

Datrics Text2SQL

A Framework for Natural Language to SQL Query Generation

Gladkykh Tetiana, Kirykov Kyrylo

Datrics*

Whitepaper v. 1.0

March 2025

Introduction

Importance of Text-to-SQL Systems

Text-to-SQL systems translate natural language questions into SQL queries, making data access more inclusive and intuitive. This is especially valuable for non-technical users in organizations who may not know how to write SQL. By bridging natural language and database queries, Text-to-SQL **empowers analysts and business users to access and analyze data without mastering complex SQL syntax**[1, 2].

In effect, it **democratizes data access** – users across different departments can retrieve insights using plain language, instead of relying on technical teams or learning query languages [3]. This lowers the barrier to entry for data analysis and fosters a more data-driven culture. With more employees able to query data directly, organizations can make faster, informed decisions at all levels.

Furthermore, allowing people to ask questions in natural language **speeds up the data retrieval process**, as business users can get answers immediately without waiting for a SQL expert to write the query [4]. By streamlining how users interact with databases, Text-to-SQL systems help unlock the full value of enterprise data and improve productivity in decision-making.

Challenges in Text-to-SQL Development

Despite their promise, developing effective Text-to-SQL systems comes with several challenges and complexities:

- **Ambiguity and Variability in Language:** Human language is inherently ambiguous and context-dependent, whereas SQL is precise and structured [4]. Users might phrase questions in unclear or conversational ways (e.g. *"show me top subscribers"*), which don't directly map to the database schema [3]. In practice, queries can also be incomplete or span multiple turns in a conversation, adding context that the system must interpret [4, 5]. Handling such ambiguity and diverse phrasing is a non-trivial task for Text-to-SQL systems. A recent study found that roughly

*th@datrics.ai, kk@datrics.ai

20% of user questions to text-to-SQL systems are problematic – often ambiguous or even unanswerable without clarification [6].

- **Domain-Specific Terminology:** Every organization or dataset may use unique vocabulary, acronyms, or business terms that do not exactly match the names of tables or columns in the database [7, 8]. Text-to-SQL models struggle when **domain terms or synonyms don't align with schema labels**, requiring the system to incorporate domain knowledge to make the correct associations [9, 10, 11]. Ensuring the model understands the context is essential but challenging.
- **Complex Schemas and Queries:** Real-world databases often have many tables with intricate relationships, and user questions may require joining multiple tables or applying advanced SQL functions. Generating correct SQL in these scenarios is difficult. Writing **efficient joins across numerous tables** or **handling nested queries** and analytics functions (e.g. window functions) is a major challenge for Text-to-SQL [12]. Large schemas also introduce many similarly named fields, increasing the risk that the model might confuse table or column names. LLMs without guidance can hallucinate – producing SQL that is syntactically incorrect or selecting the wrong table – especially when confronted with many tables or columns with similar names [13]. Ensuring the generated query uses the right database entities and is optimized for performance is non-trivial.

These challenges mean that out-of-the-box language models may falter on enterprise Text-to-SQL tasks. Handling ambiguous phrasing, understanding domain context, and coping with complex, multi-table databases all require carefully designed solutions beyond just training a model on example queries.

Datrics Text2SQL Framework

This document outlines a new Text-to-SQL framework, **Datrics Text2SQL** (GitHub Repository), which addresses the above challenges by using a *Retrieval-Augmented Generation (RAG)* approach with incorporating domain knowledge, database structure, and example-based learning to enhance accuracy and reliability. Our framework's core is a knowledge base built from database documentation and successful query examples. We store this information in a vector database and use retrieval techniques to find the most relevant context for each user question, helping us generate accurate SQL.

The described approach tackles common problems in SQL generation, including selecting the wrong tables, creating incorrect joins, and misunderstanding business rules. By following a straightforward process of knowledge collection, context retrieval, and SQL generation, we managed to produce reliable results while adapting to different types of questions and database structures.

The following sections explain how we train the knowledge base, retrieve relevant information, and generate SQL code, providing a clear guide for implementing this solution. We provided a detailed description of the methodology for training the knowledge base, retrieving relevant context, and generating SQL code, providing a clear guide for implementing an effective Text2SQL solution in real-world applications.

Datrics Text2SQL. General Concept

The Text2SQL system is designed to convert natural language queries into accurate SQL code by using a sophisticated RAG (Retrieval-Augmented Generation) framework. This framework operates based on a comprehensive Knowledge Base that combines multiple sources of information:

1. **Database Documentation:** Detailed descriptions of database tables, columns, and relationships within the overall system. This provides the basis for understanding

- the database schema.
2. **Question-Answer Examples:** A collection of question-SQL pairs that is used as a reference for mapping natural language questions to specific SQL syntax and database operations.
 3. **Domain-Specific Rules:** Specialized instructions that capture the business logic and relationships not explicitly stored in the vector database but critical for accurate query generation.

The system functions through three main phases:

1. **Knowledge Base Training:** The process of building a comprehensive understanding of the database by analyzing documentation, examples, and schema information. This involves transforming unstructured descriptions into structured metadata and vector embeddings.
2. **Content Retrieval:** This phase includes retrieving the most relevant documentation and examples from the knowledge base using semantic similarity matching with the user's request concerning the business rules.
3. **SQL Generation:** The final stage is where the system utilizes the retrieved information to generate a correct SQL query that satisfies the user's intent, leveraging both table descriptions and similar query examples.

This approach ensures that the Text2SQL system can handle complex queries by combining structural knowledge about the database with practical examples of query patterns, all while respecting domain-specific business rules that govern data relationships and access patterns.

Train Knowledge Base

The Text2SQL model training consists of two parts - training of Vector DB on the textual context, which includes documentation, database schema, DDL, and ground truth examples of the correct answers (SQL code) for user's request and Domain-specific rules extraction. The last ones are not stored in the vector DB - they are used as an additional context (together with business rules) for the increased AI agent response precision.

Train based on Documentation

The training of the Text2SQL model's Knowledge Base, which is the core part of the Text2SQL RAG framework, leads to saving the valuable content in Vector DB. The process starts with training on documentation. Each document is represented in JSON format and contains information about a specific table in the database with its detailed description, including information about columns and connection with other datasets in the database. The document should have the following structure:

- **name:** The table's name
- **summary:** A brief description of what the table contains
- **purpose:** Explains the table's role and why it exists
- **dependencies_thoughts:** Informal notes on relationships between this table's fields and fields in other tables
- **keys:** A list of key columns used to join or relate to other tables
- **connected_tables:** An array of related table names

- **columns:** An array where each element is an object describing a column (with its name and a description)
- **entities:** A list of key concepts extracted from the documentation
- **strong_entities:** A subset of entities deemed particularly significant

You don't need to manually create this structural definition. The system can automatically generate it by analyzing any free-form text that contains descriptions of the table and its columns. This automated approach saves time and reduces the potential for errors that might occur during manual creation. The system can extract the necessary information about table relationships, column definitions, and data types directly from natural language descriptions, converting them into a formal structure without requiring manual intervention.

I. DOCUMENTS PROCESSING

The automated extraction of the required fields from a document is performed via processing the document by LLM and divided into two steps:

1. Extraction of `name`, `summary`, table `purpose`, `columns` with their description, `key` columns, and `connected tables`
2. Extraction of `entities` that might be inferred from this table based on it's purpose

For the second step the voting strategy is applied - we generate several LLM responses for the same input content and assess the frequency of the concept occurrence within multiple runs. This allows extracting the strong entities from the entities' list.

DOCUMENTS AUTO-GENERATION

Suppose the user's DB contains well-named tables with columns that also have explicit names (the names reflect the columns' and table's sense). In that case, we may assume their purposes and dependencies considering the domain and the general knowledge about available data. So, if we have a DB schema, we may use LLM to generate the table description with some reasonable limitations. Generally, the generated description is applicable in the first approximation, but domain expert validation is strongly required for the complex data structure.

Here are the most important characteristics of the generated data:

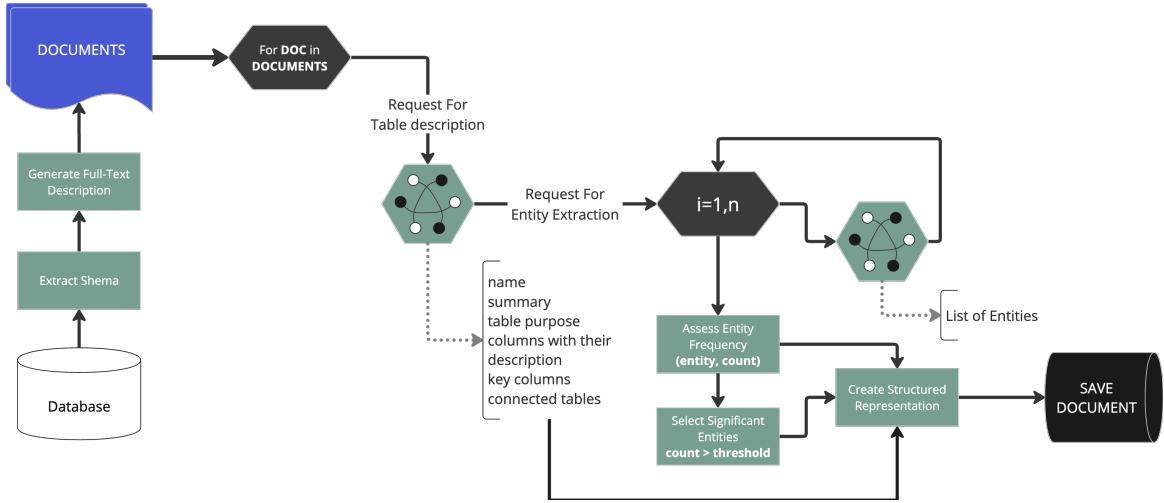
1. Auto-generated descriptions follow a clear format with sections like "Table Description," "Columns Description," and "Potential Dependencies," making them easy to navigate and review.
2. Provide detailed descriptions of columns, often attempting to explain their possible roles and relationships.
3. Highlight primary keys, sort keys, and potential table dependencies, which can be helpful in understanding database structure and optimizing queries.

Limitations:

1. While auto-generated descriptions attempt to define a table's role, they may sometimes describe it too broadly rather than focusing on its specific function.
2. In some cases, the descriptions suggest dependencies between tables that are not explicitly defined, which might lead to assumptions about relationships that do not exist.
3. While general column descriptions are useful, they may not always capture specific nuances like predefined status codes or intentionally unused fields.

DOCUMENTS STRUCTURAL REPRESENTATION

For straightforward data structures, the generated descriptions offer enough context for a Text2SQL agent to understand the table's purpose and relationships without requiring expert review. Table and column names typically provide sufficient clarity, making additional manual validation unnecessary. However, expert review may be critical for more complex schemas with nuanced relationships or specific field constraints.



Conceptual view of table documentation structure.

In order to operate with the full-text description of the table description in a more flexible way, we need to extract the local concepts from it, like *name*, short *summary*, *purpose* of the table, etc. Moreover, we need to summarise what kind of entities are connected with this table to localize the search space during documents' retrieval.

So, at the first step of the document's processing we transform the unstructured (raw) representation into the semi-structured:

1. **name**: name of table
2. **summary**: short summary about table
3. **purpose**: purpose of the table
4. **dependencies**: relations with other tables in database that are mentioned in description considering the columns that have id-markers
5. **keys**: list of columns that are keys that are used for the connections with other tables in database (frequently they are under template <name>_id)
6. **connected_tables**: names of tables that are connected with this one concerning the previously detected keys

After that we may operate with all of these properties independently both on the stage of RAG model tuning and on the SQL generation.

Next, we need to extract inferred entities from a given table by analyzing its contents and underlying purpose. Instead of directly listing the table's data, the goal is to generate meaningful, complex entities that are not explicitly mentioned but can be logically deduced from the table's context.

The process of the entities extraction begins by engaging a language model to generate *several responses simultaneously*. Instead of relying on a single output, multiple responses are produced to capture a range of interpretations. Each response is expected to include a list of entities identified within the input text.

Once all responses are received, the next step is to combine the extracted entities from each response into one comprehensive collection. This aggregation allows the system to analyze the frequency with which each unique entity appears across the multiple outputs. The underlying assumption is that entities mentioned consistently are more likely to be accurate or significant. Those entities occur just once within the series of LLM calls and are considered insignificant. This approach leverages the diversity of multiple model outputs to enhance the reliability of entity extraction. By aggregating and filtering based on frequency, the process reduces the impact of any anomalous or less confident responses, resulting in a more robust and consistent set of extracted entities.

EXAMPLE

Let us consider an example. Suppose we have well-documented database with the detailed description of each table, which looks like:

```

SMS_Statuses
-----
The `sms_statuses` table contains data about which activists are subscribed to the mobile
→ messaging list of groups being mirrored (or children, if in a network) and what their
→ current statuses are (subscribed, unsubscribed, bouncing, or spam complaint).
This table contains data about the mobile subscription status such what status the activist
→ has currently, as well as metadata about the mobile subscription status such as when it
→ was created.
#### Fields
| Field Name | Description | Type | Is Sort Key? |
| --- | --- | --- | --- |
| id | The numerical identifier of the mobile subscription status. | INT | |
| subscriber_id | The numerical identifier of the activist who is attached to this status.
→ Relates to the `id` field in the `[users]`(/mirroring/docs/users)` table. | INT | True |
| group_id | The numerical identifier of the group who's list this mobile subscription status
→ is related to. Relates to the `id` field in the `[groups]`(/mirroring/docs/groups)`-
→ table. | INT | |
| user_id | This field is intentionally blank. | INT | |
| status | The mobile subscription status of the activist. `1` if subscribed. `2` if
→ unsubscribed. `3` if bouncing. | INT | |
| source_action_id | This field is intentionally blank. | INT | |
| source_action_type | This field is intentionally blank. | VARCHAR | |
| join_date | The UTC timestamp when this mobile subscription status was last changed from
→ any status (or none) to subscribed. Ex: The date the person last joined the list. |
→ DATETIME | |
| created_at | The UTC timestamp when this mobile subscription status was created. | DATETIME
→ | |
| updated_at | The UTC timestamp when this mobile subscription status was last updated. |
→ DATETIME | |

```

It is structured representation, which is completely ready for use, will have a view:

```

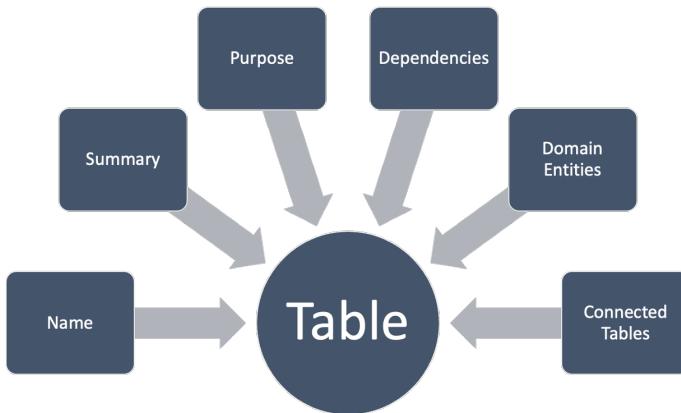
    "name": "sms_statuses",
    "summary": "Data about the mobile subscription statuses of activists within groups.",
    "purpose": "To track and manage the subscription statuses (subscribed, unsubscribed, bouncing, or spam complaint) of activists in mobile messaging lists.",
    "dependencies_thoughts": "Relates to the `users` table (subscriber_id to id) and the `groups` table (group_id to id).",
    "keys": [
        "subscriber_id",
        "group_id"
    ],
    "connected_tables": [
        "users",
        "groups"
    ],
    "columns": [
        {
            "column": "id",
            "description": "The numerical identifier of the mobile subscription status."
        },
        ...
        {
            "column": "updated_at",
            "description": "The UTC timestamp when this mobile subscription status was last updated."
        }
    ]
]

```

II. TRAIN ON DOCUMENTS

If we have a structured and comprehensive description of the database, including table names, descriptions, relationships, and metadata, we can leverage this data to enhance Retrieval-Augmented Generation (RAG) using a **Vector Database (Vector DB)**.

A **Vector DB** stores documents in a high-dimensional vector space, transforming each document into numerical representations (embeddings) that capture semantic meanings. These embeddings allow for more intelligent and context-aware searches, significantly improving how information is retrieved. Each document, representing information about a table, is stored as a **vector embedding**. The embedding is generated based on multiple document features.



Vector DB storage approach for database documentation.

These embeddings are accompanied by **metadata** that includes complete table information, relationships, and the source of embeddings. Metadata ensures the search space can be precisely controlled, enabling efficient and accurate retrieval of relevant database details.

The **training process** involves structuring documentation data into multiple embeddings to capture different aspects of the database:

1. **Table Extraction** – The function leverages a **Large Language Model (LLM)** to identify

and extract table names mentioned in the document.

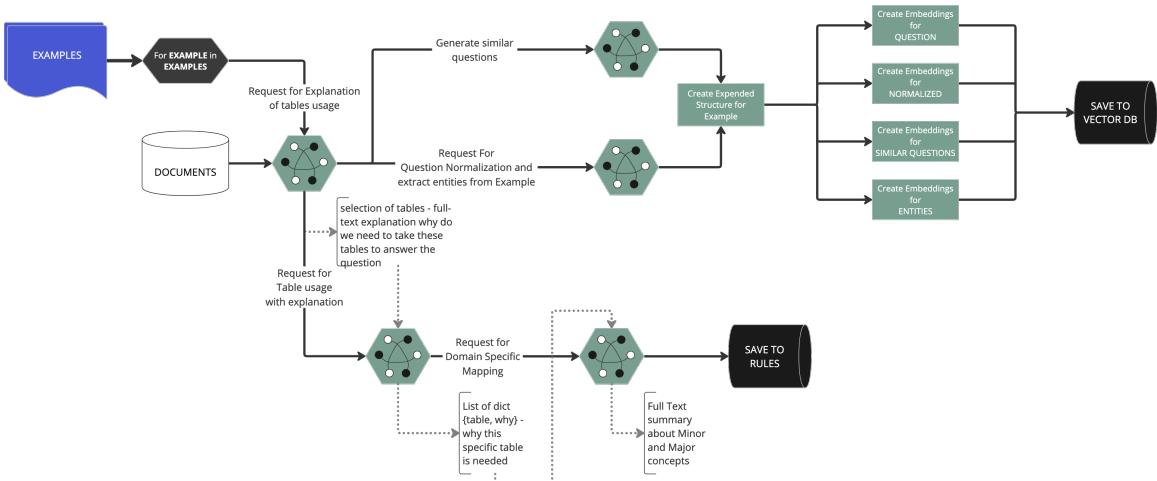
2. **Description Generation** – It dynamically creates short and long structured descriptions using predefined templates.
3. **Column Processing** – It processes table column details and attaches them to the training data.
4. **Multiple Training Entries** – Each document is transformed into different categories of embeddings:
 - **Full documentation** with table descriptions.
 - **Dependency and relationship-based embeddings** (for tracking how tables are linked).
 - **Table metadata embeddings** (name, descriptions, and keys).
 - **Connected tables and their associations**.
 - **Entity relationships** to understand interactions between database objects.

Train based on Examples

While the documentation offers a comprehensive overview of the database, including relationships, connections, and other intricacies, it does not ensure that this information alone is sufficient to accurately interpret domain-specific nuances and generate the correct SQL query in response to a user's request. That is why the ability to work with **question-answering examples** is crucial.

Each example consists of a "**Question - SQL Query**" pair, which we store in the **Vector Database (Vector DB)**. These examples serve as reference points, allowing the system to retrieve relevant patterns and structures when generating SQL queries in response to user requests. By leveraging these stored examples, the model can improve accuracy and adapt to complex, domain-specific queries more effectively.

To enhance **SQL query generation** and **natural language understanding**, we implemented a structured **normalization and extraction process**. This process performs the conversion of the user request, which is accompanied by an answer in view of SQL code, into a structured JSON format that captures key elements like **normalized questions**, **requested entities**, **data sources**, and **calculations**.



Structure of examples database with vector representations.

Each stored example \mathcal{A}_k is represented as structure:

$$\mathcal{A}_k = (\mathcal{I}_k, \mathcal{S}_{code_k}, \mathcal{N}_k, \mathcal{M}_k, \mathcal{E}_k, \mathcal{V}_k, \mathcal{O}_k); \mathcal{A}_k \in \mathbf{A}; \mathcal{V}_k = \{T_{ik}, \mathbf{C}_{ik}\}$$

where:

\mathcal{I}_k — The **initial user query** for k example.

\mathcal{S}_{code_k} — The corresponding **SQL query**, if available.

\mathcal{N}_k — The **normalized form** of the initial question.

\mathcal{E}_k — The **extracted entities** relevant to the query.

\mathcal{M}_k — The **main clause** of the query.

\mathcal{V}_k — The **identified data sources** (tables and columns).

\mathcal{O}_k — The **operations** (aggregations, filters, grouping) present in the query.

$T_{ik} \in \mathbf{T}$ — The i -th **table** from the database that are used in k example

\mathbf{C}_{ik} — set of columns from T_i that are used in k example

I. EXAMPLE NORMALIZATION AND STRUCTURING

First, we transform a **natural language query** into a standardized format while removing unnecessary details. This ensures consistency and adaptability across different database queries. Moreover, we analyze the pair of (**natural language query**, **SQL code**) to extract the entities that are connected with this request and identify data sources and operations.

This process is executed using a **Large Language Model (LLM)**, which simultaneously performs the following tasks:

$$(\mathcal{N}_k, \mathcal{E}_k, \mathcal{V}_k, \mathcal{O}_k, \mathcal{S}_{code_k}) = \text{LLM}(\mathcal{I}_k, \mathcal{S}_{code_k})$$

The following rules define how the LLM transforms and extracts structured information from natural language inputs.

1. Normalize the Question (\mathcal{N}_k)

The LLM must standardize the user query by removing unnecessary details and ensuring a structured format. The normalization process follows these steps:

- Remove database-specific names (e.g., table names, column names).
- Remove display instructions (e.g., "show", "display", "generate chart").
- Generalize conditions (e.g., "in 2023" → "for some time range").
- Abstract filtering details (e.g., replace specific IDs, names, or numbers with generalized terms).
- Maintain the directive form (e.g., "Show sales data" → "Sales data").

2. Requested Entities (\mathcal{E}_k)

The LLM identifies **key data elements** referenced in the query. These entities define the core information being retrieved.

- Identify **core data** being requested (e.g., "number of transactions per region").
- Keep significant **categorical definitions** (e.g., "mobile applications", "email campaigns").
- Strip out time filters and specific constraints if they don't affect the entity's identity.

Finally, we receive a structured set of entities:

$\mathcal{E}_k = \{(L_i, K_i, J_i)\}$, where L_i — the **label** of the entity; K_i — the **type** of entity (category, metric, or identifier); J_i — any **additional attributes** that provide context.

3. Data Sources (\mathcal{V}_{\parallel})

The LLM must determine **which tables and columns** provide the necessary data.

- If an SQL snippet is provided, extract table names and necessary columns from it.
- If no SQL code is given, keep the data source field empty to avoid assumptions.

The structured output for data sources is represented as:

$$\mathcal{V}_k = \{T_{ik}, \mathbf{C}_{ik}\}$$

4. Extract Calculations & Filters

The LLM must identify and structure any **mathematical operations, grouping fields, or filtering conditions**.

- Identify aggregation operations (e.g., SUM, AVG, COUNT).
- Extract grouping fields (e.g., BY department).
- Convert all filters and conditions into human-readable form (e.g., "where region = 'US'" → "Filter by region (United States)").

The structured output for operations is represented as:

$$\mathcal{O} = \{\text{operation, target, condition}\}$$

II. EXTRACTING THE MAIN CLAUSE & DETAILS

Additionally, we expanded the example description with isolated **request's core** from **additional details** by using Multi-Shot Learning. This helps structure the SQL query effectively.

The LLM processes the input query by splitting it into two distinct components:

$$\mathcal{I}_k \Rightarrow (\mathcal{M}_k, \mathcal{M}'_k)$$

where:

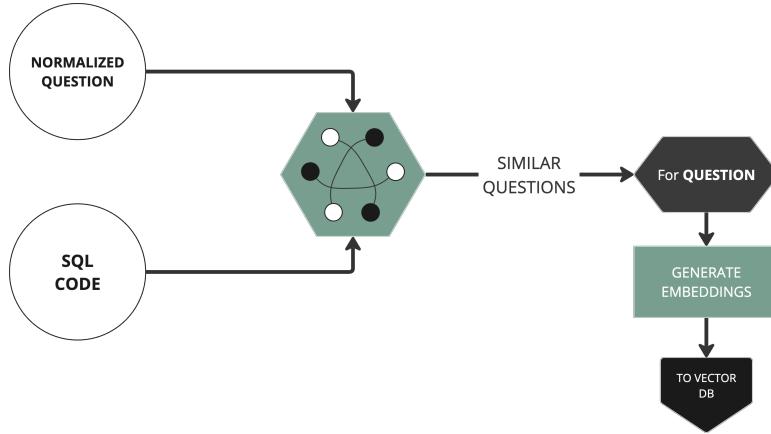
- \mathcal{M}_k — The **Main Clause**, representing the **primary data request**. Retains the essential meaning of the query. Non-critical modifiers such as filters, grouping, and sorting are excluded.
- \mathcal{M}'_k — The **Details** that expand the **Main Clause**, containing contextual refinements and constraints. Retains **all auxiliary conditions** without altering the core request. Grouping, filtering, and ordering constraints are explicitly preserved.

Example Transformations:

User Query	Extracted Main Clause	Extracted Details
"Show me a list of sales transactions for Q4, including product details and revenue"	"list of sales transactions"	"for Q4, including product details and revenue"
"Display the number of active users by country over the past month"	"number of active users"	"by country over the past month"
"Get the total revenue for each department last year"	"total revenue for each department"	"over a defined time-frame"

III. ENRICHMENT OF THE EXAMPLE WITH THE VARIATION OF QUESTION

In order to increase the number of embeddings that cover an example, we generate questions that are similar to the normalized initial question, considering SQL code as an answer.



Process of generating variations of questions to enrich examples.

Let's consider the example:

Question	Show me a list of events in the past month including name, permalink, start date, end date, and RSVP count
SQL Code	<pre> SELECT title AS name, permalink, start_at AS start_date, end_at AS end_date, rsvp_count FROM events WHERE start_at >= DATEADD(month, -1, GETDATE()) AND start_at < GETDATE(); </pre>
Normalized	"Retrieve a list of events over a defined timeframe including their name, permalink, start date, end date, and RSVP count"

The initial question was transformed to the normalized form "**Retrieve a list of events over a defined timeframe including their name, permalink, start date, end date, and RSVP count**", which

- **contains** all entities that has to be retrieved
- **contains** identifier that the request contains some specific timeframe reference
- **does not contain** any specific details that contradict to the generalisation's principals

The requested entity and main clause are "**list of events with their characteristics**" and "**list of events in the past month**" correspondingly. In order to provide the wider coverage, we generated the list of three additional questions similar to the normalized representation of the request:

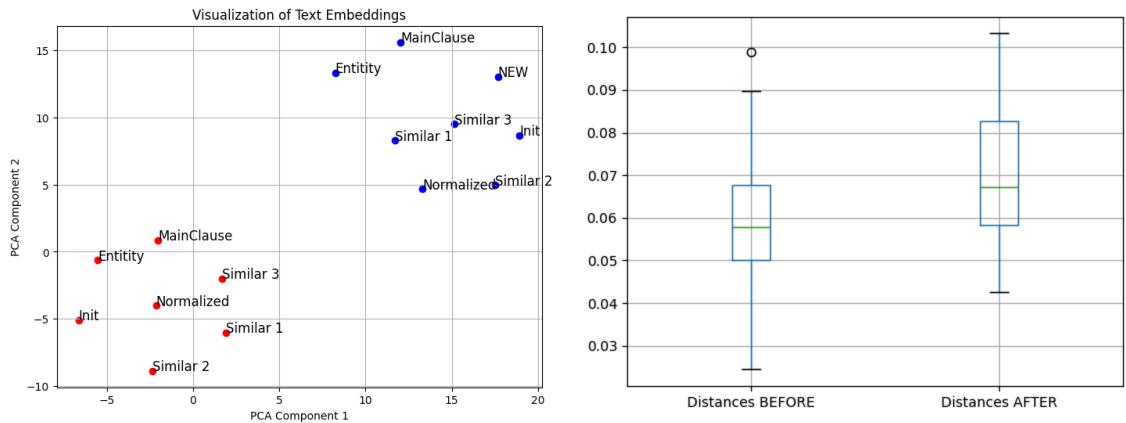
- Provide a list of events occurring within a specified time range, including their names, permalinks, start dates, end dates, and the number of RSVPs.

- Can you give me the details of events within a certain period, listing their name, permalink, start date, end date, and RSVP count?
- Generate a report of events within a certain period, showing their name, permalink, start date, end date, and number of RSVPs

So, the example will be described via seven embeddings that increase the invariance to the request's specific when the user asks about the same concepts.

Let's look at the diagrams below. The left diagram represents **PCA Visualization of Text Embeddings** for two examples. Each color-coded group relates to an example, which is represented via embeddings that originate from an initial question. Each labeled point corresponds to a text embedding, with names such as "MainClause", "Entity", "Normalized", "Similar 1", "Similar 2", "Similar 3", and "Init." We may see that these groups are well distinguished - embeddings within the same group (same example) tend to be closer together.

The diagram on the right represents the inner-cluster pairwise distance comparison **before and after** adding similar questions. It provides insight into how the embeddings' spread changed with the new data. According to the provided box plots, we may see that **before adding similar questions**, the embeddings were more compact, meaning that questions and their variations were closely related to the embedding space. **After adding similar questions**, the increase in distances suggests that the embeddings became more dispersed, possibly introducing more variety and reducing redundancy in the representations.



PCA visualization of text embeddings for two example clusters.

Comparison of intra-cluster distances before and after adding similar questions.

The wider distribution of the embedding suggests that new (similar) questions were not the exact duplicates of the ones present in the training corpus. Rather, they were a paraphrase or a different formulation of the query, which made the embeddings more diverse. This shows that the model has improved its ability to deal with the different forms of the same query, which makes the model more robust in the retrieval tasks. The increase in the median distance shows that the model has learned to tell the difference between similar questions better, which in turn enhances the generality of the encoding across the phrasing. This improvement makes the retrieval step inside a Retrieval-Augmented Generation (RAG) system less dependent on the particular words used in a request so that similar queries will always return relevant results no matter what phrase is used.

Finally, each example is stored in the Vector DB via a cloud of embeddings created for the Initial Question and its variations, Normalized question, Main Clause, and other extracted entities.

Domain-Specific Instructions

Except for the overall and comprehensive question-answer examples description to be stored in the Text2SQL model, we perform the auto-generation of the domain-specific instructions that are

inferred from the examples and contain the reasoning bridge between user questions expressed in natural language and the underlying database schema. By applying systematic analysis of question-SQL pairs, we receive structured mappings that associate domain entities with the corresponding database tables.

The domain-specific mapping may be described as a multi-stage process that extracts knowledge from each question-SQL pair to create a complete mapping framework:

I. TABLE SELECTION AND JUSTIFICATION

For each Question-SQL example \mathcal{A}_k , we identify the relevant tables required to answer the question. This selection process is performed by a large language model (LLM), which not only identifies the necessary tables but also provides explicit justification for each selection. The input to this process includes:

- \mathcal{N}_k — The **natural language question** form k -th example (normalized form)
- $\{T_{ik}\}$ — A set of **database tables** from k -th example
- \mathbf{R} — A set of **business rules** guiding table selection.
- \mathcal{S}_{code_k} — The **correct SQL query** (used for validation)

The **LLM-based table selection** function is represented as:

$$\{\mathcal{T}_{LLM_k}; \mathcal{W}_{LLM_k}\} = \text{LLM}(\mathcal{I}_k, \{T_{ik}\}, \mathbf{R}, \mathcal{S}_{code_k})$$

where

\mathcal{T}_{LLM} is the set of tables selected by the model.

\mathcal{W}_{LLM} - justifications explaining **why** each table is necessary

The language model produces a list of required tables along with justifications explaining why each table is necessary for answering the question. This output is then structured into a consistent JSON format for further processing.

II. DOMAIN-SPECIFIC ENTITIES EXTRACTION AND CLASSIFICATION

From the collected examples and table selections, we derive a comprehensive set of domain entities, categorizing them as either minor (specific attributes) or major (conceptual groupings):

1. **Minor Entities** (E_m): Specific attributes or discrete data points within the domain, such as "event name," "event start date," or "event RSVP count". These entities typically map directly to table columns or simple joins.
2. **Major Entities** (E_M): Generalized concepts that often require multiple tables or complex relationships to represent, such as "step parameters" or "activists subscribed to email." These entities frequently encapsulate business logic or domain workflows.

The **entity classification process** is performed using a **Large Language Model (LLM)**, which analyzes the retrieved tables, metadata, and business rules to assign entities accordingly. This process is formulated as:

$$E_{LLM_k} = \text{LLM}(\mathcal{T}_{LLM_k}, \mathcal{W}_{LLM_k}, R_{class})$$

where:

- E_{LLM_k} represents the **set of extracted entities**.
- R_{class} specific **rules**, which help refine the classification.

For each **extracted entity**, the model generates a structured mapping that defines its role, associated tables, and extraction methodology. This mapping is represented as:

$$\mathcal{E}'_k = (E_j \in E_{LLM_k}, T_j \subset T_{LLM_k}, \text{class}(E_j)); \mathcal{E}'_k \in \mathbf{E}'_k$$

where:

- E_j — **entity name** and its **conceptual role** within the domain.
- T_j — **database tables** required to access this entity.
- $\text{class}(E_i)$ — **classification type**, indicating whether the entity is **Minor** (E_m) or **Major** (E_M)
- \mathbf{E}'_k — full set of of domain-specific entities derived from \mathcal{A}_k example

The domain-specific mapping is the full-text instructions that are created based on union of the domain-specific entities per each example:

$$\text{Instr_Domain} = f_{text}(\bigcup \mathbf{E}'_k)$$

EXAMPLE

Let us consider the a simplified example, to illustrate the process described above.

Question: "How many people RSVP'd to the climate march event last month?"

SQL Query:

```
SELECT COUNT(rsvps.id)
FROM rsvps
JOIN events ON rsvps.event_id = events.id
WHERE events.name LIKE '%climate march%'
AND events.start_date BETWEEN DATE_SUB(CURDATE(), INTERVAL 2 MONTH)
AND DATE_SUB(CURDATE(), INTERVAL 1 MONTH)
```

Selected Tables:

- "events" - To identify the specific climate march event and its timing
- "rsvps" - To count the number of people who RSVP'd to the event

Derived Entity Mappings:

Minor Entities:

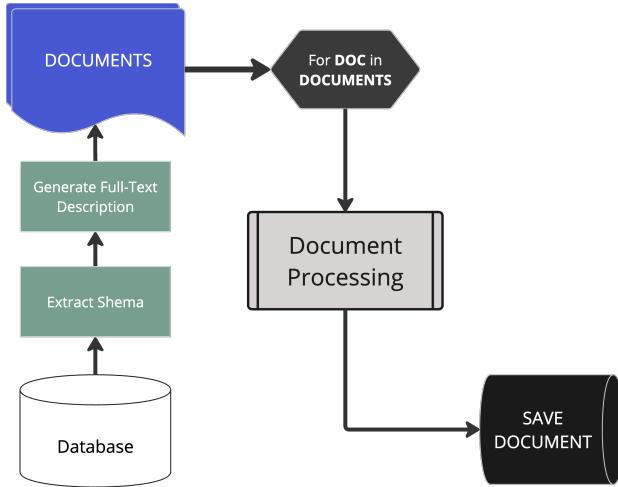
- "event name" → table: events, column: name
- "event start date" → table: events, column: start_date
- "RSVP count" → table: rsvps, aggregation: COUNT(id)

Major Entity:

- "event participation" → tables: events + rsvps, joined on event_id

Train based on DB Schema

The train based on DB schema is leaded to the process of documents creation based on full-text description generation with the following training based on documentation, as was described above.



Process flow for training based on database schema.

Conclusion

The training of the **Knowledge Base** creates the foundation for the Text2SQL model, enabling accurate and context-aware SQL query generation. By structuring the Knowledge Base with database documentation, example queries, and domain-specific mappings, the system ensures efficient retrieval and highly reliable response generation:

- The **documentation-driven training** creates a structured representation of database tables, relationships, and dependencies that allows the model to understand the database schema effectively.
- The **example-based training** enhances SQL query generation by leveraging stored question-SQL pairs, improving the system's ability to interpret natural language queries and retrieve relevant patterns. The process of query normalization, entity extraction, data source identification, and operation structuring refine query construction, leading to greater accuracy and adaptability.
- The **domain-specific instructions** serve as a reasoning bridge between natural language queries and the underlying database schema, ensuring correct table selection, attribute mapping, and relationship definition.

Retrieval Content from Knowledge Base

The Knowledge Base, which contains overall and comprehended information about the database and question-answering examples, provides our agent with the necessary context to answer the user's question via SQL prompt. The content retrieval procedure is based on the following units:

1. **Retrieve Appropriate Examples** - retrieve similar SQL-related questions from a Knowledge Base (Vector Database) using a similarity search approach. It also incorporates optional reranking to refine the results. The results of this stage are used as direct examples of code that should be either reproduced with some changes or serve as inspiration to generate SQL as an answer to the question.
2. **Retrieve Documentation on Related Tables** - determine the most relevant database tables for a given natural language query. It is based on a semantic search in Vector Database, with the following results refined using a reasoning-based large language model and the incorporation of domain-specific business rules. Documentation provides us with the overall vision of the tables used in the output SQL code, including the detailed description of columns and their dependencies, which minimizes the risk of incorrect answers.

Retrieve Appropriate Examples

In order to find appropriate examples to generate the SQL code that answers the question, we implemented the iterative approach by gradually expanding the search space due to the consequential request's specificity, reducing and decreasing the similarity threshold.

$$\mathcal{Y}^* = \{Y_j \mid \text{Sim}(X, Y_i) \geq \tau\}, \text{Sim}(X, Y_i) = \frac{\cos(\theta)+1}{2} = \frac{\mathbf{X} \cdot \mathbf{Y}_i + \|\mathbf{X}\| \|\mathbf{Y}_i\|}{2\|\mathbf{X}\| \|\mathbf{Y}_i\|}$$

where

X - vector representation of user's question or it's derivatives

Y_i - stored vector representations of example's descriptors, such as full-text question, it's normalized form, extracted entities, etc.

\mathcal{Y}^* - is the set of all retrieved examples Y_i

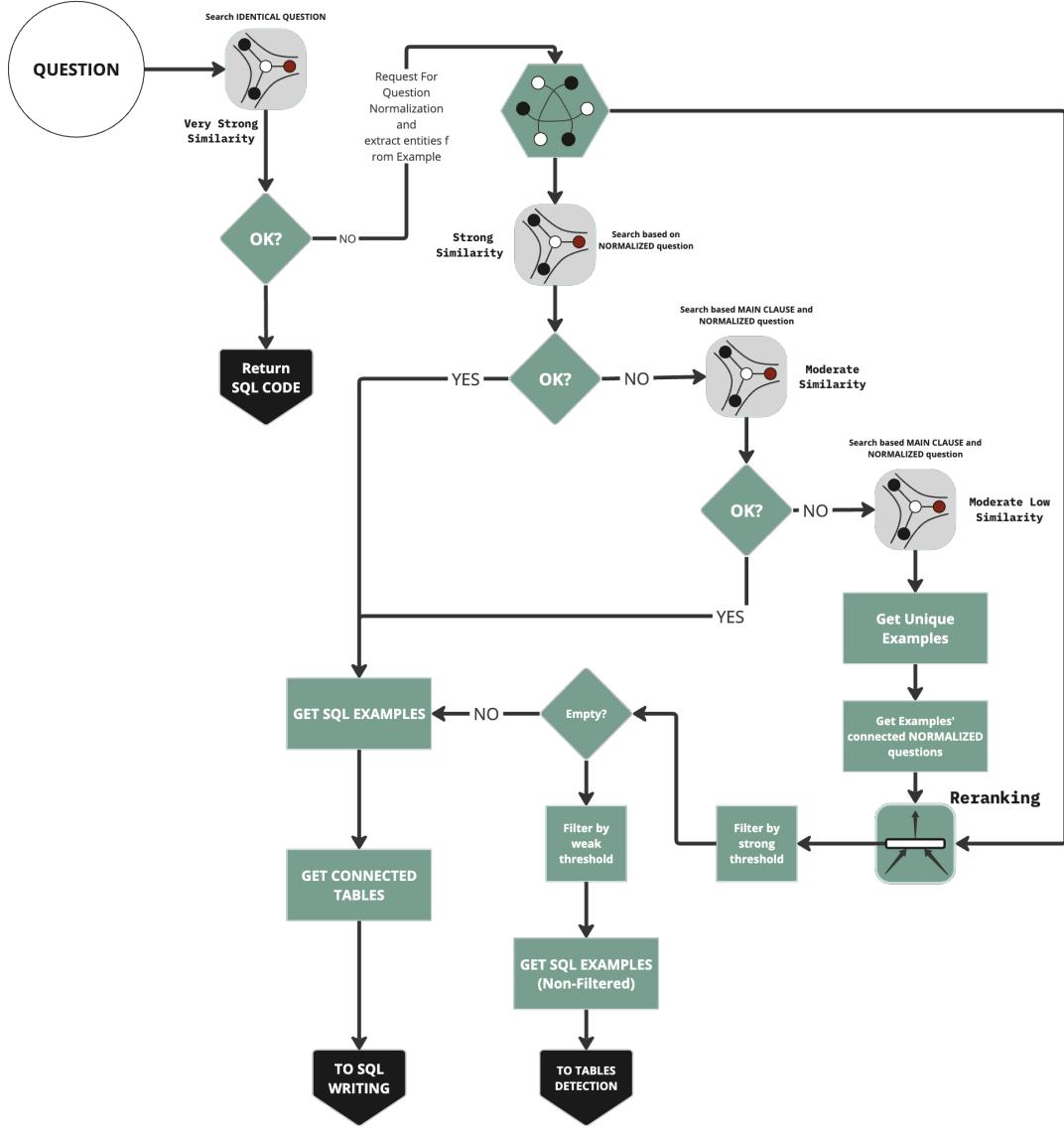
τ - similarity threshold

The process of the appropriate examples retrieval might be depicted via diagram below.

The examples' retrieval flow might be divided into two global stages:

- extract examples that perfectly match the user's request
- extract examples that are considered as additional support for the documentation in order to reduce the error risks in the SQL code-writing stage

In the first case (if we managed to find the appropriate examples), the phase of the documents's retrieval might be skipped because all necessary information for SQL generation is already in the extracted examples. The second stage is supposed to execute the **documents's retrieval** procedure as a main source of information for SQL generation.



Example retrieval flow with progressive thresholds.

I. FIND EXACT EXAMPLE

So, we start with an attempt to find the exact question already answered in the Knowledge Database. For this purpose, a very strong similarity ($\text{Sim}(X, Y) \rightarrow 1$) is expected - in the limit this step is equal to the direct match between user's request and the question in Knowledge Base. The idea here is to avoid extra calculation if we have already answered this question - we may just return the stored SQL example as an answer.

$$S_{sql}^R = \mathcal{S}_{code_k} \in \mathcal{A}_k | \text{Sim}(I, \mathcal{I}_k) \rightarrow 1$$

where

I — initial representation of the user's request

S_{code}^R — resulted SQL query

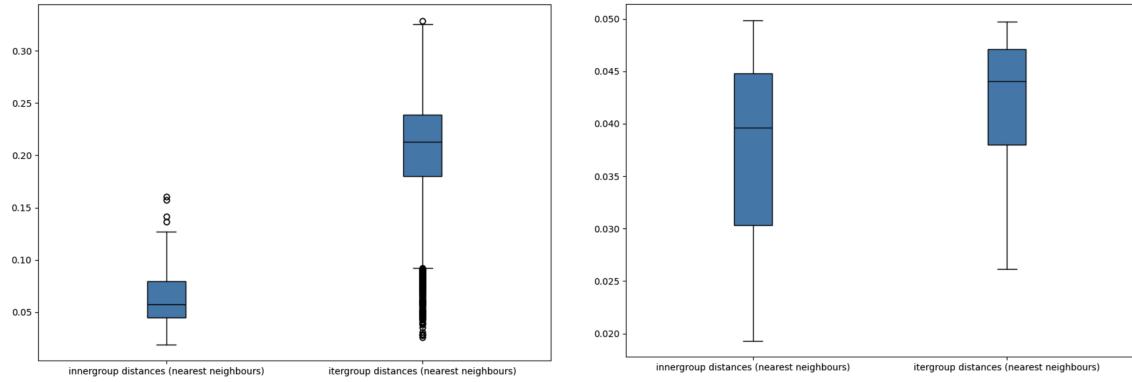
II. SEARCH BASED ON NORMALIZED REPRESENTATION

The next step is the attempt to find an example described by similar questions from several perspectives - normalized representation and main clause. For this purpose, we need to transform

the user's request to the **normalized form** and extract the **main clause** following the same process used during model training.

Since the normalized question is subject to the "**Normalize the question rules**" (see *Example Normalization and Structuring*), we may expect high similarity between normalized versions of the different questions, even if they differ with specific details or slightly differ in writing style, as long as they convey the same meaning from a generalized perspective.

The boxplots below illustrate this concept. The diagrams display the distribution of pairwise distances between normalized questions and their alternatives. The diagrams compare the distances between **nearest neighbours within the same example (intragroup)** and between different examples (*intergroup*) (for all dynamic diapason on the left, and limited by **0.05** - on the right).



Distribution of pairwise distances between normalized questions (full range).

Distribution of pairwise distances between normalized questions (limited to 0.05).

Nearest neighbours within the same example (between different examples) refer to the closest matching questions, based on cosine similarity, that belong to the same **question-answer example** or different **question-answer example** correspondingly. Given an example G_m consisting of multiple question representations $\{Q_1, Q_2, \dots, Q_k\}$, the **nearest neighbours within the same example** are defined as:

$$\mathcal{D}_{intra/int}(Q_i)_N = \arg \min_{Q_j \in G_m, j \neq i} d(Q_i, Q_j) = \arg \max_{Q_j \in G_m, j \neq i} \text{Sim}(Q_i, Q_j)$$

where

$d(Q_i, Q_j)$ - represents the pairwise cosine distance between embeddings of the two questions

For **intra-case**:

- $Q_i, Q_j \in G_m$ - are different representations of the same example
- The nearest neighbour Q_j is the one with the smallest distance to Q_i within the same example group.

For **inter-case**:

- $Q_i \in G_m$ and $Q_j \in G_n$ where $G_n \neq G_m$ (questions come from different groups).
- The nearest neighbour Q_j is selected **only from a different group**.

At this stage, we consider the questions that are the **normalized representation** of the original question and its **alternatives**, such as variations in phrasing or minor details, but with the same underlying meaning - $\mathcal{D}_{intra}(Q_i)_N$ and $\mathcal{D}_{inter}(Q_i)_N$.

Regarding the normalized representation of a question, a cosine distance of less than P_5 \mathcal{D}_{INT-N}

$P_5 = \text{percentile}(x, 5)$, where $D_{INT-N} = \{x \in D_{INT-N}\}$ - inter-group distances for **normalized questions group**, in our embedding model, enables the detection of semantically equivalent questions while reducing the risk of misidentification. Therefore, by setting the similarity threshold to

$\text{Sim}(A, B) \geq P_{95} = \text{percentile}(x, 95)$, where $S_{INT-N} = \{S_{INT-N}\}$ - inter-group similarity for **normalized questions group**, for the **normalized form** of the user's question, we ensure the correct positioning of the user's request among the saved examples. The retrieved example can be used as a template to generate the correct SQL code without needing to include the documentation in the context, as it already contains all essential logic, differing only in minor details such as timeframe, geolocation, etc.

$$\mathbf{A}_{res} = \{\mathcal{A}_k\}, \text{ where } \text{Sim}(N, Q_k) \geq \mathcal{P}_{95}^N, Q_k \in \{\mathcal{I}_k, \mathcal{N}_k\}; \mathcal{P}_{95}^N = P_{95} \quad S_{INT-N}$$

$$\mathcal{T}_{code} = \{\mathcal{T}_k | \mathcal{T}_k \in \mathcal{A}_k \forall \mathcal{A}_k \in \mathbf{A}_{res}\}$$

$$\mathcal{S}_{code}^{**} = \{\mathcal{S}_{code_k} | \mathcal{S}_{code_k} \in \mathcal{A}_k \forall \mathcal{A}_k \in \mathbf{A}_{res}\}$$

where

\mathcal{T}_{code} — full-text descriptions of all tables from the retrieved examples

\mathcal{S}_{code}^{**} — set of SQL-queries from the retrieved examples that have perfect match with the normalized question

III. SEARCH BASED ON NORMALIZED REPRESENTATION AND MAIN CLAUSE

If there is no certainty that an example in the Knowledge Base directly matches the user's request, we need to expand the search space by lowering the similarity threshold and using the question's **main clause** as an additional input for the search engine (vector database). The table below depicts the inter- and inner-group similarity distribution for the **nearest neighbours within the same example** for both normalized question and its main clause - $\mathcal{D}_{intra}(Q_i)_{N \cup M}$ and $\mathcal{D}_{inter}(Q_i)_{N \cup M}$, built on the examples from the training dataset.

Percentiles	99%	95%	90%	75%	50%	25%	10%	05%	01%
Inner-group	0.99	0.99	0.98	0.98	0.97	0.95	0.95	0.94	0.93
Inter-group	0.96	0.95	0.95	0.93	0.91	0.88	0.86	0.84	0.83

To **maximize precision while maintaining a moderate recall**, we set the similarity threshold at:

$$\text{Sim}(X, Y) > P_{99} = \mathcal{P}'_{99}^{NM}$$

where:

$S_{inter_{N \cup M}} = S_{inter_N} \cup S_{inter_M}$ represents **inter-group similarity** for the **normalized questions group expanded by main clause**.

P_{99} of **inter-group similarity** is **0.96**, meaning 99% of inter-group distances are below this threshold.

By applying the threshold \mathcal{P}'_{99}^{NM} (99% percentile of **inter-group similarity** for the **normalized questions group expanded by main clause**) instead of \mathcal{P}_{95}^N (95% percentile of **intra-group similarity** for the **normalized questions**), we ensure that **retrieved examples are strongly related to the user's request**, effectively minimizing **false positives** and **improving precision**. Since only **1%** of **inter-group examples** have a similarity above **0.96**, the risk of retrieving

incorrect examples is minimal. However, in this case, we also need to consider the table description as an additional source of information since the examples themselves may not always include the specific attributes requested.

$$\mathbf{A}_{res} = \{\mathcal{A}_k\}, \text{ where } \text{Sim}(N, Q_k) \geq \mathcal{P}'_{99}^{\mathcal{NM}}, Q_k \in \{\mathcal{I}_k, \mathcal{N}_k, \mathcal{M}_k\}$$

$$\mathcal{T}_{code} = \{\mathcal{T}_k | \mathcal{T}_k \in \mathcal{A}_k \forall \mathcal{A}_k \in \mathbf{A}_{res}\}$$

$$\mathcal{S}_{code}^* = \{\mathcal{S}_{code_k} | \mathcal{S}_{code_k} \in \mathcal{A}_k \forall \mathcal{A}_k \in \mathbf{A}_{res}\}$$

where

\mathcal{S}_{code}^* — set of SQL-queries from the retrieved examples that have sufficient match with the normalized question

IV. SEARCH BASED EXPANDED SET OF QUESTION REPRESENTATION

If we are unable to find examples that meet the similarity requirements, we need to lower the similarity threshold to expand the search space (increase the recall). To achieve this, we must consider the full set of pairwise similarities (both intra-group and inter-group) without limitation by nearest neighbours:

$$\mathbf{S} = \{\text{Sim}(Q_i, Q_j) | Q_i, Q_j \in G, i \neq j\} | Q_i \in \{\mathcal{N}_i, \mathcal{M}_i, \mathcal{E}_i, \mathcal{I}_i\} \forall \mathcal{A}_i \in \mathbf{A}$$

where:

\mathbf{S} represents the **full set of pairwise similarities**, covering both intra-group and inter-group distances.

G is the complete set of all examples.

$Q_i, Q_j \in G$ are different questions, meaning the set includes distances from **both the same and different groups**.

$\text{Sim}(Q_i, Q_j)$ is the cosine similarity between two questions

$\{\mathcal{N}_i, \mathcal{M}_i, \mathcal{E}_i, \mathcal{I}_i\}$ - set that includes normalized question, main clause, entities and init question correspondingly for \mathcal{A}_i example

As we want to maximize the coverage of the relevant examples we may set a similarity threshold equals to $\mathcal{P}_{01} = \frac{P_{01}}{S_{intra}}$.

By applying this threshold, we **increase the probability of retrieving examples relevant to the user's request**, as only **1% of intra-group examples** have a similarity score below the defined threshold. However, this comes **at the cost of reduced precision**.

$$\mathbf{A}'_{res} = \{\mathcal{A}_k\}, \text{ where } \text{Sim}(N, Q_k) \geq \mathcal{P}_{01}, Q_k \in \{\mathcal{I}_{||}, \mathcal{N}_{||}, \mathcal{M}_{||}\}$$

$$\mathcal{T}_{code} = \{\mathcal{T}_k | \mathcal{T}_k \in \mathcal{A}_k \forall \mathcal{A}_k \in \mathbf{A}'_{res}\}$$

$$\mathcal{S}_{code} = \{\mathcal{S}_{code_k} | \mathcal{S}_{code_k} \in \mathcal{A}_k \forall \mathcal{A}_k \in \mathbf{A}'_{res}\}$$

where

\mathcal{S}_{code} — set of SQL-queries from the retrieved examples that have should be used as code template examples

Looking at the distributions of intra-group and inter-group similarities, we observe a **20% chance** that the retrieved example set will include incorrect matches.

Percentiles	99%	95%	90%	80%	50%	25%	10%	05%	01%
Inner-group	0.99	0.97	0.96	0.95	0.93	0.90	0.87	0.84	0.82

Percentiles	99%	95%	90%	80%	50%	25%	10%	05%	01%
Inter-group	0.98	0.88	0.86	0.83	0.78	0.76	0.74	0.72	0.70

This means that while the retrieved set of examples contains relevant information, it also has a high level of noise. Therefore, an additional step is required to review the relevance of the extracted **question-code** representatives and filter out highly irrelevant ones.

V. RERANKING

To achieve this, we apply a cross-encoder model trained on the **Quora Duplicate Questions** dataset. This approach allows us to directly assess the semantic similarity between the **questions**, which accompany extracted examples, and the **user's request**, providing a more accurate relevance ranking due to the ability of the cross-encoder to jointly process both inputs, capturing deeper contextual relationships. The **Quora Duplicate Questions** dataset is specifically designed for **paraphrase identification**, so the cross-encoder model has the ability to recognize if two questions have the same meaning with differences in wording. This perfectly serves our goal - detect **semantically similar** examples to the **user's request**, ignoring lexical and style variations.

$$\mathbf{A}_{res} = \{\mathcal{A}_k \in \mathbf{A}'_{res} \mid getRank(\mathcal{A}_k) > \tau'\}$$

$$\mathcal{T}_{code} = \{\mathcal{T}_k \mid \mathcal{T}_k \in \mathcal{A}_k \forall \mathcal{A}_k \in \mathbf{A}_{res}\} \mid \mathbf{A}_{res} \neq \emptyset$$

$$\mathcal{S}_{code}^* = \{\mathcal{S}_{code_k} \mid \mathcal{S}_{code_k} \in \mathcal{A}_k \forall \mathcal{A}_k \in \mathbf{A}_{res}\} \mid \mathbf{A}_{res} \neq \emptyset$$

where

\mathcal{S}_{code}^* — set of SQL-queries from the retrieved examples that have sufficient match with the normalized question

τ' — the reranking threshold

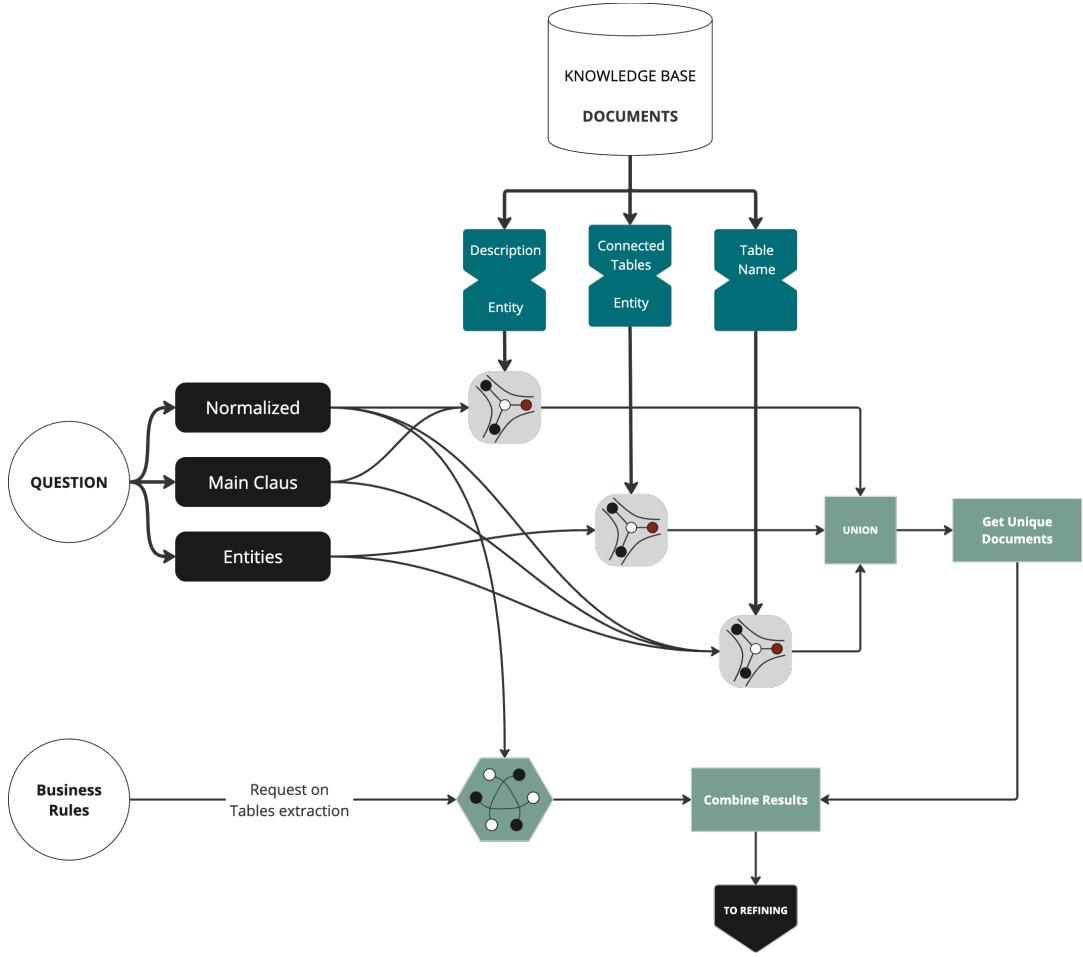
If we managed to find the relevant examples to the input request $(\mathcal{S}_{code}^*, \mathcal{T}_{code})$, we put them into context for SQL code generation, preliminary accompanied them with the related tables description. Otherwise, we need to take the codes examples and the corresponded tables $(\mathcal{S}_{code}, \mathcal{T}_{code})$ and proceed with the **Documentation on Related Tables** retrieval.

Retrieve Documentation on Related Tables

The **Retrieve Documentation** phase of the retrieval process works in two steps to identify the most relevant tables for generating SQL queries. First, we use content similarity to match the user's question with table descriptions and metadata. Then, we apply business rules to refine these results. The content similarity approach, described below, analyzes different versions of the user's question to find tables that are most likely to contain the required information. The general schema of this process is depicted on a figure below ("Document retrieval process flow").

I. SEARCH TABLES DOCUMENTATION VIA CONTENT SIMILARITY

At this stage, we try to identify the **tables** relevant to the user's request by searching in the Knowledge Base (documents collection) while taking into account the **document's metadata** - the procedure is designed to cover different perspectives of the **database tables** by querying metadata categories such as **description**, **entity**, **connected tables**, and **table names**. This process includes the search around the different representations of the user's query, including its **normalized representation**, **main clause**, and **extracted concepts**. By using this approach, we ensure a comprehensive retrieval of relevant tables while minimizing missing entries.



Document retrieval process flow.

To detect tables that match the user's request and might be used at the stage of SQL code generation, we execute multiple queries using different representations of the user's request. Each query is formulated as follows:

$$\mathcal{Z}_i = (z_i, \mathcal{F}_i, n_i)$$

where:

z_i is the query text derived from one of the following sources: **main clause** ($z_{\text{main}} = \mathcal{M}$), **normalized question** ($z_{\text{norm}} = \mathcal{N}$), **extracted concepts** ($z_{\text{concepts}} = \mathcal{E}$)

\mathcal{F}_i is the **filter condition** applied to metadata categories, which restricts the search space.

n_i is the **number of results** to retrieve per query.

The set of query parameters is defined as:

$$\mathcal{Z} = \{\mathcal{Z}_1, \mathcal{Z}_2, \dots, \mathcal{Z}_n\}$$

where each query applies different filters:

1. **Matching descriptions & entities with normalized question \mathcal{N} and main clause \mathcal{M}**
 $\mathcal{F}_1 = \mathcal{F}_2 = \{"\text{category"} \in \{\text{"description"}, \text{"entity"}\}\}$
2. **Matching connected tables & entities with extracted concepts \mathcal{E}**
 $\mathcal{F}_3 = \{"\text{category"} \in \{\text{"connected_tables"}, \text{"entity"}\}\}$
3. **Matching table names with all of these categories**

$$\mathcal{F}_4 = \mathcal{F}_5 = \mathcal{F}_6 = \{"category" = "table_name"\}$$

For each query, we perform retrieval from the documentation collection \mathcal{C} :

$$\mathcal{R}_i = \text{query}(\mathcal{C}, z_i, \mathcal{F}_i, n_i)$$

where \mathcal{R}_i represents the set of retrieved metadata entries. The final retrieved set is the union of all query results:

$$\mathcal{R} = \bigcup_{i=1}^6 \mathcal{R}_i$$

Each retrieved entry $r \in \mathcal{R}$ consists of metadata fields: **table name** (T'), **dependencies** (D'), **connected tables** (CT')

We construct a structured representation of the retrieved data:

$$\mathcal{Q} = \{\text{json}(T', D', CT') \mid (T', D', CT') \in \mathcal{R}\}$$

To ensure uniqueness, we remove duplicate representations:

$$\mathcal{Q}_{\text{unique}} = \text{unique}(\mathcal{Q})$$

Each unique metadata representation is then converted into structured documentation entries:

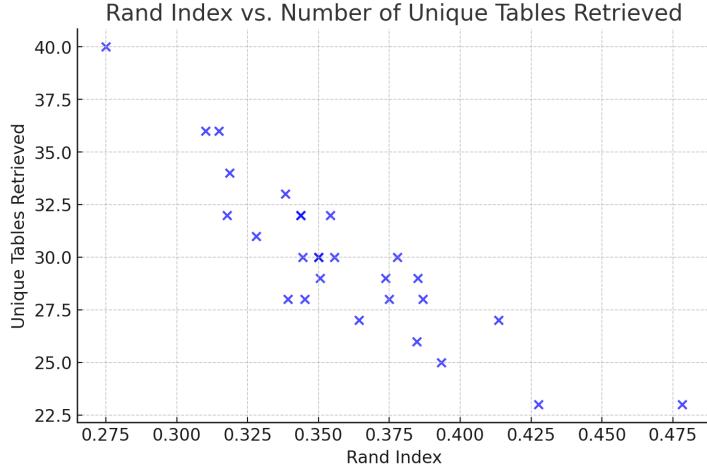
$$Doc = \{\text{get_docs}(q) \mid q \in \mathcal{Q}_{\text{unique}}\}$$

The final **tables** set is obtained by merging all document entries:

$$\mathcal{T}_{\text{det}} = \bigcup_{d \in Doc} d$$

Finally, \mathcal{T}_{det} represents the set of detected tables, ensuring that all relevant table names and their relationships are correctly identified.

In the current implementation, we set $n_i = 10$, which leads to retrieving about **30 unique documents** with table descriptions (corresponding to **30 unique tables**) on average. This ensures **broad coverage**, considering that a typical SQL query involves **4 to 5 relevant tables** while maintaining computational efficiency. The objective of the retrieval process is to maximize the recall (inclusion of relevant tables) and, at the same time, reduce the noise level, ensuring that the following steps operate on a well-curated tables' candidate set.



Relationship between Rand Index and number of unique tables retrieved.

An empirical analysis of retrieval results based on training data provides insights into the distribution of the retrieved tables and the extent of their overlap. According to the results, the **number of unique tables** is in the range of **23 to 40**, with a mean of approximately **30 tables per query**. This variation proves the ability of the search strategy to **adapt to different query structures and metadata distributions**, ensuring that a set of candidates is diverse but not exhaustive.

This statement is reflected in the **scatter plot** that depicts the relationship between the **Rand Index and the Number of Unique Tables Retrieved**.

The **moderate overlap** between retrieved sets, measured by a **Rand Index** of **0.36**, shows that different query formulations bring in **new relevant tables** while still having some common results. This balance indicates that the search strategy is **effective and reliable**, using multiple query approaches to capture more relevant tables without too much repetition. The results confirm that the retrieval process maintains a good mix of variety and consistency, ensuring that important tables are included while keeping the search efficient.

II. SEARCH TABLES DOCUMENTATION BASED ON BUSINESS RULES

In addition to the fully automated retrieval of table documentation, we incorporate **domain knowledge** to refine the selection of relevant tables. This process accounts for **business rules**, which define constraints, dependencies, and conditions that ensure table selection aligns with organizational requirements. Despite the fact that table descriptions provide a foundational understanding, they may not always explicitly capture the relationships between data elements and business-specific terminology. Variations in **wording, domain-specific concepts, and implicit assumptions** further contribute to potential ambiguities in table selection.

To solve this, we apply **table detection via an LLM query** structured as follows:

$$\mathcal{T}_B = \text{LLM}(N, R)$$

where:

\mathcal{T}_B represents the refined **set of tables selected based on business rules**

N represents the **normalized user question**.

R represents the **set of business rules** applicable to table selection.

After this step we expand the list of the tables-candidates to the SQL code writing step with \mathcal{T}_B :

$$\mathcal{T}_{total} = \mathcal{T}_{det} \cup \mathcal{T}_B$$

Refining the Search Results

Refining table selection is the process of improving retrieval accuracy by using **LLM** to incorporate domain-specific instructions, business rules, and structured metadata. This process guarantees that table selection corresponds with the business logic and the database structure, resolves ambiguities, and ensures the coverage of all the tables.

The **refinement stage** is based on results from both:

1. **Retrieve Documentation on Related Tables:** metadata-based retrieval of candidate tables from the documentation.
2. **Retrieve Appropriate Examples:** retrieval of tables referenced in similar SQL examples.

The initial retrieval is performed separately in these two stages, and therefore, there is a need to use reasoning to combine, validate, and refine the selection to retain only the most relevant tables.

I. LLM-DRIVEN TABLE REFINEMENT

The **Table Refining** prompt structures the LLM's reasoning into a systematic **step-by-step analysis**, ensuring **logical consistency** and **explicit justification** for table selection:

$$\mathcal{T}_R = \text{LLM}(N, \text{Instr_Domain}, R, \mathcal{T}_{total} \cup \mathcal{T}_{code})$$

where:

\mathcal{T}_R — is the **final refined set of tables** selected by the model.

N — normalized representation of user's request

Instr_Domain — domain-specific mapping instructions

R — business rules

\mathcal{T}_{total} — the candidate Tables with Descriptions received on the document's retrieval stage

\mathcal{T}_{code} — the candidate Tables with Descriptions received on the example's retrieval stage

II. REFINEMENT PROCESS & LLM REASONING

- **Table Coverage Analysis:**

- The model reviews table descriptions to determine what particular cases data is placed into.
- A detailed column analysis is conducted to establish whether the required characteristics are present.

- **Comprehensive Table Coverage**

- The model selects all tables that may contain the needed information while they may not have been mentioned in the question directly.
- It also considers linked or dependent tables that may be needed to get the full picture of the data.

- **Business Logic and Context Integrated**

- Domain-specific rules are directly incorporated into the reasoning process to refine table selection.
- The model's decision-making is consistent with business goals. Therefore it will select tables that are relevant to the analytical or operational goals that need to be achieved.

As a result of combining the outcomes of the two previous stages, this approach guarantees that the final **refined table selection** is:

- **Comprehensive** — includes both metadata descriptions and real query usage
- **Contextually accurate** — understanding **domain-specific mappings** and **business constraints**
- **Optimized for SQL generation** — to make sure that the right data is **retrieved, prepared, and lined up** with what the user is looking for.

This refinement process enhances the **precision and reliability** of table selection for automated query generation. The result of this stage is a cortege $(\mathcal{S}_{code}, \mathcal{T}_R)$ - SQL queries examples, received on the **Retrieve Appropriate Examples** stage, which has a recommended character for SQL-writing procedure, and the refined set of tables with their description.

Conclusion

The retrieval process from the Knowledge Base was designed to receive examples and table documentation that are relevant to the user's request, which is necessary for the SQL query generation process. With the help of **vector-based similarity searches**, the system is capable of effectively retrieving the relevant SQL query examples and their associated table documentation through **reranking techniques** and **business-rule-driven refinements**.

A multi-step approach, which is described in the document, allows for the optimization **of precision** without compromising on **low error rates** in table selection, thus ensuring that the result is consistent with both the database structure and domain requirements. The extracted context, based on a **refined set of tables** and **retrieved SQL examples**, is vital in the query generation process because it serves as a basis for the accuracy, efficiency, and contextual consistency of the whole process.

SQL Code Generation

SQL code generation is the **final step** of the Text2SQL pipeline, where structured query information is transformed into an executable SQL statement. The generation process depends on previously retrieved **SQL examples** and **table descriptions**, which guide the model in selecting the most relevant **operations, filters, and table joins**.

Based on the analysis of retrieved data, the SQL code can be generated in one of the following ways:

1. Direct Output SQL Code

- S_{code}^R — A fully formed SQL query that is **ready for execution** without further modification.

2. Exact Example-Based Generation

- \mathcal{S}_{code}^{**} — A set of SQL query examples that **perfectly match** the user's request.
- These examples are used **directly** to construct the query.
- The **table descriptions** from \mathcal{T}_{code} serve as supporting context.

3. Strongly Aligned Example-Based Generation

- \mathcal{S}_{code}^* — A set of SQL query examples that **closely match** the user's request.
- The final SQL query is **strongly influenced** by these examples, but table selection and structure depend equally on \mathcal{T}_{code} .

4. Flexible Template-Based Generation

- \mathcal{S}_{code} — A set of SQL query examples that have **partial similarity** to the user's request.
- These examples act as **non-mandatory reference templates**, while table selection follows the broader **refined table set** \mathcal{T}_R from the **Knowledge Base** analysis.

LLM-Based SQL Code Generation Process

SQL code generation follows a **structured multi-step approach**, where an **LLM** synthesizes the SQL query using both **retrieved examples** and **structured table descriptions**. The process is formalized as:

$$S_{code}^R = \text{LLM}(Q, \mathcal{T}_{code}, \mathcal{S})$$

where:

Q — The normalized **user query**.

\mathcal{T} — The set of **relevant table descriptions**. $\mathcal{T} \in \{\mathcal{T}_{code}, \mathcal{T}_R\}$

\mathcal{S} — The set of **retrieved SQL examples**. $\mathcal{S} \in \{\mathcal{S}_{code}^{**}, \mathcal{S}_{code}^*, \mathcal{S}_{code}\}$

There are two approaches to generate SQL code, depends on the input data

I. EXAMPLE-DRIVEN SQL GENERATION

- Primary Focus:
 - Leverages previous SQL query examples (\mathcal{S}_{code}^{**} or \mathcal{S}_{code}^*) that are relevant to the user's question.
 - Uses table information as supporting context (\mathcal{T}_{code}) but relies mainly on patterns from examples.
- Key Steps:
 1. Analyze relevant SQL examples to extract query patterns.
 2. Check documentation recommendations and apply direct insights.
 3. Ensure table dependencies are correctly handled.
 4. Minimize the number of tables included in the final query.
- Constraints:
 - Must not introduce extra operations unless explicitly required by the question.
 - Must follow the provided schema strictly.

II. DOCUMENTATION-DRIVEN SQL GENERATION

- Primary Focus:
 - Uses table descriptions from documentation as the primary source (\mathcal{T}_R).
 - Incorporates SQL examples (\mathcal{S}_{code}) only as secondary references, ensuring consistency with the provided dataset.
- Key Steps:
 1. Identify relevant tables from documentation, ensuring correct table selection.
 2. Verify table dependencies and construct valid joins based on relationships.
 3. Validate that table combinations satisfy the user's request.
 4. Analyze relevant SQL examples to refine query structure.
 5. Minimize unnecessary table usage to optimize performance.
- Constraints:
 - Must only use tables from the provided dataset, even if SQL examples contain additional tables.
 - Must strictly follow table relationships and dependency rules.
 - Must adhere to schema constraints at all times.

Conclusion

SQL code generation constructs accurate queries by leveraging retrieved examples and table documentation. The system dynamically chooses between example-driven and documentation-driven approaches based on relevance, ensuring alignment with the user's intent and database schema. By strictly following table relationships, dependencies, and constraints, this method ensures robust, efficient, and reliable SQL with minimal errors.

Conclusions

The Text2SQL system presented in this document establishes a comprehensive approach to converting natural language queries into correct SQL code using Retrieval-Augmented Generation (RAG). By integrating database knowledge, query examples, and domain-specific rules, this system handles the challenges of database complexity and linguistic nuance.

Key Architectural Components

The system combines three essential elements to generate accurate SQL:

1. A comprehensive **Knowledge Base** from database documentation, example queries, and business rules. This knowledge is stored as vector embeddings that capture relationships between concepts.
2. **Smart retrieval methods** adjust similarity thresholds as needed, consider multiple variations of the user’s question, and apply business rules to select the right tables.
3. A self-adapted SQL generation approach is based on the quality of retrieved examples, which can be done by closely following examples or building queries from documentation when necessary.

Real-World Applications and Benefits

This approach solves a fundamental problem — allowing non-technical users to access database insights through simple questions. By combining structured database knowledge with flexible language understanding, the system maintains accuracy while being accessible to everyone. The example-based approach means the system improves over time as more successful queries are added. Organizations can adapt the system to their specific domains by adding custom businesses, rules, and examples.

Methodological Focus and Limitations

This document presents a framework and methodology for a Text2SQL translation method based on Retrieval-Augmented Generation. Although we explain the model architecture and approach in detail, this paper does not provide any numerical comparisons with existing Text2SQL benchmarks or systems. The focus of this work is on architectural design, knowledge representation methods, and retrieval methodologies rather than on benchmarking the system performance.

We focused on methodology because of the initial requirements to develop a system in a specific organizational context with proprietary databases and custom business rules that do not directly correspond to public benchmarks. We believe that the approach described offers valuable insights for practitioners implementing similar systems, even without standardized performance metrics.

Future Work

Future developments should include empirical evaluation on publicly available benchmarks to quantify performance relative to existing approaches. Additional research could focus on optimizing the extraction process for greater efficiency, expanding the range of supported SQL constructs, and developing methods for automated extraction of domain-specific rules. We also see potential in adapting this framework to support multi-turn interactions, where users can refine their queries through conversation.

References

- [1] Naihao Deng, Yulong Chen, and Yue Zhang. Recent Advances in Text-to-SQL: A Survey of What We Have and What We Expect. *ACL Anthology*. 2022. <https://aclanthology.org/2022.coling-1.190/>
- [2] Anindyadeep Sannigrahi. State of Text-to-SQL 2024. *Premai Blog*. 2024. <https://blog.premai.io/state-of-text2sql-2024/>
- [3] AtScale. What is Text-to-SQL? Benefits, How it Works. *AtScale Glossary*. 2023. <https://www.atscale.com/glossary/text-to-sql/>
- [4] Jinqing Lian, Xinyi Liu, Yingxia Shao, Yang Dong, Ming Wang, Zhang Wei, Tianqi Wan, Ming Dong, and Hailin Yan. ChatBI: Towards Natural Language to Complex Business Intelligence SQL. *arXiv preprint*. 2024. <https://doi.org/10.48550/arXiv.2405.00527>
- [5] Hiba Fathima. Natural Language to SQL: Simplifying Data Access for Everyone. *Sequel Blog*. 2024. <https://sequel.sh/blog/natural-language-to-sql>
- [6] Bing Wang, Yan Gao, Zhoujun Li, and Jian-Guang Lou. Know What I Don't Know: Handling Ambiguous and Unanswerable Questions for Text-to-SQL. *ACL Anthology*. 2023. <https://aclanthology.org/2023.findings-acl.352.pdf>
- [7] Yujian Gan, Xinyun Chen, and Matthew Purver. Exploring Underexplored Limitations of Cross-Domain Text-to-SQL Generalization. *ACL Anthology*. 2021. <https://aclanthology.org/2021.emnlp-main.702.pdf>
- [8] Sanjeeb Panda and Burak Gözlüklü. Build a robust text-to-SQL solution generating complex queries, self-correcting, and querying diverse data sources. *AWS Machine Learning Blog*. 2024. <https://aws.amazon.com/blogs/machine-learning/build-a-robust-text-to-sql-solution-generating-complex-queries-self-correcting-and-querying-diverse-data-sources/>
- [9] Irina Saparina and Mirella Lapata. AMBROSIA: A Benchmark for Parsing Ambiguous Questions into Database Queries. *arXiv preprint*. 2024. <https://doi.org/10.48550/arXiv.2406.19073>
- [10] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. Next-Generation Database Interfaces: A Survey of LLM-based Text-to-SQL. *arXiv preprint*. 2024. <https://arxiv.org/html/2406.08426>
- [11] Longxu Dou, Yan Gao, Xuqi Liu, Mingyang Pan, Dingzirui Wang, Wanxiang Che, Dechen Zhan, Min-Yen Kan, and Jian-Guang Lou. Towards Knowledge-Intensive Text-to-SQL Semantic Parsing with Formulaic Knowledge. *ACL Anthology*. 2022. <https://doi.org/10.18653/v1/2022.emnlp-main.350>
- [12] Ali Buğra Kanburoğlu and Faik Boray Tek. Text-to-SQL: A methodical review of challenges and models. *Turkish Journal of Electrical Engineering and Computer Sciences*. 2024. <https://doi.org/10.55730/1300-0632.4077>
- [13] Tobi Beck. How to Simplify SQL with Text-to-SQL Technology. *Eckerson Group Blog*. 2024. <https://www.eckerson.com/articles/how-to-simplify-sql-with-text-to-sql-technology>