structure for document embeddings. For each document, we generate a number of *proxy embeddings*, which are embeddings that should be similar to the relevant question embeddings and which capture information that may not be present in the content of the document itself:

- We identify all the past queries which this document is relevant for. We use the embedding model (e.g., SBERT) to generate an embedding of each SQL query based on the query text, then average the embeddings across all queries: $\mathbf{E}_{\text{SQL}}$. We include this proxy embedding because documents are expected to be similar to the queries which use it, and SQL query information is not entirely present in documents and so would not be accounted for in the raw document embedding.
- For each of the relevant past queries, we prompt an LLM to generate a synthetic user question, and take the average embedding of the synthetic user questions: $\mathbf{E}_{\text{synthQ}}$. We naturally want document embeddings to be similar to the embeddings of questions which might use it.
- We identify co-occurring documents, which are other documents that are relevant to at least one of the queries that the current document is relevant to. We average the raw embeddings of all co-occurring documents: $\mathbf{E}_{\text{co-occur}}$. We include this proxy embedding because this information is useful for documents with obscure meanings, as co-occurring documents with clearer semantics can help improve their understanding. For example, in Fig. 1, `atom` and `bond` have clearer semantics than the `cnt` table.

For a given document, its embedding is the weighted sum of each of its proxy embeddings, along with its raw embedding, $\mathbf{E}_{\text{raw}}$:

$$\mathbf{E}_{\text{doc}} = \left(w_1 \cdot \mathbf{E}_{\text{raw}} + w_2 \cdot \mathbf{E}_{\text{co-occur}} + w_3 \cdot \mathbf{E}_{\text{SQL}} + w_4 \cdot \mathbf{E}_{\text{synthQ}}\right)$$

We use the same weights for all documents, i.e., the weights must be optimized once per workload, not once per document. Using the same weights for the entire workload encourages generalization. For a given workload, we find the weights that minimize the optimization objective (Eq. (2)) using gradient descent.

Proxy embeddings essentially extend the information content of the document itself. It is as if, instead of generating an embedding for a document based purely on the contents of the document itself, we are generating an embedding based on an augmented document that also includes information about relevant SQL queries and co-occuring documents. Instead of creating proxy embeddings, we could have achieved a similar effect by generating a "virtual" augmented document (which includes the original document contents along with the text of co-occurring documents, relevant SQL queries and synthetic questions) and using SBERT to generate an embedding of this augmented document. However, we found that by using a weighted sum, we have more fine-grained control over the relative importance of each piece of information when constructing the embedding, which produced better embeddings.

6 DOCUMENT RETRIEVAL

When the user asks a question to TailorSQL, we first convert the question into an embedding by feeding the question contents through the embedding model (e.g., SBERT). We then retrieve relevant documents from the document store and put their contents into the LLM prompt. One simple workload-agnostic approach is to retrieve documents in descending order of similarity between the question embedding and document embedding, until the LLM prompt context is filled.

However, there are several drawbacks to this simple retrieval approach, related to the existence of multiple document classes:

- The optimal retrieval recall and precision may differ for each document class, where recall is defined as the fraction of relevant documents that are retrieved and precision is defined as the fraction of retrieved documents that are relevant. For example, it is intuitively more critical to have high recall for schema documents than for hint documents: if a relevant schema document is not present in the context, then TailorSQL will have difficulty generating the correct SQL because it does not know what table or column name to use, but if a relevant hint is not present in the context, TailorSQL might still generate the correct SQL.
- The number of documents in each class may be vastly different. By retrieving documents in a class-agnostic manner, we may end up with many more of one document class than the other, which may not be desirable.
- The scale of embedding similarities for each document class may be different. For example, query hint documents may naturally have embeddings that are less similar to question embeddings than schema documents, due to the difference in information content of the two documents. Some of these scale differences are mitigated due to the document embedding generation process (Section 5), but there are nonetheless still effects due to the inclusion of raw document embeddings in the weighted sum that produces the tailored document embeddings.

To address these drawbacks, TailorSQL performs document retrieval in a workload-adaptive manner. TailorSQL performs an offline analysis over the past query workload to determine a *context allocation* over the document classes, i.e., a way to split up the number of tokens in the context among the document classes. When performing document retrieval for a given user question, we fill the allocated context for each document class independently. That is, for each document class, we retrieve documents of that class in descending order of similarity until the class context limit is reached.

TailorSQL determines context allocation using Bayesian optimization. Specifically, we select a sample of past queries and use an LLM to generate a synthetic question for each query. The optimization objective is to identify a context allocation that maximizes TailorSQL's accuracy on the synthetic workload while adhering to a user-specified token limit. As we will describe in Section 8.2, TailorSQL uses two classes of schema documents: table documents and column documents. Thus, the optimization is expressed as:

$$\begin{aligned} &\textbf{maximize} && \text{Accuracy}(t_{\text{tbl}}, t_{\text{col}}, t_{\text{hint}}) \\ &\textbf{subject to} && 0 \le t_{\text{tbl}} \le T, \quad 0 \le t_{\text{col}} \le T, \quad 0 \le t_{\text{hint}} \le T, \\ & && t_{\text{tbl}} + t_{\text{col}} + t_{\text{hint}} \le T \end{aligned}$$

Here, t_{tbl}, t_{col}, and t_{hint} represent the number of tokens allocated to table documents, column documents, and hint documents, respectively, subject to the total token constraint T. However, a naive Bayesian optimization implementation that samples configurations from $[0,T] \times [0,T] \times [0,T]$ would waste time exploring configurations that do not satisfy the token limit constraint. To address this, we reparameterize the problem by introducing variables that effectively eliminate the constraint:

- p: Fraction of the token limit that is allocated. The remaining tokens are not allocated to any document class and remain unused.
- p_{tbl}: Fraction of the allocated tokens that is allocated to tables.
- p_{col}: Fraction of the remaining allocated tokens (after table allocation) that is allocated to columns.

Using these new variables, the token allocations are expressed as:

$$\begin{aligned} t_{\text{tbl}} &= T \cdot p \cdot p_{\text{tbl}} \\ t_{\text{col}} &= T \cdot p \cdot (1 - p_{\text{tbl}}) \cdot p_{\text{col}} \\ t_{\text{hint}} &= T \cdot p \cdot (1 - p_{\text{tbl}}) \cdot (1 - p_{\text{col}}) \end{aligned}$$

By reparameterizing the token allocation in this manner, the original constraint $t_{\text{tbl}} + t_{\text{col}} + t_{\text{hint}} \le T$ is naturally satisfied, as all components are expressed as proportions of the total token limit. The reformulated optimization problem is now given by:

$$\begin{aligned} &\textbf{maximize} && \text{Accuracy}(p, p_{\text{tbl}}, p_{\text{col}}) \\ &\textbf{subject to} && 0 < p \le 1, \quad 0 \le p_{\text{tbl}} \le 1, \quad 0 \le p_{\text{col}} \le 1 \end{aligned}$$

This reformulation ensures a clean optimization space, making it particularly well-suited for standard Bayesian optimization.

We use Bayesian optimization, instead of a simpler method such as gradient descent, to determine context allocation for several reasons: (1) Bayesian optimization is more sample-efficient than gradient descent (i.e., it takes fewer iterations) which is ideal for cases where evaluations of the optimization function are expensive. In our case, evaluating the objective function involves running the NL2SQL pipeline and invoking the LLM for each synthetic question, which is indeed expensive. (2) There are interactions between different document classes which makes the objective function surface complex and multi-modal. For example, increasing the context allocation for a given document class is not always desirable: although larger context improves retrieval recall, it may degrade precision, and a high concentration of irrelevant documents may in fact distract the LLM [14]. Bayesian optimization is better at finding global optima, whereas gradient descent may get stuck in local optima.

7 ABSTENTION POLICY

TailorSQL specializes its NL2SQL workflow for a given query workload. However, its performance may degrade when the workload characteristics change (e.g., tables which were commonly queried in the past are now used rarely), since its specializations are no longer aligned with the user questions. In this section, we describe TailorSQL's *abstention policy*, which we use to decide whether user questions no longer align with the workload that TailorSQL was specialized for, and therefore to instead answer user questions using a generic, non-workload-specialized NL2SQL pipeline.

TailorSQL's abstention policy relies on the existence of two NL2SQL pipelines: TailorSQL's specialized runtime pipeline (i.e., Fig. 2C-D), and a generic runtime pipeline which does not use tailored embeddings or context allocations for document retrieval. Instead, the generic pipeline only retrieves schema documents by comparing similarity between the question embedding and raw document embeddings. Note that if we already ran TailorSQL's offline pipeline, there is no additional overhead for supporting a generic runtime pipeline: the schema documents and raw document embeddings needed for the generic pipeline already exist.

Conceptually, if a user question is similar to the past query workload, then we should use the specialized pipeline, and otherwise we should use the generic pipeline. Instead of directly comparing the similarity of user questions against the past query workload, TailorSQL uses runtime feedback to inform its abstention policy. We assume that after TailorSQL answers a user question with a SQL query, the user gives a binary signal (e.g., thumbs up or thumbs down) about whether they find the answer correct or useful.

TailorSQL uses a multi-armed bandit as its abstention policy: for each incoming question, TailorSQL chooses one of the two pipelines to run. After running, we collect the binary feedback from the user. We maintain the average feedback over all past questions that are run on each pipeline, where a thumbs up maps to 1 and a thumbs down maps to 0. TailorSQL chooses which pipeline to run using an ϵ-greedy strategy: with probability ϵ we choose a random pipeline, and with probability $1-\epsilon$ we choose the pipeline with the higher average historical feedback.

TailorSQL's abstention policy has two aspects which are different from a typical ϵ-greedy strategy: (1) We maintain a sliding window of feedback, so that feedback that was given earlier than the window's start boundary are not considered when computing the average feedback for a pipeline. We do not want to maintain stale feedback, since we are only concerned with determining which pipeline is better for the current workload. (2) We delete all collected feedback whenever we retrigger TailorSQL's offline pipeline, i.e., whenever we regenerate documents and embeddings (see Section 3.1).

Alternative Formulations: Instead of a multi-armed bandit, we also considered using a contextual bandit formulation. Intuitively, the multi-armed bandit determines which pipeline the *current workload* should be run on, whereas a contextual bandit determines which pipeline a *given question* (i.e., the decision "context") should be run on. Contextual bandits may do better than a multi-armed bandit at routing questions to the best pipeline if the current workload is composed of a mix of questions that are similar and dissimilar to the past queries. However, contextual bandits require many more feedback points to learn the optimal decision strategy. We were not able to justify such a long learning process through the benchmarks in our evaluation due to the low number of questions (Section 8), though contextual bandits may provide more benefit in other benchmarks or in real-world settings.

Instead of a bandit approach, we also considered using a supervised learning approach. However, this requires a different feedback mechanism, which leads to a different user experience. To train a binary classifier that decides whether a question should be processed using the generic or specialized pipeline, we would need to first collect training data by taking each question, passing it through each pipeline to produce two different responses, and asking the user to choose the better one. This type of feedback is more informative than the simple yes/no feedback used in our bandit approach, but it places a greater mental load on users.

8 EXPERIMENTAL EVALUATION

We first describe the experimental setup and then present an in-depth experimental study that shows TailorSQL's performance on three NL2SQL benchmarks. Overall, the evaluation demonstrates that:

- TailorSQL achieves 10–22% higher end-to-end SQL generation accuracy compared to other baselines while utilizing 2–15× smaller prompts for the same accuracy (Section 8.3).
- Query hint documents alone enhance performance by 4–5%, but TailorSQL achieves greater accuracy and latency improvement through tailored embeddings and context allocation (Section 8.4.1).
- In case of workload changes, TailorSQL's abstention policy adapts to the workload change to maintain high accuracy (Section 8.5).
- TailorSQL's workload specialization techniques are complementary to the methods in state-of-the-art NL2SQL systems and can improve their performance (Section 8.6).

8.1 Experimental Setup

Datasets: We use three NL2SQL benchmarks to test TailorSQL.

- **Bird-Union** combines tables from the databases in the dev set of the BIRD benchmark [13] into a single database containing 71 tables and 1200 NL2SQL question-SQL pairs[2]. By using one combined database, we simulate real-world scenarios where data is not so cleanly separated into distinct databases for every topic.
- **Spider-Union** combines tables from the databases in the dev set of the SPIDER benchmark [37] into a single database containing 221 tables and 480 NL2SQL question-SQL pairs.
- **FIBEN** [25] contains a single database containing 152 tables and 300 NL2SQL question-SQL pairs.

In general, Bird-Union and FIBEN pose a greater challenge than Spider-Union due to having more semantically intricate table names, column names, and questions.

Workloads: For each benchmark, we need to split the question-SQL pairs into two sets: one set to use for simulating historical query logs (i.e., the "training" set) and the other to use for simulating future user questions (i.e., the "test" set). We employ two types of splits:

- **Random Split:** In this case, question-SQL pairs are randomly assigned to either the query log or test set with equal probability.

[2]We exclude primary key-foreign key (PK-FK) information from the schema and omit the evidence field provided by the benchmark to simulate real-world scenarios where this data is typically unavailable.

As a result, the test set mirrors the same distribution as the query logs. This case is ideal for showcasing the benefits of TailorSQL.

- **Disjoint Split:** In this case, question-SQL pairs are divided in such a way that the SQL queries in the query log and test set do not access the same tables. Consequently, the test set exhibits a completely different distribution from the query logs. This case highlights workload drift when user questions have no similar counterparts in the query log. This is similar to the train/dev/test splits provided by the BIRD and SPIDER benchmarks, where the databases observed in the train set are completely disjoint from those in the dev or test sets.

All experiments have half the question-SQL pairs in the query log and the other half in the test set. Unless specified otherwise, we run experiments using Random Split.

Baselines: The leading techniques on the BIRD and Spider benchmarks, such as [7, 8, 19, 20, 27, 32], focus more on obtaining the best answer using superior LLMs or better decomposition or prompting techniques (e.g., least-to-most prompting). On the other hand, the goal of our evaluation is to assess how incorporating information from past query logs influences NL2SQL performance. The techniques described in this paper are orthogonal and complementary to the techniques on the benchmark leaderboards and can be combined for further improvement in accuracy (see Section 8.6). Therefore, to isolate the performance effects of using query log information and avoid confounding factors such as LLM model and prompting techniques, in our evaluation we adhere to using the same LLM (Claude 3 Haiku 1.2 [10]) and perform SQL generation via a single call to the LLM for TailorSQL and for all baselines.

We use two baselines:

- **SBERT:** In this baseline, we use SBERT [22] (specifically, the `all-MiniLM-L6-v2` model [4]) to generate embeddings for user questions and for documents. This baseline does not use any workload-related specializations, i.e., it does not store query hint documents, does not generate workload-tailored embeddings, and does not perform an offline analysis for prompt context allocation. For a given user question, this baseline retrieves documents into the LLM prompt in order of decreasing embedding similarity until hitting a specified prompt token limit.
- **BM25:** In this baseline, we use BM25 [23], a lexical retrieval method, to retrieve documents. BM25 is often used for retrieval over long documents, where SBERT-based models might not perform as well. All other aspects of the baseline are the same as **SBERT**.

Metrics: We evaluate the accuracy of each NL2SQL approach based on execution accuracy (EX), which measures the fraction of question-SQL pairs for which the execution results of the NL2SQL-generated SQL query and the ground-truth SQL query matches. Following the precedent of earlier papers, we measure improvement in accuracy in absolute numbers instead of relative numbers. For example, if execution accuracy increases from 20% to 60%, we refer to a 40% improvement in accuracy instead of a 3× improvement. We also evaluate the latency of each NL2SQL approach, which includes the latency of retrieving relevant documents to create an LLM prompt and the latency of invoking the LLM using the prompt. We repeat each experiment 5 times and we report the median value of each metric being measured in the experiment.

8.2 Implementation Details

TailorSQL uses SBERT as its embedding model for generating question embeddings and raw document embeddings.

As described in Section 4, TailorSQL's fundamental contributions are not the exact format of documents. Here, we describe the document format we used for our evaluation, but alternative formats may work better for other workloads.

8.2.1 Schema Documents. TailorSQL uses two classes of schema documents: table documents and column documents. Each column document contains information specific to a given column, including the column name, table name, and the ten most commonly-occurring column values. We separated table documents from column documents because we found that including samples for every column in a unified schema document results in very long documents which quickly use the limited context space in the LLM's prompt. Furthermore, if a question only requires a subset of columns from a table, it is unnecessary to retrieve the entire table document.

8.2.2 Query Hint Documents. TailorSQL generates the following types of query hints from past queries:

- Join path hints: for each query, we extract the set of tables which are scanned and the join conditions between the tables.
- Filter hints: for each query, we construct a document for each filter. This includes the names of the tables whose columns are referenced in that filter.
- Group-by hints: we construct one document for the entire group-by condition, including the names of the tables whose columns are referenced in the group-by.

These query hints cover the types of SQL clauses which we most commonly observed in our benchmarks and which appeared most useful to TailorSQL, but they are not exhaustive. We believe that TailorSQL can be easily extended to generate hints for other SQL clauses if needed.

If a query hint is found in multiple past queries, we merge the query hint documents and add a counter for the number of times this hint has been observed in the past.

8.3 End-to-End Evaluation

The objective of this experiment is to demonstrate that TailorSQL can enhance performance by leveraging past queries. In Fig. 5, the first row shows the SQL execution accuracy of TailorSQL and the baselines across three benchmarks. Additionally, we include a constrained TailorSQL baseline, where we artificially limit the context size for documents to 1K tokens[3], in order to evaluate how TailorSQL performs in terms of accuracy if the user requires low latency. For the baselines, we vary the limit on the LLM prompt context size. For TailorSQL, only a single point is shown since the context allocator automatically determines the context size for each document class.

Accuracy for all techniques varies across benchmarks, reflecting the varying levels of benchmark difficulty. TailorSQL consistently achieves the highest accuracy across all benchmarks compared to the baselines, while BM25 consistently underperforms. Specifically, TailorSQL achieves 12.5%, 10.9%, and 22.7% higher accuracy than the next best baseline on the three benchmarks, respectively. TailorSQL

[3]We guarantee that the Bayesian optimization algorithm will select a context allocation with at most 1K tokens by down-scaling all candidate allocations to fit within the limit.

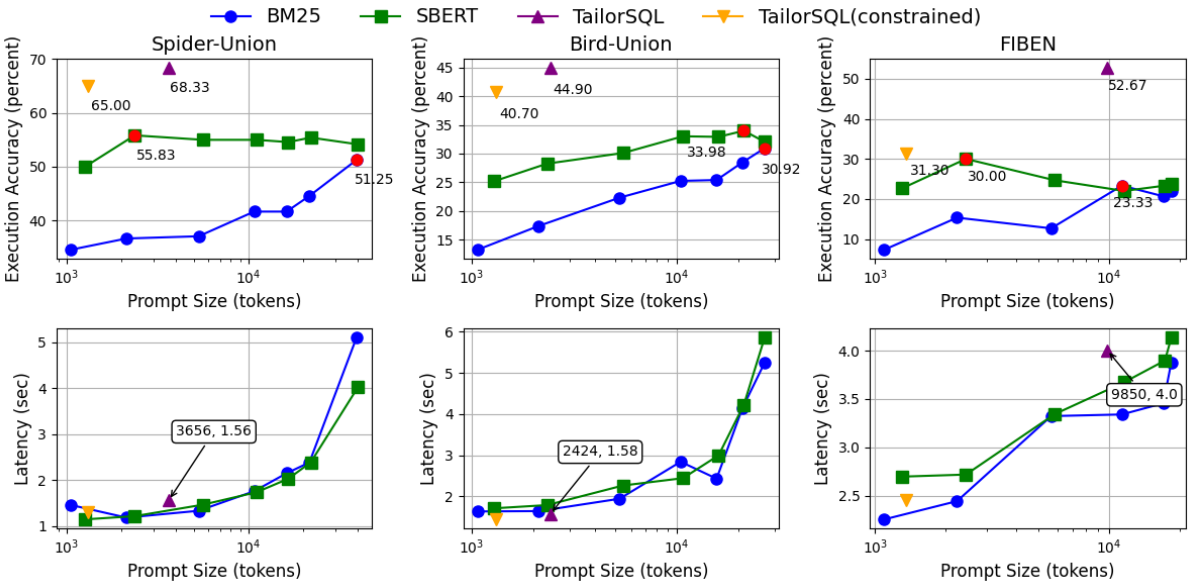

Figure 5: TailorSQL achieves 12.5%, 10.9% and 22.7% higher match execution accuracy compared to the next best baseline (SBERT). TailorSQL achieves the greatest accuracy gains on FIBEN due to the benchmark's complex queries and cryptically-named schema items, which benefit from query hints. Additionally, to achieve the same accuracy, TailorSQL (constrained) uses 2-15× times fewer tokens and incurs lower latency than the baselines.

achieves more significant accuracy improvements on FIBEN than the other benchmarks due to FIBEN's complex queries (e.g., nested queries, multi-table joins) and cryptic table names, which especially benefit from query hints and tailored embeddings.

Among the two baselines, the BM25 baseline typically requires more prompt tokens than the SBERT baseline to achieve its best accuracy. This is because BM25 is worse at document retrieval than SBERT and therefore requires more prompt tokens to achieve a sufficiently high document recall to generate accurate SQL queries. Note that for SBERT, increasing the prompt size only improves accuracy up to a certain point, after which further increases in prompt size cause accuracy to drop. This implies that as prompt size increases past the optimal point, the additional documents retrieved into the prompt have an increasing likelihood of being irrelevant, which degrades retrieval precision more than it improves recall. This observation is consistent with findings that adding irrelevant information to LLMs can reduce accuracy [26].

The SBERT baseline achieves its best accuracy with 2.3K, 21.5K, and 2.5K prompt tokens for the Spider-Union, Bird-Union, and FIBEN benchmarks, respectively. In contrast, TailorSQL matches or exceeds SBERT's best accuracy using at most 1.3K tokens[4] for the same benchmarks, as shown by the constrained TailorSQL baseline. As a result, TailorSQL achieves the same accuracy as SBERT while using 1.8×, 15×, and 1.9× fewer tokens.

The second row of Fig. 5 depicts the latency of invoking the NL2SQL pipeline for a user question, which includes both document retrieval and SQL generation. Latency is primarily influenced by the latency of the large language model (LLM) calls, which depend on factors such as the number of input and output tokens. Generally, latency increases as the number of input prompt tokens grows. Compared to the SBERT baseline, TailorSQL incurs higher latency because it uses more input prompt tokens than the SBERT baseline, since TailorSQL includes both schema and hint documents in its prompts, whereas the SBERT baseline only includes schema documents, though this effect is mitigated using TailorSQL's tailored embeddings for more effective document retrieval. However, if the user is concerned with latency, they can constrain TailorSQL to use a smaller prompt size, which achieves much lower latency than the SBERT baseline.

[4]This is larger than the context allocation limit of 1K tokens because total prompt size also includes the question and other boilerplate.

	Accuracy	Latency (s)	Prompt Tokens
TailorSQL	0.449	1.578	2424
w/o Tailored Embeddings	0.409	3.34	5999
w/o Context Allocator	0.369	6.29	41975
w/o Query Hints	0.382	3.56	6384

Table 1: TailorSQL Ablation Study on Bird-Union Benchmark

8.4 Ablation Study

We perform an ablation study for the components of TailorSQL's end-to-end workflow, as well as for the components of its workload-tailored embeddings.

8.4.1 End-to-End Workflow. In this section, we analyze the performance impact of each key technique employed by TailorSQL. Table 1 presents the impact of disabling each of TailorSQL's components, when evaluated on accuracy on Bird-Union. Disabling tailored

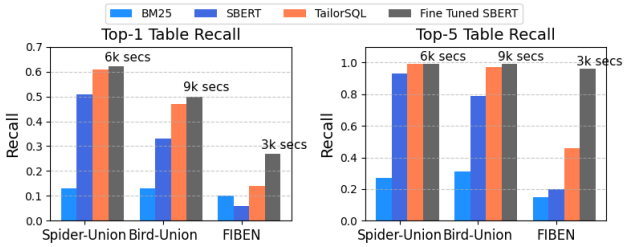

Figure 6: TailorSQL's tailored document embeddings achieve higher top-1 and top-5 table document recall compared to baselines. Fine-tuned SBERT achieves better recall than TailorSQL but requires higher training time (shown above each bar).

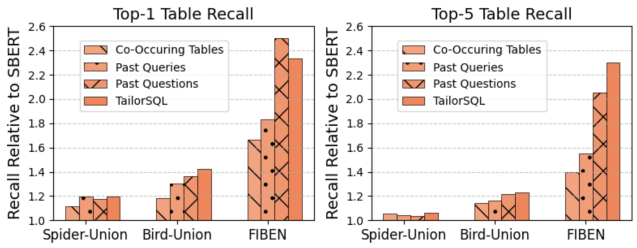

Figure 7: Improvement in recall achieved by including each individual factor into TailorSQL's tailored embeddings, while ignoring other factors. Recall improves the most when all factors are used (denoted by the bar marked "TailorSQL").

embeddings results in a noticeable increase in context size and a corresponding rise in latency. This is because less effective embeddings lead to a larger context being required to gather all the necessary information, resulting in an accuracy drop due to the reduced precision in identifying relevant documents. When the context allocator is disabled, a large fixed context is used to maintain high recall. However, this results in high latency (since LLM invocation latency is correlated with the size of the input prompt) and reduced precision when retrieving relevant documents, negatively impacting accuracy. Lastly, disabling query hints causes an increase in context usage and a reduction in accuracy. Without query hints, the system lacks crucial information that assists in SQL generation, forcing the retrieval of more schema documents, which contributes to the higher context size. In summary, each component plays a critical role in maintaining optimal accuracy and latency for TailorSQL.

8.4.2 Tailored Embeddings. This section evaluates the effectiveness of TailorSQL's tailored embeddings for documents (see Section 5). The first takeaway that TailorSQL's tailored embeddings perform better than using SBERT and BM25 to generate document embeddings. In Fig. 6, each bar represents the table recall for strategies selecting the top-K relevant tables for a user question. Top-K document recall measures the fraction of user questions where the top-K most similar documents include all relevant documents. TailorSQL consistently outperforms vanilla SBERT embeddings by specializing document embeddings based on the past query workload, leading to significant accuracy improvements (Section 8.3). BM25 performs the worst, consistent with its low accuracy in Fig. 5.

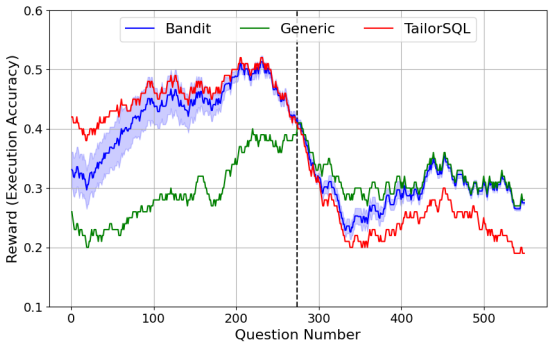

Figure 8: All questions before the vertical dotted line are similar to past queries, while all questions afterwards are dissimilar to past queries. TailorSQL's bandit-based abstention policy is able to select the better pipeline in case of this workload shift, and performs better than purely using TailorSQL's workload-tailored pipeline or a generic SBERT-based pipeline on all questions.

The second takeaway is that TailorSQL's tailored embeddings achieve worse recall than the approach of fine-tuning SBERT and using the fine-tuned model to generate embeddings. However, fine-tuning requires up to 2.5 hours of additional training time (as shown in Fig. 6), whereas TailorSQL's tailored embeddings require less than one minute of training on a CPU. Furthermore, fine-tuned models require additional storage space and may not generalize as well as TailorSQL's purposefully underparameterized model. While fine-tuned embedding models is a great approach when maximum recall is desired, we decide to use TailorSQL's method of learned embeddings to minimize training time and storage overhead.

We now explore the effect of each factor that contributes to TailorSQL's tailored document embeddings. Fig. 7 illustrates the impact on recall when integrating each of the three factors individually into the tailored embedding, along with raw SBERT document embeddings, while ignoring the two other factors. Among the three factors, embeddings based on synthetic past questions demonstrate the most substantial improvement across all benchmarks, while the incorporation of co-occurring documents contributes the least improvement.

8.5 Robustness against Workload Drift

In this section, we evaluate whether TailorSQL's abstention policy is able to correctly select whether user questions should run on TailorSQL's workload-tailored pipeline or on a generic NL2SQL pipeline. Intuitively, we expect the policy to choose to run on TailorSQL's pipeline if the user question follows similar patterns as the past query workload, and to choose to run the generic pipeline otherwise. In particular, as the user question characteristics drift over time, we expect the policy to dynamically switch from favoring TailorSQL's pipeline to favoring a generic pipeline.

Fig. 8 illustrates the behavior of the abstention policy under workload drift, which occurs over the course of many user questions. The first half of user questions follows the Random Split workload (see Section 8.1), which means that the user questions are similar to the past queries; the second half of user questions follows the Disjoint Split workload, which means that the user questions are dissimilar

Baseline	Accuracy	Latency (s)	Prompt Tokens
DIN-SQL+SBERT	0.5	61.79	40255
DIN-SQL+TailorSQL	0.6125	15	1460
MAC-SQL+SBERT	0.516	38.16	40212
MAC-SQL+TailorSQL	0.654	3.419	1397

Table 2: (Spider-Union) TailorSQL improves state-of-the-art NL2SQL systems (DIN-SQL and MAC-SQL) by using past queries.

Baseline	Accuracy	Latency (s)	Prompt Tokens
DIN-SQL+SBERT	0.219	158	26899
DIN-SQL+TailorSQL	0.4	21	1889
MAC-SQL+SBERT	0.255	34.73	26856
MAC-SQL+TailorSQL	0.47	4.03	1851

Table 3: (Bird-Union) TailorSQL improves state-of-the-art NL2SQL systems (DIN-SQL and MAC-SQL) by using past queries.

to the past queries. The plot shows the reward obtained by the abstention policy, which represents the execution match accuracy over a sliding window of 100 questions, as user questions are submitted. A 95% confidence interval is plotted around the abstention policy's reward to indicate variability. For comparison, we include the performance of only using TailorSQL's pipeline for every question and of only using a generic pipeline, equivalent to the SBERT baseline.

As expected, a policy of only using TailorSQL's pipeline performs better during the first half of the workload, while the policy of only using the generic SBERT baseline performs better in the second half. Initially, the abstention policy's bandit algorithm explores both pipelines, quickly converging to TailorSQL's pipeline in the first half. Upon detecting the distribution shift at the midpoint, the accuracy briefly drops, but the bandit algorithm soon adapts, selecting the SBERT pipeline as the optimal choice for the second half. Thus, the bandit-based abstention policy ensures adaptation to workload drift.

8.6 Impact on other NL2SQL methods

We now present empirical evidence that leveraging past queries to improve NL2SQL generation is beneficial for existing NL2SQL systems. State-of-the-art NL2SQL methods, such as those on the Spider and BIRD benchmarks, focus on enhancing SQL generation through advanced reasoning techniques like Chain-of-Thought [36] and Self-Consistency [34]. The core innovation of TailorSQL lies in its use of past queries, which is complementary and orthogonal to these techniques, allowing TailorSQL to augment their performance.

To demonstrate this, we modified two well-known NL2SQL systems, DIN-SQL [20] and MAC-SQL [32], to incorporate TailorSQL's provided prompt as the initial prompt, and compared this against the use of SBERT-based retrieval for the initial prompt. Tables 2 and 3 show that using TailorSQL as the initial prompt significantly improves accuracy and reduces SQL generation latency compared to SBERT-based retrieval across both benchmarks and NL2SQL systems. Note that the accuracy of DIN-SQL and MAC-SQL in our results differs from the values reported on the public Spider and BIRD leaderboards due to modifications in our setup—we introduced changes to the benchmarks by combining each benchmark's databases into one Union database and excluding evidence from the BIRD benchmark, and we employed a different LLM than the one originally used to tune the DIN-SQL and MAC-SQL prompts.

9 RELATED WORK

LLMs for NL2SQL: Today, the leaderboards for NL2SQL benchmarks like Spider and BIRD are dominated by LLM-based solutions [8, 9, 19, 27], while earlier methods leading the benchmarks were mostly based on manually-tweaked encoder-decoder LSTM-based architectures, e.g. [24, 33]. Top-performing NL2SQL systems primarily focus on question representation and information organization. DIN-SQL [20] uses decompositions and intermediate query representations following chain-of-thought [36] and least-to-most prompting [39] paradigms. DAIL-SQL [7] evaluates different methods of question representations, code representation, information (metadata) organization and picks the best combination of the three. CodeS [12] is a pretrained LLM designed specifically for NL2SQL. These methods are orthogonal to our idea of incorporating past query history and combining these method into TailorSQL can further improve performance. SNAILS [15] shows that NL2SQL techniques generally perform worse on databases that use less natural schema names, which further motivates the need for NL2SQL techniques like TailorSQL that can understand obscurely-named tables and columns.

Retrieval Methods: Retrieval-augmented generation (RAG) [11] has been employed to boost LLM accuracy across various NLP domains. *Bi-encoding* retrieval methods encode both question and document separately using the same embedding transformation, and compute their similarity via a distance metric such as cosine similarity. Recent literature shows that variants of the BERT model for embedding [6, 16] exhibit the best retrieval accuracy. *Cross-encoding* [5, 17] feeds the concatenated question-document pair into BERT and trains a FCN classifier layer on its vector representation. It commonly outperforms bi-encoded similarity scoring since it is able to capture more complex cross-feature interactions between the question and document. However, this approach is often prohibitively expensive at inference time, as it requires full encoding of each question-document pair at retrieval time.

Fine-tuning LLMs: Fine-tuning techniques are commonly classified into supervised [18, 28, 35], unsupervised [38], and reinforcement learning [21, 29] based methods. In the case of NL2SQL pipelines, the past query workload along with synthetically generated user questions could be used as input-output pairs for supervised fine-tuning of the model. This method could potentially deliver better results than the RAG-style solution. However, users may have privacy concerns about using their data to train LLMs shared across users, and training and maintaining a separate fine-tuned LLM per database is an expensive operation.

10 CONCLUSION

We introduced TailorSQL, an NL2SQL system that tailors itself to a specific database by leveraging the database's query history to enhance document generation, document embedding, and document retrieval in a RAG-based NL2SQL pipeline. We further introduced a bandit-based abstention policy which dynamically determines when TailorSQL's workload-specialized pipeline is no longer suitable for the current workload, thereby avoiding performance degradations due to workload drift. TailorSQL demonstrates consistent accuracy and latency improvements across three NL2SQL benchmarks.

REFERENCES

[1] [n.d.]. https://learn.microsoft.com/en-us/azure/azure-sql/copilot/query-editor-natural-language-to-sql-copilot

[2] [n.d.]. https://docs.aws.amazon.com/redshift/latest/mgmt/query-editor-v2-generative-ai.html

[3] [n.d.]. https://docs.snowflake.com/en/user-guide/snowflake-copilot

[4] 2022. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2.

[5] Zeynep Akkalyoncu Yilmaz, Wei Yang, Haotian Zhang, and Jimmy Lin. 2019. Cross-Domain Modeling of Sentence-Level Evidence for Document Retrieval. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 3490–3496. https://doi.org/10.18653/v1/D19-1352

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[7] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. arXiv:2308.15363 [cs.DB]

[8] Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. 2025. A Preview of XiYan-SQL: A Multi-Generator Ensemble Framework for Text-to-SQL. arXiv:2411.08599 [cs.AI] https://arxiv.org/abs/2411.08599

[9] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. arXiv:2406.08426 [cs.CL] https://arxiv.org/abs/2406.08426

[10] Anthropic Inc. 2024. https://www.anthropic.com/news/claude-3-haiku.

[11] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401 [cs.CL]

[12] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. CodeS: Towards Building Open-source Language Models for Text-to-SQL. *Proc. ACM Manag. Data* 2, 3, Article 127 (May 2024), 28 pages. https://doi.org/10.1145/3654930

[13] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, 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.CL]

[14] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).

[15] Kyle Luoma and Arun Kumar. 2025. SNAILS: Schema Naming Assessments for Improved LLM-Based SQL Inference. *Proc. ACM Manag. Data* 3, 1, Article 77 (Feb. 2025), 26 pages. https://doi.org/10.1145/3709727

[16] Sean MacAvaney, Andrew Yates, Arman Cohan, and Nazli Goharian. 2019. CEDR: Contextualized Embeddings for Document Ranking. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '19)*. ACM. https://doi.org/10.1145/3331184.3331317

[17] Rodrigo Nogueira and Kyunghyun Cho. 2020. Passage Re-ranking with BERT. arXiv:1901.04085 [cs.IR]

[18] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[19] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O. Arik. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. arXiv:2410.01943 [cs.LG] https://arxiv.org/abs/2410.01943

[20] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. arXiv:2304.11015 [cs.CL]

[21] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290* (2023).

[22] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Conference on Empirical Methods in Natural Language Processing*. https://api.semanticscholar.org/CorpusID:201646309

[23] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (April 2009), 333–389. https://doi.org/10.1561/1500000019

[24] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. arXiv:2109.05093 [cs.CL]

[25] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Ozcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. 2020. ATHENA++: Natural Language Querying for Complex Nested SQL Queries. *Proc. VLDB Endow.* 13, 11 (2020), 2747–2759.

[26] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*. PMLR, 31210–31227.

[27] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. arXiv:2405.16755 [cs.LG] https://arxiv.org/abs/2405.16755

[28] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html* 3, 6 (2023), 7.

[29] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[30] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. In *VLDB 2024*. https://www.amazon.science/publications/why-tpc-is-not-enough-an-analysis-of-the-amazon-redshift-fleet

[31] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, 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* (Houston, TX, USA) *(DBTest '18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3209950.3209952

[32] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. 2023. Mac-sql: Multi-agent collaboration for text-to-sql. *arXiv preprint arXiv:2312.11242* (2023).

[33] Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2021. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. arXiv:1911.04942 [cs.CL]

[34] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171 [cs.CL] https://arxiv.org/abs/2203.11171

[35] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Anjana Arunkumar, Arjun Ashok, Arut Selvan Dhanasekaran, Atharva Naik, David Stap, et al. 2022. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. *arXiv preprint arXiv:2204.07705* (2022).

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[37] 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. 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887 [cs.CL]

[38] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2023. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206* (2023).

[39] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. arXiv:2205.10625 [cs.AI]