Titre: Title:	Leveraging Query Expansion and Augmentation for Text-to-SQL Evaluation
Auteur: Author:	Mohamed Riahi
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Riahi, M. (2025). Leveraging Query Expansion and Augmentation for Text-to-SQL Evaluation [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/69007/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Leveraging Query Expansion and Augmentation for Text-to-SQL Evaluation

MOHAMED RIAHI
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
génie informatique

Septembre 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Leveraging Query Expansion and Augmentation for Text-to-SQL Evaluation

présenté par **Mohamed RIAHI**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Mohammad HAMDAQA, président
Foutse KHOMH, membre et directeur de recherche
Amine MHEDHBI, membre et codirecteur de recherche
Zohreh SHARAFI, membre

DEDICATION

To the bright soul of my mother, **Nabiha**.
This work is for you. My whole being is for you.
I miss you endlessly, and I hope that somewhere in the skies, you are smiling with pride.
Your light, your love, your empathy, and your kindness shaped the person I have become.
They are the treasures I carry within me, the inheritance that guides every step I take.
Your beauty and vitality still give meaning to my days,
even though your hands can no longer reach mine.

To my beloved sisters **Hiba**, **Sahar**, and **Anissa**.
You turned this dream into reality through your patience, your care, and your unwavering love,
even when I could no longer believe in myself.

To my grandfather **Hamda**, who once looked at a commercial of a child using a laptop and said,
"My grandson will become a better computer scientist."

To my grandmother, my uncle, and his family,
those who stood by us through sorrow,
those who carry the scent of Mom, the sound of her voice,
those who remind me where I come from.

To the wonderful friends who have crossed my path.
From my small hometown of **Mateur**,
to my years of study and work in **Tunis**,
to the second family I found here in **Montreal**.
You are the brilliant souls who made me a citizen of the world.
If I began to name you all, these pages would overflow.
But know this: if your heart stirs or your eyes well up as you read,

this dedication is for you.

To those who will read these words one day.
The path of knowledge is filled with doubt,
but resilience will always light your way.
Perfection is not the goal, for learning never ends.
Celebrate your victories, believe in yourself,
and when sadness finds you, let it pass through — it will not stay forever.

This work is for you, all of you.
My existence and my achievements are for you.

RÉSUMÉ

Le Text-to-SQL est la tâche consistant à générer une requête SQL exécutable par une base de données à partir d'une requête en langage naturel. Son objectif est de démocratiser l'accès aux données en entreprise, en permettant aux utilisateurs non techniques d'interroger directement leurs données. Les récents progrès des grands modèles de langage (LLM) ont permis d'atteindre une précision remarquable sur les jeux de tests publics et d'envisager des déploiements en production dans des environnements contrôlés. Cependant, les systèmes actuels de Text-to-SQL restent opaques, et leur évaluation comparative sur des jeux de données privés demeure difficile.

Une exigence clé pour une évaluation robuste sur des jeux de données privés est la création d'un jeu d'évaluation de référence (*golden set*). La construction manuelle d'un tel jeu est coûteuse en main-d'œuvre, tandis que s'appuyer principalement sur des LLM introduit du non-déterminisme, des hallucinations, des problèmes de reproductibilité, ainsi que des coûts importants. Les jeux de tests publics de Text-to-SQL tels que Spider et BIRD offrent une alternative, mais ne reflètent pas la complexité des requêtes présentes dans les charges de travail d'entreprise. Les requêtes en entreprise présentent souvent des structures de jointure plus riches, des schémas contenant du jargon et des acronymes propres au domaine, ainsi que de nombreuses dépendances, ce qui met les systèmes de Text-to-SQL à l'épreuve. De plus, les classements sur les tableaux d'honneur publics peuvent ne pas refléter la performance relative des systèmes sur des jeux de données privés.

Pour relever ces défis, nous proposons deux techniques d'automatisation inédites : (i) *Join Query Expansion* (JQE) et (ii) *Textual Query Augmentation* (TQA). Ces deux approches visent à créer des *points de blocage* (*choke points*) ciblés, afin de mettre systématiquement à l'épreuve les différentes parties des systèmes de Text-to-SQL. JQE se concentre sur la composante SQL et accroît systématiquement la complexité des requêtes en élargissant l'ensemble des tables jointes dans une requête SQL donnée. Elle intègre des vérifications sémantiques pour garantir des jointures valides et non transitives, ainsi qu'un mécanisme d'élagage tenant compte de la diversité afin de maximiser la couverture structurelle. TQA se concentre sur la composante en langage naturel, en traitant spécifiquement le problème de l'*élimination des tables de liaison* : il détecte les tables de liaison dans la requête SQL (souvent, mais pas toujours, des tables relationnelles traversées par des jointures), détermine si la requête en langage naturel les mentionne, et applique l'une des trois stratégies d'augmentation pour modifier ces références.

Nos expériences avec JQE montrent une augmentation substantielle de la diversité structurelle, avec un degré moyen du graphe passant de 0,82 à 1,80 et une cyclicité des jointures passant de 0,27% à 4%, deux indicateurs de requêtes plus complexes. Nous démontrons également que cette complexité accrue entraîne, pour deux systèmes à l'état de l'art, une baisse moyenne d'environ 20% de la précision d'exécution. Avec TQA, bien que les trois stratégies d'augmentation se révèlent efficaces pour éliminer les tables de liaison, les systèmes Text-to-SQL à l'état de l'art restent relativement résilients, avec une baisse maximale observée de 5,6% de la précision d'exécution.

ABSTRACT

Text-to-SQL is the task of generating a database-executable SQL query from a natural language inquiry. Its goal is to democratize data access within the enterprise, enabling non-technical users to query their data directly. Recent advances in large language models (LLMs) have yielded impressive accuracy on public benchmarks and enabled controlled production deployments. However, current Text-to-SQL systems remain opaque, and their comparative evaluation on private datasets is difficult.

A key requirement for robust evaluation on private datasets is the creation of a golden evaluation set. Manual construction of such a set is labor-intensive, while relying primarily on LLMs introduces nondeterminism, hallucinations, and reproducibility challenges along with significant cost. Public Text-to-SQL benchmarks such as Spider and BIRD offer an alternative, but they do not capture the query complexity of enterprise workloads. Enterprise queries often exhibit richer join structures and schema designs filled with domain-specific jargon and acronyms that challenge Text-to-SQL systems. Moreover, system rankings on public leaderboards may not reflect their relative performance on private datasets.

To address these challenges, we propose two novel automatic evaluation techniques: (i) *Join Query Expansion* (JQE) and (ii) *Textual Query Augmentation* (TQA). Both are designed to create targeted *choke points* that systematically challenge Text-to-SQL system parts. JQE focuses on the SQL component. It increases query complexity by expanding the set of joined tables in an input SQL query. It incorporates semantic checks to ensure valid, non-transitive joins and a diversity-aware pruning mechanism to maximize structural coverage. TQA focuses on the natural language component, specifically addressing *linker table elimination*: it detects linker tables in the SQL query, i.e., often, but not always, relationship tables traversed through joins. TQA determines whether the NL query references the linker tables, and applies one of three augmentation strategies to modify these references.

Our experiments with JQE show a substantial increase in structural diversity, with the average graph degree rising from 0.82 to 1.80 and join cyclicity increasing from 0.27% to 4%, both indicators of more challenging queries. We further demonstrate that this added complexity causes two state-of-the-art systems to experience an average drop of approximately 20% in execution accuracy. With TQA, while all three augmentation strategies show high accuracy in eliminating linker table, state-of-the-art Text-to-SQL systems remain relatively resilient, with a maximum observed drop of 5.6% in execution accuracy.

TABLE OF CONTENTS

DEDICATION iv

RÉSUMÉ vi

ABSTRACT viii

TABLE OF CONTENTS ix

LIST OF TABLES xii

LIST OF FIGURES xiii

LIST OF SYMBOLS AND ACRONYMS xiv

LIST OF APPENDICES xv

CHAPTER 1 INTRODUCTION 1
- 1.1 Limitations of Current Benchmarks 1
- 1.2 Contributions 3
- 1.3 Thesis Outline 4

CHAPTER 2 BACKGROUND 5
- 2.1 Text-to-SQL Definition 5
- 2.2 Evolution of Text-to-SQL 6
- 2.3 Systems Overview 7
  - 2.3.1 Schema Filtering 7
  - 2.3.2 Candidate Generation 8
  - 2.3.3 Query Selection 10
  - 2.3.4 Query Correction 11
- 2.4 Text-to-SQL Challenges 12
- 2.5 Text-to-SQL Systems for Evaluation 12
- 2.6 Benchmarks: Examples and Limitations 13
- 2.7 Motivation 15

CHAPTER 3 JOIN QUERY EXPANSION 16
- 3.1 Join SQL Query Expansion Example 18

3.1.1 Preliminaries on Schema Graphs . . . 18
3.1.2 Expansion Example . . . 20
3.1.3 Join Query Graph . . . 21
3.2 JQE Algorithm . . . 22
3.2.1 Stages I and II — Query Graph Mapping and Candidate Selection . . . 22
3.2.2 Stages III — Join Condition Combination (JCC) Generation . . . 22
3.2.3 Redundancy Semantic Check within Stage III . . . 24
3.2.4 Stage IV — Query Pruning Using Join Query Graphs . . . 24
3.2.5 Stage V — Expanded SQL Generation . . . 25
3.2.6 Stage VI — NL Generation . . . 25
3.2.7 User Preferences . . . 25
3.3 Evaluation . . . 26
3.3.1 Setup . . . 26
3.3.2 Experiments and Analysis . . . 26
3.3.3 Scalability Discussion . . . 35

CHAPTER 4 TEXTUAL QUERY AUGMENTATION 36
4.1 Textual Query Augmentation Example . . . 38
4.1.1 Schema Graph Example . . . 38
4.1.2 Augmentation Example . . . 39
4.2 TQA Algorithm . . . 42
4.3 Evaluation . . . 45
4.3.1 Setup . . . 45
4.3.2 Experiments and Analysis . . . 46

CHAPTER 5 CONCLUSION 55
5.1 Limitations . . . 55
5.2 Future Directions . . . 56
5.3 Closing Remarks . . . 56

REFERENCES . . . 57

APPENDICES . . . 63
A.1 New Questions for Extended SQL . . . 63
A.2 New Questions for New SQL . . . 64
A.3 New Evidence for New SQL . . . 68
B.1 Synonym Replacement . . . 71

B.2 Backtranslation 74
B.3 Contextual Augmentation 77

LIST OF TABLES

Table 1.1 Structural summary of database schemas as graphs from Text-to-SQL benchmarks. 2
Table 1.2 Join query graphs statistics for BIRD. 2
Table 3.1 Number of queries in the original and augmented BIRD sets. 27
Table 3.2 Augmented join query graphs grouped by $\Delta\bar{d}$ 29
Table 3.3 Original query graph structures 29
Table 3.4 Distribution of cyclic and acyclic join query graphs across different sets 30
Table 3.5 Cycles in BIRD Dev Schema Graphs 31
Table 3.6 Text-to-SQL execution accuracy on different sets 32
Table 3.7 Distribution of ΔEX 33
Table 3.8 Query distribution per join group 34
Table 4.1 Query distribution and node count per split 45
Table 4.2 Dataset statistics across train, dev, and test sets. 46
Table 4.3 Hiding scores by technique and dataset split. 48
Table 4.4 Automated and human hiding scores by technique and dataset. . . . 49
Table 4.5 Execution accuracy (%) across original and augmented queries. . . . 50
Table 4.6 Comparison of (ΔEX) across systems and augmentation techniques. . 51
Table 4.7 Statistics on linker table attribute mentions in questions (Qs) and evidences (Evs) across dataset splits. 52
Table 4.8 Distribution of linker table attribute mentions in questions (Qs) and evidences (Evs) across DEV and TEST splits for each augmentation technique. 53
Table 4.9 Presence of linker tables (LT) in retrieved tables across different augmentation techniques. 54

LIST OF FIGURES

Figure 2.1 Text-to-SQL: mapping natural language to SQL 6
Figure 3.1 The multi-stage pipeline for join query expansion. 17
Figure 3.2 Schema graph example. 19
Figure 3.3 Join query graph for $Q_{\texttt{bond:JC1}}$. 22
Figure 3.4 Join Redundancy: Example 24
Figure 3.5 Average degrees in the original and augmented sets. 28
Figure 3.6 EX of different text-to-SQL systems on our sampled expanded set . . 34
Figure 4.1 The three main techniques for textual query augmentation 37
Figure 4.2 Second example of a schema graph 39
Figure 4.3 Second example of a join query graph 40
Figure 4.4 Clustered databases per node distribution 45

LIST OF SYMBOLS AND ACRONYMS

NL	Natural Language
LLM	Large Language Model
SQL	Structured Query Language
Text-to-SQL	Text-to-Structured Query Language
JQE	Join Query Expansion
TQA	Textual Query Augmentation
RNNs	Recurrent Neural Networks
T5	Text-to-Text Transfer Transformer
BERT	Bidirectional Encoder Representations from Transformers
LSH	Locality Sensitive Hashing
IR	Information Retriever
SS	Schema Selector
CG	Candidate Generator
UT	Unit Tester
EX	Execution Accuracy
WikiSQL	Wikipedia Structured Query Language dataset
KaggleDBQA	Kaggle Database Question Answering
CRUD	Create, Read, Update, Delete
Seq2SQL	Sequence-to-SQL
DIN-SQL	Decomposed In-Context Learning of Text-to-SQL with Self-Correction
MAC-SQL	Multi-Agent Collaborative Framework for Text-to-SQL
CHESS	Contextual Harnessing for Efficient SQL Synthesis
BM25	Best Matching 25
PK, FK	Primary Key, Foreign Key
CT	Candidate Table
JC	Join Condition
JCC	Join Condition Combination
LT	Linker Table
EV	Evidence
BC	Betweenness Centrality
SynRep, SR	Synonym Replacement
BT, BackTrans	Backtranslation
CA, CtxAug	Contextual Augmentation

LIST OF APPENDICES

Appendix A Prompts for Join Query Expansion 63
Appendix B Prompts for Query Augmentation 71

CHAPTER 1 INTRODUCTION

Natural language interfaces to databases aspire to democratize data access by allowing users to pose questions in natural language and obtain answers directly from structured data. The Text-to-SQL task consists of translating natural language queries into executable SQL and lies at the heart of this vision. Recent progress in large language models (LLMs) has led to rapid improvements in Text-to-SQL benchmarks such as Spider [1], BIRD [2] and Beaver [3], spurring research interest and early production deployments.

Despite these advances, current evaluation practices fall short of capturing the true challenges of enterprise workloads. Enterprise queries often span multiple tables, exhibit rich join structures, and reference schema elements named with domain-specific jargon or abbreviations. By contrast, public benchmarks contain short, structurally simple queries and intuitive schema names, leading to a misleading picture of system capabilities and relative comparison. Evaluating Text-to-SQL systems on private datasets can help bridge this gap. Constructing high-quality evaluation sets however requires extensive manual annotation, domain and data analysis expertise, and possibly heavy reliance on LLMs for automation. All these approaches are costly, non-deterministic, and can be difficult to reproduce.

1.1 Limitations of Current Benchmarks

To validate the motivation of this thesis, we conducted a structural analysis of the three widely used Text-to-SQL benchmarks: Spider, BIRD, and Beaver. Each database schema was represented as a graph, where tables correspond to nodes and edges indicate possible join relationships. The results, summarized in Table 1.1, highlight a clear mismatch between academic evaluation and enterprise in: (i) database structure; and (ii) query diversity.

Database structure. Our analysis of schema graphs reveals several limitations in the structural properties of current benchmarks.

First, benchmark databases often contain multiple connected components, and the overall number of tables is small. For instance, only 82% of Spider schema graphs and 67.5% of BIRD schema graphs form a single connected component. Beaver, the benchmark most similar to enterprise settings, contains no fully connected schema graphs at all. This absence is not due to sparsity but rather to the fact that Beaver databases include a much larger number of tables, which makes complete connectivity less likely.

Second, connectivity metrics further emphasize these limitations. The average table degree,

Benchmark	# dbs	% Conn.	% Cyclic	$\bar{d}$	Avg Diam.
Spider	166	81.93	45.78	1.94	2.30
Bird	80	67.50	60.00	2.20	1.99
Beaver	6	0.00	100.00	3.92	4.33

Table 1.1 Structural summary of database schemas as graphs from Text-to-SQL benchmarks.

Num Nodes	Count	Percentage (%)	Avg Degree	Cyclic (%)
1	2172	22.40	0.00	0.00
2	5636	58.13	1.00	0.00
3	1532	15.80	1.33	0.07
4	305	3.15	1.51	1.31
5	45	0.46	1.60	0.00
6	5	0.05	1.67	0.00

Table 1.2 Join query graphs statistics for BIRD.

which reflects how many joins a table can participate in, is only 1.94 for Spider and 2.20 for BIRD, compared to 3.92 for Beaver. This shows that tables in Spider and BIRD schemas tend to be weakly connected, limiting opportunities to construct queries with rich join structures.

Third, the diameters of connected components in schema graphs, which indicate the maximum number of tables that can be connected along a single join path, further highlight these differences. Spider and BIRD have average diameters of around 2, whereas Beaver reaches an average diameter of 4. Larger diameters translate to longer join paths, which again are characteristic of enterprise workloads but underrepresented in public benchmarks.

Finally, cyclicity provides another measure of structural richness. The presence of cycles in schema graphs corresponds to more complex join patterns. Beaver contains cycles in the schema graphs of each database, making it far more likely to encounter cyclic join queries than in BIRD (60%) or Spider (45.78%).

Query diversity. We evaluate query diversity by examining SQL join structures while abstracting away query semantics. Table 1.2 presents our analysis for the BIRD benchmark (9,695 queries). The majority of SQL queries involve only two tables (58.13%), followed by single-table queries (22.4%) and three-table queries (15.80%). These join patterns are relatively simple and lack the complexity typically observed in enterprise workloads. In data warehousing scenarios, queries routinely span many more tables and incorporate more elaborate join structures. This contrast underscores a key limitation: systems that achieve high performance on public benchmarks may not generalize to production environments where multi-join queries are the norm.

Taken together, the analysis of database structure and query diversity shows that existing benchmarks fail to capture the join-heavy characteristics of enterprise workloads. Consequently, they offer limited opportunities to evaluate Text-to-SQL systems under the conditions most likely to reveal weaknesses in handling complex join queries.

Prior research has attempted to close this gap either by constructing entirely new datasets from scratch [4] (e.g., Beaver) or by modifying existing benchmarks through schema slicing [5]. While such efforts can increase structural variety, they often introduce unrealistic schemas or yield insufficient SQL diversity. *What remains missing is a systematic, controlled, and reproducible approach to stress-test existing Text-to-SQL systems, without incurring the high costs of manual evaluation set construction.*

1.2 Contributions

This thesis addresses these shortcomings by introducing two novel, automatic evaluation techniques that create targeted choke points for Text-to-SQL systems:

- Join Query Expansion (JQE), which stresses the SQL generation side by increasing query complexity through semantically valid join expansions.
- Textual Query Augmentation (TQA), which stresses the natural language (NL) understanding side, specifically schema linking, i.e., retrieving relevant schema elements, by modifying references to linker tables in NL queries.

Both query generation methods are designed to be cost-aware, largely deterministic, and reproducible. Together, JQE and TQA underscore the value of a dual perspective on Text-to-SQL evaluation: JQE targets the SQL generation process, while TQA challenges the natural language side through schema linking. Our experiments show that JQE significantly increases query diversity and exposes weaknesses in state-of-the-art systems, with execution accuracy dropping by roughly 20%. TQA, while resulting in smaller performance degradations, demonstrates system resilience under controlled perturbations that specifically stress schema retrieval.

In summary, this thesis contributes:

1. The design and implementation of Join Query Expansion (JQE) to stress-test systems on SQL-side complexity.
2. The design and implementation of Textual Query Augmentation (TQA) to stress-test systems on NL-side schema linking.

3. An experimental study showing how these techniques expand evaluation diversity, reduce reliance on costly manual sets, and provide an analysis of state-of-the-art systems on expanded and augmented queries.

1.3 Thesis Outline

The thesis is organized as follows. Chapter 2 overviews necessary background and preliminaries. Chapter 3 introduces JQE, presenting its algorithmic design and evaluation. Chapter 4 presents TQA, its algorithms and augmentation strategies. Finally, Chapter 5 concludes with a synthesis of findings, broader implications, and directions for future work.

CHAPTER 2 BACKGROUND

Across this chapter, we introduce key concepts related to the Text-to-SQL task. We start with its definition in Section 2.1 and trace its evolution over time in Section 2.2. To perform this task, various systems have been developed, which we outline along with their shared components in Section 2.3. Section 2.4 discusses the main challenges in this area, while Section 2.5 is dedicated to the three systems used in Chapters 3 and 4. Finally, we present the current benchmarks for evaluating these systems in Section 2.6 and describe the motivation for our work in Section 2.7.

2.1 Text-to-SQL Definition

Text-to-SQL is the task of mapping a natural language question to an executable SQL query. As illustrated in Figure 2.1, the objective is to convert a user's request, expressed in natural language, into a structured query that can be executed on a relational database [6, 7].

For example, consider the natural language input shown in Figure 2.1:

> *Please specify all of the schools and their related mailing zip codes that are under Avetik Atoian's administration.*

By referring to the corresponding database schema, we can infer that the intended SQL query should return the school names and their mailing zip codes where the administrator's name is Avetik Atoian. A correct translation would be:

```
SELECT School, MailZip
FROM schools
WHERE AdmFName1 = "Avetik" AND AdmLName1 = "Atoian";
```

Text-to-SQL is a prominent effort toward making data access more inclusive across professional environments. It enables individuals without strong technical backgrounds to query databases using natural language, thus removing the need to write formal SQL. This is especially useful during onboarding periods, when analysts are still becoming familiar with internal databases and the specific language or concepts used in the organization's domain [8].

In real-world settings, data analysts often encounter scenarios that involve constructing complex queries, which may involve multiple tables, conditional logic, and aggregations. Instead of writing these by hand, analysts may prefer to pose spontaneous natural language ques-

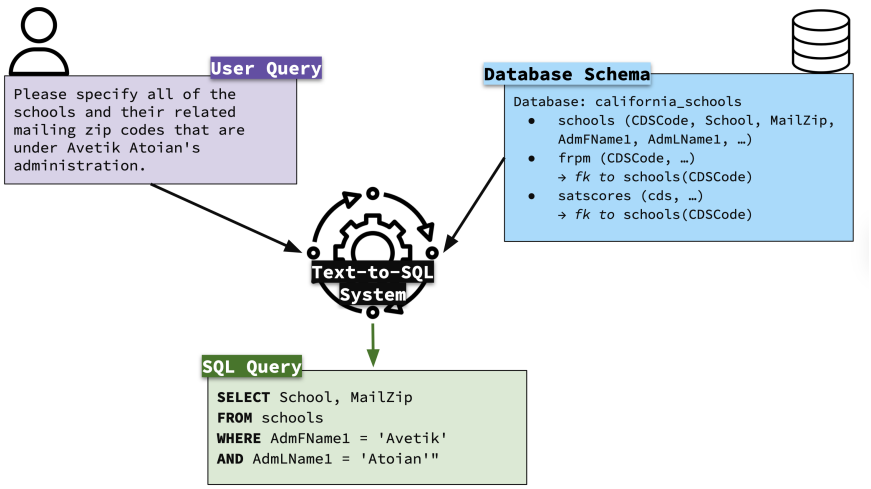

Figure 2.1 Text-to-SQL: mapping natural language to SQL

tions, expecting quick, interactive responses. A well-designed Text-to-SQL system should be capable of handling these dynamic and time-sensitive interactions efficiently [8].

In addition, Text-to-SQL facilitates on-demand exploration of data, reducing dependence on technical teams or the need for pre-built dashboards. This enables faster transitions from idea to insight, which is particularly valuable in business intelligence workflows, where timely validation of hypotheses can drive better decisions.

2.2 Evolution of Text-to-SQL

Text-to-SQL has evolved from fixed rule-based approaches to LLM-based methods. The earliest systems relied on predefined grammars and parsers to translate natural language queries into SQL. NaLIR [9] and ATHENA [10] are examples of such systems, but their reliance on fixed symbols prevented them from generalizing across domains [11]. This limitation motivated the integration of neural networks into Text-to-SQL systems, with learning driven by datasets such as KaggleDBQA [12] and WikiSQL [13], which provided benchmarks for training and evaluation. Recurrent neural networks (RNNs), particularly sequence-to-sequence architectures, proved effective in mapping natural language to SQL. A notable example is Seq2SQL, which employed an encoder-decoder architecture with reinforcement learning to perform the Text-to-SQL task [11].

Later, approaches using pre-trained language models such as T5 and BERT further improved the accuracy and generalization of Text-to-SQL systems. Their transformer architecture enabled the models to capture complex relationships between text and SQL. Cross-domain

datasets like Spider also contributed to these advances. RESDSQL [14] is one example of such techniques, where the most relevant part of the schema is first selected before constructing the SQL query [11]. However, these models face newer challenges, such as accurately retrieving the correct tables from database schemas, generalizing to unseen domains, and handling the diversity of user vocabulary in input queries.

The most recent Text-to-SQL systems leverage large language models (LLMs) such as GPT-4 and LLaMA, which achieve higher execution accuracy in SQL prediction [11]. The task is now approached through various prompting techniques, correction cycles, and learning strategies, which have reshaped how systems handle it [15–18].

2.3 Systems Overview

The Text-to-SQL task is typically handled by dedicated systems that follow multi-stage pipelines [15–25]. These systems generally include the following stages: (i) schema filtering, (ii) SQL query candidate generation, (iii) query selection, and (iv) query correction. These stages differ in how they are implemented and in their model choices (fixed or adaptive) [26].

2.3.1 Schema Filtering

In this step, Text-to-SQL systems try to capture the key elements of a natural language question and map them to the relevant parts of the database schema that can answer it. While most systems rely on step-by-step interactions with large language models (LLMs) [15–19, 27], the strategies for schema and value retrieval vary. Some systems focus on extracting keywords and entities directly from the question. For example, DIN-SQL extracts relevant keywords without relying on intermediate steps [19], while MAC-SQL reduces the schema by selecting only necessary components based on the question context [20]. Similarly, CHESS, along with XIYAN-SQL and CHASE-SQL, first extract keywords and entities but go further by retrieving values from the actual database using Locality Sensitive Hashing (LSH) [15–17]. LSH maps semantically similar data points to the same buckets with high probability using multiple hash functions and high-dimensional vector embeddings, which are stored and searched efficiently using vector databases specialized for similarity search [28]. After value retrieval, CHESS reduces the schema by selecting relevant columns based on matched values and entities, and can leverage external database catalogs with metadata like column descriptions and value annotations [15]. Ultimately, these systems share the common goal of leveraging the schema more effectively, either by narrowing it to the most relevant parts or by exploiting cues from the natural language question.

Other systems incorporate large language models more explicitly in schema linking and filtering. OpenSearch-SQL [25], after preprocessing, performs schema linking through an LLM call combined with vector similarity retrieval for value matching, then filters the schema to retain only what is needed. It also extracts keywords from the question, aligns them with expected SQL content, adds primary key constraints, and handles ambiguous columns by duplicating them when necessary [25]. N-rep generates multiple schema representations to predict the required portion, suggesting three filtering results for each: no filtering, table-only filtering, or full filtering of both tables and columns [23]. RSL-SQL employs bidirectional schema linking: it first performs a forward linking step, then generates a preliminary SQL query to extract referenced tables and columns for backward refinement [29]. TA-SQL shares the technique of generating a dummy SQL query to extract schema entities, which helps reduce hallucinations, but does not perform the backward linking step [21]. CSC-SQL does not specify any schema filtering or linking process, implying the use of the full schema [22]. Meanwhile, MCS-SQL introduces shuffling of table and column orders with union operations to avoid missing required schema elements [24]. Together, these systems enrich the schema linking phase with additional operations, diversifying how schema information is used to better align with natural language questions.

Regarding schema representation, most systems use the standard Data Definition Language schema description, but some introduce custom formats. MAC-SQL defines its own schema structure [20], and XIYAN-SQL extends this by explicitly including data types and primary key annotations [16]. E-SQL challenges the need for schema filtering, arguing it can harm performance, especially for advanced LLM-based systems like those using GPT-4o. Instead, E-SQL proposes Query Enrichment, augmenting questions with relevant database items and detailed instructions for query construction via LLM prompting. Additionally, E-SQL uses BM25, a classic information retrieval algorithm, to rank and retrieve pertinent database values and descriptions for prompt enrichment [18, 30]. Systems differ slightly in how they represent their schemas. Yet, despite these differences and the various ways this step is performed, schema filtering remains a central component of Text-to-SQL.

2.3.2 Candidate Generation

Text-to-SQL systems often generate multiple candidate SQL queries to increase the chances of producing a correct answer for the input question [15, 22–25]. While many share a broadly similar framework, they differ in prompting strategies, decomposition methods, underlying LLMs, and the number of candidates produced.

CHESS uses a straightforward approach by generating multiple candidates directly through

prompt-based sampling [15]. Similarly, OpenSearch-SQL progressively generates multiple candidates using dynamic few-shot prompting and applies an alignment process to improve output quality [25]. N-rep generates multiple candidates from three few-shot examples [23], and MCS-SQL also employs multiple prompts with high sampling temperature to produce candidate lists [24]. CSC-SQL generates multiple candidates as well, though it does not specify detailed prompting strategies [22]. Overall, these examples underscore the value of diversity in candidate generation.

DIN-SQL first classifies questions into easy, non-nested complex, or nested complex categories, adapting its strategy accordingly. It uses simple few-shot prompting for easy questions and employs intermediate representations with chain-of-thought reasoning for complex ones to guide structured query generation [19, 31]. CHASE-SQL builds on this idea with its divide-and-conquer chain-of-thought strategy, decomposing complex questions into simpler sub-questions, then aggregating their results into a final query [17]. MAC-SQL also applies a decomposition approach for complex queries but relies mainly on few-shot prompting for simpler cases [20]. CHASE-SQL also leverages a query plan chain-of-thought technique, which interprets the output of the SQL `EXPLAIN` statement (describing execution plans such as join order and index usage) into human-readable reasoning steps. This intermediate reasoning guides the LLM in generating more logically accurate SQL queries, complemented by few-shot examples [17, 31].

E-SQL follows a distinct approach by generating a single candidate using chain-of-thought reasoning, then extracting likely predicate structures from the query. It applies fuzzy matching techniques (e.g., the LIKE operator) to retrieve relevant database values for these predicates, enhancing precision [18, 31]. XIYAN-SQL combines a fine-tuned model trained in two stages: initial SQL syntax training followed by multi-task learning including SQL-to-text, with an in-context learning model that uses few-shot prompting guided by semantically similar examples to generate candidates [16]. These examples show how different prompting strategies, whether chain-of-thought or few-shot learning, can guide candidate generation from generic to more question-specific queries.

RSL-SQL first generates key SQL components to guide candidate generation, then produces refined candidates, integrating schema linking and generation phases [29]. TA-SQL shares similarities, generating SQL queries through reasoning that simulates data analysis workflows, and also produces candidates during schema linking [21, 29].

Collectively, these techniques illustrate the diversity of candidate generation approaches, ranging from simple prompt-based sampling to reasoning-driven and schema-aware methods.

2.3.3 Query Selection

In this step, the system selects the most appropriate SQL query to answer the question by evaluating all candidate queries. Some systems generate only a single query and thus do not perform any selection or ranking. For instance, DIN-SQL produces a single query per question without candidate selection [19], and similarly, E-SQL generates one query without any ranking or selection process [18].

Other systems rely on self-consistency, which selects the most frequent or consistent query among candidates. MAC-SQL decomposes complex questions into sub-questions answered by sub-queries, then applies self-consistency to pick the most consistent final query [20, 32]. CHESS uses a unit testing mechanism to evaluate candidates by constructing multiple unit tests; it selects the query that passes the most tests. If the unit tester is disabled, CHESS falls back on self-consistency for selection [15, 32]. OpenSearch-SQL and N-rep also use self-consistency with voting to select the best query; OpenSearch-SQL further filters out unfixable or empty-result queries and prioritizes results and execution speed [25, 32], while N-rep incorporates pairwise comparison with LLMs when confidence in voting is low [17, 23, 32]. RSL-SQL compares two candidates, one from the schema linking phase and one from candidate generation, using self-consistency based on execution [29]. CSC-SQL similarly applies self-consistency for query selection [22, 32]. Together, these examples highlight the diverse applications of self-consistency across techniques, emphasizing its importance in text-to-SQL query selection.

Some systems employ machine learning classifiers or models to improve selection beyond self-consistency. CHASE-SQL uses a pairwise comparison model that scores candidates according to their ability to answer the question, selecting the candidate with the highest cumulative score [17]. XIYAN-SQL fine-tunes a classifier specifically for query selection, grouping candidates by execution consistency and then choosing the best one by analyzing contextual and subtle differences rather than relying solely on consistency [16]. Such strategies demonstrate the potential of using machine learning models for query selection, providing systems with an automated and more informed approach.

MCS-SQL takes a hybrid approach by combining execution feedback, scoring, and LLM reasoning. It first executes all candidate queries and groups those with identical results, keeping the fastest query in each group. The system then assigns confidence scores, filters out low-confidence candidates, and applies an LLM-based decision step followed by majority voting [24].

2.3.4 Query Correction

Although the timing of the correction process may vary, most Text-to-SQL systems apply query correction in a similar manner.

For E-SQL, DIN-SQL, and MAC-SQL, query correction occurs at the end of the generation process, once a single final query has been produced [18–20]. E-SQL and MAC-SQL follow a similar procedure: the initial query is executed, and if it results in a syntax error or an empty result set, the system prompts the LLM to either correct the query or, in the case of E-SQL, generate a new one [18, 20]. DIN-SQL, however, performs correction regardless of execution outcome. It employs two strategies: *Generic correction*, which assumes errors are present, and *Gentle correction*, which suggests improvements without assuming the query is faulty [19].

Overall, these systems differ in how they perform query correction, even though it is typically applied at the end of query generation. There has also been prior work improving query correction and generation for future generations based on user feedback [33].

In contrast, CHESS, XIYAN-SQL, and CHASE-SQL perform correction on candidate queries before the final selection step. Their approach is generally similar: each candidate is executed, the system identifies potential issues, and the LLM is prompted to correct the query using information such as execution logs, error descriptions, database schema, and the set of candidate queries [15–17].

Other systems adopt iterative or multi-turn correction methods. OpenSearch-SQL applies an alignment process that corrects SQL output by properly matching columns, values, and SQL functions, fixing syntactic errors and handling empty results with few-shot correction examples [25]. RSL-SQL performs correction in multiple turns, detecting errors first, then iteratively refining the query using an LLM and error logs; candidate generation repeats until a valid SQL is produced or a maximum number of trials is reached [29]. CSC-SQL merges and revises inconsistent candidates to fix errors [22]. TA-SQL uses reasoning to handle complex syntax and semantics for correction [21].

These systems underscore the benefits of incorporating iterative correction into the SQL generation pipeline.

Finally, query correction is the last step to provide users with an accurate SQL query, and it is necessary regardless of whether it is applied at the end of generation, before selection, or through iterative refinement.

2.4 Text-to-SQL Challenges

Text-to-SQL techniques and systems face many challenges in converting questions to SQL queries. Part of these challenges arises from the fact that some existing techniques do not generalize well [4]. Although recent benchmarks are cross-domain, the issue becomes more pronounced with out-of-distribution databases, where many techniques struggle when exposed to such data [4].

Another challenge relates to how schema linking is performed in current systems. Natural language queries may refer to certain entities in multiple ways, such as the example of "sang by" referring equally to the tables `singer` and `artist` [6]. In addition, conditions involving database values can be interpreted differently if the model does not have access to all database values [6]. These issues occur despite the fact that most schemas in current benchmarks are human-readable and use naming conventions that can be easily captured by schema linkers using prompts, which is unlikely to be the case in industrial environments.

2.5 Text-to-SQL Systems for Evaluation

In the experimental evaluation in chapters 3 and 4, we use three text-to-SQL systems:

1. **CHESS (Contextual Harnessing for Efficient SQL Synthesis)** is a system developed by Talaei et al. [15] and it is our first used system. Its process begins with extracting keywords, values, and context from the natural language (NL) query, database files, or external metadata using an information retriever (IR). Then, a schema selector (SS) identifies the relevant parts of the database schema based on the extracted information. Next, a candidate generator (CG) produces a list of SQL queries and refines them to correct any syntactic errors. Finally, a unit tester (UT) generates unit tests for each candidate query to evaluate them and selects the one that passes the most tests. On the BIRD benchmark, two variants of CHESS are available: $\textbf{CHESS}_{\textbf{IR+CG+UT}}$, which uses only IR, CG, and UT (without the schema selector), and $\textbf{CHESS}_{\textbf{IR+SS+CG}}$, which excludes the unit tester and selects queries based on consistency. The former is ranked 19^{th}, tested on May 21, 2024, with execution accuracy (EX) scores of 68.31 on the dev set and 71.16 on the test set. The latter is ranked 34^{th}, with EX scores of 65.00 on the dev set and 66.29 on the test set. (The leaderboard includes 81 ranked techniques, with human performance, specifically, data engineers and DB students, at the top achieving EX = 92.96 on the test set.) In our experiments, we use the higher-ranked $\textbf{CHESS}_{\textbf{IR+CG+UT}}$ to better highlight the changes of performance.

2. **MAC-SQL (Multi-Agent Collaborative Framework for Text-to-SQL)**, developed by researchers from Beihang University and Tencent [20]. It follows a three-step process: a selector identifies schema parts relevant to keywords from the NL query; a decomposer breaks down complex queries into subproblems, generates subqueries, then combines them into one; and a refiner corrects syntax errors and avoids generating empty-result queries. MAC-SQL is ranked 56^{th} on the BIRD leaderboard, tested on November 21, 2023, with EX = 57.56 (dev) and 59.59 (test).

3. **DIN-SQL (Decomposed Stages with Self-Correction)**, developed by researchers from the University of Alberta [19]. It begins with a schema linking module that detects references to schema elements and filtering values in the NL query and produces a detailed description of schema and predicates. The system then classifies each query as *easy* (single-table, non-nested), *non-nested* (multi-table, no subqueries), or *nested* (includes subqueries or set operations). Depending on the classification, the SQL generation module adapts its prompting strategy (e.g., using few-shot learning or intermediate representations like NatSQL). Finally, a self-correction module performs a second pass of correction using LLM prompting. DIN-SQL was tested on August 15, 2023, with GPT-4, ranked 62^{nd} on the BIRD leaderboard with EX = 50.72 (dev) and 55.90 (test).

2.6 Benchmarks: Examples and Limitations

Text-to-SQL systems are typically evaluated using benchmarks. These benchmarks fall into two main categories: single-domain (e.g., Academic, Advising, IMDB, Yelp) [9, 34, 35] and cross-domain (e.g., BIRD, Spider, WikiSQL, KaggleDBQA, Beaver) [1–3, 12, 13]. Cross-domain benchmarks are more commonly used in recent work because they support broader generalization. They are built from varied sources, including Wikipedia tables, professional datasets, university course materials, SQL tutorials, and Kaggle challenges [36].

These benchmarks are typically evaluated using execution accuracy. This metric measures how often a model's generated query returns the same results as the gold SQL. A query is either correct (1) or incorrect (0), and the average score is reported. Leaderboards for Spider and BIRD rely heavily on this metric [1, 2].

Benchmarks also help provide structural and domain context for prompting large language models. The variety of domains supports few-shot prompting and chain-of-thought strategies, which have been shown to improve performance in recent LLM-based systems [31, 37].

Spider is one of the most widely used cross-domain benchmarks. It consists of 200 databases

from 138 domains such as TV shows, transcripts, and flights, covering 10,181 natural language questions and 5,693 SQL queries [1]. Spider 2.0 expands on this by introducing 632 workflow tasks drawn from enterprise-scale databases. These databases can include hundreds of columns and are hosted on cloud and on-premise platforms like BigQuery and Snowflake [38].

BIRD is another major cross-domain benchmark. It includes 95 large-scale databases across 37 domains, with sizes reaching up to 33.4 GB. The benchmark contains 12,751 question-query pairs [2]. Compared to Spider, BIRD databases have more tables on average (6.82 vs. 4.05). A more recent version, BIRD-SQL Pro v0.5 (LiveSQLBench), introduces an updated evaluation setting meant to reflect real-world Text-to-SQL workloads. It focuses on business intelligence and CRUD operations, with 50 new databases added in each release. The latest version, LiveSQLBench-Base-Lite, includes 18 databases and 270 evaluation tasks [39].

WikiSQL and KaggleDBQA offer different evaluation settings. WikiSQL has 80,654 question-SQL pairs across 24,241 tables from Wikipedia. KaggleDBQA is smaller, with only 272 queries over 8 databases, and is typically used in small-scale evaluations [12, 13].

Beaver is designed for industrial settings. It includes 209 question-query pairs across 6 databases, one of which is sourced from a real data warehouse. These databases are significantly larger, averaging 77.5 tables per database. The queries are also more complex than those in Spider or BIRD, featuring on average 4 joins, 2.15 aggregation functions, and 2.0 levels of nesting per query [3].

However, several challenges remain with the current benchmarks. According to Qin et al., a common issue lies in the limited quantity, quality, and diversity of data [4]. For example, Spider contains around ten thousand examples [1], which are considered insufficient for training a high-quality text-to-SQL parser or system [4]. WikiSQL [13], on the other hand, includes a large number of queries but lacks diversity in complex operations such as `GROUP BY`, nested queries, and `HAVING` clauses [4]. Furthermore, WikiSQL datasets share the same domains between them, and most databases contain only a single table, which prevents the use of joins [4].

Many benchmarks also have structural limitations. Renggli et al. [40] point out issues like incorrect or incomplete SQL labels, missing schema information (e.g., column domains, constraints), and the absence of alternative correct queries. These problems can penalize correct outputs, create inconsistencies in evaluation, and make it harder to scale benchmarks without significant manual intervention [40].

Another limitation is the lack of complex SQL structures. As noted by Mitsopoulou et al. [36], most benchmarks emphasize queries over small schemas, often avoiding deep joins, nested

queries, or aggregation patterns commonly found in production systems. Models trained on such datasets may struggle when faced with more realistic query complexity [36].

The problem of "ties" in SQL execution, described by Pourrezza and Rafiei (2023) [41], also complicates evaluation. For example, queries using LIMIT 1 or selecting based on MAX() can return multiple valid outputs. Benchmarks often label only one output as correct, which can mislead training and penalize correct predictions [41].

Natural language ambiguity further increases the difficulty. Lexical ambiguity, inference ambiguity, and user errors (e.g., grammar, typos) all affect how well a model maps questions to SQL [6]. Floratou et al. (2024) argue that many benchmarks focus on SQL correctness but ignore whether the generated SQL reflects what the user actually intended [42]. This concern overlaps with findings by Kumar et al., who report frequent misalignments between user intent and SQL semantics [43].

Benchmarks also lack standardization in evaluating large language models. Zhang et al. (2024) warn that many LLM-based systems overfit to benchmark datasets due to inconsistent evaluation setups [13]. Chen et al. observe that public benchmarks often miss the scale and complexity of enterprise databases; those with more tables, attributes, domain-specific vocabulary, and deeper join structures. These differences make schema linking and query generation more difficult in practice than in academic test sets [3]. Therefore, in this thesis, we propose two approaches to enrich existing evaluation sets: expanding existing SQL queries and augmenting the corresponding text queries.

2.7 Motivation

Prior research has shown the need for more challenging datasets in text-to-SQL evaluation. Eckermann et al. observed that many existing queries contain relatively few joins, and that joinability is a key factor in query complexity [5]. Their approach increased joins by splitting columns from existing SQL queries into different tables and adding columns to these tables using LLMs. This process revealed notable drops in system performance, underscoring the impact of join complexity.

Challenging the schema linking component by manipulating natural language queries can also expose weaknesses in system performance [4]. Motivated by these findings, our work leverages both joinability and data augmentation to generate new data, aiming to enhance diversity in join patterns and improve the robustness of text-to-SQL evaluation.

CHAPTER 3 JOIN QUERY EXPANSION

Benchmarks such as Spider and BIRD fall short in capturing the full complexity of enterprise Text-to-SQL workloads. These benchmarks tend to include structurally simple queries, often involving only a few tables and minimal join operations. In contrast, queries in enterprise environments typically span a larger number of tables and include more join operations. More importantly, since Spider and BIRD evaluate Text-to-SQL systems on public datasets, system comparative analysis might differ if tailored to a private dataset and workload.

To address this evaluation challenge, we propose *join query expansion*, a technique to systematically increase the complexity of Text-to-SQL queries by expanding the number of joined tables. Our technique takes as input a query as a natural language (NL) and SQL pair and produces multiple expanded queries. Each produced query includes one extra table linked by one or more new join conditions. As a result, for a private relational database, the approach can iteratively bootstrap an evaluation set from a small number of initial queries.

A core principle of our design is to avoid primary reliance on language models to reduce nondeterminism, improve interpretability, and limit hallucinations. Instead, we implement the core join query expansion (JQE) algorithm as a multi-stage pipeline using graph representations rather than text inputs that generates expanded SQL. Language models still play an important yet limited role. They are used within the final stages of the pipeline to generate NL equivalent queries for the expanded SQL.

Generating and evaluating on the new expanded Text-to-SQL queries is costly. Our approach takes into account cost constraints and enables users to control for it at generation time. JQE provides a user-tunable, diversity-aware pruning strategy. This enables users to cap the number of output queries while providing query topology preferences and maintaining coverage of distinct join query patterns. Alternatively, users can generate the exhaustive query set and we provide utilities to retrieve a representative subset of queries for evaluation.

Finally, while our algorithmic approach guarantees syntactic correctness by design in generating expanded SQL, end-to-end semantic validity checks remain essential. For example, we verify that no redundant join conditions are introduced. Consider a query with tables `T1`, `T2`, `T3` and the join conditions "`T1.A = T2.A AND T2.A = T3.A`". By transitivity, this implicitly entails "`T1.A = T3.A`". Such transitive join conditions are typically omitted by users when formulating queries. Accordingly, JQE should neither treat them as contributing to join complexity nor explicitly surface them when generating NL equivalent queries.

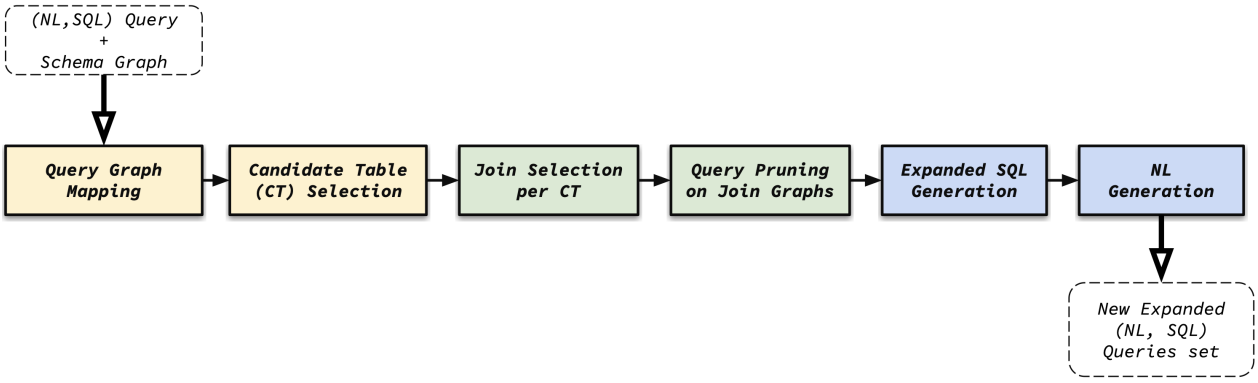

Figure 3.1 The multi-stage pipeline for join query expansion.

Before detailing our JQE approach in the remainder of this chapter, we present it as a multi-stage pipeline in Figure 3.1. The prompts used in relevant stages are provided in Appendix A. The stages are as follows:

- **Input:** JQE takes as input: (i) the database schema, represented as a schema graph (Section 3.1.2) in which nodes correspond to tables and multi-labeled edges denote *joinability*, i.e., each edge indicates that the two tables can be joined, and each label specifies the join conditions; and (ii) a query to be expanded, given as an NL–SQL pair.
- **Query Graph Mapping:** The SQL component of the input query is mapped to a *join query graph* (Section 3.1.3), where nodes represent tables and edges capture the join conditions in the query. This serves as an intermediate representation commonly used in SQL parsing.
- **Candidate Table Selection:** Using the schema graph, this stage identifies candidate tables for expansion. For each table already present in the query, its neighboring tables in the schema graph are examined, and only those not currently in the query are retained as candidates.
- **Join Condition Selection per Candidate Table:** For each candidate table, this stage enumerates all possible join conditions, each represented as a label on an edge from any original query table to the candidate. If m such conditions exist, up to 2^m expanded queries can be generated, since each expansion corresponds to a subset of the possible conditions. Join condition combinations with join redundancy or more than a join condition between two tables are pruned, and the remaining ones are ranked according to user preferences and generation constraints.

- **Query Pruning based on Structural Patterns:** This stage iterates over the possible expansions, retaining only those that correspond to one of the first n per unique join pattern, where n is a user-specified parameter for controlling generation cost.
- **Expanded SQL Generation:** For each join query graph, the selected join conditions are programmatically inserted into the `WHERE` clause to produce a new expanded query. Predicate statistics derived from the existing queries in the evaluation set are optionally used to add additional predicates.
- **NL Generation:** For each expanded SQL query, an equivalent natural language query is generated using a few-shot LLM call with fixed examples.
- **Output:** The final output is a set of expanded NL–SQL query pairs.

In this chapter, we begin by showing an example of SQL join query expansion, starting from the *Candidate Table Selection* to the *Expanded SQL Generation* stages (Section 3.1). Next, we give all algorithmic details for JQE and the rationale for the design of various stages (Section 3.2). Finally, we share detailed experimental evaluation (Section 3.3).

3.1 Join SQL Query Expansion Example

3.1.1 Preliminaries on Schema Graphs

Let S be a schema over a relational database, i.e., over a collection of tables with multiple attributes and integrity constraints such as primary keys (PK) and foreign keys (FK). We represent S as an undirected graph $G_S = (V_S, E_S)$, where:

- V_S is the set of tables in the schema.
- E_S is the set of edges, where $(t_1, t_2) \in E_S$ indicates that t_1 and t_2 can be joined.
- Each edge has one or more labels, each specifying a join condition (e.g., $t_1.a_i = t_2.a_j$).

Example. To better illustrate the schema graph, we consider a database schema defined with the following four `CREATE` commands:

```
CREATE TABLE molecule (
    molecule_id TEXT PRIMARY KEY,
    label TEXT
);
```

```
CREATE TABLE atom (
    atom_id TEXT PRIMARY KEY,
    molecule_id TEXT NOT NULL REFERENCES molecule(molecule_id),
    element TEXT
);
CREATE TABLE bond (
    bond_id TEXT PRIMARY KEY,
    molecule_id TEXT NOT NULL REFERENCES molecule(molecule_id),
    bond_type TEXT
);
CREATE TABLE connected (
    atom_id TEXT NOT NULL REFERENCES atom(atom_id),
    atom_id2 TEXT NOT NULL REFERENCES atom(atom_id),
    bond_id TEXT REFERENCES bond(bond_id),
    PRIMARY KEY (atom_id, atom_id2)
);
```

This schema models data for *toxicology*, the scientific study of the effects of chemical substances and in particular harmful ones on living organisms. We will use it as a running example throughout this section. The database stores information about molecules, the atoms comprising those molecules, and the bonds between atoms along with their types. The corresponding schema graph is shown in Figure 3.2. Each of the four tables is a node and each of the edges represents joinability based on the PK-FK and FK-FK constraints. For example, the (`atom-connected`) edge has two labels each highlighting one possible join for the many-to-many relationship table `connected` with the `atom` table.

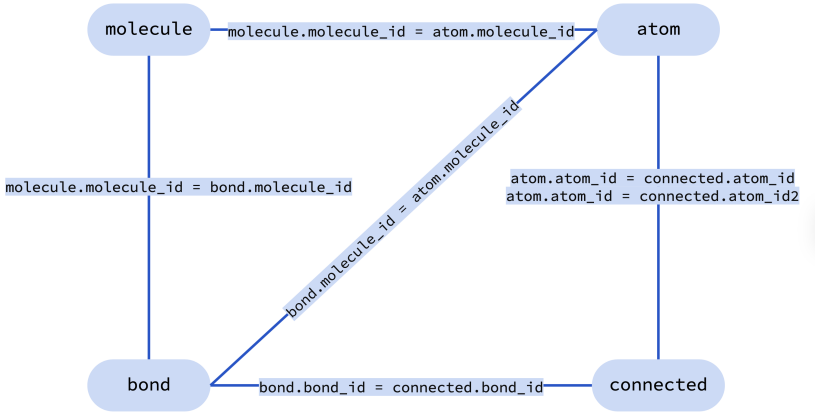

Figure 3.2 Schema graph example.

3.1.2 Expansion Example

Let the schema graph in Figure 3.2 be the input to our JQE algorithm, along with the following SQL query (we omit the NL component in this example):

```
SELECT COUNT(DISTINCT molecule.molecule_id)
  FROM molecule
  JOIN atom ON atom.molecule_id = molecule.molecule_id
 WHERE molecule.label = "-" AND atom.element = "cl"
```

The original query involves the tables `molecule` and `atom`. From the schema graph, `molecule` has an edge to the `bond` table labeled with "`molecule.molecule_id = bond.molecule_id`". We denote this join condition as JC1. And, `atom` has another edge to `bond` labeled with "`atom.molecule_id = bond.molecule_id`". We denote it as JC2.

This results in three possible expansion combinations: { {JC1}, {JC2}, {JC1, JC2} }. For {JC1, JC2}, our semantic checks consider it *redundant* and it is therefore pruned. We define redundancy formally later in this chapter.

Similarly, `atom` has one edge to the `connected` table labeled with two join conditions "`atom.atom_id = connected.atom_id`" and "`atom.atom_id = connected.atom_id2`". We denote these join conditions as JC3 and JC4, respectively. This yields three possible expansion combinations: { {JC3}, {JC4}, {JC3, JC4} }.

We denote the expansion of an input query to a table `T` with the join condition JC as `T`:JC. Using this notation, we assume for now the selected conditions are sorted as: (i) `bond`:JC1; (i) `bond`:JC2; (iii) `connected`:JC3; (iv) `connected`:JC4; (v) `connected`:JC3-JC4. The expanded SQL queries produced are:

(i) $Q_{\texttt{bond:JC1}}$:

```
SELECT COUNT(DISTINCT molecule.molecule_id)
  FROM molecule
  JOIN atom ON atom.molecule_id = molecule.molecule_id
  JOIN bond ON bond.molecule_id = atom.molecule_id
 WHERE molecule.label = "-" AND atom.element = "cl"
```

(ii) $Q_{\texttt{bond:JC2}}$:

```
SELECT COUNT(DISTINCT molecule.molecule_id)
  FROM molecule
  JOIN atom ON atom.molecule_id = molecule.molecule_id
  JOIN bond ON bond.molecule_id = molecule.molecule_id
 WHERE molecule.label = "-" AND atom.element = "cl"
```

(iii) $Q_{\texttt{connected:JC4}}$:

```
SELECT COUNT(DISTINCT molecule.molecule_id)
  FROM molecule
  JOIN atom ON molecule.molecule_id = atom.molecule_id
  JOIN connected ON connected.atom_id = atom.atom_id
 WHERE molecule.label = "-" AND atom.element = "cl"
```

(iv) $Q_{\texttt{connected:JC4}}$:

```
SELECT COUNT(DISTINCT molecule.molecule_id)
  FROM molecule
  JOIN atom ON molecule.molecule_id = atom.molecule_id
  JOIN connected ON connected.atom_id2 = atom.atom_id
 WHERE molecule.label = "-" AND atom.element = "cl"
```

(v) $Q_{\texttt{connected:JC3-JC4}}$:

```
SELECT COUNT(DISTINCT molecule.molecule_id)
  FROM molecule
  JOIN atom ON molecule.molecule_id = atom.molecule_id
  JOIN connected ON connected.atom_id = atom.atom_id
                AND connected.atom_id2 = atom.atom_id
 WHERE molecule.label = "-" AND atom.element = "cl"
```

If the query pruning stage is configured to allow only one query per unique join pattern, then processing these in the order (i), (ii), (iii) (iv) and (v) would result in retaining only the first expanded SQL query, $Q_{\texttt{bond:JC1}}$. Uniqueness is determined using the graph representation of a query's join structure, called a *join query graph*.

3.1.3 Join Query Graph

Join query graphs (JQGs) are one of the intermediate representations used by database management systems during query compilation. A JQG captures only the join structure of a SQL query, omitting projections, filters, and other clauses. Formally, let Q be a SQL query. Its join structure is represented as a graph $G_Q = (V_Q, E_Q)$ such that:

- V_Q is the set of tables appearing in the `FROM` clause of Q.
- E_Q is the set of join edges between the tables in V_Q.

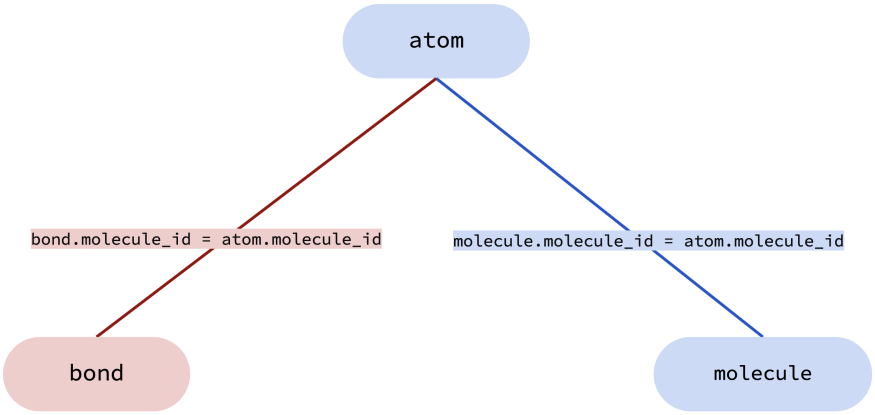

Figure 3.3 Join query graph for $Q_{\texttt{bond:JC1}}$.

Join Query Graph Example. The SQL query $Q_{\texttt{bond:JC1}}$ can be translated into a join query graph, as illustrated in Figure 3.3.

The structure of the join query graphs for SQL queries $Q_{\texttt{connected:JC2}}$ and $Q_{\texttt{connected:JC3}}$ is equivalent to that of $Q_{\texttt{bond:JC1}}$ based on *subgraph isomorphism* [44].

3.2 JQE Algorithm

The full algorithm for join query expansion is Algorithm 1. It takes as input (lines 1-3) the schema graph G_S, a NL-SQL query to expand from, the user preference order (`JCC_preference`), and the current `Eval_Set` to add the expanded queries to.

3.2.1 Stages I and II — Query Graph Mapping and Candidate Selection

In the first stage, JQE parses the input SQL query into a join query graph (Line 6). The second stage then uses the schema graph to identify candidate tables for expansion, excluding those already present in the input query (Lines 8–10).

3.2.2 Stages III — Join Condition Combination (JCC) Generation

For each candidate table (CT), JQE collects the join conditions (JCs) corresponding to the labels on the edges between the CT and the tables in the original input SQL query (Lines 17–22). It then generates all non-empty combinations of these JCs as a power set. The combinations range from a single join condition to the full set of conditions (Lines 23–31).

Algorithm 1 generate_expanded_queries() — Join Query Graph Expansion

Input: G_S /* Schema graph */, Q_{NL}, Q_{SQL} /* NL-SQL query pair */
    `JCC_preference` /* User preference for JC combination: complexity & centrality */
    `Eval_Set` /* Eval set as NL-SQL query pairs. Expanded queries added to it */

/* *Stage I. Query Graph Mapping* */
$G_Q(V_Q,E_Q) \leftarrow$ parse_query_graph(Q_{SQL})
/* *Stage II. Candidate Table (CT) Selection* */
CTs $\leftarrow$ { }
for all $v_t \in V_Q$ **do**
    $V_{expansion} \leftarrow G_S$.get_neighbours($v_t$) - V_Q *// remove tables already in the query*
    for all $v_e \in V_{expansion}$ **do**
        CTs.add([v_t, v_e])
    end for
end for

/* *Stage III. Join Condition Combination (JCC) Generation* */
JCs_per_CT $\leftarrow$ { } *// Dict. from candidate table (key) to possible JCs (value)*
for all (v_t, v_e) $\in$ CTs **do**
    for all $jc \in e(v_t, v_e)$ **do**
        JCs_per_CT.append(v_e, jc)
    end for
end for
JCCs $\leftarrow$ { }
for all $v_e \in$ JCs_per_CT **do**
    // The power set of JCs per CT gives the possible combinations
    for all JCC $\in$ get_power_set(JCs.get(v_e)) **do**
        if JCC != $\emptyset$ and !has_redundancy(JCC) **then**
            JCCs.append(JCC)
        end if
    end for
end for

/* *Stage IV—VI. Query Pruning on Join Graphs and Query Generation* */
for all JCC $\in$ JCCs.sort(JCC_preference) **do**
    new_Q_{SQL} $\leftarrow$ generate_SQL_query(Q_{SQL}, JCC)
    is_unique $\leftarrow$ True
    for all $Q_{SQL\_eval} \in$ `Eval_Set` **do**
        if is_isomorphic(new_Q_{SQL}, Q_{SQL_eval}) **then**
            is_unique $\leftarrow$ False; break
        end if
    end for
    if is_unique **then**
        `Eval_Set`.add([new_Q_{SQL}, generate_equiv_NL(new_Q_{SQL})])
    end if
end for

3.2.3 Redundancy Semantic Check within Stage III

Before using a join condition combination (JCC) for query expansion, we perform a semantic validation to ensure there are no *join redundancies*. A join redundancy occurs when a join condition is implied by others through transitivity. Such joins are uncommon in manually written queries and, more importantly, do not contribute as meaningfully to query complexity as Text-to-SQL systems have one less join condition to generate to obtain the correct result.

Consider the join query graph in Figure 3.4, which illustrates a case of join redundancy: the joins between `bond`, `atom`, and `molecule` are all on the same attribute. Due to transitivity, any two of these join conditions imply the third.

In graph terms, a redundant join condition appears when a labeled edge to be added is part of a cycle in which the two edges from the same table involve the same attribute.

To detect join redundancy (`has_redundancy` in line 27) we examine each JCC by enumerating all cycles involving the candidate table. A cycle is considered transitive if two edges from the same table reference the same attribute. Any JCC containing such a cycle is discarded.

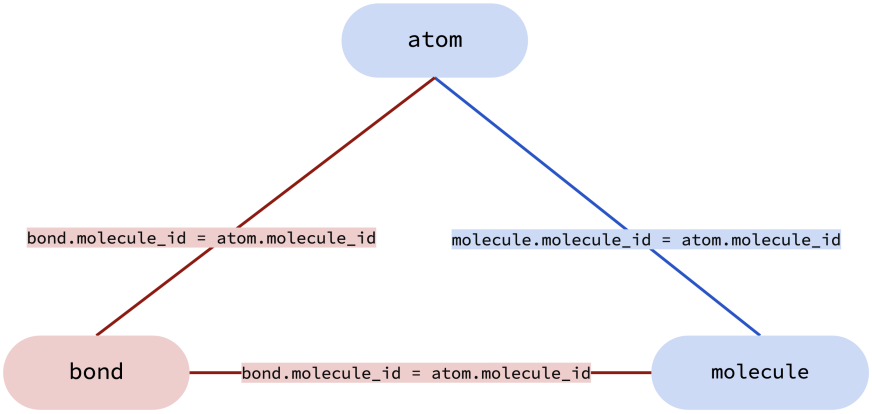

Figure 3.4 Join Redundancy: Example

3.2.4 Stage IV — Query Pruning Using Join Query Graphs

We apply a diversity-aware pruning strategy to the JCCs by performing a graph isomorphism check (Line 38). Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijection $f : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$. This ensures that structurally equivalent join patterns are not duplicated in the output, even when they involve the same number of tables or edges.

For clarity, Algorithm 1 shows a simple uniqueness check; however, our implementation supports a configurable limit on the number of queries retained per unique join pattern. The set of retained queries is referred to as the *pruned set*, while the discarded queries form the *generated set*. Together, they form a possible *expansion set*.

3.2.5 Stage V — Expanded SQL Generation

Each retained JCC is converted into an expanded SQL query via the `generate_SQL_query` function call (Line 35). This function programmatically inserts the new table and its join conditions into the query, producing an expanded SQL statement. Optionally, it may also add projections and predicates derived from statistics collected over the existing evaluation set. If the expanded query executes successfully and returns a non-empty result, it is kept in the final output.

3.2.6 Stage VI — NL Generation

For each generated SQL query, `generate_SQL_query` (Line 35) produces a corresponding NL question. The NL question is generated via a language model call, with the prompt varying according to the type of SQL query. For extended SQL queries that preserve the original portion from the input benchmark, we use Prompt A.1. For queries derived from extended graphs, optionally with added predicates or projections, we use Prompts A.2 and A.3. In all cases, the prompt instructs the model to incorporate the newly added table into the description so that the NL question remains semantically aligned with the modified SQL query. Each resulting NL-SQL pair is then stored according to the query type.

3.2.7 User Preferences

We conclude with cost-control mechanisms and user-defined preferences. Up to this point, we have only discussed the parameter limiting the number of queries retained per unique join pattern in Stage IV. In addition, users may specify a maximum total number of expanded queries to generate, as well as preferences based on query complexity, measured by the number of joins, and table centrality within the schema graph. The output queries are then ordered by complexity and subsequently by centrality (or vice versa), in either ascending or descending order, according to the user's choice. We assume a default of ordering only by complexity in descending order and do not set a maximum number of expanded queries.

3.3 Evaluation

We begin by describing the experimental setup, including the datasets, evaluation metrics, models, and text-to-SQL systems used.

3.3.1 Setup

Dataset. We use the BIRD benchmark in our evaluation. As mentioned in Chapter 2, it is a cross-domain benchmark including NL-SQL query pairs, as well as some external knowledge data called *evidence*, to explain database attributes, certain values or computations, which helps Text-to-SQL systems. BIRD queries are split into three sets: a training set with 9,428 queries over 69 databases (DBs), a development (dev for short) set with 1,534 queries over 11 DBs, and a test set that is hidden from the public and used only to evaluate text-to-SQL systems to join their leaderboard. The test set has 1,789 queries over 15 DBs.

We parse the 1,534 dev set SQL queries into join query graphs. First, we attempt to *unnest* any SQL queries containing Common Table Expressions (CTEs). Any queries that cannot be nested are disregarded in our evaluation. We ended up with 1,410 queries using a programmatic approach. Out of the rest, we used `GPT-4o` as well with checks against the ground truth to ensure we still produce correct output. We successfully unnested 70 extra queries, bringing our total to 1,480 queries as join query graphs over 11 databases.

Metrics. Execution Accuracy (EX) is the metric used by the BIRD benchmark to evaluate end-to-end Text-to-SQL pipelines. It is the proportion of queries for which the output of the predicted SQL query is identical to the output of the ground truth SQL query. The metric assigns a score of 1 if the output matches and 0 otherwise. We report EX as a percentage over queries in the evaluation set.

Models. We used `GPT-4o` and `GPT-4o-mini` in our experiments as `GPT-4o-mini` is more cost-efficient, while `GPT-4o` achieves better EX on more challenging Text-to-SQL queries.

Text-to-SQL Systems. We use three text-to-SQL systems from Section 2.5: (i) CHESS; (ii) MAC-SQL; and (iii) DIN-SQL.

3.3.2 Experiments and Analysis

Remark. The set of retained queries from is referred to as the *pruned set*, while the discarded queries form the *generated set*. Together, they form a possible *expansion set*.

Structural Impact of Join Query Expansion

JQE led to exactly 6,873 queries, divided as follows: 58 queries had join query graphs that were unique, meaning they were non-isomorphic to the original graphs in the initial set or had shapes not encountered during the extension process (also verified through isomorphism checks). We call this the *pruned set*. The remaining 6,815 queries had shapes already seen in either the initial set or the extension process; this is called the *generated set*.

Overall, the expanded set is about five times larger than the initial development set, with queries that are valid in terms of execution.

Set	**Number of Queries**
Original BIRD Dev Set	1534
Augmented Set	6873
Pruned Set	58
Generated Set	6815

Table 3.1 Number of queries in the original and augmented BIRD sets.

First, we began by exploring the join query graphs extended from this rule. Their numbers are similar to the query counts shown in Table 3.1, and our goal was to study their structural distribution. Our rule essentially adds a table, which is represented as a node in our join query graph representation, along with a set of edges corresponding to join relations. This made it interesting to assess how our augmentation performs in terms of increasing the connectivity of the original graphs.

To evaluate this, we examine the *average degree* of nodes in each graph. The *degree* of a node v, denoted $\deg(v)$, is defined as the number of edges connected to v. Since inspecting each node's degree individually can be impractical, we rely on the average degree, denoted $\bar{d}$, which is calculated using the formula:

$$\bar{d} = \frac{2 \times |E|}{|V|}$$

where $|E|$ is the number of edges, $|V|$ is the number of nodes, and the factor of 2 accounts for the fact that each edge connects two nodes. A higher value of $\bar{d}$ indicates a more connected graph. In our case, a more connected join query graph implies more joins between tables, which translates to increased query complexity.

As shown in Figure 3.5, the average degree $\bar{d}$ increases from 0.82 in the original BIRD dev set to 1.35 in our augmented set. This reveals that our augmentation results in more connected graphs, even though we only add a single node and its corresponding edges.

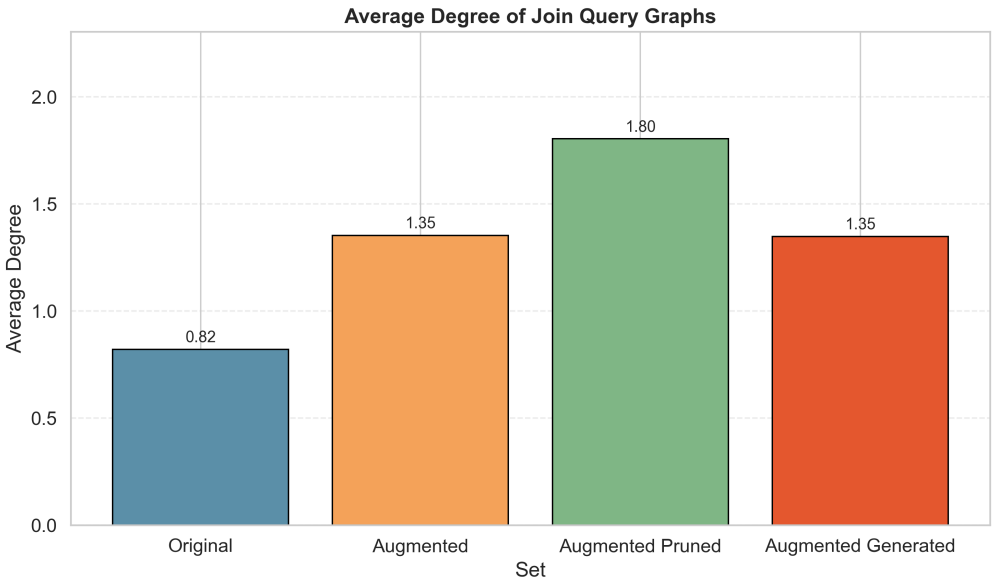

Figure 3.5 Average degrees in the original and augmented sets.

Furthermore, our splits within the augmented sets reveal a variation in average degree. The *pruned set* has a higher average degree of $\bar{d} = 1.80$, compared to the *generated set* with $\bar{d} = 1.35$. This is because the pruned set retains graphs that were not seen in the original set and are structurally unique. That is, each graph shape in this set is encountered only once, resulting in 58 entirely new graph patterns. Our approach made these new structures possible through a guided expansion of input graphs, leading to more connected variants.

We also wanted to explore more closely how the average degrees vary between each augmented join query and its corresponding original version (i.e., before augmentation). Since a single input join query graph can give rise to multiple augmented graphs, each resulting from connecting a new node with various combinations of edges to the original nodes, this naturally leads to multiple new SQL queries per original example. To measure this variation, we compute the *delta average degree*, defined as:

$$\Delta\bar{d} = \bar{d}(G_{\text{Q-aug}}) - \bar{d}(G_{\text{Q-orig}})$$

For each augmented graph, we calculated its average degree and subtracted the average degree of its original counterpart. This difference, $\Delta\bar{d}$, provides a useful signal for how much connectivity changed due to augmentation and explains why.

As shown in Table 3.2, of the 6873 augmented queries we generated, these are the most prevalent $\Delta\bar{d}$:

$\Delta\bar{d}$	# Queries	$(\lvert V_{\text{aug}}\rvert, \lvert E_{\text{aug}}\rvert)$	$(\lvert V_{\text{orig}}\rvert, \lvert E_{\text{orig}}\rvert)$	# Instances
0.33	3910	(3, 2)	(2, 1)	3910
0.17	1356	(4, 3)	(3, 2)	1356
1.00	1166	(2, 1)	(1, 0)	1077
		(3, 3)	(2, 1)	89
0.10	226	(5, 4)	(4, 3)	226
0.67	145	(4, 4)	(3, 2)	145
0.50	32	(5, 5)	(4, 3)	32
0.00	27	(5, 5)	(4, 4)	27
0.07	8	(6, 5)	(5, 4)	8
0.40	2	(5, 6)	(4, 4)	1
		(6, 6)	(5, 4)	1
0.90	1	(5, 6)	(4, 3)	1

Table 3.2 Augmented join query graphs grouped by $\Delta\bar{d}$

$(\lvert V_{\text{orig}}\rvert, \lvert E_{\text{orig}}\rvert)$	Number of Queries
(2, 1)	901
(1, 0)	348
(3, 2)	195
(4, 3)	29
(4, 4)	4
(2, 0)	2
(5, 4)	1

Table 3.3 Original query graph structures

- 3910 queries with $\Delta\bar{d} = \mathbf{0.33}$.
- 1356 queries with $\Delta\bar{d} = \mathbf{0.17}$
- 1166 queries with $\Delta\bar{d} = \mathbf{1.0}$.

The queries with $\Delta\bar{d} = 0.33$ result from adding one node and one edge to existing join query graphs with 2 nodes and 1 edge. This is because such graphs make up a large portion of the original set (around 61%), as shown in Table 3.3, so most extensions naturally stem from them.

The 1,356 queries with $\Delta\bar{d} = 0.17$ result from extending 3-node, 2-edge graphs by adding one node and one edge. These graphs are the third most frequent in the BIRD dev set, with 195 instances (Table 3.3). Their structure allows for more expansion possibilities, especially since 9 out of 11 BIRD schema graphs have at least 4 nodes and $\bar{d}$ ranging from 2 to 6.15,

which shows high connectivity, often with edge counts at least equal to and sometimes up to three times the number of nodes, as in the *formula_1* and *codebase_community* databases (Table 3.5).

Queries with $\Delta\bar{d} = 1.0$ fall into two cases. The majority (1077 out of 1166) come from adding one node and one edge to single-node graphs, which represent about a quarter of the input. The rest (89 out of 1166) are from extending single-node graphs with two existing edges by adding one node and two edges.

Overall, the variation in $\Delta\bar{d}$ values is mainly driven by the imbalance in input graph types. The most common structures are path-like graphs with n nodes and $n-1$ edges, and graphs with fewer nodes and edges, as reflected in Table 3.3.

Set	**Total**	**Acyclic %**	**Cyclic %**
Original (BIRD dev)	1480	99.73	0.27
Augmented	6873	95.69	4.31
Augmented Pruned	58	48.28	51.72
Augmented Pruned by Tables	1106	93.04	6.96

Table 3.4 Distribution of cyclic and acyclic join query graphs across different sets

The cycles in join query graphs indicate more complex joins. Since our goal in augmentation was to increase the complexity of SQL queries, we focused on introducing a portion of cyclic graphs. As shown in Table 3.4, the BIRD dev set contains very few cyclic graphs (0.27%), but our augmentation raised that proportion to 4.31%.

We pruned our generated augmented query graphs in two ways. First, we kept only graph shapes that were new compared to the original input set. This was done through an isomorphism check: if the generated extended graph was isomorphic to one in the input, it was excluded from the pruned set. This pruning ensured that each graph shape was unique in terms of extension, so if two extended graphs were isomorphic, only one was kept. Second, we pruned graphs that shared the same node names and edges—that is, they had the same table names and joins between tables, regardless of the join keys used.

In the first pruned set, 51.72% of the graphs included at least one cycle. This is understandable because the augmentation introduced new graph shapes, and graphs with only one instance can cause the percentage of cyclic graphs to be high. Also, the schema graphs from which we took extra tables and edges are always cyclic, with the number of cycles reaching up to 65,817, as shown in Table 3.5. Most of these graphs (7 out of 11) have at least 3 cycles, which explains the high probability of introducing cycles during augmentation.

For the second pruned set (by tables), 6.96% of the graphs were cyclic.

This shows that even a simple extension, based on schema graphs that include thousands of cycles (especially for graphs with many nodes, see Table 3.5), can significantly increase the number of cycles in the augmented queries when adding one or more nodes.

Databases	$(\lvert V_S\rvert, \lvert E_S\rvert)$	**# Cycles**	$\bar{d}$	**Cycles by Node Size (%)**
formula_1	(13, 40)	65817	6.15	3(0.1%), 4(0.5%), 5(1.8%), 6(5.7%), 7(14.7%), 8(27.4%), 9(32.3%), 10(17.5%)
codebase_community	(8, 24)	2283	6.00	3(1.6%), 4(4.9%), 5(12.6%), 6(25.2%), 7(33.6%), 8(22.1%)
financial	(8, 15)	70	3.75	3(17.1%), 4(27.1%), 5(30.0%), 6(17.1%), 7(8.6%)
student_club	(8, 12)	18	3.00	3(33.3%), 4(33.3%), 5(22.2%), 6(11.1%)
card_games	(6, 9)	8	3.00	3(62.5%), 4(37.5%)
european_football_2	(7, 9)	3	2.57	3(100.0%)
toxicology	(4, 5)	3	2.50	3(66.7%), 4(33.3%)
california_schools	(3, 3)	1	2.00	3(100.0%)
debit_card_specializing	(5, 5)	1	2.00	3(100.0%)
superhero	(10, 10)	1	2.00	3(100.0%)
thrombosis_prediction	(3, 3)	1	2.00	3(100.0%)

Table 3.5 Cycles in BIRD Dev Schema Graphs

Finding. Our approach increases the structural complexity of BIRD dev queries. The expansion generated 6,873 queries, roughly five times the size of the original set, including 58 structurally unique queries. Average node degrees rose from 0.82 to 1.35, indicating higher connectivity, and further analysis showed that this increase was related to the structure of the input graphs. Additionally, more cyclic graphs were introduced, with their proportion rising from 0.27% to 4.31%, demonstrating that even single-node extensions can create more complex join patterns, especially when derived from cyclic schema graphs.

Execution Accuracy under Join Query Expansion

Our expansion led to 58 queries associated with join query graphs that were unique in structure, so we wanted to see their impact on EX, specifically the effect of adding a node and a set of edges resulting in these non-isomorphic graphs. To do so, we evaluated the per-

formance of three text-to-SQL systems: **CHESS**, **DIN-SQL**, and **MAC-SQL**, on the full development set of BIRD (1534 queries), using GPT-4o. As shown in Table 3.6, CHESS leads with approximately 65%, followed by DIN-SQL at 59.11%, and MAC-SQL at 56%.

Text-to-SQL	$\mathbf{EX_{dev}}$	$\mathbf{EX_{ori}}$	$\mathbf{EX_{exp}}$
CHESS	64.86%	63.33%	44.83%
DIN-SQL	59.11%	60.00%	32.76%
MAC-SQL	55.90%	40.00%	39.66%

Table 3.6 Text-to-SQL execution accuracy on different sets

We then focused on the 30 original queries whose join graph expansions produced the 58 new queries. Each system was first tested on the 30 original queries, then on the 58 expanded ones. We denote EX_{ori} as the execution accuracy on the original 30, and EX_{exp} as the accuracy on the 58 expanded ones. Table 3.6 reports these results as well. In terms of the drop between EX_{ori} and EX_{exp}, DIN-SQL drops the most (around 27.24%), followed by CHESS (around 18.5%), and MAC-SQL with only a slight decrease (0.34%). This can be explained by the relatively strong performance of CHESS and DIN-SQL on the original queries, whereas MAC-SQL showed nearly identical performance across both sets.

To better understand the differences, we compute $\mathbf{\Delta EX}$, defined as follows:

- $\mathbf{\Delta EX = 1}$: the original query was incorrectly predicted, while the expanded one was correct;
- $\mathbf{\Delta EX = 0}$: both queries were either predicted correctly or incorrectly;
- $\mathbf{\Delta EX = -1}$: the original was correct, but the expansion failed.

As shown in Table 3.6, most queries fall under $\mathbf{\Delta EX = 0}$, suggesting that systems either succeeded or failed consistently across original and expanded versions. For instance, CHESS had approximately 36.21% of its attempts fail, with the same proportion succeeding, while MAC-SQL succeeded on about 22% of its attempts but failed on nearly twice that proportion.

MAC-SQL had the highest proportion of $\mathbf{\Delta EX = 1}$ cases ($\mathbf{\approx 17\%}$), followed by CHESS with $\mathbf{\approx 9\%}$, and DIN-SQL with $\mathbf{\approx 5\%}$.

However, the most telling category is $\mathbf{\Delta EX = -1}$, capturing cases where performance degraded due to expansion. Here, DIN-SQL performed the worst, with 24.14% of queries falling in this group, followed by CHESS, and finally MAC-SQL, which maintained stable performance across both sets.

To expand the analysis, we manually reviewed the $\mathbf{\Delta EX = -1}$ cases. For DIN-SQL, 4 out of 14 failures were clearly caused by our expansion, confirmed by comparing both ground-truth and predicted join query graphs before and after expansion. For CHESS, 2 out of 11 failures were similarly due to changes in the join graph. For MAC-SQL, only 1 out of 5 failures was directly attributable to the expansion. The remaining failures across all systems were due to related but secondary effects of the expansion, such as missing `DISTINCT`, incorrect projected attributes, or mismatches in `GROUP BY` clauses.

System	$\mathbf{\Delta EX = 1}$	$\mathbf{\Delta EX = -1}$	$\mathbf{\Delta EX = 0 \mid EX_{ori} = 1}$	$\mathbf{\Delta EX = 0 \mid EX_{ori} = 0}$
CHESS	8.62%	18.97%	36.21%	36.21%
DIN-SQL	5.17%	24.14%	27.59%	43.10%
MAC-SQL	17.24%	8.62%	22.41%	51.72%

Table 3.7 Distribution of $\mathbf{\Delta EX}$

Finding. Text-to-SQL systems showed decreased performance when evaluated on our expanded queries, especially those with unique, non-isomorphic join query graphs. DIN-SQL experienced the largest drop in execution accuracy (27.2%), followed by CHESS (18.5%), while MAC-SQL remained relatively stable (0.3% drop). The $\mathbf{\Delta EX = 0}$ values indicate that most predictions remained consistent, but performance failures due to expansion ($\mathbf{\Delta EX = -1}$) were sometimes directly caused by the added nodes and edges, and in other cases triggered secondary errors such as missing `DISTINCT`, incorrect projected attributes, or mismatched `GROUP BY` clauses. These results demonstrate that systems respond differently to our expansion, with MAC-SQL being least affected and DIN-SQL most affected.

Query Complexity and Performance under Expansion

We sampled queries from our expanded set. However, our setting is different: rather than slicing existing tables, we introduce one additional table from the schema and connect it to existing tables using various join paths. In our context, a join corresponds to an edge in the *join query graph*.

We grouped queries by the number of nodes and edges in their join graphs and identified four main categories: 2 tables with 1 edge, 3 tables with 2 edges, 4 tables with 3 edges, and 5 tables with 4 edges. After inspecting the generated queries, we applied consistent selection criteria: for each group, we ensured an equal number of conditions and projections, a consistent number of queries per group, and the same databases under each condition/projection

combination. The final filtered set includes 102 queries for each group, and 408 in total, as summarized in Table 3.8.

#Conditions	#Projections	Databases Involved	#Queries
0	1	student_club	8
1	1	codebase_community (11) european_football_2 (4) financial (14) formula_1 (20) superhero (18)	67
1	2	student_club	5
1	3	financial (4) toxicology (4)	8
1	4	formula_1	10
		Total	**102**

Table 3.8 Query distribution per join group

We then evaluated three publicly available text-to-SQL systems used in our experiments: **CHESS**, **DIN-SQL**, and **MAC-SQL**, all backboned by GPT-4o-Mini. Our results align with those reported by Eckmann et al., even though we did not apply their schema-splitting technique. As shown in Figure 3.6, execution accuracy consistently decreases with the number of joins, dropping below 15% for one of the systems at higher join counts.

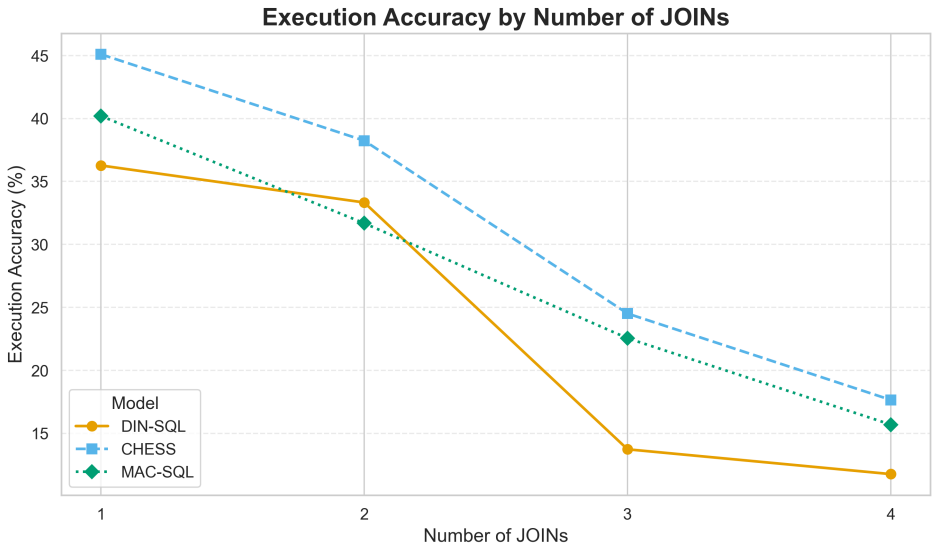

Figure 3.6 EX of different text-to-SQL systems on our sampled expanded set

Finding. This experiment shows that execution accuracy decreases as query complexity increases, where complexity is measured by the number of joins (represented as edges in the join query graph). CHESS, DIN-SQL, and MAC-SQL achieved higher EX values on simpler queries with fewer joins and tables, but performance dropped as the number of joins increased. For queries with 4 joins, EX fell to around 15% or less for all three systems. This confirms that expanding queries, and thereby increasing their complexity, challenges system performance.

3.3.3 Scalability Discussion

In this work, we considered single-threaded execution and presented a simplified description of the JQE algorithm. Several stages of the algorithm can be parallelized. For example, candidate table selection through join selection are independent and thus embarrassingly parallel. In contrast, query pruning depends on graph isomorphism and requires a divide-and-conquer strategy, as a final pass is needed to enforce a uniqueness check among the generated queries before inserting them.

It is worth noting that the scalability of JQE is also influenced by data characteristics, particularly schema sparsity. Denser schemas naturally introduce more potential joins and a greater number of unique patterns to track during query pruning, which can add substantial computational overhead.

CHAPTER 4 TEXTUAL QUERY AUGMENTATION

An important step of converting the question into an executable SQL query is knowing what the question is referring to. The user input must be well interpreted to identify the relevant schema parts in the database. This notion, known as *schema linking*, is where the textual question is linked to the entities and relations within the specified database. In professional environments, language capacities vary between employees, and databases are named after their domains in non-intuitive standards. This makes it even harder to parse the questions into meaningful output. However, public benchmarks tend to have simple names for their databases, tables, and attributes, where you can easily tell what is inside them without needing prior knowledge.

This is where we suggest *textual query augmentation*, an approach designed to challenge the capacities of Text-to-SQL systems in schema linking, precisely in identifying linker tables. Linker tables are the core element of a join pattern within a SQL query; they serve as a hub, such that whenever you need to move between tables, you must pass through them. Out of a natural language (NL) query with its mapped SQL query, received as input to our approach, three new NL queries are generated. Each one contains modified references to linker tables, if such tables exist. The linker tables are found using join query graphs of SQL queries, by retrieving the most central table. We implement three well-known natural augmentation techniques to substitute these references, which provide diverse augmented queries.

The purpose of using augmentation techniques, while tailoring them to our needs through prompting, is that each technique has a different way of hiding these references, either by directly replacing the tokens, by translating the whole text into another language and back while keeping the reference hidden, or by masking the reference and predicting the missing token using context. This reflects the lexical variety of natural language, where words are not always written the same, and large language models (LLMs) make it highly likely to generate different phrasings.

Changing the references to all involved tables from the textual query can also serve to challenge systems in extracting necessary parts of the schema, but it will not be specific to stress the system for evaluation, since it introduces many other possibilities for failed predictions. Instead, we rely on this notion of linker table, which we consider important both for navigating between the involved tables in joins and for constructing the join patterns themselves.

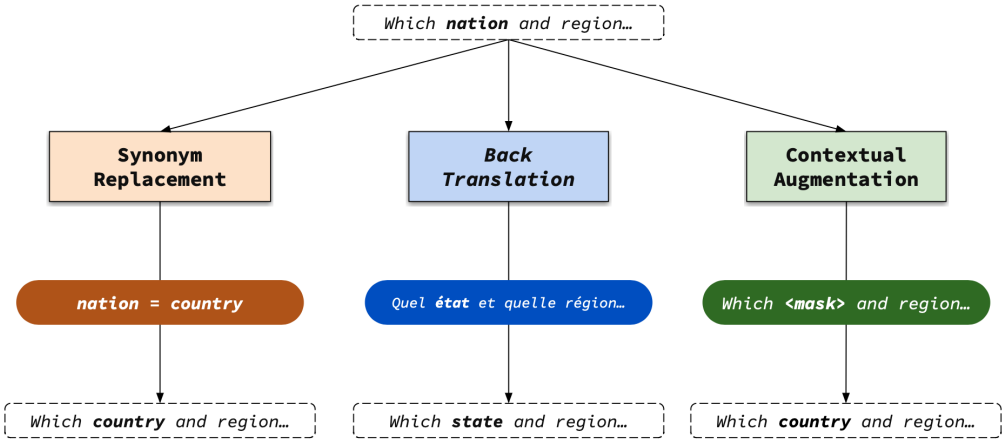

Figure 4.1 The three main techniques for textual query augmentation

In the following lines, we briefly introduce our approach, while focusing on the three augmentation techniques used to hide references to the linker table, as illustrated by Figure 4.1.

TQA takes as input a pair of NL-SQL queries where the NL query being augmented, along with its join query graph, which is an intermediate graph representation, and a database schema graph (Sections 3.1.3 and 3.1.2).

Then, the most central table, named the linker table, is detected by calculating centrality scores for each node in the join query graph. If ties occur, the schema graph is used to identify the linker table, since such tables are likely to be linked to the largest number of tables while building the database.

The table name is split into segments. Each token of the NL query is checked to see whether it refers to the entire linker table name or just one of its segments.

The NL query, with its tokens referencing the linker table, is passed to the three augmentation techniques:

Synonym Replacement. It is a technique that replaces specified tokens with their synonyms while keeping the rest of the sentence intact. These synonyms might come from lexical libraries like WordNet or large language models. In Figure 4.1, we see the input sentence *"Which nation and region..."*, in which the token *"nation"* was specified for replacement. In the intermediate step, *"nation"* is matched with *"country"*, so *"country"* appears in the output while the rest is kept as is [45, 46].

Backtranslation. The input text is translated twice, in two directions: forward translation, where the text starts in English and is translated into the second language, often with

modifications to the entire text, and back translation, where the sentence is translated back into its original language. This back-and-forth translation can lead to changes in certain words, which is useful, especially when we specify what to alter. If we take the example in Figure 4.1, the input *"Which nation and region..."* is translated into French *"Quel état et quelle région..."*, and then back into English: *"Which state and region..."*, where *"nation"* became *"state"* [47].

Contextual Augmentation. This augmentation technique replaces a token with another that matches the context of the surrounding tokens. These replacements are predicted using language models. Unlike simple synonym replacement, this technique takes into account the context of the whole sentence. In Figure 4.1, the sentence *"Which nation and region..."* first goes through the masking phase, where the token to substitute is replaced with a mask and the phrase becomes *"Which <mask> and region..."*. Then, based on the surrounding context, the token *"country"* is predicted as the replacement [48].

The output of our approach is a set of augmented NL queries mapped to their SQL queries.

This chapter will start by illustrating the three augmentation techniques with an example (Section 4.1). Then, we detail the implementation of the previously mentioned steps in a structured algorithm (Section 4.1), before ending with the experimental evaluation (Section 4.3).

4.1 Textual Query Augmentation Example

4.1.1 Schema Graph Example

For this technique, we will use this database schema illustrated by the three `CREATE` commands:

```
CREATE TABLE actor (
  ActorID INTEGER PRIMARY KEY, Name TEXT,
DateOfBirth DATE, BirthCity TEXT, BirthCountry TEXT, HeightInches INTEGER,
    Biography TEXT, Gender TEXT, Ethnicity TEXT, NetWorth TEXT
);
CREATE TABLE movie (
    MovieID INTEGER PRIMARY KEY,
    Title TEXT,
    MPAARating TEXT,
    Budget INTEGER,
    Gross INTEGER,
```

```
    ReleaseDate TEXT,
    Genre TEXT,
    Runtime INTEGER,
    Rating REAL,
    RatingCount INTEGER,
    Summary TEXT
);
CREATE TABLE characters (
    MovieID INTEGER NOT NULL REFERENCES movie(MovieID),
    ActorID INTEGER NOT NULL REFERENCES actor(ActorID),
    CharacterName TEXT,
    CreditOrder INTEGER,
    Pay TEXT,
    Screentime TEXT,
    PRIMARY KEY (MovieID, ActorID)
);
```

This schema describes the *movie* database, which gives us details about actors that played certain characters in different movies. The translated schema graph, as described in Section 3.1.2, is in Figure 4.2. It is a three-node graph with two edges, each labeled by the join keys `MovieID` and `ActorID`, respectively.

4.1.2 Augmentation Example

We consider the following NL/SQL pair, as our input, with the schema graph described in Figure 4.2.

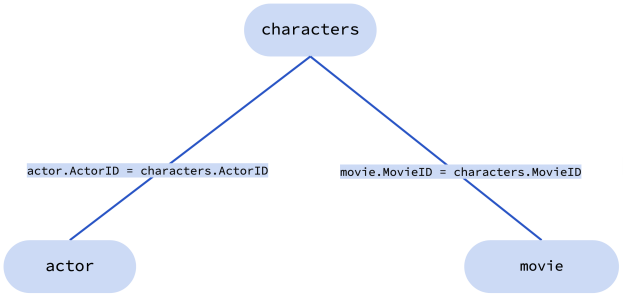

Figure 4.2 Second example of a schema graph

Natural Language Query

Who played the No.2 character in the credit list of the movie "American Hustle"?

```
SELECT T3.Name
 FROM movie AS T1
  INNER JOIN characters AS T2 ON T1.MovieID = T2.MovieID
  INNER JOIN actor AS T3 ON T3.ActorID = T2.ActorID
  WHERE T1.Title = 'American Hustle'
 AND T2.creditOrder = '2'
```

This query is translated to the join query graph as defined in Section 3.1.3. It involves three tables: `actor` linked `characters` through `actor.ActorID = characters.ActorID`, and `movie` also linked to `characters` via `movie.MovieID = characters.MovieID`. When we calculate the centrality of each node of the join query graph, shown in Figure 4.3, we see that the table `characters` is the most central one, so it is considered as a *linker table*.

Then we check the NL query if it involves the linker table name, through tokenizing and lemmatizing it and seeing if it is referring to the linker table.

Natural Language Query

Who | played | the | No.2 | character | in | the | credit | list | of | the | movie | "American Hustle" | ?

After locating the exact tokens referencing the linker table, it is time to apply the three augmentation techniques mentioned earlier to substitute this reference. The output might sometimes be the same, but the approach of each technique is different.

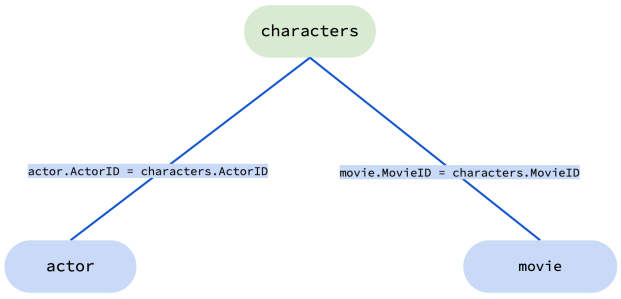

Figure 4.3 Second example of a join query graph

Synonym Replacement. Here, we give the token `character` to a language model (LM), and we ask it to suggest a synonym that is not fully or partially similar (e.g: `southern` for `south`). The LM suggests the token `figure` and when replaced in the NL query, we get:

NL Query with Synonym Replacement

Who played the No.2 figure in the credit list of the movie "American Hustle" ?

Backtranslation. This augmentation translates the entire sentence to a second language, French in this example, using an LM, while instructing it to change the word `character` to a synonym, where the sentence becomes:

NL Query with Backtranslation

Qui a interprété le rôle No.2 dans le générique du film "American Hustle" ?

Then, the back translation to English results in:

NL Query with Backtranslation

Who portrayed the No.2 role in the credit list of the movie "American Hustle" ?

Since the sentence is translated back and forth, tokens like `portrayed` might appear as modifications to the original `played`, even though the LM was not explicitly meant to make this change.

Contextual Augmentation. This technique starts by masking the token to substitute in the initial textual query:

NL Query with Contextual Augmentation

Who played the No.2 `<mask>` in the credit list of the movie "American Hustle" ?

Then, we instruct the LM to predict the masked token, using the context from the surrounding words. Since here the context is about someone who plays in a movie and who shows in the credit list, the predicted token is also `role`:

NL Query with Contextual Augmentation

Who played the No.2 role in the credit list of the movie "American Hustle" ?

In this example, the change happened in one token, but the fact that the whole input sentence is given to the LM, might result in changing some other terms, like in Backtranslation.

4.2 TQA Algorithm

Algorithm 2 is describing the implemented steps of textual query augmentation. The input to this algorithm is a schema graph G_S, a join query graph G_Q and a triplet of NL-SQL-Evidence (i.e., evidence is metadata providing guidance to systems on query construction, common in BIRD benchmark) (Lines 1 - 2). The output will be the `Aug_Set` including the augmented triplet $(Q_{NL}^{aug}, Q_{SQL}, Q_{EV}^{aug})$ (Line 3).

The algorithm only considers join query graphs with more than three nodes (Lines 6). We compute the *betweenness centrality* of its nodes. It quantifies the importance of a node based on how often it lies on shortest paths between other node pairs. Formally, for a node v, the score $C_B(v)$ is:

$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where σ_{st} is the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number that pass through v. Nodes with high scores are considered structural bridges.

We select the node with the highest centrality as the *linker table* (Line 9). In case of a tie, we re-evaluate centrality in the broader schema graph to break it (Line 11).

We then verify if the textual query (NL or EV if exists) is referring to the linker table. We first extract the names of linker tables, segment them into components, and label each component as either an abbreviation or not, since the linker tables can include these kind of segments (Line 16).

Due to the fact that text-to-sql systems cannot digest the full database files and rely on metadata in their schema linking, we preprocess the evidence by removing database values and attributes to prevent augmentation techniques from altering them (Line 18).

We tokenize and lemmatize the text of both NL and evidence queries, storing the position of each token w.r.t the original raw text (Lines 20 - 21). For each lemmatized component,

we compare it to the matched tokens. For *abbreviated components*, we apply a fuzzy vowel-stripped match. Vowels are removed from both the component and the tokens, and the *partial ratio* from the `fuzzywuzzy` library is computed. A match is detected when this score equals 100. For instance, "img" and "image" would yield a match under this rule (Lines 24 - 25). As for *non-abbreviation components*, we compare the lemmatized forms using WordNet's Wu-Palmer similarity [49], with a threshold of 0.95. Wu-Palmer similarity measures how closely related two word senses are, based on the depth of their most specific shared ancestor. Scores range from 0 (no similarity) to 1 (identical), allowing us to detect references such as "sold" referring to "sales" (Lines 27- 28).

Inspired by NLP augmentation techniques [50], we implement three prompting strategies to substitute references to linker tables (Line 34):

Synonym Replacement. (see Prompt B.1) The input here consists of tokens that match references to linker tables in either the question or the evidence. The prompt instructs the language model to suggest synonyms that differ from the input tokens. These synonyms are then inserted back into the original text in the exact same positions, ensuring that only the specified tokens are modified, while the rest of the sentence remains unchanged.

Backtranslation. (see Prompt B.2), the input instead includes the full source text (either the question or the evidence), along with the list of matched tokens and their positions, (database names are also used for additional context). The entire source text is given to the LM to first translate it to a second language, French in our case, while instructing it to replace the matched tokens with their synonym. Then, the LM will translate the sentence back to English. This might result in changing non-highlighted tokens in the original text as possible references.

Contextual Augmentation. (see Prompt B.3), this technique takes the whole text as input like Backtranslation, with the list of matched tokens and database names for context. The LM is instructed here to first mask the matched tokens in the original text. Then, through predicting those masked tokens, the LM tries to put a word that should be different and that is derived from the surrounding context of the other words.

Algorithm 2 generate_augmented_queries() - Textual Query Augmentation

Input: $\boldsymbol{G_S}$ /* Schema graph */, $\boldsymbol{G_Q}$ /* join query graph */
$\boldsymbol{Q_{NL}}$, $\boldsymbol{Q_{SQL}}$, $\boldsymbol{Q_{EV}}$ /* NL-SQL-EV query triplets */
Output: `Aug_Set` /* Aug set as NL-SQL-EV query triplet. Aug. queries added to it */

/* *Linker Table (LT) Identification* */
if $|\boldsymbol{V_Q}| > 3$ **then**
  $\boldsymbol{M} \leftarrow \arg\max_{v_t \in V_Q} \mathrm{BC}(\boldsymbol{v_t})$ // *BC: Betweenness Centrality*
  if $|\boldsymbol{M}| = 1$ **then**
    $\boldsymbol{LT} \leftarrow \boldsymbol{M}[1]$ // *the unique element of* $\boldsymbol{M}$
  else
    $\boldsymbol{LT} \leftarrow \arg\max_{v_t \in V_S} \mathrm{BC}(\boldsymbol{v_t})$ // $\boldsymbol{G_S(V_S, E_S)}$
  end if
end if

/* *Reference Detection* */
Segment $\boldsymbol{LT} = \{\boldsymbol{s_1}, \ldots, \boldsymbol{s_m}\}$ and classify $\boldsymbol{s_i}$ as `abbr` or `non-abbr`
if $\boldsymbol{Q_{EV}} \neq \emptyset$ **then**
  $\boldsymbol{Q_{EV}} \leftarrow \mathrm{Preprocess}(\boldsymbol{Q_{EV}})$ //remove DB values/attributes
end if
$\boldsymbol{T_{NL}} \leftarrow \mathrm{Tokenize{+}Lemmatize}(\boldsymbol{Q_{NL}})$
$\boldsymbol{T_{EV}} \leftarrow \mathrm{Tokenize{+}Lemmatize}(\boldsymbol{Q_{EV}})$
for $\boldsymbol{s} \in \boldsymbol{LT}$ **do**
  if $\mathrm{type}(\boldsymbol{s}) =$ `abbr` **then**
    $\boldsymbol{M_s^{NL}} \leftarrow \{\boldsymbol{t} \in \boldsymbol{T_{NL}} \mid \mathrm{FuzzyPartial}(\boldsymbol{s}, \boldsymbol{t}) = \mathbf{100}\}$
    $\boldsymbol{M_s^{EV}} \leftarrow \{\boldsymbol{t} \in \boldsymbol{T_{EV}} \mid \mathrm{FuzzyPartial}(\boldsymbol{s}, \boldsymbol{t}) = \mathbf{100}\}$
  else
    $\boldsymbol{M_s^{NL}} \leftarrow \{\boldsymbol{t} \in \boldsymbol{T_{NL}} \mid \mathrm{WuPalmer}(\boldsymbol{s}, \boldsymbol{t}) \geq \mathbf{0.95}\}$
    $\boldsymbol{M_s^{EV}} \leftarrow \{\boldsymbol{t} \in \boldsymbol{T_{EV}} \mid \mathrm{WuPalmer}(\boldsymbol{s}, \boldsymbol{t}) \geq \mathbf{0.95}\}$
  end if
end for
/* *Augmentation Techniques* */
for $\boldsymbol{s} \in \boldsymbol{LT}$ **do**
  if $\boldsymbol{M_s^{NL}} \neq \emptyset$ or $\boldsymbol{M_s^{EV}} \neq \emptyset$ **then**
    for all strategy $\in$ {SynRep, BackTrans, CtxAug} **do**
      if $\boldsymbol{M_s^{NL}} \neq \emptyset$ **then**
        $\boldsymbol{Q_{NL}^{aug}} \leftarrow \mathrm{Apply}(\boldsymbol{strategy}, \boldsymbol{Q_{NL}}, \boldsymbol{M_s^{NL}})$
      end if
      if $\boldsymbol{M_s^{EV}} \neq \emptyset$ **then**
        $\boldsymbol{Q_{EV}^{aug}} \leftarrow \mathrm{Apply}(\mathrm{strategy}, \boldsymbol{Q_{EV}}, \boldsymbol{M_s^{EV}})$
      end if
      `Aug_Set` $\leftarrow$ `Aug_Set` $\cup \{(\boldsymbol{Q_{NL}^{aug}}, \boldsymbol{Q_{SQL}}, \boldsymbol{Q_{EV}^{aug}})\}$
    end for
  end if
end for

4.3 Evaluation

4.3.1 Setup

Dataset Similar to the prior chapter, we use the BIRD benchmark. We group all queries to evaluate our linker table elimination approaches without following the three BIRD splits. We split all queries into train, dev, and test sets in a logical manner. The train and dev sets were used to verify and refine prompts, while the test set was reserved for evaluating generalization to unseen data. To ensure a meaningful split, we applied K-Means clustering with $\boldsymbol{k = 2}$ on the databases, using the node distribution of their join query graphs as features. This was done to maintain a balanced node distribution across the sets.

The resulting clusters are shown in Figure 4.4. We selected the cluster with the larger number of databases and performed a split that preserved node distribution, number of queries, and database diversity. We fixed the random seed at 35 to ensure reproducibility. Table 4.1 summarizes the number of queries, databases, and node distributions in each split: the train set includes 466 queries from 31 databases, the dev set 125 queries from 10 databases, and the test set 106 queries from 7 databases.

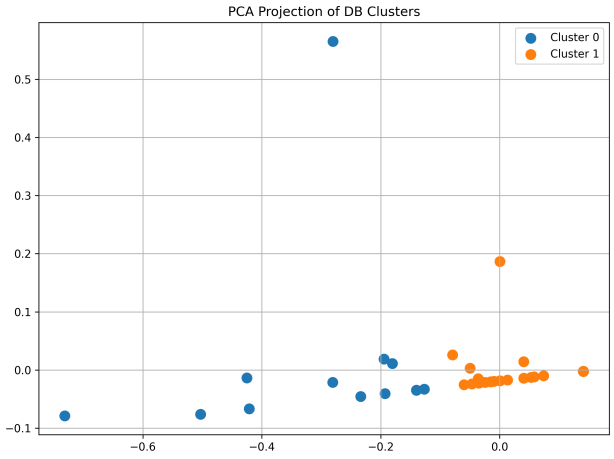

Figure 4.4 Clustered databases per node distribution

Split	#Queries	#DBs	Node Count Distribution
Train	466	31	3: 92.70%, 4: 6.22%, 5: 1.07%
Dev	125	10	3: 92.00%, 4: 7.20%, 6: 0.80%
Test	106	7	3: 93.40%, 4: 5.66%, 5: 0.94%

Table 4.1 Query distribution and node count per split

Used Metrics Alongside the EX metric used for this rule, we define our own metric called the *hiding score*. This score checks, for each component of the linker table name, whether that component is still present in the natural language (NL) query (referred to as the *question*) or in the external knowledge (referred to as the *evidence*). If the tokens referring to these components remain unchanged, the score is 0; if they have been replaced, the score is 1. This metric allows us to determine whether the token replacement has effectively occurred.

4.3.2 Experiments and Analysis

Linker Table References in Questions vs. Evidence

As shown in Table 4.2, we observe that, on average, 1.34 parts of linker tables appear in the train set, 1.2 in the dev set, and 1.18 in the test set. This indicates a high likelihood of referencing at least one part in the questions across all splits.

In contrast, this is less common in the evidence, where we find only 0.62 parts referenced in the train set, 0.49 in dev, and 0.48 in test. This is largely due to some evidence lacking explicit or even approximate mentions of the linker table name.

Looking at exact tokens referring to the linker table, questions contain about 1.6 such tokens on average in train, 1.3 in dev, and 1.4 in test.

For evidence, the average is lower: 0.72 in train, 0.58 in dev, and 0.59 in test. This again stems from the fact that evidence often omits direct references to linker tables. Evidence typically provides external knowledge about what is in the database in terms of attributes and values that may be relevant for predicting the SQL query. Since this concerns a linker table name, it is rarely mentioned directly in evidence and tends to appear only when its attributes are required for projections, filters, or aggregations.

Statistic	**Train**	**Dev**	**Test**
# Queries	466	125	106
# Questions	466	125	106
# Evidence	443	122	106
# LT Components in Questions	625	151	126
# LT Components in Evidence	276	60	51
# Matching Question Tokens	754	161	149
# Matching Evidence Tokens	320	71	63

Table 4.2 Dataset statistics across train, dev, and test sets.

Finding. Linker table names appear more often in questions (up to 1.34 components; up to 1.6 tokens) than in evidence (up to 0.62 components; minimum 0.58 tokens). This indicates that augmentation strategies targeting linker table references should focus on questions, while evidence only needs processing if it contains explicit mentions.

Augmentation Effects on Linker Table References

To refine our prompts and verify whether they effectively replace tokens referring to the linker table, we sampled 50 queries from the train set and used GPT-4o for higher-quality generations. As shown in Table 4.3, Synonym Replacement achieved a 100% hiding score in both questions and evidence. This is expected, as it directly replaces the target tokens with synonyms in place. For Backtranslation, the generated questions hid 96% of linker table components, and evidence reached 100%. Contextual Augmentation performed slightly lower, with 94% in questions and 89% in evidence.

We then validated the prompts on the dev set. Although Synonym Replacement is more straightforward in its approach, it succeeded in hiding 99.2% of linker table components in questions. This small drop is due to one case where a synonym used was also a token that needed to be replaced. Backtranslation matched this 99.2% in questions and reached 90.7% in evidence. Contextual Augmentation achieved 100% in questions and 88.9% in evidence.

Finally, we evaluated the techniques on a new, unseen test set. Synonym Replacement maintained its perfect score in both questions and evidence. Backtranslation hid 97.2% of relevant components in questions and 100% in evidence. Contextual Augmentation reached 97.2% for questions and 82.2% for evidence.

The success of Synonym Replacement lies in its simplicity: it replaces tokens directly in position using only the list of matched tokens. In contrast, Backtranslation and Contextual Augmentation take the full question or evidence text along with the matched tokens, perform intermediate transformations, and return an output where target tokens may still remain unchanged. This variability comes from the nature of each method. Backtranslation involves translating to another language and back, while Contextual Augmentation masks tokens and selects the most probable replacements. These steps introduce uncertainty, which makes the outcome less consistent.

Set	Technique	Question Score (%)	Evidence Score (%)
Train	Synonym Replacement	100	100
Train	Backtranslation	96	100
Train	Contextual Augmentation	94	89.3
Dev	Synonym Replacement	99.2	100
Dev	Backtranslation	99.2	90.7
Dev	Contextual Augmentation	100	88.9
Test	Synonym Replacement	100	100
Test	Backtranslation	97.2	100
Test	Contextual Augmentation	97.2	82.2

Table 4.3 Hiding scores by technique and dataset split.

Finding. Synonym Replacement achieves nearly perfect scores in modifying tokens that refer to the linker table across all dataset splits. In contrast, Backtranslation and Contextual Augmentation perform slightly worse due to intermediate transformations that may leave some references unchanged. This indicates that Synonym Replacement is directly focused on changing the target tokens, making it a better option when the goal is to alter linker table references while preserving the rest of the sentence.

Alignment of Automated Accuracy Scores with Human Judgments

We also manually reviewed the dev and test sets to assess the reliability of our newly introduced hiding accuracy metric. This involved comparing the original and augmented versions of each question and evidence. A score of 0 was given if the linker table was still referenced (due to tokens not being replaced or replaced by very similar synonyms), 1 if the replacement was successful, and `None` when there was no reference to judge. This last case mostly occurred in the evidence.

As shown in Table 4.4, for questions in the dev set, Contextual Augmentation showed the largest gap between automated and human scores (5.6%), followed by Backtranslation (4.0%), and then Synonym Replacement with the smallest gap (2.4%). For evidence, the same pattern holds: Contextual Augmentation had the largest difference (4.7%), Backtranslation had 2.2%, and Synonym Replacement showed no gap (0%). This is mainly because Contextual Augmentation and Backtranslation tend to alter more than just the targeted tokens due to their intermediate steps. In contrast, Synonym Replacement performs direct and more efficient substitutions.

On the test set, automated and human scores aligned perfectly in both questions and evidence for Synonym Replacement and Contextual Augmentation, with a difference of 0%. For Backtranslation, the human score was slightly higher for questions, resulting in a negative difference of **−0.9%**, while for evidence, the gap reached 4.3%. This can be attributed to more substantial changes in the text.

Overall, the differences between automated and human scores remain modest, with a maximum of 5.6% for questions and 4.7% for evidence. This is expected, since the primary goal of these techniques is to replace tokens with synonyms. The automated score is designed to capture that, while the human evaluation also considers the quality of the surrounding text in the question or evidence.

Set	**Technique**	**Metric**	**Question**	**Evidence**
Dev	Synonym Replacement	Automated Score	99.2%	100.0%
		Human Score	96.8%	100.0%
		Difference	2.4%	0.0%
	Backtranslation	Automated Score	99.2%	90.7%
		Human Score	95.2%	88.5%
		Difference	4.0%	2.2%
	Contextual Augmentation	Automated Score	100.0%	88.9%
		Human Score	94.4%	84.2%
		Difference	5.6%	4.7%
Test	Synonym Replacement	Automated Score	100.0%	100.0%
		Human Score	100.0%	100.0%
		Difference	0.0%	0.0%
	Backtranslation	Automated Score	97.2%	100.0%
		Human Score	98.1%	95.7%
		Difference	-0.9%	4.3%
	Contextual Augmentation	Automated Score	97.2%	82.2%
		Human Score	97.2%	82.2%
		Difference	0.0%	0.0%

Table 4.4 Automated and human hiding scores by technique and dataset.

Finding. The automated hiding metric aligns closely with human judgments across dataset splits. Synonym Replacement shows the smallest differences, with a maximum of 2.4% for questions, while Backtranslation and Contextual Augmentation exhibit larger discrepancies, reaching up to 5.8% for evidence. This confirms that our metric reliably captures changes to linker table references, and that Synonym Replacement is the most precise method, modifying target tokens while minimally affecting the rest of the text.

Impact of Augmentation Techniques on Execution Accuracy

We evaluated the three text-to-SQL systems on the test set using the original queries from BIRD, as well as their augmented versions produced by Synonym Replacement (SR), Back-translation (BT), and Contextual Augmentation (CA). As shown in Table 4.5, CHESS maintained nearly the same execution accuracy (EX) across all versions, with values ranging between 62.26% and 63.21%. DIN-SQL showed consistent improvements over the original EX (58.49%) with all three techniques: 60.38% for SR, 62.26% for BT, and 61.32% for CA. In contrast, MAC-SQL showed mixed results. While BT preserved the original EX (56.60%), SR and CA led to performance drops, reaching 54.72% and 50.94%, respectively.

System	**Original Queries**	**SR Queries**	**BT Queries**	**CA Queries**
CHESS	63.21%	62.26%	63.21%	63.21%
DIN-SQL	58.49%	60.38%	62.26%	61.32%
MAC-SQL	56.60%	54.72%	56.60%	50.94%

Table 4.5 Execution accuracy (%) across original and augmented queries.

We then compared the execution accuracy (EX) of the predicted SQL queries after applying each augmentation technique to the EX of the original queries, for all three systems.

$\mathbf{\Delta}$EX can take three main values: 1, 0, or $\mathbf{-1}$. A value of $\mathbf{-1}$ indicates that the prediction failed after applying the augmentation, while the original prediction was successful. A value of 1 means that the augmented query led to a successful prediction, whereas the original one failed. A value of 0 indicates that both predictions (before and after augmentation) either failed or succeeded, within the same system. Specifically:

- $\mathbf{\Delta}\text{EX} = 0 \mid \text{EX} = 1$: Both the original and augmented queries were predicted correctly.
- $\mathbf{\Delta}\text{EX} = 0 \mid \text{EX} = 0$: Both the original and augmented queries failed in prediction.

As shown in Table 4.6, most cases across all systems fall under the category $\mathbf{\Delta}\text{EX} = 0 \mid \text{EX} = 1$. This suggests that the augmented queries generally led to the same prediction outcomes as the original ones, regardless of the technique. Regarding $\mathbf{\Delta}\text{EX} = 0 \mid \text{EX} = 0$, up to 39% of test queries with augmentation were predicted incorrectly, matching the failure of their original counterparts. As for $\mathbf{\Delta}\text{EX} = 1$, DIN-SQL shows the highest proportion, peaking at 6.6%, meaning that more queries were predicted correctly only after augmentation. MAC-SQL follows, with CHESS showing the lowest share at just 1.89%. Finally, for $\mathbf{\Delta}\text{EX} = \mathbf{-1}$, where augmented queries caused incorrect predictions while the originals were successful,

MAC-SQL ranks highest at 9.43% for Contextual Augmentation. CHESS and DIN-SQL both show 2.83% under this category for Backtranslation and Contextual Augmentation.

We also manually inspected cases where $\mathbf{\Delta}\text{EX} = 1$ and $\mathbf{\Delta}\text{EX} = \mathbf{-1}$ to determine whether the changes were due to the inclusion or exclusion of linker tables in the predicted SQL queries.

CHESS: Across SR, BT, and CA, the cases with $\mathbf{\Delta}\text{EX} = \mathbf{-1}$ were not caused by missing linker tables, which were present in both predictions. The errors typically stemmed from handling ties (e.g., `MAX` vs. `LIMIT 1`), incorrect conditions, missing tables, or unwanted groupings. For $\mathbf{\Delta}\text{EX} = 1$, improvements were mostly due to correcting projections, replacing linker tables with valid equijoins, or adding previously omitted tables and again, not due to missing or inserted linker tables.

DIN-SQL: In all techniques, $\mathbf{\Delta}\text{EX} = \mathbf{-1}$ cases were rarely due to linker table absence. Instead, they resulted from projection errors, tie handling, grouping mistakes, or missing tables. For $\mathbf{\Delta}\text{EX} = 1$, the improvements came from similar areas: better handling of projections, table inclusion, or corrected join conditions. The presence or absence of the linker table was not a determining factor in most cases.

MAC-SQL: The causes of $\mathbf{\Delta}\text{EX} = \mathbf{-1}$ across techniques included missing tables, projection issues, aggregation errors, and sometimes the use of `DISTINCT`. Only one case involved a missing linker table. For $\mathbf{\Delta}\text{EX} = 1$, most changes were due to improved join construction or fixing omissions in tables or projected attributes rather than changes in linker table usage.

System	Technique	Δ EX = 1	Δ EX = 0 | EX = 1	Δ EX = 0 | EX = 0	Δ EX = -1
CHESS	SR	1.89%	60.38%	34.91%	2.83%
	BT	2.83%	60.38%	33.96%	2.83%
	CA	2.83%	60.38%	33.96%	2.83%
DIN-SQL	SR	6.6%	53.77%	34.91%	4.72%
	BT	6.6%	55.66%	34.91%	2.83%
	CA	5.66%	55.66%	35.85%	2.83%
MAC-SQL	SR	3.81%	51.43%	39.05%	5.71%
	BT	5.66%	50.94%	37.74%	5.66%
	CA	3.77%	47.17%	39.62%	9.43%

Table 4.6 Comparison of ($\mathbf{\Delta}$EX) across systems and augmentation techniques.

Finding. Text-to-SQL systems responded differently to augmented queries: CHESS maintained its performance, DIN-SQL improved, and MAC-SQL generally decreased in accuracy compared to the original queries. Most predictions retained the same execution accuracy after augmentation (over half of queries with **Δ**EX = 0 | EX = 1). Failures caused by augmentation after previously correct predictions (**Δ**EX = -1) occurred mostly in DIN-SQL, but were not due to missing linker tables. They stemmed from other factors such as projection or join issues. This indicates that augmentation techniques modified linker table references without substantially harming system performance.

Linker Attribute References and System Reliance under Augmentation

The outputs made us investigate what might contribute to correct linker table predictions, even in the absence of explicit references. We hypothesized that the text-to-SQL could exploit implicit cues from linker table attributes, especially when they were mentioned in the question or evidence. To explore this, we analyzed all three data splits by extracting linker table attributes from the SQL queries and categorizing them as used in either projections, conditions, or joins. We then checked whether these attributes were directly referred to in the corresponding questions or evidence.

As shown in Table 4.7, in the train set, most linker table attributes used in joins are potentially referred to in the questions (51.9%) and are even more frequently referred to in the evidence (53.5%). In the dev set, join attributes remain the most commonly referenced, with 63.2% in questions and 68.0% in evidence. A similar thing was observed in the test set, where join attributes are the most referred to in both questions (37.7%) and evidence (52.8%).

Split	# Queries	LT Join Atts in Qs	LT Proj/-Filter Atts in Qs	LT Join Atts in Evs	LT Proj/-Filter in Evs
Train	466 (466 Qs | 443 Evs)	51.9%	18.9%	53.5%	29.8%
Dev	125 (125 Qs | 122 Evs)	63.2%	22.4%	68.0%	30.3%
Test	106 (106 Qs | 106 Evs)	37.7%	27.4%	52.8%	40.6%

Table 4.7 Statistics on linker table attribute mentions in questions (Qs) and evidences (Evs) across dataset splits.

For the Dev and Test sets, we examined whether augmentation techniques could reduce the system's reliance on these implicit cues. The results are shown in Table 4.8. In Dev, the

proportion of join attributes referred to in questions was nearly halved across all techniques, while projections and filter attributes dropped by approximately 4 to 6%. However, in the evidence, although the percentage of join attributes slightly decreased, it remained high across all techniques, and the proportions for projections and filters appeared largely unaffected.

In Test, the decreases were more pronounced across all techniques for both join attributes and projections/filtering attributes in questions. In contrast, the percentages in evidence remained nearly unchanged for both types of linker table attributes.

Split	Tech	# Queries	Join Atts in Qs	Proj/Filter Atts in Qs	Join Atts in Evs	Proj/Filter in Evs
Dev	Original	125 (125 Qs | 122 Evs)	63.2%	22.4%	68.0%	30.3%
	SR	125 (125 Qs | 122 Evs)	30.4%	18.4%	63.9%	30.3%
	BT	125 (125 Qs | 122 Evs)	32.0%	16.8%	61.8%	29.3%
	CA	125 (125 Qs | 122 Evs)	29.6%	18.4%	63.2%	29.6%
Test	Original	106 (106 Qs | 106 Evs)	37.7%	27.4%	52.8%	40.6%
	SR	106 (106 Qs | 106 Evs)	25.5%	19.8%	52.8%	39.6%
	BT	106 (106 Qs | 106 Evs)	26.4%	17.9%	51.9%	39.6%
	CA	106 (106 Qs | 106 Evs)	26.4%	18.9%	50.9%	39.6%

Table 4.8 Distribution of linker table attribute mentions in questions (Qs) and evidences (Evs) across DEV and TEST splits for each augmentation technique.

Finding. Linker table attributes can also help predict linker tables when they are mentioned in textual queries. Our analysis of tokens potentially referring to linker tables via their attributes shows that augmentation techniques substantially reduce their presence in questions, while the effect on evidence is minimal. This suggests that text-to-SQL systems may rely on more than just linker table names for correct predictions, with evidence serving as an important source of implicit knowledge when evaluating on BIRD.

Frequency of Schema Linking Retrieving Linker Tables

The investigation of potential implicit cues made us curious about how schema linking performs in retrieving necessary tables. Schema linking is an early step in most text-to-SQL systems that interacts with the database schema. It simply connects keywords extracted from the question to possible tables and columns, which are later used for SQL prediction

via prompting techniques [27]. Although the representations may vary, the goal remains the same: to retrieve relevant parts of the schema from the question and/or evidence as external knowledge.

We wanted to evaluate how schema linking handles augmented questions and evidence. To do so, we fed our test set, with different augmentation techniques applied, into a schema linker that first extracts tables and then relevant columns. Since we only needed tables, we simply checked whether the linker table was included in the returned list. Table 4.9 presents these results. Out of 106 queries in the test set, the percentage of queries for which the linker table was not retrieved remained roughly constant across all techniques and compared to the original queries, with about 8 to 9 queries where the linker table did not appear in the retrieved table list.

Technique	**LT in Retrieved Tables**	**LT not in Retrieved Tables**
Original	97 / 106 (91.51%)	9 / 106 (8.49%)
SR	97 / 106 (91.51%)	9 / 106 (8.49%)
BT	98 / 106 (92.45%)	8 / 106 (7.55%)
CA	97 / 106 (91.51%)	9 / 106 (8.49%)

Table 4.9 Presence of linker tables (LT) in retrieved tables across different augmentation techniques.

Finding. A naive schema linking module successfully retrieves the linker table in over 91% of queries across all three augmentation techniques, with results nearly identical to the original textual queries. This indicates that the augmentation approaches have minimal impact on challenging the schema linker in retrieving the correct linker tables.

CHAPTER 5 CONCLUSION

This thesis set out to address a central challenge in the evaluation of Text-to-SQL systems: the mismatch between public benchmarks and the complexity of enterprise workloads. While benchmarks such as Spider, BIRD, and Beaver have driven substantial progress, our structural analysis (Chapter 1) demonstrated that they remain limited in both database structure and query diversity. As a result, they fail to capture the join-heavy, schema-rich characteristics of real-world deployments. Without more representative evaluation methods, systems that perform well on academic benchmarks may offer misleading indications of their robustness in practice on private datasets.

To bridge this gap, we introduced two complementary techniques that enable systematic stress-testing of Text-to-SQL systems without requiring costly manual evaluation set construction. The first, **Join Query Expansion (JQE)** (Chapter 3), increases SQL query complexity by expanding join structures. By relying on schema and query graph representations, pruning redundant joins, and generating natural language counterparts in a programatic fashion, JQE creates structurally richer evaluation sets that expose weaknesses in SQL generation. The second, **Textual Query Augmentation (TQA)** (Chapter 4), targets the natural language side of the task by identifying linker tables and applying augmentation strategies to modify their references. TQA probes schema linking robustness, introducing common linguistic variability in enterprise environments but absent from most benchmarks.

Our experiments of both JQE and TQA highlight the effectiveness of these approaches. JQE significantly increases structural diversity, raising join complexity and cyclicity, and leads to execution accuracy drops of around 20% in state-of-the-art systems. TQA, while yielding smaller degradations of up to 5.6%, demonstrates the resilience of current systems under controlled perturbations with little weaknesses in schema retrieval. cTogether, these findings underscore the value of a dual perspective on evaluation, in which both the SQL and NL-side challenges are systematically explored.

5.1 Limitations

While this work advances the state of Text-to-SQL evaluation, it is not without limitations. Our methods were developed and tested primarily on the BIRD benchmark, which, although representative in some respects, does not encompass the full diversity of Text-to-SQL workloads. The expansion process focuses on structural aspects of SQL queries—adding tables

and joins—while leaving aside other dimensions of complexity such as filtering conditions, aggregations, and nested subqueries. Similarly, TQA operates only on the natural language side and does not alter SQL queries themselves, potentially limiting the scope of augmentation. Both methods also rely on LLMs for natural language generation and augmentation, introducing challenges related to prompt sensitivity, possible hallucination, and reproducibility. Finally, our evaluation primarily used execution accuracy (EX), which, while widely adopted, may not fully capture partial correctness or semantic fidelity.

5.2 Future Directions

All of the prior limitations can be a fertile ground for future research directions. Expanding JQE to incorporate richer SQL constructs, such as aggregations, subqueries, and grouping, would enable broader stress-testing of system capabilities. Incorporating more semantic checks alongside structural ones could further improve the quality and interpretability of generated queries. On the augmentation side, extending TQA to operate jointly on natural language and SQL would create more realistic perturbations, while experimenting with varied prompting strategies and multiple language models could increase diversity and robustness.

Beyond extending the methods themselves, future work should explore their application to a wider range of benchmarks and industrial datasets. This would provide a deeper understanding of how structural properties of schemas influence Text-to-SQL system performance and reveal whether the trends observed here generalize across domains. Additional evaluation metrics, including measures of partial correctness and semantic integrity, would also complement execution accuracy and provide a more nuanced view of system behavior after the expansion and augmentation of an evaluation set.

5.3 Closing Remarks

In summary, this thesis contributes two novel, automatic techniques: Join Query Expansion (JQE) and Textual Query Augmentation (TQA). They provide systematic, controlled, and reproducible means of stress-testing Text-to-SQL systems. By exposing weaknesses in SQL generation and probing schema linking robustness, these methods help close the gap between academic evaluation and enterprise reality. Ultimately, they represent a step toward more reliable, interpretable, and enterprise-ready Text-to-SQL systems through rigorous evaluation. This in turn, would advance the long-standing goal of natural language access to data that is both accurate and trustworthy.

REFERENCES

[1] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," Brussels, Belgium, 2018, pp. 3911 – 3921.

[2] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls," vol. 36, New Orleans, LA, United states, 2023.

[3] P. B. Chen, F. Wenz, Y. Zhang, D. Yang, J. Choi, N. Tatbul, M. Cafarella, C. Demiralp, and M. Stonebraker, "Beaver: An enterprise benchmark for text-to-sql," 2024.

[4] B. Qin, B. Hui, L. Wang, M. Yang, J. Li, B. Li, R. Geng, R. Cao, J. Sun, L. Si, F. Huang, and Y. Li, "A survey on text-to-sql parsing: Concepts, methods, and future directions," 2022.

[5] T. Eckmann, M. Urban, J.-M. Bodensohn, and C. Binnig, "HLR-SQL: Human-like reasoning for text-to-SQL," in *Novel Optimizations for Visionary AI Systems Workshop at SIGMOD 2025*, 2025. [Online]. Available: https://openreview.net/forum?id=NZLm4vzXfm

[6] G. Katsogiannis-Meimarakis and G. Koutrika, "A survey on deep learning approaches for text-to-sql," *VLDB Journal*, vol. 32, no. 4, pp. 905 – 936, 2023.

[7] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, "Text-to-sql empowered by large language models: A benchmark evaluation," vol. 17, no. 5, Guangzhou, China, 2024, pp. 1132 – 1145.

[8] K. Maamari and A. Mhedhbi, "End-to-end text-to-sql generation within an analytics insight engine," 2024.

[9] F. Li and H. Jagadish, "Nalir: An interactive natural language interface for querying relational databases," Snowbird, UT, United states, 2014, pp. 709 – 712.

[10] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan, "ATHENA: an ontology-driven system for natural language querying over relational

data stores," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 1209–1220, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1209-saha.pdf

[11] B. Li, Y. Luo, C. Chai, G. Li, and N. Tang, "The dawn of natural language to SQL: are we fully ready?" *Proc. VLDB Endow.*, vol. 17, no. 11, pp. 3318–3331, 2024.

[12] C.-H. Lee, O. Polozov, and M. Richardson, "Kaggledbqa: Realistic evaluation of text-to-sql parsers," vol. 1, Virtual, Online, 2021, pp. 2261 – 2273.

[13] V. Zhong, C. Xiong, and R. Socher, "Seq2sql: Generating structured queries from natural language using reinforcement learning," 2017.

[14] H. Li, J. Zhang, C. Li, and H. Chen, "RESDSQL: decoupling schema linking and skeleton parsing for text-to-sql," in *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, B. Williams, Y. Chen, and J. Neville, Eds. AAAI Press, 2023, pp. 13 067–13 075. [Online]. Available: https://doi.org/10.1609/aaai.v37i11.26535

[15] S. Talaei, M. Pourreza, Y.-C. Chang, A. Mirhoseini, and A. Saberi, "Chess: Contextual harnessing for efficient sql synthesis," 2024.

[16] Y. Gao, Y. Liu, X. Li, X. Shi, Y. Zhu, Y. Wang, S. Li, W. Li, Y. Hong, Z. Luo, J. Gao, L. Mou, and Y. Li, "A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql," 2024. [Online]. Available: http://dx.doi.org/10.48550/arXiv.2411.08599

[17] M. Pourreza, H. Li, R. Sun, Y. Chung, S. Talaei, G. T. Kakkar, Y. Gan, A. Saberi, F. Ozcan, and S. O. Arik, "Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql," 2024.

[18] H. A. Caferolu and O. Ulusoy, "E-sql: Direct schema linking via question enrichment in text-to-sql," 2024.

[19] M. Pourreza and D. Rafiei, "Din-sql: Decomposed in-context learning of text-to-sql with self-correction," 2023.

[20] B. Wang, C. Ren, J. Yang, X. Liang, J. Bai, L. Chai, Z. Yan, Q.-W. Zhang, D. Yin, X. Sun, and Z. Li, "Mac-sql: A multi-agent collaborative framework for text-to-sql," 2025.

[21] G. Qu, J. Li, B. Li, B. Qin, N. Huo, C. Ma, and R. Cheng, "Before generation, align it! A novel and effective strategy for mitigating hallucinations in text-to-sql generation," in *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 5456–5471. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-acl.324

[22] L. Sheng and S. Xu, "CSC-SQL: corrective self-consistency in text-to-sql via reinforcement learning," *CoRR*, vol. abs/2505.13271, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.13271

[23] Y. D. Dönder, D. Hommel, A. W. Wen-Yi, D. Mimno, and U. E. S. Jo, "Cheaper, better, faster, stronger: Robust text-to-sql without chain-of-thought or fine-tuning," *CoRR*, vol. abs/2505.14174, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2505.14174

[24] D. Lee, C. Park, J. Kim, and H. Park, "MCS-SQL: leveraging multiple prompts and multiple-choice selection for text-to-sql generation," in *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, O. Rambow, L. Wanner, M. Apidianaki, H. Al-Khalifa, B. D. Eugenio, and S. Schockaert, Eds. Association for Computational Linguistics, 2025, pp. 337–353. [Online]. Available: https://aclanthology.org/2025.coling-main.24/

[25] X. Xie, G. Xu, L. Zhao, and R. Guo, "Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment," *Proc. ACM Manag. Data*, vol. 3, no. 3, pp. 194:1–194:24, 2025. [Online]. Available: https://doi.org/10.1145/3725331

[26] M. Malekpour, N. Shaheen, F. Khomh, and A. Mhedhbi, "Towards optimizing SQL generation via LLM routing," *CoRR*, vol. abs/2411.04319, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.04319

[27] K. Maamari, F. Abubaker, D. Jaroslawicz, and A. Mhedhbi, "The death of schema linking? text-to-sql in the age of well-reasoned language models," *CoRR*, vol. abs/2408.07702, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.07702

[28] M. Datar, P. Indyk, N. Immorlica, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," Brooklyn, NY, United states, 2004, pp. 253 – 262.

[29] Z. Cao, Y. Zheng, Z. Fan, X. Zhang, W. Chen, and X. Bai, "RSL-SQL: robust schema linking in text-to-sql generation," *CoRR*, vol. abs/2411.00073, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.00073

[30] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends in Information Retrieval*, vol. 3, no. 4, pp. 333 – 389, 2009.

[31] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," vol. 35, New Orleans, LA, United states, 2022.

[32] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," 2023.

[33] K. Maamari, C. Landy, and A. Mhedhbi, "Genedit: Compounding operators and continuous improvement to tackle text-to-sql in the enterprise," 2025. [Online]. Available: https://arxiv.org/abs/2503.21602

[34] C. Finegan-Dollak, J. K. Kummerfeld, L. Zhang, K. Ramanathan, S. Sadasivam, R. Zhang, and D. Radev, "Improving text-to-sql evaluation methodology," vol. 1, Melbourne, VIC, Australia, 2018, pp. 351 – 360.

[35] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Sqlizer: Query synthesis from natural language," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 2017.

[36] A. Mitsopoulou and G. Koutrika, "Analysis of text-to-SQL benchmarks: Limitations, challenges and opportunities," in *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2025, pp. 199–212.

[37] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," vol. 2020-December, Virtual, Online, 2020, pp. Apple; et al.; Microsoft; PDT Partners; Sony; Tenstorrent –.

[38] F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, W. Hu, P. Yin, V. Zhong, C. Xiong, R. Sun, Q. Liu, S. I. Wang, and T. Yu, "Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows," 2024.

[39] B. Team, "Livesqlbench: A dynamic and contamination-free benchmark for evaluating llms on real-world text-to-sql tasks," GitHub: https://github.com/bird-bench/livesqlbench, 2024, accessed 2025-05-22.

[40] C. Renggli, I. F. Ilyas, and T. Rekatsinas, "Fundamental challenges in evaluating text2sql solutions and detecting their limitations," 2025.

[41] M. Pourreza and D. Rafiei, "Evaluating cross-domain text-to-sql models and benchmarks," Hybrid, Singapore, Singapore, 2023, pp. 1601 – 1611.

[42] A. Floratou, F. Psallidas, F. Zhao, S. Deep, G. Hagleither, W. Tan, J. Cahoon, R. Alotaibi, J. Henkel, A. Singla, A. v. Grootel, B. Chow, K. Deng, K. Lin, M. Campos, V. Emani, V. Pandit, V. Shnayder, W. Wang, and C. Curino, "NL2SQL is a solved problem... not!" *CIDR*, 2024.

[43] A. Kumar, P. Nagarkar, P. Nalhe, and S. Vijayakumar, "Deep learning driven natural languages text to sql query conversion: A survey," 2022.

[44] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976. [Online]. Available: https://doi.org/10.1145/321921.321925

[45] X. Zhang, J. J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., 2015, pp. 649–657.

[46] J. W. Wei and K. Zou, "EDA: easy data augmentation techniques for boosting performance on text classification tasks," 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 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 6381–6387.

[47] R. Sennrich, B. Haddow, and A. Birch, "Improving neural machine translation models with monolingual data," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[48] S. Kobayashi, "Contextual augmentation: Data augmentation by words with paradigmatic relations," in *Proceedings of the 2018 Conference of the North American Chapter of*

the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers), M. A. Walker, H. Ji, and A. Stent, Eds. Association for Computational Linguistics, 2018, pp. 452–457.

[49] Z. Wu and M. S. Palmer, "Verb semantics and lexical selection," in *32nd Annual Meeting of the Association for Computational Linguistics, 27-30 June 1994, New Mexico State University, Las Cruces, New Mexico, USA, Proceedings*, J. Pustejovsky, Ed. Morgan Kaufmann Publishers / ACL, 1994, pp. 133–138. [Online]. Available: https://aclanthology.org/P94-1019/

[50] S. Y. Feng, V. Gangal, J. Wei, S. Chandar, S. Vosoughi, T. Mitamura, and E. H. Hovy, "A survey of data augmentation approaches for NLP," in *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021*, ser. Findings of ACL, C. Zong, F. Xia, W. Li, and R. Navigli, Eds., vol. ACL/IJCNLP 2021. Association for Computational Linguistics, 2021, pp. 968–988. [Online]. Available: https://doi.org/10.18653/v1/2021.findings-acl.84

APPENDIX A PROMPTS FOR JOIN QUERY EXPANSION

In this appendix, we enumerate the various prompts used as part of the query expansion and query augmentation techniques.

A.1 New Questions for Extended SQL

```
You are given a text-to-SQL example where a query has been extended by adding a
new table and its join conditions. Your task is to write a new natural language
question that corresponds to the updated SQL query.

For reference, you will be provided with the schema, the original SQL query,
its associated question, and the new SQL query:
### Schema:
{{ schema }}

### Old query:
{{ old_query }}
### Old question:
{{ old_question }}

### New query:
{{ new_query }}
### New question:

### Instructions:
- Write a single-line question (no newlines) that accurately reflects the new
SQL query.
- Return only the question and do not include explanations or commentary.
```

A.2 New Questions for New SQL

```
Your task is to generate a natural language question from the given SQL query,
following the format of a dev set question in a Text-to-SQL benchmark as the
examples below. Use the provided schema provided below to better understand the
relationships in the database. Work through the reasoning behind the question
to ensure it aligns with the structure of the SQL query, and use the examples
provided to guide your understanding of the question format.

### Chain of Thought (CoT):
1. **Understand the SQL query**: Analyze the SQL query and identify the key
entities (tables, columns) and operations (joins, filters, aggregations). What
is the query trying to extract from the database? For example, in the SQL query
"SELECT COUNT(T1.Id) FROM users AS T1...," the focus is on counting the number
of posts made by a user, which can help form a "How many" style question.

2. **Break down the question**: What specific details or relationships from the
SQL query should be captured in the question? For instance, in the query
"SELECT T1.client_id FROM client AS T1...", the question needs to reflect the
user (client) and the condition of being female, and it asks "Who are the
female account holders who own credit cards and also have loans?"

3. **Match the schema**: Refer to the schema to understand the relationships
between tables and their attributes. For example, in the query "SELECT DISTINCT
t4.name FROM Player_Attributes AS t1...," the schema tells us that "name" is an
attribute of "Country," and this helps form the question around the country of
the players with certain attributes.

4. **Formulate the question**: Based on the analysis, generate a clear, concise
question that corresponds to the SQL query. Ensure the question is structured
similarly to the examples provided. For instance, if the SQL query is about
counting or summarizing certain data (like the "wins" in the race query), the
question might follow the format of "How many times did [person] do [action]?"


### Schema:
{{ schema }}

#### Examples:
```

```
SQL query:
SELECT COUNT(T1.Id) FROM users AS T1
  JOIN postHistory AS T2 ON T1.Id = T2.UserId
  JOIN posts AS T3 ON T2.PostId = T3.Id
  JOIN votes AS T4 ON T4.PostId = T3.Id
 WHERE T1.DisplayName = "Matt Parker"
 GROUP BY T2.PostId, T4.Id HAVING COUNT(T4.Id) > 4
Question: How many posts by Matt Parker have more than 4 votes?

SQL query:
SELECT T1.client_id FROM client AS T1 INNER JOIN disp AS T2 ON T1.client_id =
T2.client_id
  INNER JOIN account AS T5 ON T2.account_id = T5.account_id
  INNER JOIN loan AS T3 ON T5.account_id = T3.account_id
  INNER JOIN card AS T4 ON T2.disp_id = T4.disp_id
 WHERE T1.gender = 'F'
Question: Who are the female account holders who own credit cards and also have
loans?

SQL query:
SELECT DISTINCT t4.name FROM Player_Attributes AS t1
  INNER JOIN Player AS t2 ON t1.player_api_id = t2.player_api_id
  INNER JOIN Match AS t3 ON t2.player_api_id = t3.home_player_8
  INNER JOIN Country AS t4 ON t3.country_id = t4.id
 WHERE t1.vision > 89
Question: Locate players with vision scores of 90 and above, state the country
of these players.

SQL query:
SELECT SUM(T2.wins) FROM drivers AS T1
  INNER JOIN driverStandings AS T2 ON T2.driverId = T1.driverId
  INNER JOIN races AS T3 ON T3.raceId = T2.raceId
  INNER JOIN circuits AS T4 ON T4.circuitId = T3.circuitId
 WHERE T1.forename = 'Michael'
 AND T1.surname = 'Schumacher'
 AND T4.name = 'Sepang International Circuit'
Question: How many times did Michael Schumacher won from races hosted in Sepang
International Circuit?
```

```
SQL query:
SELECT T2.publisher_name FROM superhero AS T1
  INNER JOIN publisher AS T2 ON T1.publisher_id = T2.id
  INNER JOIN hero_attribute AS T3 ON T1.id = T3.hero_id
  INNER JOIN attribute AS T4 ON T3.attribute_id = T4.id
 WHERE T4.attribute_name = 'Speed'
 ORDER BY T3.attribute_value
 LIMIT 1
Question: Which publisher published the slowest superhero?

SQL query:
SELECT T2.forename, T2.surname FROM qualifying AS T1
  INNER JOIN drivers AS T2 on T1.driverId = T2.driverId
  INNER JOIN races AS T3 ON T1.raceid = T3.raceid
 WHERE q3 IS NOT NULL AND T3.year = 2008
 AND T3.circuitId IN (
    SELECT circuitId FROM circuits
    WHERE name = 'Marina Bay Street Circuit'
    )
 ORDER BY CAST(SUBSTR(q3, 1, INSTR(q3, ':') - 1) AS INTEGER) * 60 +
 CAST(SUBSTR(q3, INSTR(q3, ':') + 1, INSTR(q3, '.') - INSTR(q3, ':') - 1) AS
 REAL) + CAST(SUBSTR(q3, INSTR(q3, '.') + 1) AS REAL) / 1000 ASC
 LIMIT 1

Question: What is full name of the racer who ranked 1st in the 3rd qualifying
race held in the Marina Bay Street Circuit in 2008

### Given the SQL query:
SQL query: {{ main_query }}

### Instructions:
1. Follow the Chain of Thought reasoning steps to generate a question that
matches the SQL query.
2. Ensure that the question matches the style and format of the examples above.
It should be clear, concise, and related to the SQL query.
3. The question should be formatted as a single line with no additional
explanations.
```

```
4. Refer to the schema to better understand the relationships in the database.
Make sure the question aligns with the schema and captures the important
details of the query.
```

A.3 New Evidence for New SQL

```
Your task is to generate **evidence** for the given SQL query and question.

The evidence refers to external knowledge needed to help map the **natural
language question** to its corresponding **SQL query**, especially for values
in filtering conditions and the meaning of projected values (i.e., what is
being selected). This is useful when such details are **implicit** or not
clearly stated in the question.

When writing evidence:
- Focus only on filtering conditions and projections.
- You may mention column names, relevant values, and the names of the tables to
which those columns belongbut only if they are involved in filtering or
projection, not joins.
For example, consider the question: Which accounts are held by male clients who
have loans, placed orders, and possess credit cards?
The correct evidence should be: Male clients refer to gender = 'M'; accounts
refers to account.account_id.
There is no need to mention tables involved in join conditions, such as:
accounts with loans, orders, and credit cards involve account_id in account,
loan, "order", and card tables.  this would be incorrect.
- Do not mention any join conditions or join keys used in the SQL query (e.g.,
`Tab1.id1 = Tab2.id2`) or refer to the keys themselves (e.g., `id1` as a join
key).
- Your output must be **a single, concise line** starting with `Evidence:` and
must contain **no additional explanation**.

### Schema:
{{ schema }}

### Examples:
SQL query:
SELECT COUNT(T1.Id) FROM users AS T1
  INNER JOIN postHistory AS T2 ON T1.Id = T2.UserId
  INNER JOIN posts AS T3 ON T2.PostId = T3.Id
  INNER JOIN votes AS T4 ON T4.PostId = T3.Id
 WHERE T1.DisplayName = 'Matt Parker'
 GROUP BY T2.PostId, T4.Id
```

```
 HAVING COUNT(T4.Id) > 4
Question: How many posts by Matt Parker have more than 4 votes?
Evidence: more than 4 votes refer to PostId > 4; DisplayName = 'Matt Parker'

SQL query:
SELECT T1.client_id FROM client AS T1
  INNER JOIN disp AS T2 ON T1.client_id = T2.client_id
  INNER JOIN account AS T5 ON T2.account_id = T5.account_id
  INNER JOIN loan AS T3 ON T5.account_id = T3.account_id
  INNER JOIN card AS T4 ON T2.disp_id = T4.disp_id
 WHERE T1.gender = 'F'
Question: Who are the female account holders who own credit cards and also have
loans?
Evidence: Female refers to gender = 'F'

SQL query:
SELECT DISTINCT t4.name FROM Player_Attributes AS t1
  INNER JOIN Player AS t2 ON t1.player_api_id = t2.player_api_id
  INNER JOIN Match AS t3 ON t2.player_api_id = t3.home_player_8
  INNER JOIN Country AS t4 ON t3.country_id = t4.id
 WHERE t1.vision > 89
Question: Locate players with vision scores of 90 and above, state the country
of these players.
Evidence: vision scores of 90 and above refers to vision > 89

SQL query:
SELECT SUM(T2.wins) FROM drivers AS T1
  INNER JOIN driverStandings AS T2 ON T2.driverId = T1.driverId
  INNER JOIN races AS T3 ON T3.raceId = T2.raceId
  INNER JOIN circuits AS T4 ON T4.circuitId = T3.circuitId
 WHERE T1.forename = 'Michael' AND T1.surname = 'Schumacher'
 AND T4.name = 'Sepang International Circuit'
Question: How many times did Michael Schumacher won from races hosted in Sepang
International Circuit?
Evidence: win from races refers to max(points)

SQL query:
SELECT T2.publisher_name FROM superhero AS T1
```

```
  INNER JOIN publisher AS T2 ON T1.publisher_id = T2.id
  INNER JOIN hero_attribute AS T3 ON T1.id = T3.hero_id
  INNER JOIN attribute AS T4 ON T3.attribute_id = T4.id
 WHERE T4.attribute_name = 'Speed'
 ORDER BY T3.attribute_value
 LIMIT 1
Question: Which publisher published the slowest superhero?
Evidence: the slowest superhero refers to attribute_name = 'Speed' where
MIN(attribute_value); publisher refers to publisher_name

### Now for the following SQL query and the question:
**SQL query:**
{{ main_query }}
**Question:**
{{ question }}

**Instructions:**
- Generate a **single-line evidence** starting with ‘Evidence:‘ based on the
query above.
- **Do not include anything else**.
```

APPENDIX B PROMPTS FOR QUERY AUGMENTATION

B.1 Synonym Replacement

```
You are a **synonym generator**.

## TASK:
Given a list of tokens with their positions, return a new list where each token
is replaced by a synonym.

### FORMAT:
Input is a list of dictionaries which includes tokens and a table name:
Tokens:
[
  {"token": "word1", "position": position1},
  {"token": "word2", "position": position2},
  ...
]
Table name: table1
Database name: db_id1

Output should match this format **as a raw string** (no markdown blocks, no
headings, no commentary):
[
  {"token": "synonym1", "position": position1},
  {"token": "synonym2", "position": position2},
  ...
]



### RULES:
0. The synonym **must not appear as a substring** in any of the segments of the
table name (case-insensitive).
1. The synonym must be contextually aligned with the database name. If not
possible, suggest a generic synonym.
2. The synonym must not be equivalent to any of the original tokens. Your
performance will be evaluated based on whether you generate **new synonyms**
```

```
that do **not** appear among the input tokens.
  - For example: if the input tokens are [{"token": "start", "position": 5},
  {"token": "begin", "position": 25}], then for the second token "begin", you
  **should not** suggest "start" as a synonym, because it already appears in
  the input.
3. The synonym must not be the full form of an abbreviation, if the token
itself is an abbreviation.
  - For example: if the input token is "msg", *do not use* "message" as a
  synonym.
4. The synonym must not be a morphological variant that is too similar to the
original token.
  - For example: if the input is "north", *do not suggest* "northern" *as a
  synonym; if the input is "nation", *do not use* "national" *as a synonym.
5. The synonym must have the **same grammatical category** as the original
token (e.g., noun, verb, adjective).
6. The synonym must preserve the **grammatical form**:
   - Verbs -> same tense (e.g., present/past/gerund).
   - Nouns -> same number (singular/plural).
   - Adjectives -> same degree (positive/comparative/superlative).
7. The synonym **must NOT include the original token as a substring**
(case-insensitive).
8. The **position value must stay the same**.
9. One synonym per token.
10. The **output must be a valid JSON-like string**. No markdown formatting, no
extra explanations.

---

### INPUT:
Tokens:
{{ tokens }}
Table name:
{{ table_name }}
Database name:
{{ db_id }}
---

### STEP-BY-STEP:
```

```
{% for item in tokens %}
Token: "{{ item.token }}", Position: {{ item.position }}
-> Identify the part-of-speech and grammatical form for "{{ item.token }}"
-> Generate a synonym that:
   - Has the same POS and grammatical form
   - Does **not** contain "{{ item.token }}" as a substring
   - Does **not** appear as a substring in any segment of the table name "{{
   table_name }}"
-> Final: {"token": "synonym", "position": {{ item.position }}}
{% endfor %}

---

### FINAL OUTPUT:
[
{% for item in tokens %}
  {"token": "{{ "SYNONYM_" + loop.index|string }}", "position": {{
  item.position }}}{% if not loop.last %},{% endif %}
{% endfor %}
]
```

B.2 Backtranslation

```
{% if source_type == "question" %}
Let's think step by step. You are working in a text-to-SQL context, where a
natural language question is intended to be answered by a SQL query over a
given database schema.
{% elif source_type == "evidence" %}
Let's think step by step. You are working in a text-to-SQL context, where an
external evidence passage may provide background knowledge necessary to
interpret or support the execution of the SQL query.
{% endif %}

In this task, you are provided with:
- A central table and its components (i.e., the words or abbreviations that
make up the table name).
- A {{ source_type }} text that may contain references to those components.

Each component is represented as:

components = [
  {
    "segment": "comp1",
    "is_abbreviation": True,
    "matched_tokens": [
      {"token": "word1", "position": position1}
    ]
  },
  {
    "segment": "comp2",
    "is_abbreviation": False,
    "matched_tokens": [
      {"token": "word2", "position": position2}
    ]
  },
  ...
]

Where:
```

```
- ‘segment‘ is a part of the central table name (can be a word or an
abbreviation).
- ‘is_abbreviation‘ indicates whether the segment is an abbreviation.
- ‘matched_tokens‘ lists the tokens from the {{ source_type }} that match this
segment, along with their positions.

---

### Your Goal:
Apply **Back Translation (BT)** to obfuscate direct references to the central
table by:
1. Translating the original {{ source_type }} into **French**.
2. In the French version, replacing each matched token with a **French
synonym** that:
   - Preserves the meaning of the original token.
   - Does **not** translate back to the same token in English.
3. Translating the modified French sentence back to **English**.

---

### Instructions:
0. If the {{ source_type }} is empty, return an empty string. If the list of
components is empty, return the {{ source_type }} as is, without explanations
or headings.
1. Translate the full {{ source_type }} into French.
2. For each component in ‘components‘, iterate over its ‘matched_tokens‘ list.
3. For each token:
   - Locate the token in the translated French sentence.
   - Replace it with a **French synonym** that:
     - Preserves the meaning in context.
     - Will not translate back to the original English token.
     - Do not replace the token with any word that appears in the original
     component’s segment field. For example, if the segment is "wooden" and the
     token to change is "timber", do not replace "timber" with "wooden".
4. Translate the modified French sentence back into English.
5. Ensure the final English version:
   - Keeps the original sentence structure and meaning.
   - Obfuscates all references to the central table components.
```

```
{% if source_type == "evidence" %}
6. DO NOT CHANGE tokens used in the SQL query as a **value** (e.g., inside
WHERE) or as an **attribute/column name** - **do not replace them**.
{% endif %}
{% if source_type in ["evidence", "question"] %}
{% if source_type == "question" %}6{% else %}7{% endif %}. Avoid trivial
changes. Substitutions should be **meaningful and non-obvious**.
   - For example, if the component is `"chair"`, do **not** use `"wooden
   chair"` or `"chairs"` - the new word must be **semantically distinct and
   non-derivative**.
{% if source_type == "question" %}7{% else %}8{% endif %}. Do **not** include
the French or intermediate outputs.
{% if source_type == "question" %}8{% else %}9{% endif %}. Output only the
final rewritten {{ source_type }} as a single line, with no commentary or
formatting.
{% endif %}
--


### Input:
- Database ID: {{ db_id }}
- {{ source_type|capitalize }}: {{ source }}
- SQL Query: {{ sql_query }}
- Candidate Table: {{ candidate_table }}
- Components: {{ components }}
{% if source_type == "evidence" %}
- New Question: {{ new_question }}
{% endif %}

---

### Output Format:
New {{ source_type }}: rewritten_{{ source_type }}
```

B.3 Contextual Augmentation

```
{% if source_type == "question" %}
Let's think step by step. You are working in a text-to-SQL context, where a
natural language question is intended to be answered by a SQL query over a
given database schema.
{% elif source_type == "evidence" %}
Let's think step by step. You are working in a text-to-SQL context, where an
external evidence passage may provide background knowledge necessary to
interpret or support the execution of the SQL query.
{% endif %}

In this task, you are provided with:
- A central table and its components (i.e., the words or abbreviations that
make up the table name).
- A {{ source_type }} text which may contain words that correspond to these
components.

Each component is represented as:

components = [
  {
    "segment": "comp1",
    "is_abbreviation": True,
    "matched_tokens": [
      {"token": "word1", "position": position1}
    ]
  },
  {
    "segment": "comp2",
    "is_abbreviation": False,
    "matched_tokens": [
      {"token": "word2", "position": position2}
    ]
  },
  ...
]

Where:
```

```
- ‘segment‘ is a part of the central table name (can be a word or an
abbreviation).
- ‘is_abbreviation‘ indicates whether the segment is an abbreviation.
- ‘matched_tokens‘ lists the tokens from the {{ source_type }} that match this
segment, along with their positions.

---

### Your Goal:
Perform **Contextual Augmentation (CA)** to rewrite the {{ source_type }}. This
involves:
- Masking each matched token using its position.
- Thinking like a contextual language model (e.g., BERT or T5) to predict a
grammatically and semantically valid replacement that **excludes** the original
token.

---

### Instructions:
0. If the {{ source_type }} is empty, return an empty string. If the list of
components is empty, return the {{ source_type }} as is, without explanations
or headings.
1. Iterate through the list of components.
2. For each ‘matched_token‘:
   - Use its ‘position‘ to locate the token in the original {{ source_type }}.
   - Mask only the exact matched token (do not mask phrases or overlapping
   spans).
3. For each masked span, generate a replacement using the surrounding context.
4. Ensure the replacement is grammatically correct and contextually appropriate.
5. Do **not** reuse the original token or include it as a substring.
{% if source_type == "evidence" %}
6. DO NOT CHANGE tokens used in the SQL query as a **value** (e.g., inside
WHERE) or as an **attribute/column name** - **do not replace them**.
{% endif %}

{% if source_type in ["evidence", "question"] %}
{% if source_type == "evidence" %}7{% else %}6{% endif %}. Do **not** replace
the token with any word that appears in the original component’s segment field
```

```
either. For example, if the segment is "wooden" and the token to change is
"timber", do not replace "timber" with "wooden".
{% if source_type == "evidence" %}8{% else %}7{% endif %}. Avoid trivial edits
- replacements must reflect genuine augmentation.
   - For example, if the component is `"chair"`, do **not** substitute it with
   `"wooden chair"` or `"chairs"` - the new word must **not** contain or derive
   from the original.
{% if source_type == "evidence" %}9{% else %}8{% endif %}. Do **not** add
formatting like asterisks, quotes, or markdown.
{% if source_type == "evidence" %}10{% else %}9{% endif %}. Output only the
rewritten {{ source_type }} as a single sentence. No explanations, no headings.
{% endif %}

--

### Input:
- Database ID: {{ db_id }}
- {{ source_type|capitalize }}: {{ source }}
- SQL Query: {{ sql_query }}
- Candidate Table: {{ candidate_table }}
- Components: {{ components }}
{% if source_type == "evidence" %}
- New Question: {{ new_question }}
{% endif %}

---

### Output Format:
New {{ source_type }}: rewritten_{{ source_type }}
```