TU WIEN Informatics

Large Language Model-Powered Query Answering

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Vladimir Panin, B.Sc.
Matrikelnummer 12238682

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Emanuel Sallinger
Mitwirkung: Dr. Eleonora Laurenza

Wien, 5. Mai 2025

Vladimir Panin	Emanuel Sallinger

TU WIEN Informatics

Large Language Model-Powered Query Answering

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Vladimir Panin, B.Sc.
Registration Number 12238682

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Emanuel Sallinger
Assistance: Dr. Eleonora Laurenza

Vienna, May 5, 2025

Vladimir Panin	Emanuel Sallinger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Vladimir Panin, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 5. Mai 2025

Vladimir Panin

Danksagung

Mein herzlicher Dank gilt meinen Betreuern Emanuel Sallinger und Eleonora Laurenza, die mich mit großer Unterstützung und wertvollen Anregungen während der Entstehung dieser Arbeit begleitet haben. Emanuels konstruktive Rückmeldungen und seine breite Expertise haben wesentlich zur Qualität der Arbeit beigetragen, während Eleonora durch zahlreiche hilfreiche Gespräche und konkrete Vorschläge wichtige Impulse gegeben hat. Beide standen mir mit Rat und Motivation zur Seite, wofür ich sehr dankbar bin.

Acknowledgements

My deepest gratitude goes to my supervisors Emanuel Sallinger and Eleonora Laurenza, who supported me in an excellent way. Emanuel's feedback and extensive experience has considerably improved the quality of this thesis, while Eleonora directly had part in its creation by devoting much of her time to fruitful discussions and constant ideas for improvement. Their advice and mentoring has been very important to me.

Kurzfassung

Die jüngsten Fortschritte im Bereich großer Sprachmodelle, Large Language Models (LLMs), haben deren herausragende Fähigkeiten im Verstehen, Interpretieren und Generieren von Sprache eindrucksvoll unter Beweis gestellt. Beim Abfragen von Datenbanken in domänenspezifische Sprachen wie der Structured Query Language (SQL), die speziell für relationale Datenbanken entwickelt wurden, können Fehler auftreten, die durch Probleme wie Rauschen in den Daten, inkonsistente Formate oder fehlende Standardisierung bedingt sind. LLMs bieten das Potenzial, diese Herausforderungen zu überwinden. In dieser Arbeit werden mehrere Pipelines vorgeschlagen, die die Interpretationsfähigkeit natürlicher Sprache durch LLMs nutzen, um den ursprünglichen Query zu modifizieren und somit das Ergebnis zu verändern. Dies wird erreicht, indem die ursprüngliche Abfrage angepasst und bei Bedarf zusätzliche Übersetzungstabellen integriert werden.

Zunächst entwerfen wir einen ersten Testdatensatz, der potenzielle Herausforderungen exemplarisch darstellt und als Grundlage für nachfolgende Experimente dient. Die Experimente werden mithilfe dieser Pipelines durchgeführt und demonstrieren deren Leistungsfähigkeit im Umgang mit einer Vielzahl von SQL-Vergleichsoperatoren. Darüber hinaus werden die vorgeschlagenen Methoden mit dem traditionellen Ansatz verglichen, bei dem ausschließlich die ursprüngliche, unveränderte Abfrage ausgeführt wird. Dieser Vergleich zeigt eine deutliche Verbesserung zentraler Leistungskennzahlen um bis zu 50% zugunsten der entwickelten Pipeline.

Zudem identifizieren wir potenzielle Fehlerquellen und zeigen, dass Abweichungen von korrekten Ergebnissen in mehreren Phasen der Pipeline auftreten können, insbesondere beim semantischen Vergleich sowie bei der Modifikation der Abfragen. Abschließend evaluieren wir den vorgeschlagenen Ansatz anhand zweier größerer Testdatensätze. Die Ergebnisse belegen eine gute Leistung hinsichtlich dieser wichtigen Metriken. Darüber hinaus werden die drei implementierten Ansätze—Zero-Shot Prompting, Embedding-Only und Two-Step—miteinander verglichen, wobei sowohl Leistungsunterschiede als auch potenzielle Zielkonflikte hinsichtlich Genauigkeit und Ausführungszeit herausgearbeitet werden.

Schließlich fassen wir die zentralen Erkenntnisse dieser Arbeit zusammen und skizzieren vielversprechende Ansätze für zukünftige Forschung in diesem Bereich.

Abstract

Large Language Models (LLMs) have shown remarkable capabilities in understanding and generating natural language. However, when querying databases in domain-specific languages, such as the Structured Query Language (SQL), designed specifically for relational databases, errors can arise due to issues such as noisy data, inconsistent formats, or lack of standardization. LLMs have the potential to mitigate these challenges. This work proposes several query evaluation pipelines that leverage the semantic strengths of LLMs to modify query outputs to account for residual noise. This is accomplished by modifying the original query and, when applicable, incorporating additional translation tables to enhance accuracy.

First, we design an initial test dataset that exemplifies potential issues, which will serve as a foundation for experimentation. Experiments are conducted using a LLM, showcasing its capability across a range of SQL comparison operators. Additionally, these methods are evaluated against the traditional baseline of executing the original, unmodified query, revealing a strong improvement in key performance metrics.

We also identify potential sources of errors and show that deviations from correct results can occur at various stages of the chosen pipeline, including the semantic comparison and the modification of the queries. Finally, we evaluate the proposed approach on two larger datasets. The results indicate good performance across key performance indicators. Moreover, the evaluation compares the three implemented approaches—zero-shot prompting, embedding-only, and two-step—and highlights their performance differences, including potential trade-offs in terms of both key performance metrics and execution efficiency.

In conclusion, we synthesize the key findings of this work and outline promising directions for future research in the domain.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In many practical use cases, data gathered from diverse sources, such as web crawls, may contain inconsistencies or noise [DBES09]. In the domain of Natural Language Processing (NLP) some common noise includes misaligned sentences, misordered words, wrong language, untranslated sentences and short segments [KK18]. Consequently, such incomplete, inconsistent, or erroneous data contained in a database may result in queries yielding unintended outcomes, even when the queries themselves are syntactically and logically correct.

Let us consider the exemplary scenario in which an original user is querying a database for all dogs in a potential column of animals.as shown in Table 1.1. However, in the creation of the database, datasets from different sources could have been combined together without accounting for different languages. Hence, a potential query written in English and filtering for "dog" might miss all additional records written in another language. Therefore, the resulting output might not be complete and not aligned with user intent, who wanted to retrieve all records associated with a dog.

Table 1.1: Language Inconsistency in Category Column

animalname	**category**	**owner_id**
bill	chien	1
diego	chat	2
chris	dog	3
juan	perro	4

1.2 Problem Statement

Traditional database systems store structured information within predefined sachems, enabling efficient querying and data retrieval. These systems operate according to formalized rules and languages designed for database querying, such as the Structured Query Language (SQL), ensuring deterministic and logically consistent results [KE09]. Consequently, when confronted with ambiguous or imprecise data, traditional databases may return incomplete results or fail to retrieve relevant information, limiting their adaptability in real-world applications [ZP97].

Additionally, Large Language Models (LLMs) have undergone significant progress in recent years. LLMs are trained on massive datasets spanning different data formats, learning to predict the next token in a sequence based on the context of preceding tokens. Their ability to comprehend, interpret, and generate various forms of language is currently influencing numerous fields in remarkable ways [MMN+24]. The output of a LLM is also highly dependent on the input prompt, giving rise to the field of prompt engineering [SSS+24].

In order to combine the strengths of symbolic systems and statistical learning models such as LLMs, hybrid reasoning approaches have emerged, resulting in the field of neurosymbolic AI [GL20].

1.2.1 Research Gap

LLMs have shown considerable strengths in natural language understanding, contextual interpretation, and multilingual processing [MMN+24]. Despite these advances, their potential in query processing, particularly for improving the correctness of query outputs in cases of residual noise, such the example scenario outlined in the Section 1.1, remains underexplored. Systems like Chat-DB [HFD+23] or Dater[YHY+23] enhance the reasoning capabilities of a LLM on a database by facilitating intermediate interactions via queries to the database. However, to the best of our knowledge, there is no existing pipeline that leverages a LLM in a neurosymbolic reasoning setting to resolve noise on a data level. Such a neurosymbolic reasoner is illustrated in Figure 1.1. This a pipeline would aim to match entities based on their semantic meaning, rather than relying on traditional syntactic matching in SQL queries. This gap in the current research highlights the opportunity to enhance SQL query processing by aligning user intent with more accurate, semantically-informed data retrieval. This work aims to address the issue of noise leverage both a LLM and SQL to design and implement such a pipeline for query processing, addressing the gap in current research.

1.3 Research Questions

Based on the stated research gap we formulate the resulting research questions. In the following, we refer to token cost as the number of tokens processed across all LLM inputs, outputs, and function calls, which directly impacts computational cost and latency. We

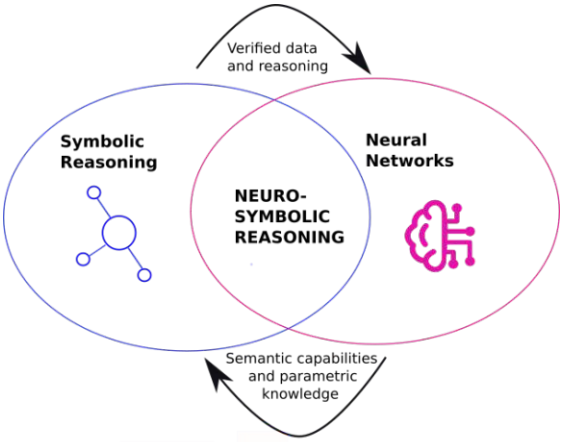


Figure 1.1: Neurosymbolic Reasoner

refer to soft binding as a binding procedure designed for query answering, where variables are bound based on their semantic meaning rather than strict syntactic matching, utilizing a LLM. This contrasts with hard binding, where variables are only linked when there is an exact, literal match. The relevant Key Performance Indicators examined in this work are precision, recall and F1-score. Going on, the research questions guiding this thesis are the following.

- **Research Question 1:** How do LLM-based *softening* techniques perform in terms of relevant KPIs compared to approaches relying on strict binding?
- **Research Question 2:** What is the computational and token cost of the LLM-integrated pipeline compared to a approach with strict binding and how does this affect the query execution time?
- **Research Question 3:** How do these *softening* techniques perform for relevant KPIs as well as computational cost when applied to pre-existing datasets for entity matching and classification?

1.4 Main Contributions

The challenge addressed in this work is to leverage the semantic capabilities of a LLM to return intended query results in the presence of imperfect or inconsistent data. This thesis investigates LLM-powered query answering by integrating the semantic understanding of LLM into SQL-based data retrieval. To achieve this, we develop a query processing pipeline that builds on the parametric knowledge embedded within a LLM, enabling robust handling of unstructured, ambiguous, or incomplete data during query execution.

For this purpose, a binding procedure based on the concept of soft binding [BBB+24], tailored for query answering, is designed, implemented, and evaluated. This procedure utilizes a LLM to bind variables based on their semantic meaning, rather than syntactic matching - resulting in the soft binding procedure of the pipeline.

The main contributions of the presented work are the following:

1. This thesis presents a method for translating predicate calculus expressions into executable SQL queries, enabling noise-aware query processing over databases. The proposed approach leverages the semantic capabilities of a LLM to address residual noise and enhance the robustness of the query execution process.
2. A test dataset designed for development and evaluation, incorporating various forms of residual noise. The performance of the pipeline is partly assessed using this dataset.
3. A comprehensive analysis is conducted on two external datasets, considering precision, recall, F1-score, computational cost, and BLEU metrics to evaluate the pipeline's effectiveness.

1.5 Methodology

In this work, we adopt Design Science as the methodology to guide both the research process and the development of artifacts, with the aim of addressing the identified problem and achieving the outlined objectives.

1. **Literature Review** This chapter provides a literature review covering reasoning, neurosymbolic integration of LLMs with Knowledge Graphs (KGs), LLM interaction with structured data, prompting techniques, Text-to-SQL approaches, query expansion, data integration, entity matching, and blocking techniques.
2. **Pipeline Design** In a next step, we discuss and implement a query processing pipeline designed to mitigate the impact of data noise on query results in relational databases. The pipeline employs a LLM to analyze database content, identify noise sources, and refine query outputs. We create two different types of pipelines tailored towards different use-cases.
3. **Evaluation** Going on, we outline initial considerations, the KPIs used and explain the design of the initial test dataset. In a next step, we evaluate the effectiveness of the pipeline leveraging a product classification and an entity matching dataset.
4. **Limitations and Discussion** Next, we address the limitations of the approach and conduct the discussion.
5. **Conclusion and Future Work** In a final step, we draw the conclusion and also suggests future directions of research.

1.6 Structure of the Thesis

The first chapter introduces the research topic, outlining its significance and objectives. A comprehensive review of the state of the art follows in Chapter 2, examining existing methodologies and relevant approaches in the field. Next, Chapter 3 explores the implemented pipeline in detail, discussing its design and functionality. The creation of a test dataset is the focus of Chapter 4, along with the identification of cases where adjustments to the binding procedure are necessary. Additionally, this chapter evaluates the implemented pipeline on the test dataset and two external datasets, assessing the results through different metrics. Limitations of the chosen approaches are explored in Chapter 5, accompanied by a critical discussion of the findings. Finally, Chapter 6 concludes the study by summarizing key insights and offering perspectives for future research.

CHAPTER 2

Literature Review

The following chapter gives an overview of the related work to this thesis. First of all, the field of reasoning is touched upon in Chapter 2.1. Chapter 2.2 discusses the different neurosymbolic approaches of integrating LLMs with KGs. The interaction of a LLM with structured data, Text-to-SQL approaches and prompting techniques are evaluated in Chapter 2.3. Next, Chapter 2.4 focuses on query expansion, while Chapter 2.5 elaborates on data integration, entity matching, and blocking.

2.1 Reasoning and Soft-Chase Algorithm

The development of Knowledge Graphs (KGs) is linked to the necessity of organizing data and knowledge into a structured format. It was defined by Marchi et al. [MM74] in 1974 as "A mathematical structure with vertices as knowledge units connected by edges that represent the prerequisite relation".

Furthermore, it is possible to define rules inside of the KG. Often used in the context of KGs are ontologies, which are defined by Studer et al. as a "formal, explicit specification of a shared conceptualization" [SBF98]. They allow to organize the entities inside a KG into different concepts with properties alongside explicitly defined rules. This structure paves the road for reasoning with knowledge and generating new facts.

The combination of data D with definitions of a set of rules Σ in declarative language like for example *Vadalog* [BSG18] in a KG results in a Knowledge Representation and Reasoning (KRR) system [Bra88]. Their advantage is the ability to formally represent and reason over both explicit data and implicit knowledge encoded in the rules. These systems have been applied in various fields, for example in finance, in order to reason about company takeovers [BBB+22] or in the field of Genomics Problems to understand the

evolution of SARS-CoV-2 variants [ABBC22]. In rule-based languages such as Vadalog, the application of logical rules is often operationalized using the chase algorithm. At its core it enriches the database by applying a set of rules Σ to the available data D to enrich the data. This process involves applying dependencies, such as functional dependencies (FDs) or join dependencies (JDs), until a fix-point is reached[AHV95]. Due to the advancement of Artificial Intelligence (AI) there have been a variety of neurosymbolic systems, combining KRR systems with AI [GL20]. One area of research within this context is the extension of the chase algorithm such as the **Soft Chase** 2.1 Algorithm [BBB$^+$24]. It introduces a neurosymbolic approach, combining the robustness of KRR systems with the flexibility of LLMs to enhance the adaptability of ontological reasoning. The innovation of the soft chase particular lies in the identification of bindings through a LLM based on natural language input facts and their verification. In the Soft Chase Algorithm 2.1, natural language statements are converted into potential mappings in line 6, which are subsequently validated in line 12. Thus, unstructured data and external data can be incorporated into the processing pipeline without abandoning the reasoning engine. Consequently, the result is more interpretable and is less prone to hallucinations [XJK24, HYM$^+$24], particularly in comparison to approaches that rely solely on the LLM without the support of a structured reasoning component.

2.2 LLMs and Knowledge Graphs

Several methods exist for integrating LLMs with KGs. Generally three different approaches can be identified: KG-enhanced LLMs , where KGs can be used for pre-training [SMH$^+$20] or to inject knowledge [YZD$^+$22], LLM-augmented KGs, [KPGM20], where a LLM can be used to populate a KG, and Synergized LLMs and KGs, where KGs can be used to inject knowledge as context to a LLM [DHX$^+$22].

To improve the performance of a LLM or a Pretrained Language Model (PLM) on domain-specific tasks, such as Question Answering (QA), it is essential to incorporate structured knowledge. One common source of such knowledge is a KG. In LLMs the deeper layers encode the local syntax and semantic meaning, while higher layers focus more on the complex semantic relationships [TDP19, CKLM19]. Therefore, frequently fine-tuning approaches are used to infuse domain-specific knowledge.

For QA based on text, a temporary KG can be constructed from the text, allowing relevant facts to be retrieved for specific questions. This enables the construction of information-rich prompts, mitigating the limitations of finite context windows in Large Language Models (LLMs) [ANC$^+$22]. The SKILL method [MDAJ22] enhances closed-book QA by pre-training T5 [RSR$^+$19] directly on knowledge graph triples, achieving performance comparable to models trained on natural language sentences. Building on the SKILL method, the KITLM framework [AGAB23] makes use of a T5-pretrained model for multi-hop question answering using pre-existing triples from a KG. Also, fine-tuning a LLM can be performed on a database leveraging the database itself, the intensional definitions and the domain glossary [BBC$^+$23]. By the help of an ontological

Algorithm 2.1: Soft Chase Procedure

Input : $D, \Sigma, G,$ model
Output : $\Sigma(D)$

$\sigma(D) \leftarrow D$
while *VADALOG.hasNext()* **do**
  $\sigma, i \leftarrow$ VADALOG.next();
  $i_{\text{mappings}}, \text{attempts} \leftarrow \varnothing, 0;$
  if $\mathit{LINEAR}(\sigma)$ **then**
    $i_{\text{mappings}} \leftarrow \text{model.bindLinear}(\sigma, i);$
  end
  else if $\mathit{JOIN}(\sigma)$ **then**
    $i_{\text{mappings}} \leftarrow \text{model.bindAndMatchJoin}(\sigma, i);$
  end
  while $\mathit{attempts} < \mathit{LIMIT}$ **do**
    feedback $\leftarrow$ validate(i_{mappings}, model);
    if $\mathit{feedback} == \text{"OK"}$ **then**
      break;
    end
    else
      $i_{\text{mappings}}, \text{attempts} \leftarrow \varnothing, \text{attempts} + 1;$
      if $\mathit{attempts} < \mathit{LIMIT}$ **then**
        $i_{\text{mappings}} \leftarrow \text{model.refineMappings}(\sigma, i, \text{feedback});$
      end
    end
  end
  if $i_{\mathit{mappings}} \neq 0$ **then**
    $i'.\text{logic} \leftarrow \text{VADALOG.apply}(\sigma, i_{\text{mappings}});$
    $i'.\text{metadata} \leftarrow \text{storeMetadata}(i_{\text{mappings}});$
    $i'.\text{nl} \leftarrow \text{verbalize}(i', G);$
    if $\mathit{model.checkTermination}(\Sigma(D), i'.nl)$ **then**
      $\Sigma(D) \leftarrow \Sigma(D) \cup i';$
    end
  end
end
return $\Sigma(D)$

reasoner and applying the chase these inputs are combined to a verbalized plan leading to question-response pairs, which are used to fine-tune the LLM. This procedure constituted a minor improvement over the general purpose LLM.

Another approach to fine-tuning LLMs involves verbalizing ontologies into text and

training the model using a contrastive learning framework [RN24], which involves positive and negative pairs of data. This method showed notable improvements in sentence similarity tasks for domain-specific models, though gains were smaller for more advanced LLMs.

There are also approaches leveraging a Graph Neural Network (GNN) [ZCH+20], which is a neural network architecture designed to process graph-structured data. The QA-GNN [YRB+21] constructs a joint graph through connecting the given query answering context with the KG, using relevance scoring and then designing a GNN module for reasoning. GreaseLM [ZBY+22] combines a LLM with information from a KG using a GNN, which helps the model better handle complex language features like negation or uncertainty.

A structure reasoning skill can be embedded into the LLM by leveraging the framework for Unifying Structure Reasoning [WWX+23], which uses a PLM. Through geometric embedding-based methods, representations of entities and relations from text are constructed by mapping them to geometric shapes in a representation space. Then, during pre-training, structure-aware language representations are learned and fine-tuned for complex reasoning tasks.

However, there is evidence suggesting that a PLM lacks general deduction capabilities. Yuan et al. [YHV+23] show that PLMs are prone to forgetting previously acquired knowledge during fine-tuning and often fail to effectively capture the logic rules embedded in the fine-tuning corpus. To overcome this limitation, a translation-based approach that leverages existing solvers and tools constitutes a sound alternative. LOGIC-LM [PAWW23] combines a LLM with the symbolic reasoner for the areas of Logic Programming, First-order Logic, Constraint Optimization and SMT Solver. The query is transformed into the target formalism, processed using an appropriate solver, and subsequently mapped back to the original representation. The findings demonstrate that Logic-LM outperformed standard LLMs and also the Chain-of-Thought (CoT)-reasoning[WWS+22].

2.3 Text-to-SQL, Prompting and LLM-Based Interaction with Databases

In NLP, *zero-shot learning* refers to a model's ability to perform tasks without receiving any task-specific training examples. Instead, the model uses its pre-trained knowledge to generalize and make predictions on unseen tasks or classes [KGR+22]. On the other hand, *few-shot learning* involves providing the model with a limited number of examples to guide task performance. These examples are typically presented within a *prompt*, a structured input that includes natural language instructions and illustrative instances, to contextualize the task for the model. There is evidence indicating that LLMs can produce decent results relying exclusively on such few-shot demonstrations [BMR+20].

Text-to-SQL refers to the task of automatically translating natural language questions into SQL queries, enabling non-technical users to retrieve information from relational

databases without needing to know the SQL language. A common method relies on encoder-decoder architectures, such as the Seq2Seq model[SVL14]. In these systems, the natural language question and the table schema are encoded and used as input, while the decoder generates the corresponding SQL query. Two notable examples include Seq2SQL [ZXS17], which enhances the basic Seq2Seq model by incorporating reinforcement learning or the approach developed by Iyer et al. [IKC+17] incorporating user feedback.

Additionally, there are sketch-based methods. These approaches in Text-to-SQL involve decomposing SQL query generation into structured components, enabling more interpretable and modular models. Two main contributions include SQLNet [XLS17], which employs a dependency graph to predict SQL components, and IRNet[GZG+19], which introduces an intermediate representation.

Further there are various approaches for integrating LLMs with databases, often leveraging a Text-to-SQL methodology. For example, DIN-SQL [PR23] decomposes the SQL query generation process into multiple sub-steps, with the results being combined in the final step. This approach leverages techniques such as CoT prompting [WWS+22], which guides the model through a series of reasoning steps using multiple prompts, and least-to-most prompting [ZSH+22], where problems are broken down into smaller subproblems and solved incrementally. TAPEX[LCG+21] uses a pre-training approach where the training is based on SQL queries and their execution results. Furthermore, KB-BINDER [LMZ+23] is a query answering framework over diverse datasets generating logical forms using a LLM and then binding the entity and the relation. Nan et al. [NZZ+23] demonstrate the effectiveness of different prompt design strategies and show that using both similar and diverse prompts for demonstration leads to improved results. Going beyond the Text-to-SQL approach is the Chain-of-Table approach [WZL+24], incorporating the structure of the table inside the reasoning chain. Using CoT-Reasoning and 1-shot demonstration, without finetuning, LLMs can match the performance of pre-trained architecture like GraPPa [YWL+20] on datasets like WikiTableQuestions [Che22]. However, performance is still not satisfactory and can't substitute symbolic models. Moreover the performance decreases for bigger table inputs. Furthermore, there is a hybrid approach like Dater [YHY+23], where a LLM is used as a decomposer to generate relevant sub-tables and intermediate SQL-queries, whose results are given to a LLM to generate the final answer.

There is the potential to automatically enhance the database with data inferred from the LLM. OmniscientDB [UNB23] integrates LLMs within databases, allowing users to query both explicit data and implicit parametric knowledge of the LLM using virtual tables. These virtual tables are treated like regular tables, enabling all according operations.

A framework for QA with structured data is StructGPT [JZD+23]. It introduces an integrated framework for QA for multiple types of structured data, namely KGs, tables and databases. For a table, the pipeline consists in extracting the table names, their columns and a relevant sub-tables. In a next step, the information is invoked and linearized and finally a LLM generation step occurs. Furthermore, an implementation like DB-GPT [XJS+23] includes a conversational agent that enables users to interact

with the system through natural language, enabling intuitive and efficient access to data analytics.

2.4 Query Expansion

Query expansion is a technique that enhances the query terms in a retrieval system by incorporating similar terms frequently found in a set of relevant documents. This process helps capture the user's intent more accurately by adding relevant terms [CR12]. The initial query terms are expanded with the aim to improve the recall of the system while not degrading the precision. Initially, Automatic query expansion (AQE) was introduced by leveraging the concept of relevance and designing a *Probabilistic Indexing* technique, which assigns a numerical estimate of relevance to each document with respect to a given query, thereby enabling the ranking of documents based on their estimated relevance[MK60].

A variety of approaches have been developed to implement query expansion. One such approach involves the use of lexical knowledge bases, which are structured repositories of semantically and lexically related terms constructed around a word [GEKM16]. In this approach, the original query terms are expanded by incorporating lexically related terms derived from the knowledge base[Voo94].

Relevance Feedback (RF) is an iterative process that refines search results by incorporating user feedback on retrieved documents to dynamically expand the initial query. By leveraging user feedback on retrieved documents, it iteratively improves the retrieval process [Roc71]. The idea of RF can be expanded to Psuedo-Relevance Feedback (PRF) to incorporate implicit feedback by the user by assuming that the top-ranked documents are relevant [SB97].

As LLMs have advanced, they have been increasingly applied to query expansion tasks by fine-tuning or modifying training procedures. Imani et al. [IVMS18] utilize a neural classifier to predict the usefulness of candidate expansion terms based on word embeddings. Similarly, Zhang et al.[ZHH+20] leverage BERT to select contextually relevant text chunks from top-ranked documents for query expansion, thereby improving document re-ranking performance. Furthermore, also fine-tuned, open-source LLMs can be leveraged for query expansion without any training steps and therefore decreasing the resource need [WBZ+21]. Going on, query expansion can be implemented making use of few-shot demonstration or CoT on natural language queries investigating a variety of prompt templates [JZQ+23], showing that LLM-based approach combined with CoT lead to a superior result. Furthermore, Query2doc [WYW23] leverages a LLM to generate a pseudo-document followed by a retrieval stage for query expansion. Another approach to leveraging a general-purpose LLM via few-shot prompting for information retrieval, is to use OpenAI's GPT-2 to expand a given query by generating artificial texts for query expansion [Cla21].

2.5 Data Integration, Entity Matching and Blocking

Data integration addresses the challenge of combining data from multiple sources, which may involve different formats and structures, into a standardized format. This process ensures that disparate datasets can be queried and analyzed cohesively, as if they were part of a single unified dataset [ZD07]. A main problem of this topic is the need for labeled data along with a ground truth. Oftentimes these datasets have to be manually curated by domain experts and are therefore subject to human error. A solution for solving the problem of data generation is crowd sourcing, [VBD14] leveraging humans to verify the equality of a subset. However, when leveraging crowd sourcing this can be prone to errors due to unscientific labeling quality by potentially unexperienced humans [DTWP18]. Another approach leverages Active Learning [BIPR12] by selecting relevant pairs to improve recall while maintaining a precision threshold. However, it can be argued that active learning alone does not provide sufficient comprehensive quality guarantees. To address this, the Human and Machine Cooperation Framework (HUMO) [CCF+17] divides the workload between machines and humans, offering the flexibility to enforce both high precision and recall.

Entity matching is a research field that focuses on identifying and linking different data representations that refer to the same real-world entity [CEP+21]. This task, commonly performed in data integration, arises from the need to merge information from multiple sources, accounting for potential duplicates and variations in representation. Possible problems arise from poor data quality, the quadratic complexity of potential matches and oftentimes the dependency on external human knowledge and interaction [BG21].

To tackle the quadratic complexity of potential matches, many blocking procedures have been introduced to reduce computational cost. Traditional blocking [I. 69] partitions records into blocks based on Blocking Key Values (BKVs), thereby reducing the number of candidate record pairs by limiting comparisons to records within the same block.

The Sorted Neighborhood Indexing technique, first proposed in the mid-1990s [CG08], involves sorting records based on a BKV and then sliding a fixed-size window over the sorted list to generate candidate record pairs. The inverted index-based approach indexes records using sorted BKVs and applies a sliding window over these keys to generate candidate record pairs from the corresponding index lists [dVKCC11].

Beyond indexing techniques, various clustering techniques efficiently handle large, high-dimensional datasets. These techniques leverage approximate distance measures to create overlapping subsets, canopies. Then the exact distances are only calculated in the respective canopy [MNU00] leveraging the Term Frequency-Inverse Document Frequency (TF-IDF) cosine-similarity [SB88].

The TF-IDF cosine similarity between two documents A and B is given by:

$$\text{cosine similarity}(A, B) = \frac{\sum_{i=1}^{n} \text{TF-IDF}_{A,i} \cdot \text{TF-IDF}_{B,i}}{\sqrt{\sum_{i=1}^{n} \text{TF-IDF}_{A,i}^2} \cdot \sqrt{\sum_{i=1}^{n} \text{TF-IDF}_{B,i}^2}} \tag{2.1}$$

where TF-IDF$_{A,i}$ and TF-IDF$_{B,i}$ represent the TF-IDF values of term i in documents A and B, respectively.

An alternative measure is the Jaccard similarity s_J between record r_A and record r_B, based on the intersection and union of their token sets, which can be calculated as:

$$s_J(r_A, r_B) = \frac{|\text{token}(r_A) \cap \text{token}(r_B)|}{|\text{token}(r_A) \cup \text{token}(r_B)|} \tag{2.2}$$

Cluster calculation can be performed using a threshold-based approach, where records are assigned to a cluster based on a similarity measure and the selection of centroids [CR02], as well as by a nearest-neighbor approach, which involves clustering records based on their proximity to other records [CG08].

Additionally, research has explored the application of deep learning architectures for blocking [TLT+21], utilizing a similarity-based pairing approach and calculating threshold-based similarity in the final stages. This demonstrates the feasibility of incorporating these strategies into the blocking process. DeepER [ETJ+18] utilizes a bidirectional Long Short-Term Memory (LSTM), a neural network designed to learn sequential data, for generating word embeddings and applies cosine similarity in each dimension. Another approach, AutoBlock [ZWS+19], combines bidirectional LSTM and self-attention mechanisms for efficient word representations and leverages Fast Nearest Neighbor Search to retrieve potential candidates for blocking.

Rule-based entity matching [DRD+13, EIO+14] refers to matching and linking entities from different datasets using pre-defined rules based on specific attributes or patterns, such as exact or fuzzy matching. This approach can be advantageous for end-users by potentially reducing execution time and conserving computational resources. These rules can be synthesized using different logical structures. Entity matching rules can be synthesized using a General Boolean Formula (GBF) rule structure [SME+17], which applies logical operators to combine conditions for matching entities, or a Disjunctive Normal Form (DNF) rule structure [WSSJ14], which organizes conditions into a series of disjunctions and conjunctions to improve matching accuracy and flexibility. Another more user-centric approach to develop rules is the application of TuneR [PGSV19], which is a framework that leverages fine-tuning to optimize rule sets grounded in user input criteria, hence decreasing user interaction.

Additionally, deep learning methods, including those based on LLMs, have been explored for entity matching due to their strong semantic capabilities. There are fine-tuning approaches like BERT [DCLT18], Sentence-BERT [RG19], Ditto [LLS+20], a meta-learning approch like Rotom [MLW21] and transfer learning approach DADER [TFT+22] and also PromptEM [WZC+22], leveraging prompt tuning for a low-resource environment.

Evaluations showed decent results for a variety of data cleaning and data integration task leveraging few-shot prompting with the GPT-3-175B parameter model [NCO+22]. Additionally, BatchER [FHF+23] implements cost-effective batch prompting to reduce

computational cost leveraging LLAMA2 and GPT-3.5-turbo. In terms of reducing computational complexity the COMEM framework [WCL+24] extends entity matching to input a multitude of candidates . Furthermore, LLMs can be fine-tuned for entity matching, though the effectiveness varies across different models. It was demonstrated that LaMA 3.1-3B improved performance while GPT-4o Mini exhibited a decline [SPB24]. Further, Peeters et al. [PB23] studied various prompt design strategies some of which outperformed RoBERTa[LOG+19], which was fine-tuned on a sampled training set. In addition, Peeters et al. [PSB23] showed that LLMs exhibit decent zero-shot performance, which can match that of fine-tuned PLMs.

CHAPTER 3

Query Processing Pipeline

The following Chapter deals with designing the query processing pipeline. In Section 3.1 we describe the problems encountered, specifically regarding the availability of data. Section 3.2 then introduces the concept of soft binding, a core component of the proposed pipeline that leverages the semantic understanding of LLMs to redefine SQL comparison operations. Section 3.3 describes the design considerations for the query pipeline, outlining three distinct strategies. The next Section 3.4, details the architecture and implementation of the proposed query processing pipeline. Section 3.5 details the process of operator transformation to ensure correct integration of soft-bound entities into the final query and introduces a duplicate elimination mechanism. Going on, Section 3.6 introduces the concept of blocking. In a final step, Section 3.7 outlines the computational resources and API limitations encountered during the development of the query processing pipeline.

3.1 Problem Description

Querying relational databases using SQL can produce incomplete or undesired results due to residual noise in the data. This noise may stem from factors such as incorrect schema alignment, insufficient data cleaning, or erroneous entries [KK18, RD00].

LLMs have demonstrated significant semantic understanding and reasoning capabilities [KGR+22]. These capabilities offer the potential to analyze database contents, facilitating the automatic identification and mitigation of potential sources of noise. Additionally, they than provide the opportunity to refine query outputs to mitigate the impact of such noise. The goal of this thesis is to develop a pipeline that accounts for residual noise affecting query results, ensuring more complete and better-aligned data retrieval with user intent.

In the SQL language, a variety of comparison operators are utilized, including “=”, “<”, “”>, “>=”, “<=”, and “<>” [Pos25]. These operators are fundamental for comparing

values and establishing relationships between them, primarily within WHERE clauses or JOIN conditions, to filter or combine data based on specified criteria.

The core methodology of this research involves modifying the standard comparison operations by introducing a novel approach referred to as soft binding. This concept, developed in this thesis, redefines how SQL comparison operations evaluate to TRUE or FALSE by leveraging the semantic capabilities of a LLM. Soft binding enhances traditional entity matching techniques by utilizing all available SQL comparison operators, thereby extending its applicability to a wider range of use cases while seamlessly integrating into the SQL framework.

Potential residual noise can be exemplified by language inconsistencies, as shown in Table 1.1.

For instance, the entities “chien” and “perro” can be compared to “dog” using the “=” comparison operator. In this case the soft binding enables the matching of multiple candidates, “chien” and “perro”, to a single real-world entity, the concept of a dog. Hence for the scope of this query, the respective entities should be softly bound to the real-world entity “dog”. This example highlights potential database inconsistencies. The key inconsistencies that guided the development of this thesis are as follows:

- **Handling String Representations of Numbers**: If numerical values are represented as strings in the dataset, the soft binding mechanism should still function as if all values were properly typed as numbers. For example, when querying for values smaller than 12, the system should correctly identify both the numerical values like for example “11” and also potential string representations like “ten” as valid results.

- **Inconsistent Data Formats**: In cases where entries are represented inconsistently, such as differing formats for dates or large numbers, the soft binding mechanism should internally standardize these representations. For example, dates like “18.04.1968” and “April 1986 on the day number 18” should be interpreted as equivalent. This ensures that relevant data entries are accurately matched.

- **JOIN Attribute Modification**: When performing JOINs, the JOIN keys might be represented differently across datasets. For example, the numerical value “1” in one source may correspond to the string “one” in another. The soft binding mechanism should account for such cases by treating numerically equivalent values as equal.

- **Multilingual Support**: The soft binding should recognize multilingual variants. For example, when querying for animal types, a search for “dog” should also return entries labeled as “perro” in Spanish and “chien” in French, while excluding unrelated terms such as “chat,” the French word for “cat.”

3.2 Soft Binding

SQL utilizes a range of comparison operators [Pos25]. These operators are integral to the functionality of WHERE and JOIN clauses, which are used to filter and combine data. Within these clauses, arrays of values corresponding to specific columns are compared against each other according to the defined operator. The records where the comparison condition evaluates as true are included in the result set, while those that do not satisfy the condition are excluded. These operators form the foundation of querying in SQL, allowing for data retrieval and manipulation through comparisons.

The soft binding procedure redefines the evaluation criteria of comparison operators by utilizing a LLM, enabling comparison outcomes to be determined based on semantic similarity rather than syntactic equivalence or similarity. In order to achieve that, the comparison operation is verbalized and fed to a LLM leveraging a zero-shot prompting approach. In the process, all entities involved in the comparison are formatted into a structured prompt string, in which the verbalized comparison operator serves as the connective element between them. The LLM is then instructed to return a boolean value indicating whether the statement is correct or incorrect. The verbalization of the operators is facilitated through a mapping, which defines the semantic meaning of each comparison operator. This mapping is provided in Table 3.1.

Comparison Operator	**Meaning**
=	has the same meaning as (also in another language) or is the same as
<	is smaller than
<=	is smaller or equal compared with
>	is bigger than
>=	is bigger or equal compared with
!=	has a different meaning than
<>	has a different meaning than

Table 3.1: Semantic Meaning of Comparison Operators

Revisiting the multilingual example from Table 1.1, we can see how different prompts are constructed based on a comparison operation. Let's assume the initial comparison operation is:

```
SELECT * FROM animalowner WHERE category = "dog"
```

For each row a distinct prompt is generated. In the prompt a `f`-string used. It is a Python syntax feature that allows embedding variables or expressions directly within a string using curly braces `{}`. The construction of the prompt follows the following general format:

```
prompt = f"{value1}"  {comparison operator}  "{value2}"
```

For the first row of the table, the resulting prompt would be:

```
"dog" has the same meaning as (also in another language)
or is the same as "chien"
```

The prompt is presented to the LLM using a zero-shot prompting methodology, wherein the LLM is instructed to generate an accurate boolean value based on the provided input. In this specific instance, the expected output is true, as the values correspond to the same real-world entity. This evaluation process is repeated for each comparison operation. Then, the values for which a true boolean value was returned are selected as the soft bound entities. In the next step, a dictionary is created using the soft-bound values. The hard-bound entity serves as the key, while its corresponding soft-bound entities form the values. Each comparison operation generates a new dictionary, which is then added to a list of dictionaries - the semantic list. A dictionary has the following structure:

```
{hard_value :  soft_values}
```

In a final step the semantic list is incorporated to write the final query, a process which is discussed in much detail in Section 3.3.

It is important to note that the interpretation of a comparison operator can be adjusted based on the specific task at hand. For instance, when applying the pipeline to a classification task, the "=" operator could be reformulated to "belongs to the category" to better align with the task requirements. This flexibility allows for dynamic adaptation of the comparison semantics, ensuring that the comparison logic aligns with the particular context or domain of the problem being addressed, thus broadening the application possibilities.

3.3 Design Considerations

The mechanism for the soft bound entities was already discussed in Section 3.2. In order to develop a pipeline leveraging the soft binding to modify user output, various approaches can be considered. The following discussion outlines three potential pipeline implementations: a parameter re-implementation, a translation-based approach, and the auxiliary table strategy. These approaches are further elaborated in the subsequent sections.

3.3.1 Parameter Re-Implementation

As previously discussed, the SQL language utilizes a set of comparison operators to compare values. One possible approach to implement the soft binding for these comparison

operators is to create a custom querying language or even a new database engine. In this new system, a soft binding version of each comparison operator would be introduced. This would alter the standard behavior of these operators to leverage the semantic capabilities of a LLM.

Other potential pipelines that leverage the SQL framework face challenges when altering the behavior of comparison operators and handling soft-bound values. Once the comparison is executed, additional steps are required to modify the query to incorporate the soft-bound values, ensuring they are accurately reflected in the final output. This often involves rewriting or adding extra conditions to the query, which increases both the complexity of the process and the potential for errors.

In contrast, the re-implementation of comparison operators to their soft-bound versions removes the need for such intermediate steps. By integrating these modified operators directly into the querying process, the database engine can automatically return the correct results without the need for manual adjustments to the query.

However, a significant drawback is the lack of transparency in this approach. A user receiving a result might struggle to understand its validity, potentially eroding trust in the system. This lack of interpretability makes this approach of re-implementing operators less attractive. In consequence, it could hinder system use and be ineffective for data cleaning, as intermediate steps, such as displaying the final query before execution, are not realizable.

3.3.2 Consequences

Due to the transparency issues within the scope of this thesis, a **translation-based approach** and a **auxiliary table** strategy were adopted, both making use of SQL. Instead of re-implementing the operators, the original SQL query is modified to return also the soft bound entities. The following sections elaborate on the chosen approaches.

3.3.3 Translation-Based Approach

A possibility to leverage the SQL framework is to use a translation-based approach. Instead of re-implementing the parameters themselves the query can be modified by the possible extension with *CASE* and *OR* statements. For this approach the resulting SQL query, if generated, is returned to the user, along with the database output.

In the case of the translation-based approach the SQL syntax is leveraged to include all soft bound entities. In the scope of this thesis, possible WHERE statements along with JOIN statements are analyzed. Soft bound entities are included via referencing them in SQL itself. This process can be exemplified by the following example.

```
{dog :  chien, perro }
```

with a general structure of

```
{hard_value :  soft_values}
```

We consider a scenario with noise in which soft-bound entities are inferred. Consequently, to ensure that the correct records are retrieved, the standard JOIN operation must be adapted toward these different representations.

Thus, an initial JOIN query:

```
JOIN table2 ON table1.column1 = table2.column2
```

can be modified to:

```
JOIN table2 ON table1.column1 = CASE table2.column2
                WHEN 'soft_value' THEN 'hard_value'
```

Similarly, the WHERE clause can be extended with additional OR statements. Revisiting the example above, this would mean transforming the query from the original query:

```
WHERE animal.category = 'dog'
```

to the modified query:

```
WHERE animal.category = 'dog'  OR  animal.category = 'perro'
                               OR  animal.category = 'chien'
```

Overall in the translation-based approach, soft-bound entities are incorporated through the use of *CASE* and *OR* statements. By examining the modified SQL queries users can identify potential noise and subsequently remove it. Over time, through iterative querying by individual users or user groups, this process can lead to substantial improvements in data quality. Therefore, the proposed query pipeline can be considered a tool for detecting residual noise and supporting a late-stage data cleaning process.

3.3.4 Auxiliary Table

Another possibility exists next to the translation-based approach. Instead of modifying the original query itself via the extensions of *CASE* and *OR* statements, intermediate tables can be defined. Similar to the OmniscientDB [UNB23], the database is extended through the LLM's parametric knowledge. The results, which consist of both hard-bound and soft-bound values, are written into a separate table. This procedure applies to both *WHERE* and *JOIN* statements, with the data being stored in the auxiliary table.

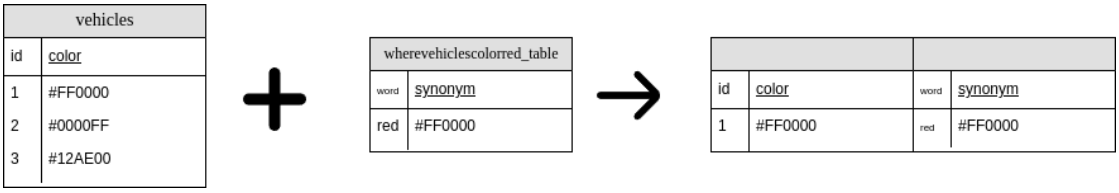


Figure 3.1: Auxiliary Approach

The idea behind the auxiliary table can be seen in the Figure 3.1. The original color column in the table vehicles is expressed as a color code instead of the natural language description of the color itself.

Exemplary for this would be an example of filtering vehicles based on color. However, the initial query not leveraging the color hex code, would not return any result, when querying for the original color. Hence, the auxiliary table is created. The potential initial query could be the following:

```
SELECT * FROM vehicles
INNER JOIN owners ON vehicles.owner_id = owners.id
WHERE vehicles.color = 'red';
```

After the initial query is constructed and the soft-bound values inferred, the intermediate table has to be constructed. In a final step,it is necessary to add an additional JOIN with the auxiliary table in order to deliver the wanted result. Then the modified query would be the following:

```
SELECT * FROM vehicles
INNER JOIN owners ON vehicles.owner_id = owners.id
INNER JOIN wherevehiclescolorred_table
ON wherevehiclescolorred_table.synonym = vehicles.color
WHERE wherevehiclescolorred_table.word = 'red';
```

Also, it is to note that also the evaluation of the metrics becomes more complex as the output has more columns due to the intermediary tables as the ground truth. More columns are returned that are contained in the ground truth due to the auxiliary table. Therefore, the evaluation had to be adjusted. Hence, an output row is regarded as correct if the ground truth is a subset of the output row.

Finally, this has the advantage that the user can access the created table and therefore can reasonably assess the quality of the LLM's answer and potentially adjust the created table. It further provides an implicit request for subsequent data cleaning steps that may be necessary to ensure data quality.

To illustrate the differences between the translation-based and auxiliary table approaches, Figure 3.2 presents the flowchart for the auxiliary method, while Figure 3.3 depicts the

flowchart for the translation-based approach. The three key components namely the user, the database and the pipeline, meaning the program, are displayed. A principal distinction between the two approaches is the explicit write operation required by the auxiliary table method, wherein intermediate results are written to the database. Hence, an advantage of the auxiliary method is that it allows users to inspect and, if necessary, modify the auxiliary table and possibly re-execute the final query.

3.4 Pipeline Design

In this section, we describe how the motivations and requirements outlined in Section 3.1 shape the design and implementation of the structured pipeline. Taking a predicate calculus query as input, the pipeline generates a corresponding modified SQL query, which is then executed on the database to produce the desired output. The modification of the initial query involves multiple stages of query processing ensuring the soft bound entities are incorporated in the output. The main steps of the pipeline can be seen in in Figure 3.4. The key stages of the pipeline are as follows:

1. An initial SQL query is generated based on the predicate calculus expression and the schema level information of the database by prompting the LLM. The schema-level information, including the column name, data type, nullability, and possible constraints, is retrieved for each table **t**. This is done by executing the SQL Query 6.2. The context for each table is written to a file, in order to avoid multiple querying for various runs. The output of such a context for a potential table *doctors* could be for example:

```
The name of the table is doctors

Columns in the table doctors (in correct order):
id
name
patients_pd

Schema Information:
[('id', 'NO', 'integer', 'PRIMARY KEY'),
('name', 'YES', 'text', None),
('patients_pd', 'YES', 'text', None)]
```

2. A SQL parser checks the initial SQL query for the presence of at least one *WHERE* clause and/or at least one *JOIN* clause. Steps 3–4 are executed if at least one *JOIN* clause exists, whereas steps 5–6 are executed if at least one *WHERE* clause exists.

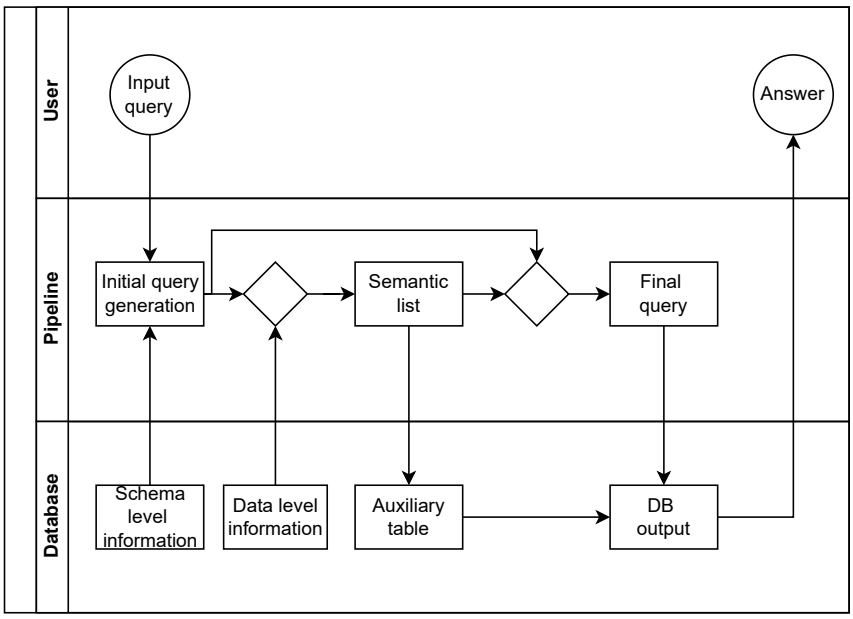


Figure 3.2: Flowchart Auxiliary Approach

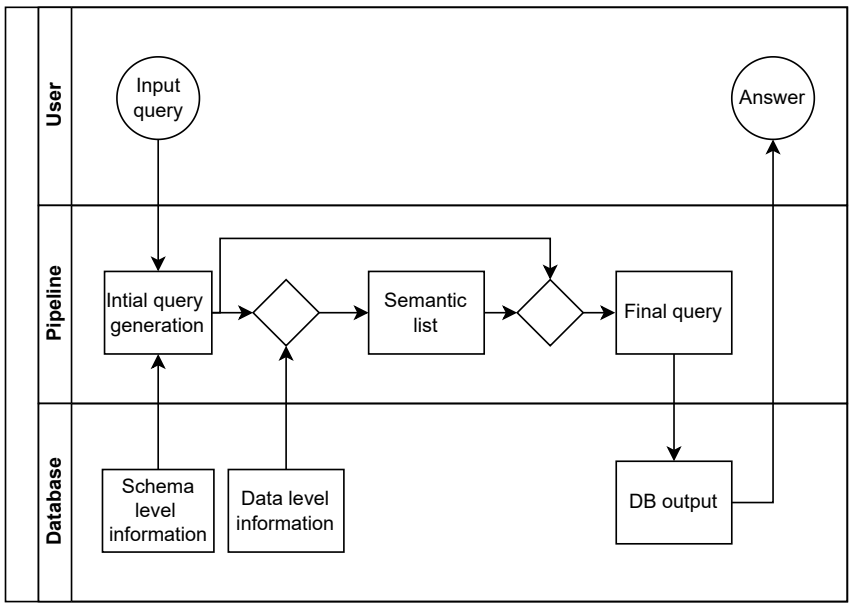


Figure 3.3: Flowchart Translation-Based Approach

3. For each JOIN condition, the two JOIN values as well as their unique values are retrieved. Then the soft-binding procedure, which was explained in detail in Section 3.2, returns a dictionary of semantically relevant terms and adds it to the semantic list. The resulting semantic list may take the following form:

```
[ {hard_value1 :  soft_values1},
{hard_value2 :  soft_values2} ]
```

4. The LLM, using a few-shot prompting approach, combines the semantic list of the *JOIN* with the original query. The approach is further discussed in Section 3.5. Soft bound entities are included in the output via the query modification through the LLM using either the auxiliary table or the translation-based approach.

5. The semantic list based on all *WHERE* conditions is created using the the soft binding methodology explained in Section 3.2.

6. The final output query is generated using the semantic list from the previous step. If the JOIN pipeline was used, its query output is incorporated. Otherwise, the original SQL query is used instead. This process is driven by a LLM with few-shot prompting to ensure accurate query generation.

7. The final query is executed on the database, and the result is returned along with the modified query displayed to the end user.

Overall, every WHERE and JOIN operation is checked towards potential binding. If no soft bindings are retrieved for a specific variable, the result is equivalent to the hard binding and the condition is not modified.

The pipeline is implemented using the **combined pipeline**, which internally reference the **JOIN pipeline** and the **row calculus pipeline**, which deals with WHERE clauses. This means that the the steps 1-2 are managed by the **combined pipeline**, steps 3-4 are managed by the **JOIN pipeline** and the steps 5-6 are dealt with by the **row calculus pipeline**. The general structure of the **combined pipeline** is given by the 3.1 algorithm.

The pseudo code for the *join_pipeline* 6.2 and the *row_calculus_pipeline* 6.2 functions is provided in the appendix for enhanced clarity and detailed step-by-step descriptions.

3.5 Operator Processing and Duplicate Elimination

Once a semantic list has been generated, it is essential to integrate it correctly into the final modified query. This step is crucial for both the translation-based and auxiliary table approaches to ensure that the selected entities are accurately incorporated into the query output. For this, the pipeline is designed to transform the comparison operator into the equality operator "=" in the final query.

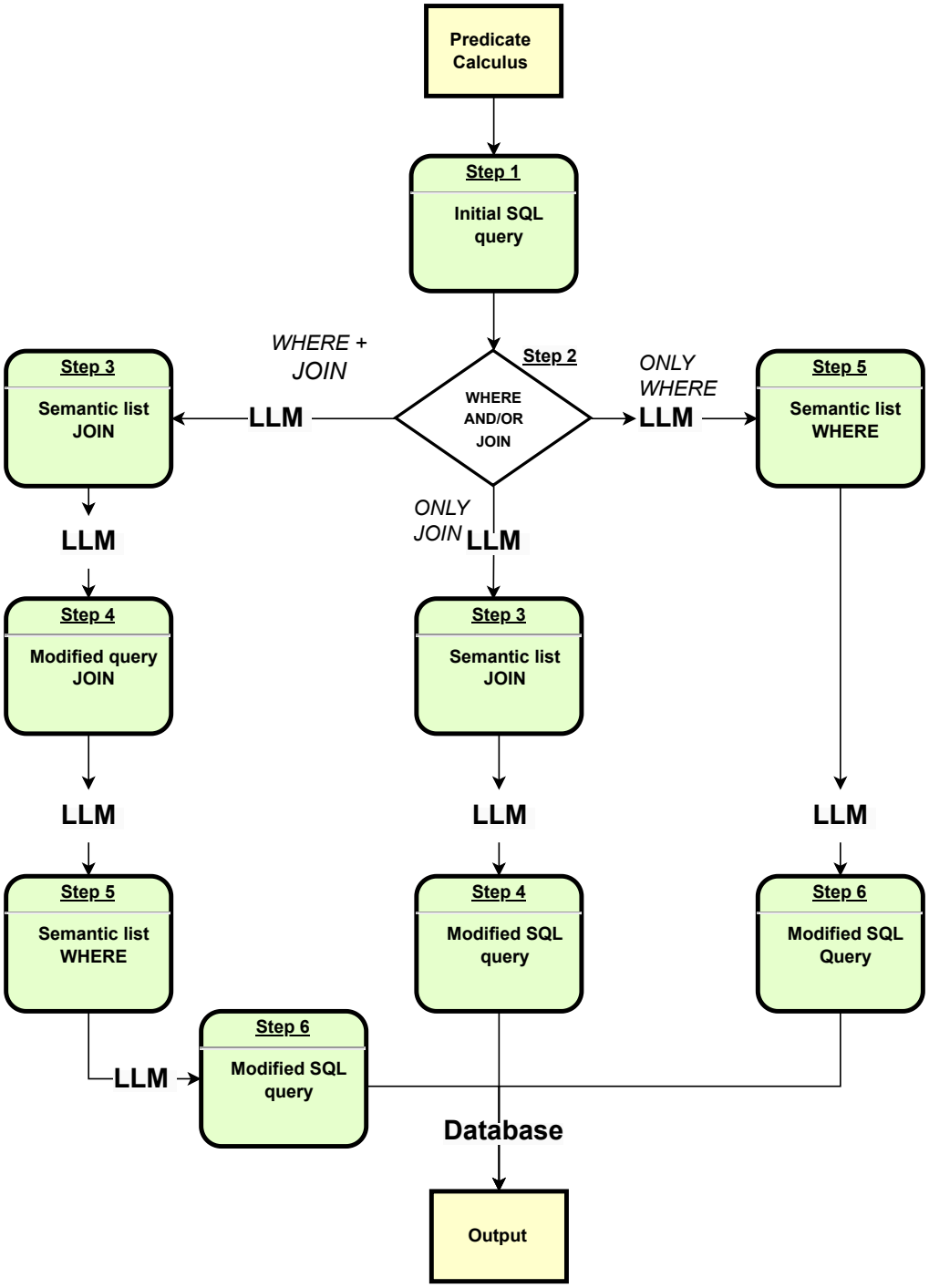


Figure 3.4: High-Level Query Pipeline

Algorithm 3.1: Combined Pipeline

```
Input  : query
Output : output
total_retries ← 4
count ← 0
while count<total_retries do
    tables ← get_relevant_tables(query)
    if tables is not None then
        break
    end
    else
        count ← count + 1
    end
end
context ← get_context(tables) Get schema level information
sql_query ← intial_query(query, context) Construct initial query
where_conditions, join_conditions ← analyze_sql_query(ql_query)
Look for WHERE and JOIN conditions
if join_conditions and where_conditions then
    output_query ← join_pipeline(sql_query)
    output ← row_calculus_pipeline(output_query)
end
else if where_conditions then
    output ← row_calculus_pipeline(sql_query)
end
else if join_conditions then
    output ← join_pipeline(sql_query)
end
else
    output ← query_database(sql_query)
end
return output
```

For instance, the "!=" operator serves as the logical inverse of the "=" operator. Consequently, the construction of the semantic list should produce the exact complement of the entities returned by the "=" operator. This can be illustrated via the Table 1.1. Consider the following initial query:

```
SELECT * FROM animalowner WHERE category != "dog"
```

In this case, the phrase comparing the two values would be "has a different meaning than." Therefore, all elements that satisfy this condition should be selected. As a result,

the ideal dictionary within the semantic list is defined as:

```
{dog :  chat}
```

Consequently, the comparison operator must be transformed into the equality operator "=" in the final query. This transformation applies to both the translation-based and auxiliary table approaches. Furthermore, the conversion to the "=" comparison operator is systematically performed for all comparison operations and is applicable within both *WHERE* and *JOIN* clauses.

These modifications are executed during the SQL query adaptation process after the semantic list has been inferred in the steps four and six in the pipeline explained in Section 3.4. To facilitate this process, a few-shot prompting technique is employed. Although LLMs have demonstrated strong zero-shot reasoning capabilities [KGR+22], a few-shot approach can be more effective in task-specific scenarios, as it helps guide the LLM toward appropriately adapting the output, in this case, the modified query [BMR+20]. The prompt includes manually constructed illustrative examples that demonstrate the handling of each specific operator. Each example consists of an input, comprising the original query and its corresponding semantic list, and an output in the form of the modified query. This approach enables the LLM to effectively guide the appropriate query modification for both processing pipelines.

Continuing, the pipeline approach addresses the issue of residual noise in the database through either the translation-based method or the auxiliary table method. However, when using soft binding, additional output rows may be generated, potentially leading to duplicates. For instance, consider the following dictionary:

```
{ dog:  chien, perro }
```

Assuming the output tuples are:

```
( bill, chien, 1 )
```

```
( bill, perro, 1 )
```

By examining the output along with the dictionary it can be abstracted that the output is redundant. Hence in order to solve the redundancy problem, a function was defined to remove output tuples that are identical. If any row can be created from another row by substituting a value using the dictionary, then one of the rows has to be left out. The binding process is inherently bidirectional, as it establishes relationships between keys and values in both directions. Consequently, the reversed dictionary, which reflects this inverse relationship, must also be considered, and is therefore created inside the process.

The algorithm works by creating a reversed dictionary, where the values and keys are interchanged. Subsequently, it generates all possible variants by substituting each key with its corresponding values from both the original and the reversed dictionaries. During the final step, each generated row is evaluated for duplication. If a duplicate is detected, the row is discarded, otherwise it is added to the output.

The pseudo code can be seen in Algorithm 3.2.

Algorithm 3.2: Duplicate Row Removal with Synonyms (Direct and Reverse)

```
   Input  : rows, total_dic
   Output : cleaned_rows
 1 cleaned_rows ← [];
 2 seen_rows ← [];
 3 reverse_dic ← {};
 4 foreach key k, synonyms syns in total_dic do
 5 |   foreach synonym s in syns do
 6 |   |   reverse_dic[s] ← k;
 7 |   end
 8 end
 9 foreach row ∈ rows do
10 |   variants ← [];
11 |   foreach element e in row do
12 |   |   if e in in total_dic.keys() then
13 |   |   |   foreach synonym s of e in total_dic.keys() do
14 |   |   |   |   variant ← row with e replaced by s;
15 |   |   |   |   variants ← variants + variant;
16 |   |   |   end
17 |   |   end
18 |   |   if e in reverse_dic.keys() then
19 |   |   |   foreach key k in reverse_dic.keys() do
20 |   |   |   |   variant ← row with e replaced by k;
21 |   |   |   |   variants ← variants + variant;
22 |   |   |   end
23 |   |   end
24 |   end
25 |   if row or any row in variants is in seen_rows then
26 |   |   continue ;                                  // Skip duplicates
27 |   end
28 |   cleaned_rows ← cleaned_rows + row;
29 |   seen_rows ← seen_rows + row + variants;
30 end
31 return cleaned_rows
```

3.6 Blocking

As outlined in Chapter 2, blocking is a key technique for minimizing computational costs. Given the high computational demand of utilizing the LLM for zero-shot comparisons [ZWS+19], blocking helps conserve computational resources, reduce token usage, which translates to monetary costs when using an API, and decrease execution time.

In information retrieval, potential candidates are often embedded, and similarity scores are computed, such as with SBERT [XLS17], a modification of BERT, and cosine similarity [VSAB+24]. For the "=" comparison operator, distinct entities are compared based on their semantic meaning, hence achieving blocking. Initially, all potential candidates are embedded, and cosine similarity is computed between the original entity and each candidate. The cosine similarity between two terms A and B is calculated as:

$$\text{cosine similarity}(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{|\mathbf{A}||\mathbf{B}|} \tag{3.1}$$

Candidates that surpass the manually input threshold proceed to the second stage, while those below the threshold are excluded. In the second stage, zero-shot prompting is employed to compare the entities. Finally, entities deemed equivalent are soft-bound in the last step. The threshold for cosine similarity is determined during the evaluation phase.

This two-step process is optional within the pipeline.Alternatively, we can employ the embedding-only method, which utilizes only the first stage, or the zero-shot prompting method, which operates without embeddings. Additionally, it is important to note that blocking does not apply to the "<" and ">" comparison operators, as cosine similarity is not a suitable measure for such comparisons. For the “!=“ and “<>” operators, the process is inverted: entities below the threshold are included, rather than those exceeding it, as is the case for the "=" operator.

Figure 3.5 illustrates the process and its various settings.

3.7 Computational Resources and API Limits

The PostgreSQL version used during development was 17.2. The system used was Ubuntu 24.04.1 LTS along with a AMD Ryzen™ 5 4600H with Radeon™ Graphics × 12 on par with 16 GB of RAM. It is to mention that the LLM mentioned during the development step was Gemini-1.5-Flash through Google API [Goo25a]. The rate limits [Goo25c] during the development of the query pipeline were :

- 15 RPM (Request per minute)
- 1 million TPM (Tokens per minute)
- 1,500 RPD (Requests per day)

Therefore, the development of the pipeline had to take the limitations of the API into account as no computational resources could be accessed during the majority of the this thesis, which allowed only for limited execution of the pipeline hence reducing evaluation possibilities.

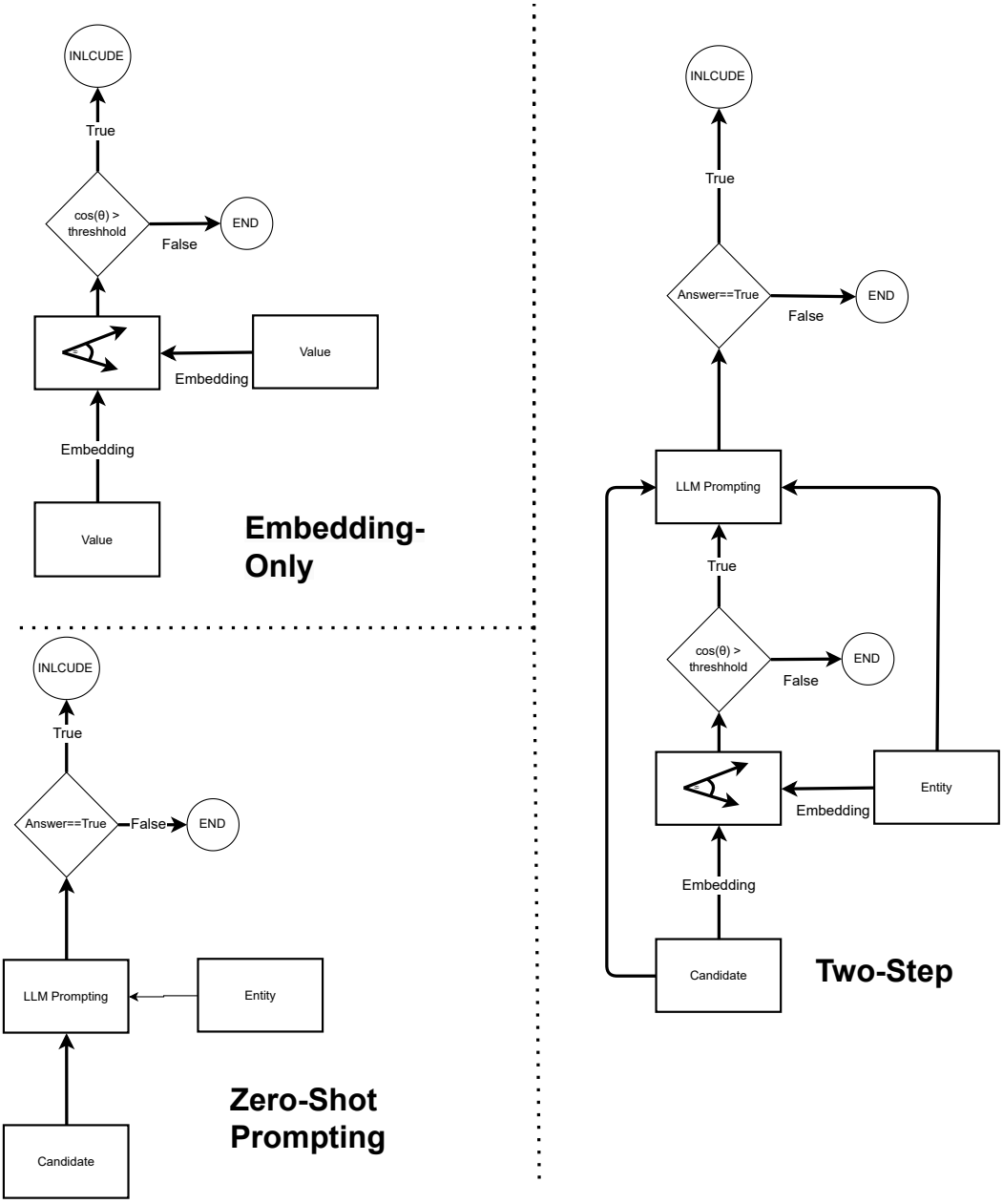


Figure 3.5: Blocking

CHAPTER 4

Evaluation

The following chapter presents a comprehensive evaluation process, encompassing multiple analytical steps. To begin, key initial considerations are outlined in Section 4.1, providing the foundation for the subsequent analysis. The discussion then shifts to the design of the test dataset in Section 4.2, where the covered cases and their relevance are examined in detail. Equally important is describing the necessary KPIs and measurements, which are introduced and justified in Section 4.3. The gathered results, obtained from various models and evaluation strategies, are then explored in Section 4.4, offering insights into the effectiveness of different approaches. To conclude, the proposed method is put to the test on two additional datasets in Section 4.5, verifying the validity and performance of the approach.

4.1 Considerations for Evaluation and Dataset Limitations

A key challenge in evaluating the pipeline is the necessity for high-quality data. This challenge is further exacerbated by the limited availability of suitable datasets that feature noisy data paired with predicate calculus queries and their corresponding ground truth. In the field of entity matching, some datasets have been curated, such as those provided in DeepMatcher[MLR+18] or the WDC Products Dataset [PDB23].

However, many available datasets are tailored to specific use cases, such as entity matching, as in the example above. In contrast, the implemented approach encompasses a broader scope by incorporating multiple comparison operators beyond simple matching, which is analogous to the "=" operator in SQL. Standard entity-matching datasets fail to account for other SQL comparison operators. For instance, a statement such as "many > 1" can potentially be interpreted as true, a relationship that can be abstracted by a LLM. However, to our knowledge, such statements are not typically captured in traditional entity-matching datasets. Furthermore, the user input in the proposed pipeline consists

of a predicate calculus query, and to our knowledge no public datasets are available that align with this input format for evaluation purposes.

This established the groundwork for evaluating the proposed approach using a manually curated small-scale test dataset to demonstrate its effectiveness. Additionally, further assessments were conducted on supplementary datasets, though these datasets only capture a subset of the capabilities of the approach.

It is important to acknowledge the substantial hardware constraints and dependency on API access, which significantly limited the scope of experimentation. Running a LLM locally, particularly for generating structured output, frequently resulted in system crashes due to hardware limitations. Moreover, the API's restriction of 15 calls per minute imposed severe constraints on the evaluation process, further restricting the extent of the analysis.

4.2 Test Dataset Construction

The following subchapters present a categorized overview of the predicate calculus queries created to test various semantic and structural challenges encountered in real-world data. Each category addresses a specific type of complexity, such as multilingual representations, inconsistent data formats, or ambiguous JOIN conditions. Within each category, the corresponding predicate calculus queries are listed along with a description of the particular challenge they aim to evaluate.

Since no publicly available datasets were suitable for this particular evaluation, a custom dataset was developed.

Different Languages / Multilingual Support

This section states predicate calculus queries associated with multilingual data to test the pipeline's ability to provide multilingual support.

- ∃id ∃shares ∃name (shareowner1row(id, name, shares) ∧ animalowner1row(id, _, 'dog'))

  The entity "dog" appears in the animalowner Table 2 in multiple languages: "chien" in French and "perro" in Spanish. The predicate calculus query searches for owners of "dog" and then performs a JOIN with the shareowner Table 1 based on the *id* column.

- ∃id ∃shares ∃name(shareowner(id, name, shares) ∧ ¬animalowner(id, _, 'dog'))

  The query retrieves owners of animals not labeled "dog" in the animalowner Table 2. The only entry, which is not a dog, is "chat", which is the French word for "cat". Also, a JOIN based on the *id* column is performed with the shareowner Table 1.

- ∃ movies("The sky over Berlin",_,_)

  This predicate calculus represents a filtering operation applied to the movies Table 20. The title to filter for is a literal translation of the German title "Der Himmel über Berlin" into English. However, the English title is not a direct syntactic translation but conveys a different meaning. As a result, the LLM faces the challenge of interpreting the movie's context to ensure accurate retrieval. The correct result in this case would be "Wings of Desire".

Different Scales / Inconsistent Data Formats

This subsection highlights predicate calculus queries that address the challenges posed by inconsistent data formats and varying scales, testing the pipeline's ability to handle different numerical representations and units effectively.

- ∃m ∃f ∃i (influencers(m, f) ∧ f > 500 ∧ followers(i, m, z))

  This example uses the influencers Table 11 together with the followers Table 12. This demonstrates the need to handle different representations of numerical quantities, such as "1000 thousand" and "1 million" in the *clicks* column of the influencers table. Moreover, instead of directly referencing the column names in the predicate calculus query, some variables are used as arguments to test the system's abstraction capability.

- ∃item bakery_sales(item, _, _) ∧ oven_temperature(item, >200 °C)

  This predicate calculus expression deals with the bakery_sales Table 18, which is filtered in degrees Celsius, whereas the records are in degrees Fahrenheit. In the next step, a JOIN occurs with the oven_temperature Table 19 based on the *Item* column.

- ∃item bakery_sales(item, < 55, _) ∧ oven_temperature(item, _) This predicate calculus expression queries the bakery_sales Table 18 for items with a quantity less than 55. However, the quantities are listed in dozens rather than as individual numbers in the *Quantity* column. The query then joins with the oven_temperature Table 19 to retrieve the corresponding oven temperature for the same item.

- ∃item bakery_sales(item, > 90, _) ∧ oven_temperature(item, >180 °C)

  This predicate calculus expression searches the bakery_sales Table 18 for items where the quantity exceeds 90, though the quantities are recorded in dozens. The query then performs a JOIN with the oven_temperature Table 19, filtering for items with oven temperatures greater than 180°C. The system must correctly handle the conversion of quantities from dozens to individual units, as well as convert the oven temperatures from Fahrenheit, as recorded in the oven_temperature table, to Celsius, as used in the query.

Inconsistencies

- ∃id ∃name ∃patients_pd (doctors(id, name, patients_pd) ∧ patients_pd < 12)

  This predicate calculus query operates on the ‘doctors‘ Table 5, where the *patients_pd* column sometimes contains numeric values as strings, such as “ten”, “11”. This creates a mismatch in data types when trying to compare string representations like “ten” or “11” with a numeric value like “12”.

- ∃id ∃patients_pd (doctors(id, ’Peter’, patients_pd) ∧ patients_pd < 12)

  This query builds upon the previous one, but it specifically filters for the doctor named “Peter” from the ‘doctors‘ Table 5. The *patients_pd* column still contains string representations of numbers, so the challenge remains in handling those correctly during the comparison operation.

- ∃id (tennis_players(id, _, ’January’) ∧ tournaments(id, name, price_money))

  This query highlights inconsistent date formats in the *born* column of the tennis_players Table 9. Some players’ birth dates are given as full dates like “20.02.2003”, while others are listed only by month and year like “January 1986”.

JOIN Attribute Modification

This subsection explores predicate calculus queries that tackle the challenge of handling inconsistencies in data, including string representations of numbers and varying date formats, evaluating the system’s ability to process such irregularities effectively.

- ∃x ∃y ∃z (children_table(x, y) ∧ fathers(x, z))

  In the children_table Table 13, the *id* attribute is expressed as a numerical value, whereas in the fathers Table 14, the attribute is expressed as text.

- ∃id (children_table(id,) ∧ fathers(id, _) ∧ mothers(id, _))

  In this example, the children_table Table 13 and the mothers Table 15 use numerical values for the id attribute, while the fathers Table 14 expresses the id attribute as text.

- ∃id (children_table(id, >1) ∧ fathers(id, _))

  In this case, the id attribute in the children_table Table 13 is a numerical value, while in the fathers Table 14, the id attribute is expressed as text. Additionally,

the children_table Table 13 contains the string “many” as a representation for a number greater than one, complicating the comparison logic.

- ∃movie movies(movie, _, _) ∧ movies_personal(movie, _)

  In this case, the movies Table 20 and movies_personal Table 22 express movie titles in different languages: English in movies and German in movies_personal. The system needs to handle these multilingual representations.

- ∃movie movies(movie, _, _) ∧ movies_personal(movie, >70%)

  This query introduces the issue of different representations of movie ratings. While the movies Table 20 uses a rating scale of 1 to 5, the movies_personal Table 22 provides ratings as percentages. The system must interpret the 70% filter in terms of the 1-5 scale in the movies Table 20.

- ∃clicks influencers(_, clicks) ∧ publication_clicks(_, clicks)

  The clicks attribute in the influencers Table 11 and publication_clicks Table 21 is represented in various formats, including “1000 thousand,” “1 million,” and “10^6.”

- ∃d weather(d, city, temperature, rainfall) ∧ website_visits(d, page, visits)

  This query demonstrates the challenge of joining the weather Table 17 and website_visits Table 16, where dates are represented in different formats, such as “2023 10 26” and “2023 October 26”. The system must reconcile these different date formats before performing the JOIN operation.

Uncategorizable

This predicate calculus query addresses an initial scenario to evaluate the ability of the query processing pipeline to interpret and align the data appropriately across different tables.

- ARTISTS(a,,), ALBUMS(,a,"Reputation",2017),SONGS(,a2,song_name,), ALBUMS(a2,a,)

  In the artists Table 8, the artist information is not fully aligned with the albums Table 7 and the songs Table 6.

4.3 Key Performance Indicators

For evaluation, the retrieved results are compared against the ground truth using KPIs such as precision, recall, and F1-Score. These metrics are defined as follows:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4.2}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.3}$$

- TP (True Positive): The number of correct positive predictions.
- FP (False Positive): The number of incorrect positive predictions.
- FN (False Negative): The number of incorrect negative predictions.

The query pipeline aims to improve recall, potentially at the cost of precision. This is because the initial query, lacking context or complete information, might miss some relevant records, resulting in a potentially lower recall score. The pipeline can infer additional records, increasing recall, but there's a risk of incorporating irrelevant or inaccurate data, leading to lower precision. To balance these opposing goals, the F1-score is used as a harmonic mean of precision and recall, providing a measure of the overall performance.

Additionally the amount of used tokens is captured for every query. It is important to note, that currently for Gemini models a token consists of roughly 4 characters, leading to the fact that that 100 tokens are approximately 60-80 words [Goo25b]. Relevant for this work are the following metrics:

- **Total Calls:** The total number of successful API requests for a given input query.
- **Prompt Token Count:** The number of tokens used in the request.
- **Candidates Token Count:** The number of tokens contained in the response(s).
- **Total Token Count:** The combined number of tokens in both the request and response(s).
- **Average Processing Time:** The mean time taken to process a query.

Tracking input tokens is essential due to their limitations. Furthermore, monitoring response tokens is important as comparing input and output token counts provides a measure of how elaborate the LLM's answers are. The total number of calls is a critical metric, as it is subject to both per-minute and daily limits. Additionally, monitoring the total calls provides a means to identify potentially redundant or unnecessary requests

during development, enabling the optimization of the pipeline to minimize resource consumption[Goo25d]. Additionally it is to note, that the relevance of the total number of calls diminishes when the LLM is executed on local hardware, as it no longer depends on API usage and associated rate limits. Finally, the average time constitutes an important measure as this is the processing time faced by the end user.

4.4 Test Dataset Evaluation

Beyond the KPIs outlined in Section 4.3, additional evaluation tools are essential for refining the pipeline. In the query modification process described in Section 3.4, multiple intermediate steps are executed. To enable a thorough analysis, each predicate calculus expression undergoes a complete execution, with all intermediate results systematically stored. This allows for direct comparisons with potentially erroneous outputs, facilitating the identification of the specific step responsible for inaccuracies.

Moreover, it is important to note that intermediate steps are assigned a *None* value if they are not utilized during execution. This occurs, for instance, when a SQL query lacks either a *WHERE* or *JOIN* condition, rendering certain steps unnecessary.

The intermediate steps are the following:

- initial_sql_query_join (Initial SQL query , if at least one JOIN present)
- semantic_list_join (Semantic list with soft bound entities for JOIN attributes)
- result_join (Modified query using the semantic list for the JOIN entities)
- initial_sql_query_where (Resulting query from JOIN or initial SQL query)
- semantic_list_where (Semantic list with soft bound entities for WHERE attributes)
- result_where (Output of modified WHERE query on database)
- output (Final output on database)

Due to the number of intermediate steps, some adjustments in the evaluation process needed to be made when comparing SQL queries returned in the intermediate steps. Executing the SQL query on the database and comparing the result with the expected outcome is the most widely used approach in the evaluation. However, sometimes initial SQL queries cannot be executed because of the data type of a respective column. For example, a column that contains numbers might be stored as a string. Hence, potentially using a comparing operator like "<" would be inappropriate since the type of the column would be a string. An example would constitute the comparison "many > 1". In such a case, although the schema level information was incorporated, the initial SQL query is not executable leading to an Error message. Therefore, comparing the two non-executable

SQL queries regarding the output is not plausible as both queries are erroneous. The other possibility, namely comparing syntactic equality, can be erroneous as two distinct queries can have the same meaning semantically but employ different notations. Hence as a solution, the error message is compared. A potential error message in this could be *"ERROR: operator does not exist: text < integer"*. If the error messages are equal for two distinct SQL queries, it is therefore assumed this intermediate step is not erroneous.

This method allows for the visualization of the step at which errors occur, thereby helping to identify and understand the bottlenecks in the pipeline. Additionally, the erroneous intermediate steps were listed per category as described in Section 4.2, enabling a more thorough analysis.

4.4.1 Intermediate Steps Evaluation

For each predicate calculus query of the manually curated test set, the run was conducted three times. The following plots display the error counts for each step. Each respective subplot deals with one category, while the lower-right plot shows the total error counts. The x-axis represents the intermediate step at which the respective error occurs. The title is a legend describing the encoding of the position of the x-axis to the name of the intermediate step. At step 7, highlighted in blue, the output matches the defined ground truth and is therefore correct. It is important to note that an intermediate step is highlighted in red if it doesn't match the intermediate step of the correct result. This, however, does not mean that the results evaluate to zero in terms of precision, recall, and F1-score. Instead, it indicates that the output differs from the ideal output, without specifying the extent of the difference.

Figure 4.1 shows the error distribution for the Gemini-1.5-Flash version, while the Figure 4.2 shows the error distribution leveraging Gemini-2.0-Flash. The results for the Gemini-1.5-Flash version can be considered promising, as a relative majority of the runs are entirely correct. However it is notable that the errors are distributed across all categories indicating a variety of error sources. Especially high is the error count for semantic_list_where and initial_sql_query_join highlighting the wide distribution of the possible errors. Going on, for the Gemini-1.5-Flash version the correct results strongly outweigh the incorrect results except for the category "Inconsistencies". This suggests that this particular LLM experiences challenges with inconsistencies and varying scales, resulting in reduced performance. However, this does not imply that the results are entirely incorrect, but rather that they fall short of ideal accuracy.

In comparison to the Gemini-1.5-Flash version the Gemini-2.0-Flash version in Figure 4.2 produces much better results. A majority of runs is successful. The application of different LLMs using the same pipeline indicates strong performance differnce. Therefore, it can be emphasized that deploying more powerful models is essential for achieving optimal results.

The sources of errors are fairly widespread. No clear conclusions can be drawn about a specific point of improvement due to the broad distribution of errors. Positions 1 and 5,

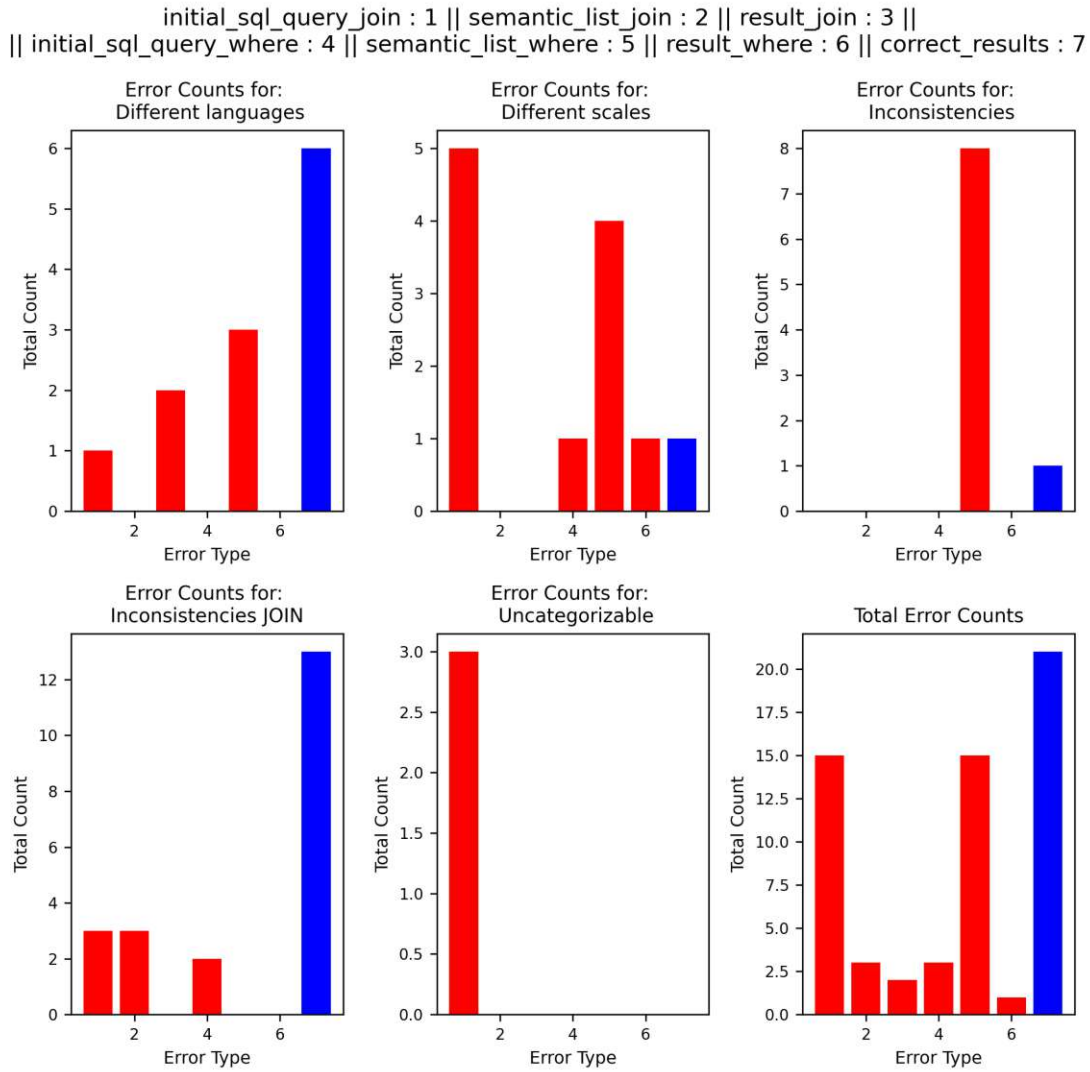


Figure 4.1: Error Distribution for Test Set for Gemini-1.5

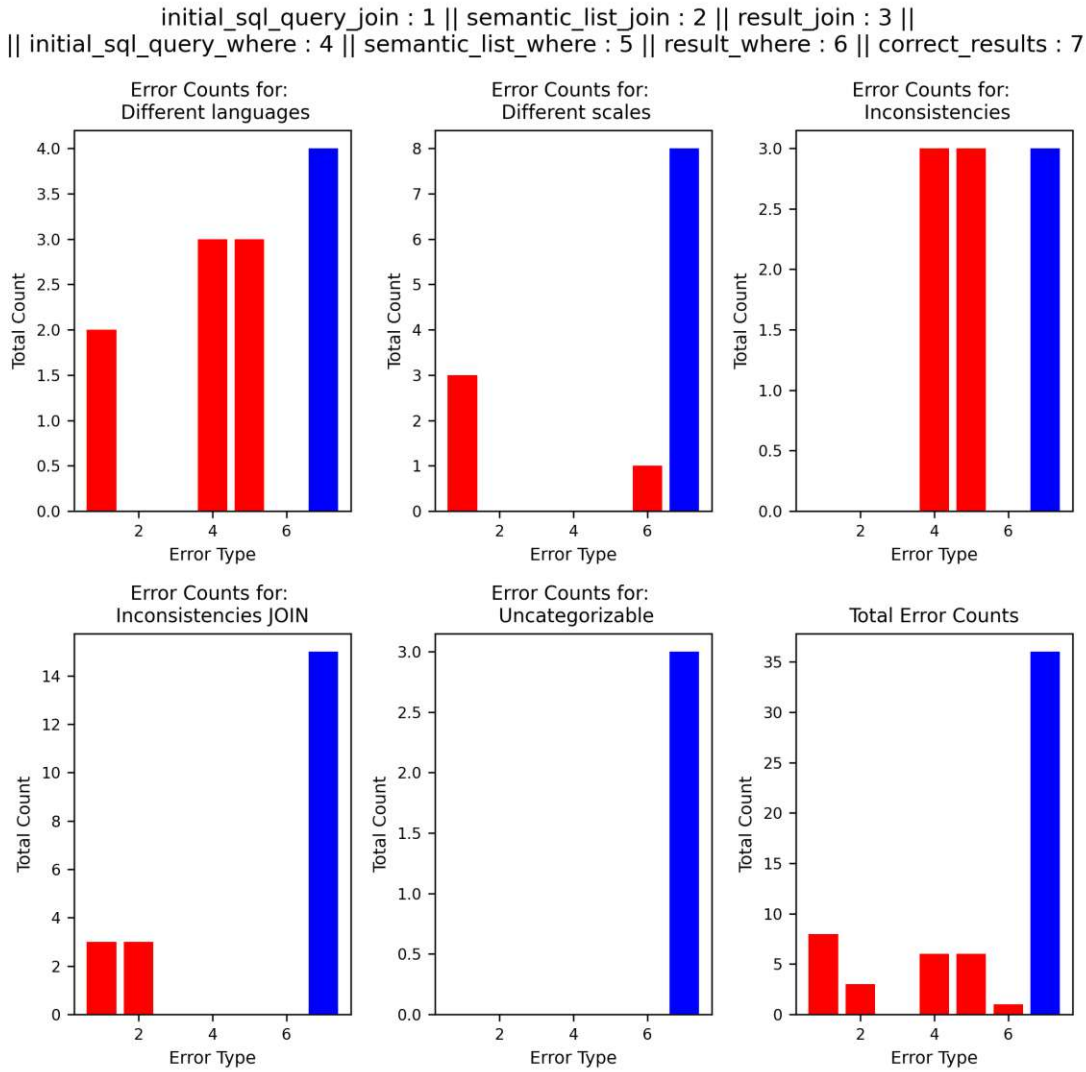


Figure 4.2: Error Distribution for Test Set for Gemini-2.0

can be singled out as a strong source of error for the Gemini-1.5-Flash version. However, for the more advanced Gemini version, errors occur at nearly all intermediate stages in lower numbers, but at comparable levels between the intermediate steps themselves. Therefore, these deviations must be accepted as part of the pipeline's design and cannot be entirely eliminated.

4.4.2 Key Performance Indicator Results

Building on the observations of the Section 4.4.1 the KPIs are calculated for the three runs for each predicate calculus expression. For that, the runs for both the Gemini-1.5-version along with the Gemini-2.0-version are compared. The respective results are also compared with the hard binding. The hard binding refers to the execution of the initial query without adjusting the query for both the *WHERE* and *JOIN* statements. The KPIs for soft binding for the Gemini-1.5-version are shown in Figure 4.5, while the KPIs for the hard binding can be seen in Figure 4.6. Regarding the Gemini-2.0-version the KPIs for the soft binding are displayed in Figure 4.3 and for the hard binding in Figure 4.4. It is important to note that this comparison between soft binding and hard binding is not entirely even. The dataset was deliberately designed to incorporate various forms of noise, thereby posing significant challenges for the hard binding. The objective of this comparison is to verify the superior performance of the soft binding method on datasets characterized by a significant degree of noise.

The results indicate that the metrics for soft binding significantly outperform those for hard binding. Specifically, for the Gemini-2.0-version, soft binding achieves a precision greater than 0.7 and a recall of 0.68, yielding an F1-score of 0.69, which suggests a decent balance between precision and recall. In contrast, hard binding, as anticipated, exhibits considerably lower performance, with an average precision and recall of 0.18. Especially the bad precision can be attributed to the nature of the hard binding procedure itself. In some instances for the hard binding, th initial query is non-executable, often due to data inconsistencies, such as when a numerical column is represented as text. Therefore, relying solely on schema data for the initial query generation may result in erroneous queries. Consequently, when no results are returned, all KPIs are assigned a value of zero, which accounts for the relatively poor performance of hard binding, even for precision.

Furthermore, the performance difference between the models is substantial. The F1-score utilizing the soft binding for the Gemini-1.5-version is 0.48, compared to 0.69 for the Gemini-2.0-version. It is also noteworthy that for both models, the precision and recall values are within 0.10 of each other, indicating a relatively balanced performance. However, for the hard binding procedure, the metrics for the Gemini-1.5-version are somewhat lower than those of the Gemini-2.0 version. This discrepancy can be attributed to the fact that the hard binding procedure consists solely of the SQL generation step, and notably lacks the generation of a semantic list.

In conclusion, the soft binding procedure yields promising results, with an F1-Score just below 0.7 for the superior model. Additionally, the LLM-version demonstrates a

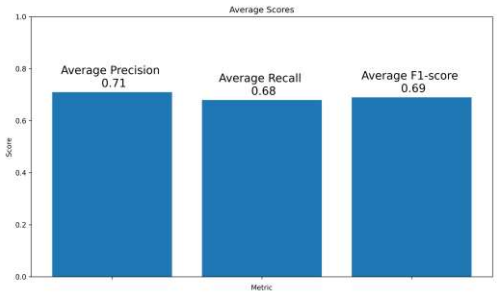


Figure 4.3: Average Metrics for Gemini-2.0-Flash for Soft Binding

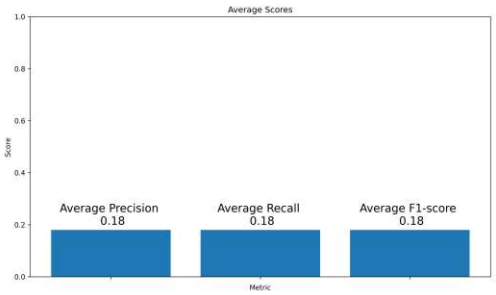


Figure 4.4: Average Metrics for Gemini-2.0-Flash for Hard Binding

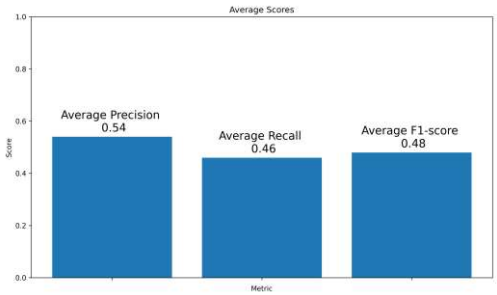


Figure 4.5: Average Metrics for Gemini 1.5-Flash for Soft Binding

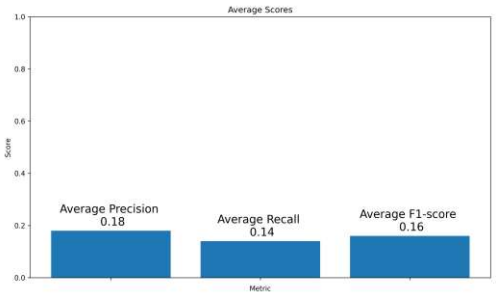


Figure 4.6: Average Metrics for Gemini 1.5-Flash for Hard Binding

significant impact on the metrics, emphasizing the importance of leveraging the more advanced model whenever feasible.

4.4.3 Token Cost and Execution Time

The following Table 4.1 presents the average tokens and execution time for each query. An analysis of the table reveals, as expected, that the hard binding procedure, which relies solely on initial query generation, utilizes significantly fewer tokens and fewer calls. Additionally, the average execution time is substantially shorter compared to the soft binding procedure, as expected. While for the soft binding the different models show similar token usage, it is noteworthy that the execution time for the more advanced Gemini-2.0 model is lower than that of the Gemini-1.5 version. This observation suggests that more advanced models can improve KPIs while also decreasing execution time.

It is to be noted, that the amount average tokens input values are much higher compared to the output tokes. This can be explained as there are several few-shot prompts, including examples.

Table 4.1: Computational Cost

Config.	Avg. Input Tokens	Avg. Output Tokens	Avg. Total Tokens	Avg. Total Calls	Average Time
Gemini 2.0 + Soft	3533.77	608.21	4143.74	8.89	0.38
Gemini 2.0 + Hard	274.02	36.79	310.81	1.00	0.07
Gemini 1.5 + Soft	3384.19	477.47	861.67	8.26	0.46
Gemini 1.5 + Hard	268.07	39.93	308.00	1.00	0.09

4.5 External Dataset Evaluation

In this section, we discuss the evaluation process using external datasets. Subsection 4.5.1 addresses the challenges encountered during the evaluation design. Subsection 4.5.2 covers the additional metrics explored. Next, the evaluation of the *JOIN* operation is presented in Subsection 4.5.3, followed by the evaluation of the *WHERE* clause in Subsection 4.5.4.

4.5.1 Evaluation Challenges

Evaluating the proposed approach presents substantial hurdles. This evaluation process is not straightforward, particularly given the current absence of datasets that truly capture the nuances of residual noise we aim to address with predicate calculus expressions as input. To the best of our knowledge, no existing resources fully encompass these specific characteristics. Moreover, the inherent diversity of comparison operators applicable within our pipeline introduces a significant evaluation challenge. However, the use of self-constructed datasets alone is insufficient for a robust evaluation. Despite the drawbacks presented, datasets related to similar tasks, such as entity matching or classification, exist and can be adapted to assess the proposed approach. It is important to note that such comparisons may not be entirely equitable, as many of these systems are specifically trained for a singular task, whereas the proposed solution is a general-purpose SQL framework designed to assist users in identifying and mitigating residual noise.

4.5.2 Metrics

For a rigorous evaluation, it is crucial to consider metrics beyond precision, recall, and F1-score, as these do not fully capture semantic and structural similarities between model outputs and the ground truth. To address this, the Bilingual Evaluation Understudy (BLEU) metric [PRWZ01] is employed, which quantifies textual overlap based on n-gram matching. By comparing outputs with ground truth, BLEU serves as an automated approximation of human evaluation, providing a numerical estimate of textual similarity. This enables a more refined assessment, particularly in scenarios where exact syntactic matches are not required, but semantic consistency remains crucial. Consequently, a correctly predicted output with a lower BLEU score demonstrates the pipeline's capability

to generate accurate predictions even when textual similarity is relatively low, highlighting its robustness in handling variations in expression.

We compare database outputs. Therefore, a best-average BLEU score is calculated. It can be calculated by taking the max BLEU score for each ground truth row to all of the output row. In a final step, the BLEU score is averaged for all ground truth rows. Additionally, the BLEU score depends on the the length of the n-grams. The formula for this BLEU-score is:

$$\text{Average Best BLEU} = \frac{1}{m}\sum_{i=1}^{m}\max_{j}\left(\text{BLEU}(C_i, R_j, n)\right) \tag{4.4}$$

Where:

- m is the number of ground truth rows.
- C_i is the i-th ground truth row.
- R_j is the j-th output row.
- $\text{BLEU}(C_i, R_j, n)$ is the BLEU score between ground truth row C_i and the output row R_j considering an n-gram.
- The function $\max_j\left(\text{BLEU}(C_i, R_j, n)\right)$ yields the maximum BLEU score for each ground truth row when compared against all output rows.

The concept of BLEU score for referring to tuples is primarily derived from Paganelli et al. [PBGF22].

4.5.3 SQL JOIN Performance Evaluation

It was possible to make use of the following computational resources for conducting more extensive experiments. The system used is running Ubuntu 24.04.2 LTS, with a hardware configuration that includes a 13th Gen Intel(R) Core(TM) i7-13700K processor, 32 GB of RAM, and an NVIDIA GeForce RTX 4070 GPU with 12 GB of VRAM. Additionally, the LLM employed for computation is the Llama-3.2-3B model.

To rigorously evaluate the performance of the JOIN operation within the proposed pipeline, we select an entity matching dataset. Specifically, the DBLP-ACM dataset [Rah25] is chosen. It contains information about various articles published by the Association for Computing Machinery (ACM), a major organization in the field of computer science. The dataset consists of two separate files, namely “ACM.csv” and “DBLP2.csv”, which provide details about the respective attributes. Additionally, a mapping file, “DBLP-ACM_perfectMapping.csv”, links the corresponding matches. The dataset comprises the following attributes:

- **id**: The unique identifier of the paper, represented in various formats.
- **title**: The title of the research paper.
- **authors**: The list of authors who contributed to the paper.
- **venue**: The conference or journal where the paper was published.
- **year**: The year in which the paper was published.

Using the provided solution file "DBLP-ACM_perfectMapping.csv", matches are identified through their respective identifiers. To facilitate comparison, all relevant columns, title, authors, venue, and year, are concatenated. Due to computational constraints, a subset of 100 matches is selected for evaluation. From the two dataset files, two separate tables are created. A JOIN operation is then performed on the concatenated column between these tables. Since each table contains 100 unique values, the process results in a total of 10,000 comparisons. The ground truth consists of correctly matched entities. The input is chosen to be an initial SQL query, which joins the two distinct tables based on the aggregated column :

```
SELECT * FROM unique_right JOIN unique_left
ON unique_left.aggregate = unique_right.aggregate;
```

For this evaluation, the KPIS are precision, recall, and F1-score. Ideally, out of the 10,000 comparisons, only 100 should be correct matches, while the remaining 9,900 should be true negatives.

The evaluation is conducted using the three approaches laid out in the previous chapter in Section 3.6. First, the standard zero-shot prompting method is applied. Second, the blocking procedure is incorporated, utilizing a two-step approach with threshold values of 0.5, 0.625, 0.75, 0.875, 0.9, 0.95, 0.975 and 1.0. Lower thresholds are not considered, as results do not significantly differ from the lowest selected value of 0.5. Third, an approach relying solely on embeddings is evaluated. Additionally, the BLEU scores and runtime performance are analyzed.

Figures 4.7 and 4.8 display the results obtained from the execution. The x-axis represents the threshold, while the y-axis indicates the corresponding score. The solid lines, each in a distinct color, correspond to either the embedding-only approach or the two-step approach. In contrast, the dotted lines, which remain unaffected by the threshold, represent the zero-shot approach.

Figure 4.7 displays the comparison of the embedding-only approach as well as the zero-shot evaluation. Specifically, the results highlight the trade-off between precision and recall across various threshold settings for the embedding-only method. As the threshold increases, precision generally improves, while recall decreases. However, at a threshold

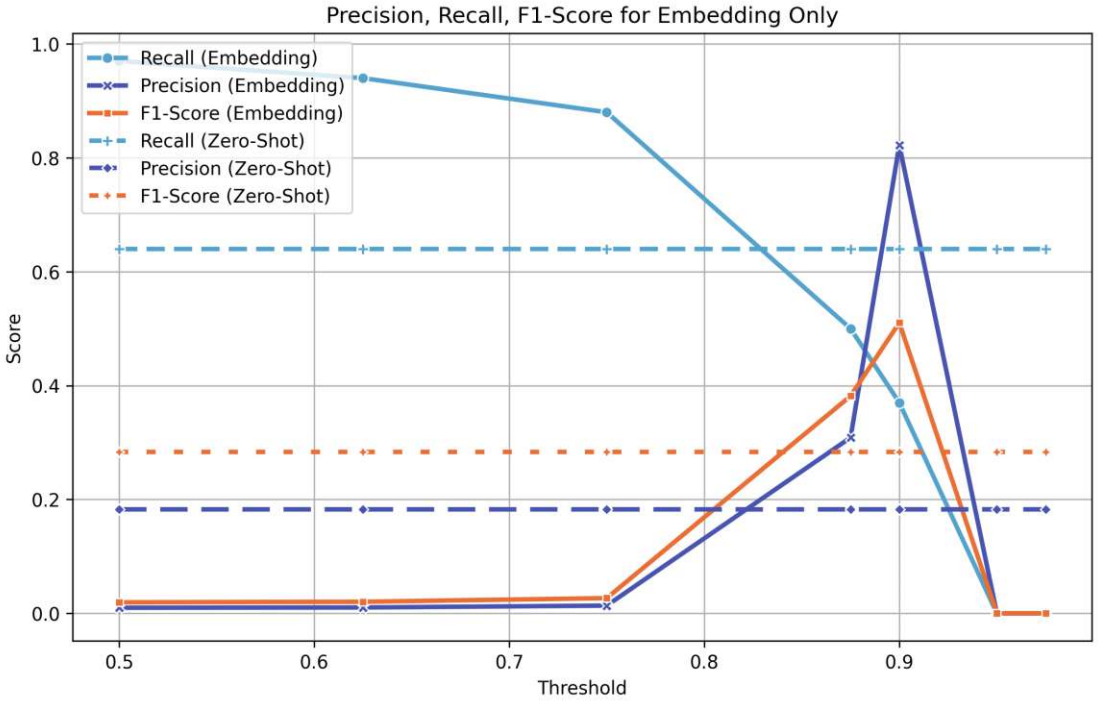


Figure 4.7: Recall, Precision, and F1-Score for Embedding-Only and Zero-Shot Prompting

of 0.95, both precision and recall drop to zero, as the embedding stage filters out all outputs at this level, effectively blocking all records. The highest observed F1-score of approximately 0.5 for the embedding-only approach occurs at a threshold of 0.9. Furthermore, for thresholds below 0.75, precision approaches zero due to the high number of true negatives, which also diminishes the F1-score. Notably, for the embedding-only method, a precision spike of approximately 0.8 can be observed at a threshold of 0.9. This spike in precision is paired with a substantially lower decrease in recall, leading to a higher F1-Score. For thresholds higher than 0.9, all metrics drop to 0, as no records can pass due to the chosen threshold.

For the zero-shot prompting method, a F1-score of approximately 0.28 can be considered satisfactory, particularly given the large number of comparisons involved. Notably, the precision is slightly below 0.2, while recall is much higher. Furthermore, the highest F1-Score was not achieved by zero-shot prompting but by the embedding-based approach. In this case, the outcome underscores the effectiveness of leveraging embeddings in the soft binding approach. In summary, the embedding-only method outperforms zero-shot prompting for the tresholds 0.875 and 0.9. However, the performance is highly dependent on the particular entities being compared. Additionally, accurately estimating an optimal threshold remains challenging. Based on the observations in this study, a reasonable range for the threshold appears to lie between 0.75 and 0.95.

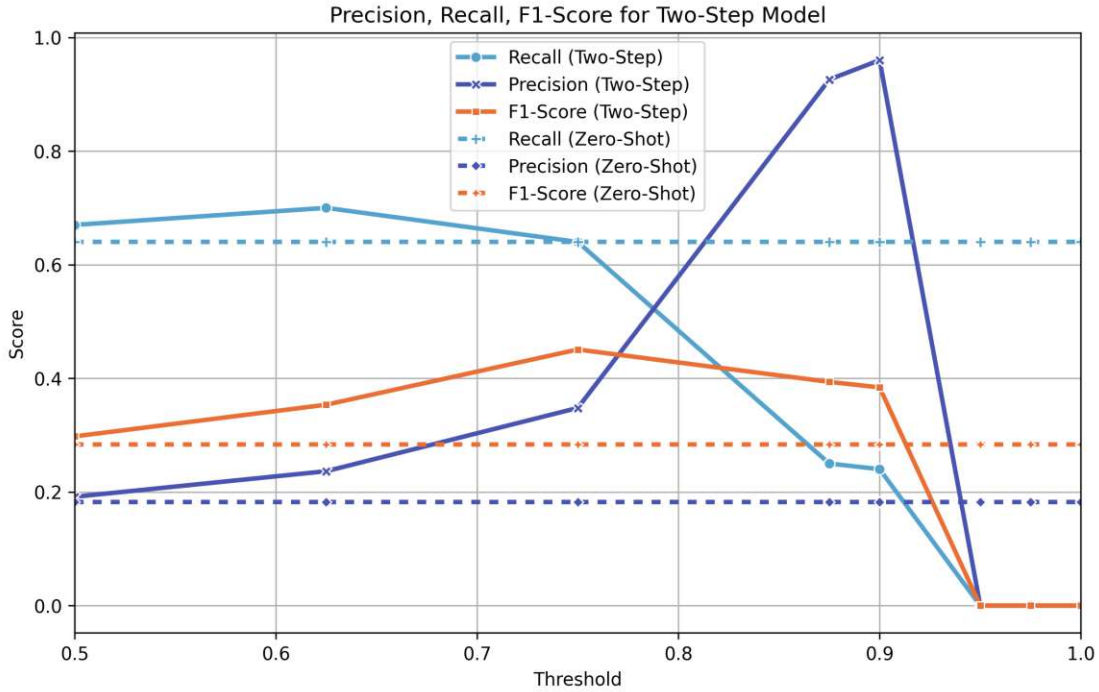


Figure 4.8: Recall, Precision, F1-Score for Two-Step and Zero-Shot Prompting

Figure 4.8 presents the results for the two-step model, which incorporates the use of embeddings as a blocking stage followed by a zero-shot prompting approach. As illustrated, the F1-Score for the two-step model is consistently higher than that of the zero-shot prompting method except for thresholds above 0.95. Notably, for the two-step model for a threshold of up to 0.9, the precision score rises sharply before decreasing to zero at a threshold of 0.95. This indicates that while increasing the threshold can improve precision, higher thresholds also risk filtering out nearly all output records. Additionally, it is important to note that the two-step model achieves higher precision than the zero-shot prompting approach, while recall can be lower. This can be attributed to the more restrictive nature of the two-step model compared to the zero-shot prompting method. Therefore, a trade-off is evident: higher precision is achieved by the two-step model, whereas higher recall is generally attained, except for the threshold being below 0.75, by the zero-shot prompting method. The optimal threshold regarding the current example for the two-step model appears to lie between 0.7 and 0.8. Overall for this example, the two-step model outperforms zero-shot prompting in terms of the F1-score for thresholds of up to 0.9.

An additional comparison considers execution time. Figure 4.9 shows the execution times for the two-step model, the embedding-only approach, and the zero-shot prompting method. The two-step model is significantly more computationally expensive than the other two when the threshold is below 0.9. However, at higher threshold values, it blocks

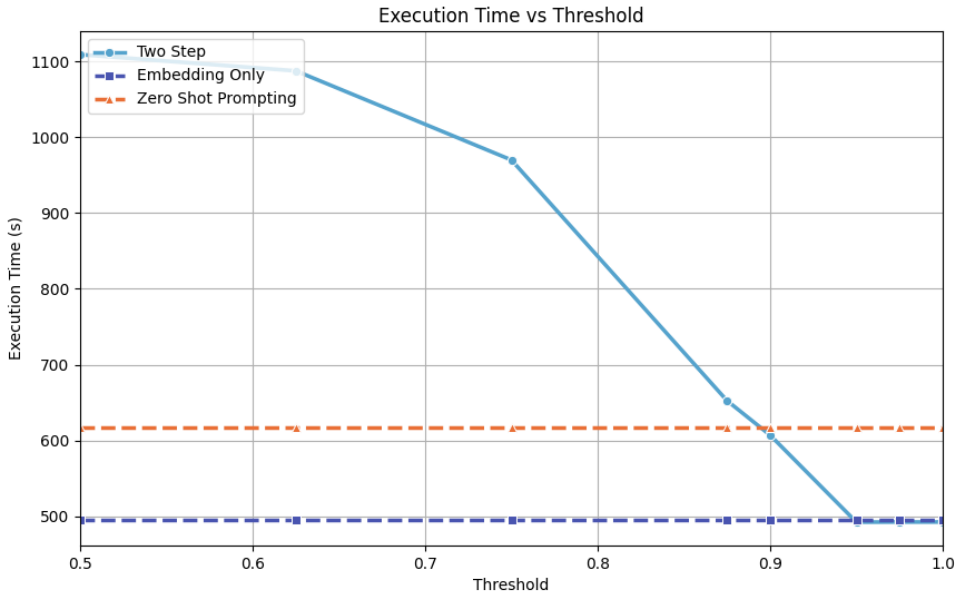


Figure 4.9: Execution Times for Embedding-Only, Two-Step and Zero-Shot Prompting

more candidate records, which reduces computation time. When all records are blocked in the the first blocking step, the execution speed of the two-step model aligns with that of the embedding-only model, as no candidates are passed towards the LLM for zero-shot prompting. Notably, the zero-shot prompting method exhibits a computational cost higher than the embedding-only model, placing it in the middle of the comparison. For threshold values below 0.9, the execution time for two-step model is higher than that of the zero-shot prompting method. In summary, the key observation is that increasing the threshold value leads to a higher number of entities being blocked in the two-step approach. Hence, execution time decreases, leading to computational cost savings. This effect is particularly pronounced at higher threshold values, where the two-step approach demonstrates significant savings compared to the zero-shot prompting technique, thereby achieving one of the intended outcomes of this method.

Finally, the evaluation of BLEU scores serves as a key indicator of how closely the predicted values align with the ground truth. The formula for the BLEU score used in this study is given in Equation 4.4. A high recall should therefore be reflected in both the BLEU1 and BLEU2 scores. As shown in Figure 4.10, for thresholds below 0.75, the BLEU1 score consistently remains highest for the embedding-only model. For thresholds below 0.6, the BLEU1 scores for the zero-shot and two-step models are comparable. The BLEU1 score of the two-step model decreases rapidly starting from a threshold of 0.75. At threshold values around 0.9, the two-step and embedding-only models converge, as the embedding filtering process dominates. Overall, this indicates that leveraging prompting

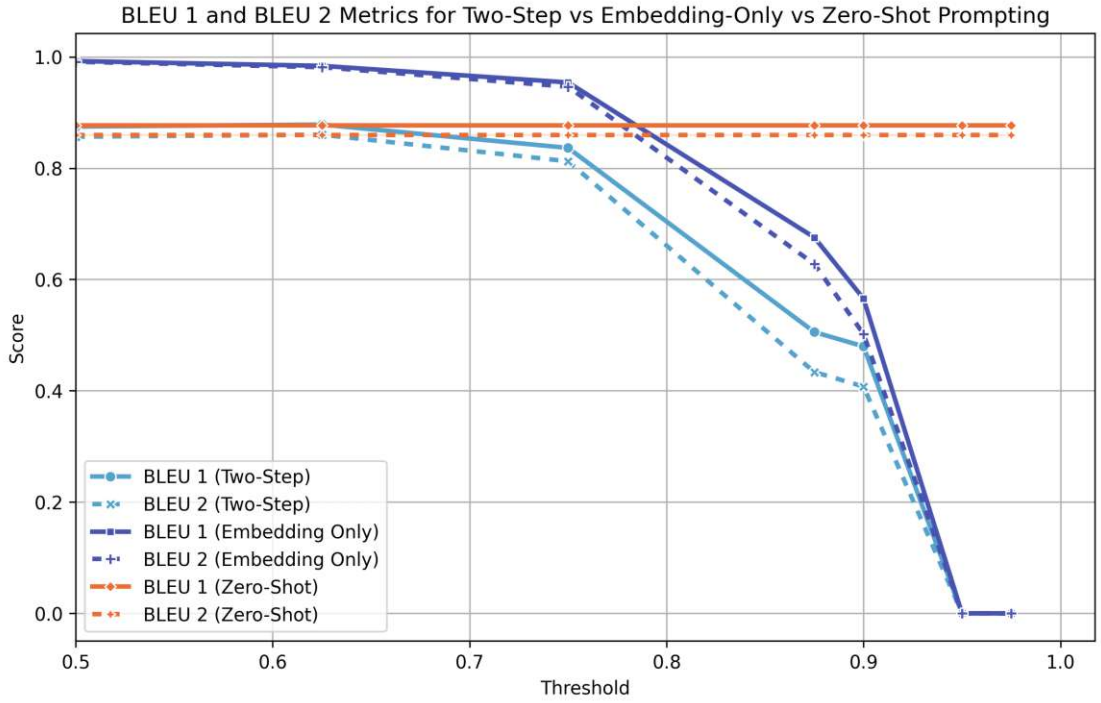


Figure 4.10: BlEU1 and BLEU2 Score for Embedding-Only, Two-Step and Zero-Shot Prompting

in the two-step and zero-shot prompt approach yields more diverse results in terms of text similarity as the BLEU metrics are consistently lower. Additionally, the BLEU2 score exhibits a similar trend to BLEU1, but within the range of 0.75 to 0.9, it remains notably lower than the BLEU1 score.

In conclusion, threshold-based filtering demonstrates reasonable performance for the JOIN operation. However, it is important to note that the two-step model yields inferior results when compared to the embedding-only approach for certain thresholds. For this example, although the two-step approach incurs a higher execution time, its performance metrics are superior to those of the zero-shot prompting approach for a treshhold up to 0.9, achieving higher F1-scores with a maximum of 0.45. Whether this trade-off between execution time and improved accuracy is acceptable ultimately depends on the specific requirements and preferences of the user or application. Finally, a trade-off between precision and recall was observed up to a threshold of 0.9 when comparing the two-step model with the zero-shot model. Increasing the threshold was associated with a decrease in recall, while precision either improved or plateaued. The selection of the optimal threshold is ultimately at the discretion of the user.

4.5.4 WHERE and WHERE NOT Evaluation

Additionally, the `WHERE` clause also was evaluated on a larger dataset. For this purpose, we select a product classification dataset. This selection enables a thorough investigation of the filtering operation using the `WHERE` clause. The dataset used for this experiment is obtained from Kaggle [Moh25]. It contains data sourced from the online retailer "JioMart". The primary attributes relevant to this analysis are the `items` column and the `category` column.

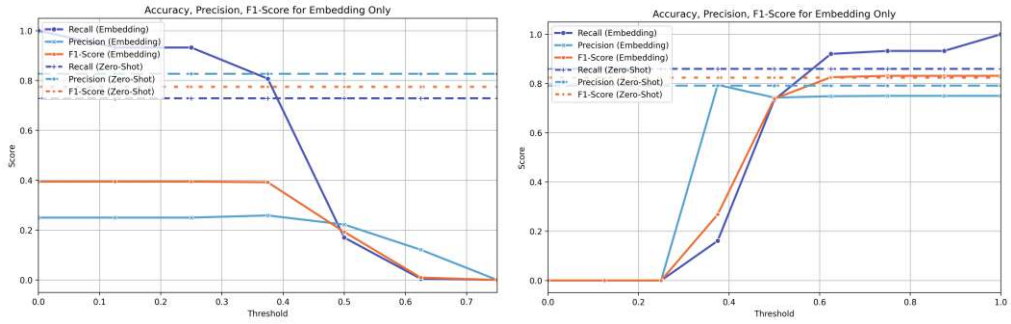
The task within the scope of this evaluation is to retrieve the category of each item listed in the category *items*. The `items` column contains information such as the item "Venus Vector Hi Speed Ceiling Fan V1200 (White)" belonging to the category "Electronics". The original dataset contains 162,313 rows along with 5 columns. Initially, there were 6 categories present, namely: "Groceries", "Home & Kitchen", "Fashion", "Electronics", "Beauty", and "Jewellery". However, initial experimentation showed that the categories "Home & Kitchen" and "Groceries" are quite ambiguous and are disproportionately assigned to other categories. For this reason, along with the large size of the dataset, we select a subset containing 1,000 rows in order to better evaluate and process the results. The four remaining categories, "Fashion", "Electronics", "Beauty", and "Jewellery", are equally distributed in the dataset. Based on the item names, the pipeline is tasked with identifying the correct category. This evaluation is carried out across all four investigated columns. Also, the verbal translation of the "=" operator was changed to "belongs to the category". The corresponding SQL query used is:

```
SELECT * FROM jio_smart
WHERE jio_smart.items = 'current_category'
```

Again we test the three methodologies of embedding-only, two-step and zero-shot prompting for various thresholds. This is performed for both the equality operator "=" and the inequality operator "!=". It is important to note that for the "!=" operator, the embedding comparison is inverted from ">" to "<" in order to account for negation. Additionally, instruction prompts are used to better facilitate the process. For "=", the instruction prompt is: *"You are a machine returning boolean values. Given the categories "Fashion", "Electronics", "Beauty", and "Jewellery', each object belongs to exactly one. Answer 'yes' if an item belongs to the stated category, otherwise answer 'no'. Validate the following statement using 'yes' or 'no' only!"*

For "!=", the instruction prompt is: *"You are a machine returning boolean values. Given the categories "Fashion", "Electronics", "Beauty", and "Jewellery", each object belongs to exactly one. Answer 'yes' if an item does *not* belong to the stated category, otherwise answer 'no'. Validate the following statement using 'yes' or 'no' only!"*

First, we compare the KPIs for the different methods between the embedding-only approach and the zero-shot prompting approach. The corresponding figures, namely Figure 4.11a and Figure 4.11b, present the KPIs in relation to the threshold values. It can be observed that for the "WHERE" clause, as the threshold increases, the KPIs



(a) WHERE: Embedding-Only and Zero-Shot Prompting

(b) WHERE NOT: Embedding-Only and Zero-Shot Prompting

Figure 4.11: Embedding-Only and Zero-Shot Prompting for Both WHERE and WHERE NOT

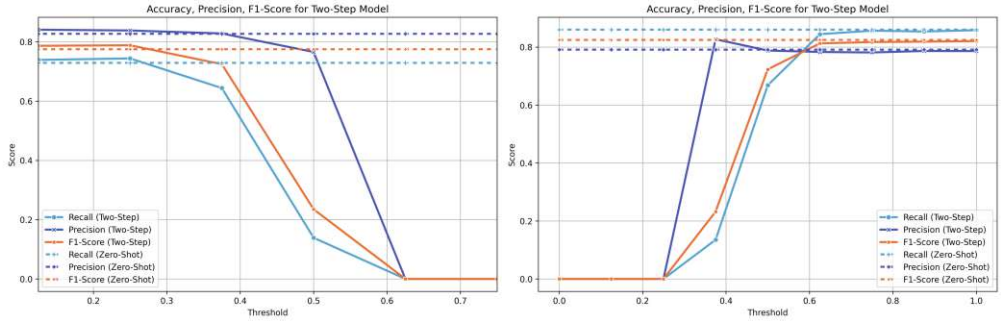
generally decrease, whereas for the "WHERE NOT" condition, the KPIs rise. This can be explained by the fact that, with increasing thresholds, the “WHERE” clause blocks more candidates, whereas the “WHERE NOT” clause blocks fewer entities due to the differing nature of the comparison. Notably, when comparing with the JOIN evaluation, a smaller trade-off between precision and recall is observed. However, the main trend is that the F1-score either remains constant or continuously decreases for the “WHERE” clause, whereas it either increases or remains constant for the “WHERE NOT” clause. For the “WHERE NOT” filtering condition, precision is relatively high for higher thresholds, as nearly all instances pass through, resulting in a precision close to 0.75. However, the precision for the zero-shot prompting approach exceeds this value. For the “WHERE NOT” condition, the F1-score is comparable for both approaches at higher thresholds. This suggests that in this scenario applying a threshold does not result in effective blocking for this operation. In contrast, for the "WHERE" filtering condition, precision drops as the threshold increases. The same trend is observed in Figure 4.11b. It is important to note that for the "WHERE NOT" condition, the logic is reversed, meaning that higher thresholds block fewer entities. In summary, the overall results indicate that the zero-shot prompting approach outperforms the embedding-only phase for the "WHERE" condition and yields similar performance for the "WHERE NOT" condition for this example.

The evaluation of the two-step model against the zero-shot prompting method reveals notable differences in performance. As illustrated in Figure 4.12b, the KPIs for the “WHERE NOT” condition are considerably lower for the two-step model compared to the zero-shot prompting approach at lower threshold values. However, as the threshold increases, the KPIs of the two-step model converge with those of the zero-shot prompting method, with the threshold where they become closely aligned occurring at approximately 0.6.

Similarly, for the evaluation of the “WHERE” clause, shown in Figure 4.12a, the two-step

model exhibits KPI values comparable to the zero-shot prompting method at lower threshold values, up to around 0.4, where nearly no candidate records are blocked. Beyond this point, as the threshold increases, the KPIs for the two-step model decrease significantly.

The analysis for this reveals that, in terms of F1-score, the zero-shot prompting approach either outperforms or is closely comparable to any method that employs threshold-based filtering.



(a) WHERE: Two-Step and Zero-Shot Prompting

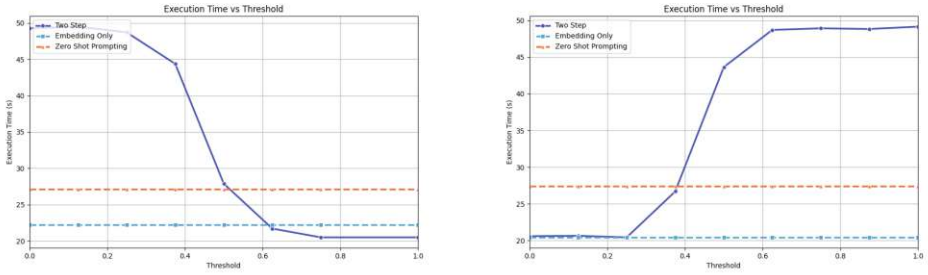
(b) WHERE NOT: Two-Step and Zero-Shot Prompting

Figure 4.12: Two-Step and Zero-Shot Prompting for Both WHERE and WHERE NOT

Figure 4.13 presents the execution times for the three methods—two-step, embedding-only, and zero-shot prompting—under both filtering conditions: "WHERE" and "WHERE NOT".

For lower thresholds in the condition, respectively higher thresholds in the "WHERE NOT" condition, the two-step model exhibits higher execution times compared to the embedding-only and zero-shot prompting methods. As the blocking threshold increases, respectively decreases for the "WHERE NOT" clause, as can be seen in Figure4.13b, execution time decreases, eventually matching the embedding-only model at comparably high thresholds, respectively comparably low thresholds for the "WHERE NOT" clause.

In this evaluation, the execution time of the two-step and embedding-only models intersects at approximately 0.6 for the "WHERE" clause and slightly above 0.2 for the "WHERE NOT" condition.

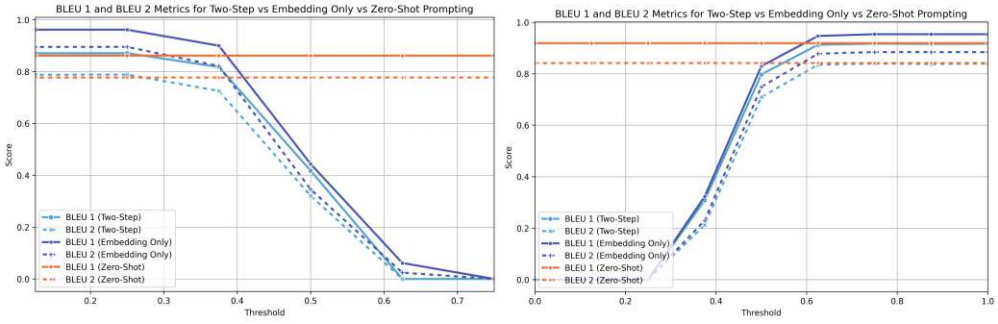


(a) WHERE: Execution Time (b) WHERE NOT: Execution Times

Figure 4.13: Execution Times for WHERE and WHERE NOT

Overall, for the evaluation of the WHERE clause in this example, threshold-based filtering is not practical. While recall is higher for the embedding-only approach than in the zero-shot prompting approach for lower thresholds in the "WHERE" condition, respectively higher thresholds in the "WHERE NOT" condition, precision was never significantly better. Moreover, as shown in Figure 4.13a and Figure 4.13, execution time in these cases is considerably higher than that of the zero-shot prompting approach.

Additionally, BLEU1 and BLEU2 scores offer valuable insights into the evaluation process. The left Figure 4.14a displays these metrics for the "WHERE" clause across all three methods, while the right Figure 4.14b illustrates the corresponding results for the "WHERE NOT" clause, with respect to the threshold value. It is evident that the BLEU scores for both the two-step approach and the embedding-only approach exhibit similar trajectories for BLEU1 and BLEU2, highlighting the comparable performance between these methods. As the threshold increases for the "WHERE" clause, or decreases for the "WHERE NOT" clause, both BLEU1 and BLEU2 scores decline. This decline can be attributed to a decrease in recall, leading to the exclusion of correct records, thus reducing the overall BLEU scores. Of particular note, the zero-shot prompting approach achieves a notably high BLEU1 score of approximately 0.9 for both clauses.



(a) WHERE: BLEU Metrics (b) Where not: BLEU metrics

Figure 4.14: WHERE and WHERE NOT: BLEU Metrics

Overall, it can be stated that for both clauses, the pipeline produces good results in terms of precision, recall, and F1-score. With results exceeding 0.8 for the F1-score, the zero-shot prompting approach can be considered effective.

However, it is important to note that the embedding method does not appear to perform well . Results comparable to the zero-shot prompting approach, which does not leverage embeddings, in terms of KPIs can be observed for lower thresholds in the “WHERE” operation or higher thresholds in the “WHERE NOT” clause. However, at these thresholds, the execution time of the zero-shot prompting remains lower and is therefore superior.

Ultimately, for the "WHERE" and "WHERE NOT" clause for the following example, threshold-based blocking is not recommended, while the zero-shot prompting approach yields good results. Hence, it can be employed without the need to recommend a specific threshold range.

CHAPTER 5

Limitations and Discussion

In this chapter, we discuss the limitations of the thesis in Section 5.1 and perform the discussion in Section 5.2.

5.1 Limitations

In this subsection, we discuss the inherent limitations of the chosen approach and also the limitations associated with the evaluation.

In terms of limitations, it is important to highlight that both the auxiliary tables Approach 3.3.4 and the translation-based Approach 3.3.3 leverage the SQL framework. This has inherent advantages and also disadvantages. The advantage lies in SQL's status as a widely adopted querying language for relational databases [MMR23], underpinned by its computational efficiency and extensive optimization techniques [RIKJ24]. However, relying on the already existing SQL comparison operators means adjusting the query for the translation-based approach or writing an intermediary table for the auxiliarty table approach, complicating the initial query. This adds complexity to the original query and can lead to errors if the rewriting is not done correctly.

The translation-based approach is quite impractical, when there are many deviations from the original query. The additional statements expressed through *OR* and *CASE* clauses for a small number of corrections are a good visual representation for the user. However, in the case of highly noisy datasets, a substantial number of additional entities must be incorporated. Hence, the total expression can become extremely long, potentially leading to difficulty of comprehension for the end user, when trying to analyze the cause of the errors. The expression may grow significantly in length, making it difficult to analyze. This extended complexity can hinder the end user's ability to identify the root cause of errors when reviewing the query. For the current version of PostgreSQL there is no defined limit for the input prompt [The25]. Despite that, it can't be recommended to

make use of this method for larger deviations between the hard and the soft binding.

The auxiliary table strategy simplifies SQL query adjustment by adding an intermediate expression for each clause, requiring exactly one intermediate JOIN per modified clause. However, this incorporates writing and modifying the database while querying. However, in many real-world environments, users, especially developers, are often granted only read-only or very limited privileges on the live database. This follows the Principle of Least Privilege, which dictates that identities should only be permitted to perform the smallest set of actions necessary to fulfill a specific task [SdLJ18]. Consequently, users are often restricted from making modifications to production databases, thereby complicating the implementation of this approach. However on the other hand, the user retains the ability to access and modify the created table according to their preferences, should their judgment differ from that of the LLM.

Additionally, more powerful LLMs could not be investigated. A valuable comparison for evaluating extensive external datasets would have been to include results from a different Llama version with a higher amount of parameters. However, the employed machine lacked the capacity to load it and execute it. For instance, running a model such as LLaMA 7B typically demands approximately 28 GB of GPU memory to operate efficiently [Sam23]. Also, it is possible to leverage an API like for example with Gemini like in Section 4.4.2, however there are substantial access and rate limits leading to significant financial costs [Goo25d] when performing an extensive analysis. Furthermore, the maximum JOIN size examined in this study was 100×100, primarily due to significant hardware limitations and execution time constraints. For a more comprehensive analysis, higher computational resources and extended access time could be utilized to increase the scope of the work.

Furthermore, it is important to emphasize that the approach remains constrained by the syntactic limitations of SQL. Despite these promising results, the approach is still prone to various sources of error. These include inaccuracies in the initial query generation, the creation of the semantic list, and the subsequent adjustment of the query. Since these steps are highly interdependent, errors introduced at one stage may propagate throughout the entire process, thereby increasing the overall likelihood of failure.

Finally, the total evaluation of the approach was only possible in the limited fashion due to the lack of specific data. The evaluation was performed first on the manually curated dataset along with an analysis of the error stage. The final evaluation was performed on a entity matching and a classification dataset. However, the nature of general purpose LLMs is far wider and more targeted evaluation could be performed leveraging for example the “<” and “>” comparison operators on a bigger scale, provided that test such data were available.

5.2 Discussion

The proposed approach integrates the semantic capabilities of LLMs into SQL systems by introducing multiple methods for softening bindings in both JOIN and WHERE evaluations. A key challenge in this research domain is the lack of suitable benchmark datasets, necessitating the creation of a small, task-specific dataset. Within the SQL framework, two primary methods were proposed: the auxiliary approach and the translation-based approach. The auxiliary method is generally more suitable when dealing with substantial residual noise, as it provides flexibility in modifying the translation table and is reusable, allowing the user to re-execute the relevant queries as needed. In contrast, the translation-based approach is recommended only when residual noise is minimal. Additionally, the auxiliary method offers users a clearer overview of modifications compared to directly altering the initial query, enhancing interpretability and allowing the user to modify the translation table after processing by the pipeline to incorporate manual judgments.

Furthermore, the proposed architecture demonstrates a significant improvement in KPIs compared to hard binding, which solely relies on executing the initially generated query. This finding highlights the potential of the approach, as the pipeline led to enhanced results in terms of metrics. However, it is important to note that these methods are computationally expensive, resulting in higher execution times, which must be carefully considered in practical applications.

The final evaluation of the JOIN condition involved comparing three approaches: zero-shot prompting, embedding-only, and two-step. The F1-score reached up to 0.5 for the embedding-only method and a F1-score of up to around 0.45 for the two-step model. This can be considered a strong outcome in such a sparse positive environment. The importance of selecting an appropriate threshold was evident, with an optimal range identified between 0.75 and 0.95. Overall, incorporating a blocking stage reduced execution time for some threshold ranges, namely for thresholds above the threshold above 0.9. While the two-step method was computationally more expensive than the zero-shot approach, it resulted in a higher F1-score inside a reasonable threshold range. The two-step model performed better on precision, whereas in most cases the zero-shot prompting method had a higher recall. Notably, the two-step method achieved significantly higher precision, albeit at the cost of recall. Therefore, the choice of method depends on whether precision or recall is prioritized for a given application.

The evaluation of the “WHERE” and “WHERE NOT” filtering operations highlights the advantages of the zero-shot prompting approach over the two-step and embedding-only methods. The zero-shot method consistently performed better in F1-score and in terms of execution time. Also the zero-shot prompting results for the “WHERE NOT” condition produced similar results compared to the "WHERE" condition, indicating a strong generalization capability across various comparison operators. It is to mention that the two-step model first blocking procedure did not produce better results in terms of the F1-score, but was computationally more expensive. Execution time compared with the two-step model was also lower for the zero-shot prompting approach, making it more

efficient overall. In contrast, the embedding-only had a significantly worse precision than the zero-shot prompting approach indicating to be an ineffective filtering mechanism. Overall, the results indicate that threshold-based filtering is less effective, and zero-shot prompting is the preferred method for WHERE clause filtering, providing both higher accuracy and efficiency.

Hence, the implemented pipeline demonstrated satisfactory results, showcasing its potential to extend beyond traditional SQL queries. The pipeline's ability to semantically interpret the meaning of comparison operators, particularly in the context of data with residual noise, has been effectively demonstrated. However, it is important to consider the resource intensity associated with invoking the LLM, which necessitates attention to performance optimization.

The threshold-based filtering yielded favorable results for the JOIN evaluation, with an optimal threshold range identified between 0.7 and 0.9. Although the peak F1-score of around 0.5 was lower than that observed for the WHERE evaluation, the result remains reasonable given the large number of comparisons and the significant proportion of true negatives. In contrast, for the WHERE evaluation, the F1-score was substantially higher. However, in this case, threshold-based filtering proved ineffective.

Moreover, it is crucial to note that the JOIN operation inherently involves more comparisons, as it requires comparing each row from one dataset with every row from another, leading to a significant increase in the total number of comparisons. Consequently, threshold-based filtering is more essential for JOIN filtering in comparison to WHERE filtering. Furthermore, the results for the filtering operation demonstrate a respectable F1-score. In conclusion, the pipeline has proven to be effective, offering a promising approach for JOIN and categorization tasks.

CHAPTER 6

Conclusion and Future Work

This final chapter deals with the conclusion of the entire thesis in Section 6.1 and describes possible future work in Section 6.2.

6.1 Conclusion

The main contribution of this thesis was the development of a query processing pipeline designed to address the issue of residual noise in the context of *WHERE* and *JOIN* statements. To evaluate the proposed approach, a hand-crafted dataset was created, accounting for various types of residual noise, including multilingual support, handling of string representations of numbers, inconsistent data formats, and modifications of JOIN attributes.

Experimental results demonstrated that the *soft binding* strategy consistently outperformed *hard binding*, with particularly notable improvements when using the Gemini-2.0-Flash version, achieving an increased F1-score exceeding 50%. These findings highlight the effectiveness of the proposed pipeline in dealing with residual noise.

Going further regarding execution considerations, a blocking procedure leveraging the employed LLM was implemented, and its functionality and utility were thoroughly investigated. In the evaluation on the DBLP-ACM dataset [Rah25], the results demonstrated the efficacy of utilizing embeddings and zero-shot prompting. Given the sparse-positive nature of the dataset, achieving a F1 score exceeding 0.4 represents a notable success. Moreover, the benefits of incorporating embeddings through the LLM became evident, providing users with the ability to control the trade-off between recall and precision when applying the embedding-only and two-step approaches, provided they stay within a reasonable threshold range. As a general proposition, users can prioritize recall by selecting a lower threshold or emphasize precision by setting a higher threshold. However, it should be noted that this trade-off behavior is dependent on the specific characteristics

of the dataset and may not consistently generalize across all types of data distributions. This flexibility constitutes a powerful instrument for end-users, allowing them to adapt the approach to their specific requirements and objectives.

Additionally, the evaluation of the categorization dataset [Moh25] highlighted the strong classification capabilities of the employed LLM using zero-shot prompting, achieving a F1-score close to 0.8 for the "WHERE" clause and a F1-score of slightly above 0.8 for the "WHERE NOT" clause. Notably, comparable F1-scores were obtained for both "WHERE" and WHERE NOT" conditions, demonstrating the method's applicability across different types of comparison operators. These results further illustrate the model's ability to deliver strong classification performance without relying on pre-training or few-shot prompting.

In conclusion, the developed query processing pipeline provides substantial utility for end-users. It offers flexibility by allowing adjustments of threshold settings, the selection of alternative LLMs, and the customization of textual descriptions for comparison operators. This adaptability enables users to fine-tune the pipeline according to specific task requirements, thereby enhancing its applicability across a range of practical scenarios.

6.2 Future Work

There are several promising directions to extend the research presented in this thesis. One potential avenue is the re-implementation of query parameters directly within the querying process itself, which would eliminate the need for dynamic adjustments of SQL queries and streamline query generation. In the case of the re-implementation it would be important to implement a predicate calculus layer, to translate from the initial predicate calculus expression to a SQL query.

It could also be valuable to deploy the model in real user environments, allowing users to evaluate helpfulness of the generated suggestions. Both explicit feedback like user ratings and implicit feedback, such as user interactions or corrections, could be collected and utilized to fine-tune the underlying LLM, enabling more task-specific modifications. Furthermore, such deployment would allow for a comprehensive evaluation effectiveness of the proposed pipeline in real-world scenarios.

Additionally, similar to DB-GPT [XJS+23], a multi-turn conversational agent could be developed, enabling users to interact with the system iteratively rather than receiving query suggestions in a single-shot manner. Such an agent would allow users to handle each individual database operation step by step and request additional suggestions throughout the whole query process. While this approach may increase the complexity of interaction, it would also provide greater flexibility and enable the system to better support highly tailored and complex use cases.

Another promising approach for future work involves integrating additional SQL operations, such as GROUP BY, COUNT, and other advanced operators. This would expand the range of supported SQL queries, enhancing the system's flexibility and applicability

across more complex analytical tasks. By incorporating these operations, the system could better more complex query structures, ultimately improving user experience.

In conclusion, these proposed extensions offer significant opportunities to enhance the practicality and performance of the system, ultimately leading to more intuitive user interactions with databases in real-world applications.

Overview of Generative AI Tools Used

1. To help refine certain formulations in the thesis, the Writefull add-on for Overleaf (https://www.writefull.com/writefull-for-overleaf) was used occasionally. Its use was limited due to the daily limit on suggestions in the free version.
2. ChatGPT (version GPT-4) was also utilized in several instances to provide additional formulation suggestions.

List of Figures

List of Tables

List of Algorithms

Acronyms

ACM Association for Computing Machinery. 48

AI Artificial Intelligence. 8

AQE Automatic query expansion. 12

BKV Blocking Key Value. 13

BLEU Bilingual Evaluation Understudy. 47–49, 52, 53, 57

CoT Chain-of-Thought. 10–12

GNN Graph Neural Network. 10

HUMO Human and Machine Cooperation Framework. 13

KG Knowledge Graph. 4, 7, 8, 10, 11

KGs Knowledge Graphs. 7

KRR Knowledge Representation and Reasoning. 7, 8

LLM Large Language Model. xi, xiii, xv, 2–4, 7–12, 15, 17–24, 26, 29, 31, 35–37, 40–42, 45, 48, 52, 60–64

LLMs Large Language Models. 8, 11, 12, 14, 15, 42, 60

LSTM Long Short-Term Memory. 14

NLP Natural Language Processing. 1, 10

PLM Pretrained Language Model. 8, 10, 15

PRF Psuedo-Relevance Feedback. 12

QA Question Answering. 8, 11

RF Relevance Feedback. 12

SQL Structured Query Language. 2–4, 10, 11, 17–22, 24, 26, 29, 35, 41, 42, 45, 47–49, 59–62, 64

TF-IDF Term Frequency-Inverse Document Frequency. 13

Bibliography

[ABBC22] Tommaso Alfonsi, Luigi Bellomarini, Anna Bernasconi, and Stefano Ceri. Expressing Biological Problems with Logical Reasoning Languages. Technical report, 2022.

[AGAB23] Ankush Agarwal, Sakharam Gawade, Amar Prakash Azad, and Pushpak Bhattacharyya. KITLM: Domain-Specific Knowledge InTegration into Language Models for Question Answering. 8 2023.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[ANC+22] Berkeley R Andrus, Yeganeh Nasiri, Shilong Cui, Benjamin Cullen, and Nancy Fulda. Enhanced Story Comprehension for Large Language Models through Dynamic Document-Based Knowledge Graphs. Technical report, 2022.

[BBB+22] Luigi Bellomarini, Lorenzo Bencivelli, Claudia Biancotti, Livia Blasi, Francesco Paolo Conteduca, Andrea Gentili, Rosario Laurendi, Davide Magnanimi, Michele Savini Zangrandi, Flavia Tonelli, Stefano Ceri, Davide Benedetto, Markus Nissl, and Emanuel Sallinger. Reasoning on company takeovers: From tactic to strategy. *Data and Knowledge Engineering*, 141, 9 2022.

[BBB+24] Teodoro Baldazzi, Davide Benedetto, Luigi Bellomarini, Emanuel Sallinger, and Adriano Vlad. Softening Ontological Reasoning with Large Language Models. Technical report, 2024.

[BBC+23] Teodoro Baldazzi, Luigi Bellomarini, Stefano Ceri, Andrea Colombo, Andrea Gentili, and Emanuel Sallinger. Fine-tuning Large Enterprise Language Models via Ontological Reasoning. 6 2023.

[BG21] Nils Barlaug and Jon Atle Gulla. Neural Networks for Entity Matching: A Survey, 4 2021.

[BIPR12] Kedar Bellare, Suresh Iyengar, Aditya G. Parameswaran, and Vibhor Rastogi. Active sampling for entity matching. In *Proceedings of the 18th ACM*

SIGKDD international conference on Knowledge discovery and data mining, pages 1131–1139, New York, NY, USA, 8 2012. ACM.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. 5 2020.

[Bra88] Ronald J Brachman. The basics of knowledge representation and reasoning. *AT&T Technical Journal*, 67(1):7–24, 1988.

[BSG18] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The Vadalog system: Datalogbased reasoning for knowledge graphs. In *Proceedings of the VLDB Endowment*, volume 11, pages 975–987. Association for Computing Machinery, 2018.

[CCF+17] Zhaoqiang Chen, Qun Chen, Fengfeng Fan, Yanyan Wang, Zhuo Wang, Youcef Nafa, Zhanhuai Li, Hailong Liu, and Wei Pan. Enabling Quality Control for Entity Resolution: A Human and Machine Cooperation Framework. 9 2017.

[CEP+21] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. An Overview of End-to-End Entity Resolution for Big Data. *ACM Computing Surveys*, 53(6):1–42, 11 2021.

[CG08] Peter Christen and Ross Gayler. Towards Scalable Real-Time Entity Resolution using a Similarity-Aware Inverted Index Approach. *AusDM '08: Proceedings of the 7th Australasian Data Mining Conference - Volume 87*, 10 2008.

[Che22] Wenhu Chen. Large Language Models are few(1)-shot Table Reasoners. 10 2022.

[CKLM19] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What Does BERT Look At? An Analysis of BERT's Attention. 6 2019.

[Cla21] Vincent Claveau. Neural text generation for query expansion in information retrieval. In *ACM International Conference Proceeding Series*, pages 202–209. Association for Computing Machinery, 12 2021.

[CR02] William W. Cohen and Jacob Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 475–480, New York, NY, USA, 7 2002. ACM.

[CR12] Claudio Carpineto and Giovanni Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44(1), 1 2012.

[DBES09] Xin Luna Dong, Laure Berti-Equille, and Divesh Srivastava. Truth discovery and copying detection in a dynamic world. *Proceedings of the VLDB Endowment*, 2(1):562–573, 8 2009.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. 10 2018.

[DHX+22] Shizhe Diao, Zhichao Huang, Ruijia Xu, Xuechun Li, Yong Lin, Xiao Zhou, and Tong Zhang. Black-box Prompt Learning for Pre-trained Language Models. 1 2022.

[DRD+13] N Dalvi, V Rastogi, A Dasgupta, A Das Sarma, and T Sarlós. Optimal Hashing Schemes for Entity Matching. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 295–306, 2013.

[DTWP18] Mohamad Dolatshah, Mathew Teoh, Jiannan Wang, and Jian Pei. Cleaning crowdsourced labels using oracles for statistical classification. *Proceedings of the VLDB Endowment*, 12(4):376–389, 12 2018.

[dVKCC11] Timothy de Vries, Hui Ke, Sanjay Chawla, and Peter Christen. Robust Record Linkage Blocking Using Suffix Arrays and Bloom Filters. *ACM Transactions on Knowledge Discovery from Data*, 5(2):1–27, 2 2011.

[EIO+14] A K Elmagarmid, I F Ilyas, M Ouzzani, J Quiané-Ruiz, N Tang, and S Yin. NADEEF/ER: Generic and Interactive Entity Resolution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1071–1074, 2014.

[ETJ+18] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. Distributed representations of tuples for entity resolution. In *Proceedings of the VLDB Endowment*, volume 11, pages 1454–1467. Association for Computing Machinery, 2018.

[FHF+23] Meihao Fan, Xiaoyue Han, Ju Fan, Chengliang Chai, Nan Tang, Guoliang Li, and Xiaoyong Du. Cost-Effective In-Context Learning for Entity Resolution: A Design Space Exploration. 12 2023.

[GEKM16] Iryna Gurevych, Judith Eckle-Kohler, and Michael Matuschek. Lexical Knowledge Bases. pages 1–20. 2016.

[GL20] Artur d'Avila Garcez and Luis C. Lamb. Neurosymbolic AI: The 3rd Wave. 12 2020.

[Goo25a] Google AI. Gemini API Documentation. https://ai.google.dev/api?lang=python, Accessed: 2025-02-02 at 21:17, 2025.

[Goo25b] Google AI. Gemini API Tokens Documentation. https://ai.google.dev/gemini-api/docs/tokens?lang=python, Accessed: 2025-02-02 at 21:17, 2025.

[Goo25c] Google AI. Google Cloud AI Pricing. https://ai.google.dev/pricing, Accessed: 2025-02-03 at 10:17, 2025.

[Goo25d] Google AI. Vertex AI Generative AI Usage Metadata Documentation. https://cloud.google.com/vertex-ai/generative-ai/docs/reference/nodejs/latest/vertexai/usagemetadata, Accessed: 2025-02-02 at 21:31, 2025.

[GZG+19] Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. Towards Complex Text-to-SQL in Cross-Domain Database with Intermediate Representation, 2019.

[HFD+23] Chenxu Hu, Jie Fu, Chenzhuang Du, Simian Luo, Junbo Zhao, and Hang Zhao. ChatDB: Augmenting LLMs with Databases as Their Symbolic Memory. 6 2023.

[HYM+24] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Transactions on Information Systems*, 11 2024.

[I. 69] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Society*, 64(328), 1969.

[IKC+17] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. Learning a Neural Semantic Parser from User Feedback, 2017.

[IVMS18] Ayyoob Imani, Amir Vakili, Ali Montazer, and Azadeh Shakery. Deep Neural Networks for Query Expansion using Word Embeddings. 11 2018.

[JZD+23] Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. StructGPT: A General Framework for Large Language Model to Reason over Structured Data. 5 2023.

[JZQ+23] Rolf Jagerman, Honglei Zhuang, Zhen Qin, Xuanhui Wang, and Michael Bendersky. Query Expansion by Prompting Large Language Models. 5 2023.

[KE09] Alfons Kemper and André Eickler. *Datenbanksysteme: Eine Einführung*. Oldenbourg, Munich, 7 edition, 2009.

[KGR+22] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners. 5 2022.

[KK18] Huda Khayrallah and Philipp Koehn. On the Impact of Various Types of Noise on Neural Machine Translation. *CoRR*, abs/1805.12282, 2018.

[KPGM20] A Kumar, A Pandey, R Gadia, and M Mishra. Building knowledge graph using pre-trained language model for learning entity-aware relationships. In *2020 IEEE International Conference on Computing, Power and Communication Technologies (GUCON)*, pages 310–315. IEEE, 2020.

[LCG+21] Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. TAPEX: Table Pre-training via Learning a Neural SQL Executor. 7 2021.

[LLS+20] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, Anhai Doan, and Wang Chiew Tan. Deep entity matching with pre-trained language models. *Proceedings of the VLDB Endowment*, 14(1):50–60, 9 2020.

[LMZ+23] Tianle Li, Xueguang Ma, Alex Zhuang, Yu Gu, Yu Su, and Wenhu Chen. Few-shot In-context Learning for Knowledge Base Question Answering. 5 2023.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach, 2019.

[MDAJ22] Fedor Moiseev, Zhe Dong, Enrique Alfonseca, and Martin Jaggi. SKILL: Structured Knowledge Infusion for Large Language Models. 5 2022.

[MK60] M. E. Maron and J. L. Kuhns. On Relevance, Probabilistic Indexing and Information Retrieval. *Journal of the ACM*, 7(3):216–244, 7 1960.

[MLR+18] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, Anhai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 19–34. Association for Computing Machinery, 5 2018.

[MLW21] Zhengjie Miao, Yuliang Li, and Xiaolan Wang. Rotom. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1303–1316, New York, NY, USA, 6 2021. ACM.

[MM74] E. Marchi and O. Miguel. On the structure of the teaching-learning interactive process. *International Journal of Game Theory*, 3(2):83–99, 6 1974.

[MMN+24] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large Language Models: A Survey. 2 2024.

[MMR23] Saikat Mondal, Debajyoti Mondal, and Chanchal K Roy. Investigating Technology Usage Span by Analyzing Users' Q&A Traces in Stack Overflow, 2023.

[MNU00] Andrew Mccallum, Kamal Nigam, and Lyle H Ungar. Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. Technical report, 2000.

[Moh25] Mohit2512. Jio Mart Product Items. https://www.kaggle.com/datasets/mohit2512/jio-mart-product-items, Accessed: 2025-03-22 at 12:33, 2025.

[NCO+22] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. Can Foundation Models Wrangle Your Data? 5 2022.

[NZZ+23] Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. Enhancing Few-shot Text-to-SQL Capabilities of Large Language Models: A Study on Prompt Design Strategies. 5 2023.

[PAWW23] Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning. 5 2023.

[PB23] Ralph Peeters and Christian Bizer. Using ChatGPT for Entity Matching. 5 2023.

[PBGF22] Matteo Paganelli, Francesco Del Buono, Francesco Guerra, and Nicola Ferro. Evaluating the integration of datasets. In *Proceedings of the ACM Symposium on Applied Computing*, pages 347–356. Association for Computing Machinery, 4 2022.

[PDB23] Ralph Peeters, Reng Chiz Der, and Christian Bizer. WDC Products: A Multi-Dimensional Entity Matching Benchmark. 1 2023.

[PGSV19] Matteo Paganelli, Francesco Guerra, Paolo Sottovia, and Yannis Velegrakis. TuneR: Fine tuning of rule-based entity matchers. In *International Conference on Information and Knowledge Management, Proceedings*, pages 2945–2948. Association for Computing Machinery, 11 2019.

[Pos25] PostgreSQL Global Development Group. PostgreSQL Comparison Functions and Operators. Technical report, PostgreSQL, 2025.

[PR23] Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction, 2023.

[PRWZ01] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*, page 311, Morristown, NJ, USA, 2001. Association for Computational Linguistics.

[PSB23] Ralph Peeters, Aaron Steiner, and Christian Bizer. Entity Matching using Large Language Models. *28th International Conference on Extending Database Technology (EDBT)*, 10 2023.

[Rah25] Erhard Rahm. Benchmark Datasets for Entity Resolution. https://dbs.uni-leipzig.de/research/projects/benchmark-datasets-for-entity-resolution, Accessed: 2025-04-28, 2025.

[RD00] Erhard Rahm and Hong Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.*, 23:3–13, 4 2000.

[RG19] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. 8 2019.

[RIKJ24] Md Mostafizur Rahman, Siful Islam, Md Kamruzzaman, and Zihad Hasan Joy. ADVANCED QUERY OPTIMIZATION IN SQL DATABASES FOR REAL-TIME BIG DATA ANALYTICS. 4:1–14, 6 2024.

[RN24] Francesco Ronzano and Jay Nanavati. Towards Ontology-Enhanced Representation Learning for Large Language Models. 5 2024.

[Roc71] J J Rocchio. Relevance feedback in information retrieval. In G Salton, editor, *The Smart retrieval system - experiments in automatic document processing*, pages 313–323. Englewood Cliffs, NJ: Prentice-Hall, 1971.

[RSR+19] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. 10 2019.

[Sam23] Sam Stoelinga. Calculating GPU Memory for Serving LLMs. Accessed: 2025-03-17 at 16:40, 2023.

[SB88] G Salton and C Buckley. Term weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[SB97] Gerard Salton and Chris Buckley. Improving retrieval performance by relevance feedback. In *Readings in Information Retrieval*, pages 355–364. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[SBF98] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. Knowledge Engineering: Principles and methods. *Data and Knowledge Engineering*, 25(1-2):161–197, 1998.

[SdLJ18] Stuart Steiner, Daniel Conte de Leon, and Ananth A. Jillepalli. Hardening web applications using a least privilege DBMS access model. In *Proceedings of the Fifth Cybersecurity Symposium*, pages 1–6, New York, NY, USA, 4 2018. ACM.

[SME+17] Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. Synthesizing entity matching rules by examples. *Proceedings of the VLDB Endowment*, 11(2):189–202, 10 2017.

[SMH+20] Tao Shen, Yi Mao, Pengcheng He, Guodong Long, Adam Trischler, and Weizhu Chen. Exploiting Structured Knowledge in Text via Graph-Guided Representation Learning. 4 2020.

[SPB24] Aaron Steiner, Ralph Peeters, and Christian Bizer. Fine-tuning Large Language Models for Entity Matching. 9 2024.

[SSS+24] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. 2 2024.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. In Z Ghahramani, M Welling, C Cortes, N Lawrence, and K Q Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[TDP19] Ian Tenney, Dipanjan Das, and Ellie Pavlick. BERT Rediscovers the Classical NLP Pipeline. In *Association for Computational Linguistics*, 2019.

[TFT+22] Jianhong Tu, Ju Fan, Nan Tang, Peng Wang, Chengliang Chai, Guoliang Li, Ruixue Fan, and Xiaoyong Du. Domain Adaptation for Deep Entity Resolution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 443–457. Association for Computing Machinery, 6 2022.

[The25] The PostgreSQL Global Development Group. PostgreSQL Documentation: Appendix K. PostgreSQL Limits. https://www.postgresql.org/docs/current/limits.html, Accessed: 2025-02-27 at 09:42, 2025.

[TLT+21] Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and Anhai Doan. Deep learning for blocking in entity matching: A design space exploration. In *Proceedings of*

the VLDB Endowment, volume 14, pages 2459–2472. VLDB Endowment, 2021.

[UNB23] Matthias Urban, Duc Dat Nguyen, and Carsten Binnig. OmniscientDB: A Large Language Model-Augmented DBMS That Knows What Other DBMSs Do Not Know. In *Proceedings of the 6th International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM 2023 - In conjunction with the 2023 ACM SIGMOD/PODS Conference*. Association for Computing Machinery, Inc, 6 2023.

[VBD14] Norases Vesdapunt, Kedar Bellare, and Nilesh Dalvi. Crowdsourcing algorithms for entity resolution. *Proceedings of the VLDB Endowment*, 7(12):1071–1082, 8 2014.

[Voo94] Ellen M. Voorhees. Query expansion using lexical-semantic relations. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 1994*, pages 61–69. Association for Computing Machinery, Inc, 8 1994.

[VSAB+24] K. Venkatesh Sharma, Pramod Reddy Ayiluri, Rakesh Betala, P. Jagdish Kumar, and K. Shirisha Reddy. Enhancing query relevance: leveraging SBERT and cosine similarity for optimal information retrieval. *International Journal of Speech Technology*, 27(3):753–763, 9 2024.

[WBZ+21] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned Language Models Are Zero-Shot Learners. 9 2021.

[WCL+24] Tianshu Wang, Xiaoyang Chen, Hongyu Lin, Xuanang Chen, Xianpei Han, Hao Wang, Zhenyu Zeng, and Le Sun. Match, Compare, or Select? An Investigation of Large Language Models for Entity Matching. 5 2024.

[WSSJ14] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for Similarity Search: A Survey. *CoRR*, abs/1408.2927, 2014.

[WWS+22] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. 1 2022.

[WWX+23] Siyuan Wang, Zhongyu Wei, Jiarong Xu, Taishan Li, and Zhihao Fan. Unifying Structure Reasoning and Language Model Pre-training for Complex Reasoning. 1 2023.

[WYW23] Liang Wang, Nan Yang, and Furu Wei. Query2doc: Query Expansion with Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9414–9423, Stroudsburg, PA, USA, 2023. Association for Computational Linguistics.

[WZC+22] Pengfei Wang, Xiaocan Zeng, Lu Chen, Fan Ye, Yuren Mao, Junhao Zhu, and Yunjun Gao. PromptEM: Prompt-tuning for Low-resource Generalized Entity Matching. In *Proceedings of the VLDB Endowment*, volume 16, pages 369–378. VLDB Endowment, 2022.

[WZL+24] Zilong Wang, Hao Zhang, Chun-Liang Li, Julian Martin Eisenschlos, Vincent Perot, Zifeng Wang, Lesly Miculicich, Yasuhisa Fujii, Jingbo Shang, Chen-Yu Lee, and Tomas Pfister. Chain-of-Table: Evolving Tables in the Reasoning Chain for Table Understanding. 1 2024.

[XJK24] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. Hallucination is Inevitable: An Innate Limitation of Large Language Models. 1 2024.

[XJS+23] Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, Wang Zhao, Fan Zhou, Danrui Qi, Hong Yi, Shaodong Liu, and Faqiang Chen. DB-GPT: Empowering Database Interactions with Private Large Language Models. 12 2023.

[XLS17] Xiaojun Xu, Chang Liu, and Dawn Song. Sqlnet: Generating structured queries from natural language without reinforcement learning. *CoRR*, abs/1711.04436, 2017.

[YHV+23] Zhangdie Yuan, Songbo Hu, Ivan Vulić, Anna Korhonen, and Zaiqiao Meng. Can Pretrained Language Models (Yet) Reason Deductively? In Andreas Vlachos and Isabelle Augenstein, editors, *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1447–1462, Dubrovnik, Croatia, 5 2023. Association for Computational Linguistics.

[YHY+23] Yunhu Ye, Binyuan Hui, Min Yang, Binhua Li, Fei Huang, and Yongbin Li. Large Language Models are Versatile Decomposers: Decompose Evidence and Questions for Table-based Reasoning. 1 2023.

[YRB+21] Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. QA-GNN: Reasoning with Language Models and Knowledge Graphs for Question Answering. 4 2021.

[YWL+20] Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan, Xinyi Yang, Dragomir R Radev, Richard Socher, and Caiming Xiong. GraPPa: Grammar-Augmented Pre-Training for Table Semantic Parsing. *CoRR*, abs/2009.13845, 2020.

[YZD+22] Hongbin Ye, Ningyu Zhang, Shumin Deng, Xiang Chen, Hui Chen, Feiyu Xiong, Xi Chen, and Huajun Chen. Ontology-enhanced Prompt-tuning for Few-shot Learning. In *WWW 2022 - Proceedings of the ACM Web*

Conference 2022, pages 778–787. Association for Computing Machinery, Inc, 4 2022.

[ZBY+22] Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D. Manning, and Jure Leskovec. GreaseLM: Graph REASoning Enhanced Language Models for Question Answering. 1 2022.

[ZCH+20] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 1 2020.

[ZD07] Patrick Ziegler and Klaus R. Dittrich. *Data Integration — Problems, Approaches, and Perspectives*, pages 39–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[ZHH+20] Zhi Zheng, Kai Hui, Ben He, Xianpei Han, Le Sun, and Andrew Yates. BERT-QE: Contextualized Query Expansion for Document Re-ranking. 9 2020.

[ZP97] Esteban Zimányi and Alain Pirotte. Imperfect Information in Relational Databases. In *Uncertainty Management in Information Systems*, pages 35–87. Springer US, Boston, MA, 1997.

[ZSH+22] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. 5 2022.

[ZWS+19] Wei Zhang, Hao Wei, Bunyamin Sisman, Xin Luna Dong, Christos Faloutsos, and David Page. AutoBlock: A Hands-off Blocking Framework for Entity Matching. *CoRR*, abs/1912.03417, 2019.

[ZXS17] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning, 2017.

Appendix

This chapter presents supplementary material that complements the main body of this work. Section Query for Schema-Level Data details the SQL query used to retrieve schema-level data. Next, the Section Test Dataset lists all the tables of the test dataset. Section Test Cases presents the corresponding queries along with their respective ground truth. Finally, Section Pseudo Code includes additional pseudo code from the implementation.

Query for Schema-Level Data

The following SQL query is executed to retrieve the schema-level information regarding a specific table *t*.

```
SELECT
            c.column_name,
            c.is_nullable,
            c.data_type,
            constraints.constraint_type
        FROM
            information_schema.columns c
        LEFT JOIN (
            SELECT
                kcu.column_name,
                tc.constraint_type
            FROM
                information_schema.table_constraints tc
            JOIN
                information_schema.key_column_usage kcu
            ON
                tc.constraint_name = kcu.constraint_name
            WHERE
                tc.table_name = '{t}'
        ) AS constraints
```

```
ON c.column_name = constraints.column_name
WHERE
    c.table_name = '{t}';
```

Test Dataset

This section shows all tables of the manually crafted test dataset.

Table 1: shareowner

id	name	shares
1	Pierre	20
2	Vladi	10
3	Diego	15
4	Marcel	11

Table 2: animalowner

animalname	category	owner_id
bill	chien	1
diego	chat	2
chris	dog	3
juan	perro	4

Table 3: shareowner1row

id	name	shares
1	Pierre	20

Table 4: animalowner1row

animalname	category	owner_id
bill	chien	1

Table 5: doctors

id	name	patients_pd
2	Giovanni	11
3	Hans	fourty
4	Lukas	44
1	Peter	ten
5	Dr. Smith	150

Table 6: songs

id	album_id	song_name	duration
1	1	Delicate	3:52
2	2	New Year's Day	3:55

Table 7: albums

id	artist_id	album_name	release_year
1	1	Reputation	2017
2	2	Reputation	2017

Table 8: artists

id	name	language
1	Taylor Swift	English
2	Reputation Artist	English

Table 9: tennis_players

id	name	born
1	Juan	20.02.2003
2	Paul	18.04.1968
3	Xi	January 1986
4	Michael	18.01.1997

Table 10: tournaments

winner id	name	price money in million
4	Berlin Open	4
3	Warsaw Open	3
2	Jakarta Open	1.5
3	Osaka Open	0.5

Table 11: influencers

media_name	clicks
makeuptutorial	1000 thousand
outsideguy	50
surviver1000	1 million
princess	one thousand

Table 12: followers

id	following	adult
1	surviver1000	True
3	makeuptutorial	False
2	surviver1000	True
3	princess	True

Table 13: children_table

id	children
0	4
1	1
2	many
3	2

Table 14: fathers

id	name
zero	Gerhard
one	Joachim
four	Simon
two	Dieter

Table 15: mothers

id	name
1	Julia
2	Petra
3	Claudia
4	Lena

Table 16: website_visits

date	page	visits
2023 October 26	homepage	1000
2023 October 26	about	500
2023 October 27	homepage	1200
2023 October 27	contact	200

Table 17: weather

date	city	temperature	rainfall
2023 10 26	London	12	0
2023 10 26	New York	15	2
2023 10 27	London	10	5
2023 10 27	New York	13	1

Table 18: bakery_sales

Item	Quantity	Price
Croissants	5 dozen	12.00 per dozen
Baguettes	8 dozen	10.00 per dozen
Macarons	7 dozen	12.00 per dozen
Pain au Chocolat	3 dozen	15.00 per dozen

Table 19: oven_temperature

Item	Temperature
Croissants	200 °F
Baguettes	400 °F
Macarons	350 °F
Pain au Chocolat	200 °F

Table 20: movies

Movie	category	rating
Raiders of the Lost Arc	action	4/5
The Shawshank Redemption	thriller	3/5
Wings of Desire	fantasy	4/5
Amélie	comedy	5/5

Table 21: clicks

publication	clicks
17.01.2011	1000000
08.03.2016	500
22.11.2014	10^6
24.12.2022	1000

Table 22: movies_personal

Movie	Personal rating
Die Flucht aus Shawshank	3/5
Der Himmel über Berlin	5/5
Die fabelhafte Welt der Amélie	4/5
Lola rennt	2/5

Test Cases

This section presents the predicate calculus expression along with the ground truth used in the initial evaluation.

Test Cases for Predicate Calculus to SQL Conversion

(a) **Input:** ∃id ∃name ∃patients_pd (doctors(id, name, patients_pd) ∧ patients_pd < 12)
Output: {(2, 'Giovanni', '11'), (1, 'Peter', 'ten')}

(b) **Input:** ∃id ∃patients_pd (doctors(id, 'Peter', patients_pd) ∧ patients_pd < 12)
Output: {(1, 'Peter', 'ten')}

(c) **Input:** ∃id ∃shares ∃name (shareowner1row(id, name, shares) ∧ animalowner1row(id, _, 'dog'))
Output: {(1, 'Pierre', 20, 1, 'bill', 'chien')}

(d) **Input:** ∃id ∃shares ∃name (shareowner(id, name, shares) ∧ animalowner(id, _, 'dog'))
Output: {(3, 'Diego', 15, 3, 'chris', 'dog'), (4, 'Marcel', 11, 4, 'juan', 'perro'), (1, 'Pierre', 20, 1, 'bill', 'chien')}

(e) **Input:** ∃id ∃shares ∃name(shareowner(id, name, shares) ∧ ¬animalowner(id, _, 'dog'))
Output: {(2, 'Vladi', 10, 2, 'diego', 'chat')}

(f) **Input:** ∃x ∃y ∃z (children_table(x, y) ∧ fathers(x, z))
Output: {(0, '4', 'zero', 'Gerhard'), (1, '1', 'one', 'Joachim'), (2, 'many', 'two', 'Dieter')}

(g) **Input:** ∃id (children_table(id, _) ∧ fathers(id, _) ∧ mothers(id, _))
Output: {(1, '1', 'one', 'Joachim', 1, 'Julia'), (2, 'many', 'two', 'Dieter', 2, 'Petra')}

(h) **Input:** ∃id (tennis_players(id, _, 'January') ∧ tournaments(id, name, price_money))
Output: {(4, 'Michael', '18.01.1997', 4, 'Berlin Open', 4.0), (3, 'Xi', 'January 1986', 3, 'Warsaw Open', 3.0), (3, 'Xi', 'January 1986', 3, 'Osaka Open', 0.5)}

(i) **Input:** ∃m ∃f ∃i (influencers(m, f) ∧ f > 500 ∧ followers(i, m, z))
Output: {('surviver1000', '1 million', 1, 'surviver1000', True), ('makeuptutorial', '1000 thousand', 3, 'makeuptutorial', False), ('surviver1000', '1 million', 2, 'surviver1000', True), ('princess', 'one thousand', 3, 'princess', True)}

(j) **Input:** ∃id (children_table(id, >1) ∧ fathers(id, _))
Output: {(0, '4', 'zero', 'Gerhard'), (2, 'many', 'two', 'Dieter')}

(k) **Input:** ARTISTS(a, _), ALBUMS(_, a, "Reputation", 2017), SONGS(_, a2, song_name), ALBUMS(a2, a)
Output: {(1, 1, 'Reputation', 2017, 1, 'Taylor Swift', 'English', 1, 1, 'Delicate', '3:52'), (2, 2, 'Reputation', 2017, 2, 'Reputation Artist', 'English', 2, 2, 'New Year's Day', '3:55')}

(l) **Input:** ∃d weather(d, city, temperature, rainfall) ∧ website_visits(d, page, visits)
Output: {(2023 10 26, 'London', 12, 0, '2023 October 26', 'about', 500), (2023 10 26, 'London', 12, 0, '2023 October 26', 'homepage', 1000), (2023 10 26, 'New York', 15, 2, '2023 October 26', 'about', 500), (2023 10 26, 'New York', 15, 2, '2023 October 26', 'homepage', 1000), (2023 10 27, 'London', 10, 5, '2023 October 27', 'contact', 200), (2023 10 27, 'London', 10, 5, '2023 October 27', 'homepage', 1200), (2023 10 27, 'New York', 13, 1, '2023 October 27', 'contact', 200), (2023 10 27, 'New York', 13, 1, '2023 October 27', 'homepage', 1200)}

(m) **Input:** ∃item bakery_sales(item, _, _) ∧ oven_temperature(item, >200 °C)
Output: {('Baguettes', '400 °F', '8 dozen', '10.00 per dozen')}

(n) **Input:** ∃item bakery_sales(item, < 55, _) ∧ oven_temperature(item, _)
Output: {('Pain au Chocolat', '3 dozen', '15.00 per dozen', '200 °F')}

(o) **Input:** ∃item bakery_sales(item, > 90, _) ∧ oven_temperature(item, >180 °C)
Output: {('Baguettes', '400 °F', '8 dozen', '10.00 per dozen')}

(p) **Input:** ∃movie movies(movie, _, _) ∧ movies_personal(movie, _)
Output: {('Wings of Desire', 'fantasy', '4/5', 'Der Himmel über Berlin', '5/5'), ('Amélie', 'comedy', '5/5', 'Die fabelhafte Welt der Amélie', '4/5'), ('The Shawshank Redemption', 'thriller', '3/5', 'Die Flucht aus Shawshank', '3/5')}

(q) **Input:** ∃movie movies(movie, _, _) ∧ movies_personal(movie, >70%)
Output: {('Wings of Desire', 'fantasy', '4/5', 'Der Himmel über Berlin', '5/5'), ('Amélie', 'comedy', '5/5', 'Die fabelhafte Welt der Amélie', '4/5')}

(r) **Input:** ∃ movies("The sky over Berlin", _, _)
Output: {('Wings of Desire', 'fantasy', '4/5')}

(s) **Input:** ∃clicks influencers(_, clicks) ∧ publication_clicks(_, clicks)
Output: {('princess', 'one thousand', '24.12.2022', '1000'), ('makeuptutorial', '1000 thousand', '17.01.2011', '1000000'), ('surviver1000', '1 million', '17.01.2011', '1000000')}

Pseudo Code

In this section, we present the pseudo code for both the *row calculus pipeline* and the *join pipeline*.

Row Calculus Pipeline

The following provides the pseudo code for the *row calculus pipeline*.

```
1
2   sql_query = INPUT  #Get predicate
3
4   #Extract all WHERE comparisons (e.g. WHERE animal.category='dog') using a SQL parser and divide inot the different parts
5   conditions = extract_where_conditions_sqlparse(sql_query):
6      conditions= [] # List to fill with divided WHERE statments, divided into left part, right part and compariosn operator
7      for token in sqlparse(sql_query): # Iteration over all tokens in SQL query
8         if token is where.Clause and isinstance Comparison: # If is part of WHERE clause e.g. 'WHERE animal.category='dog''
9
10           # Use column to generate a SQL query e.g. 'animal.category' -> 'SELECT category FROM animal;'
11           token.left <- Convert_to_SQL(token.left)
12
13           # Append to conditions, structure: {('SELECT category FROM  animal', "=",'WHERE animal.category='dog'', 'dog'),
             ↪ (...)}
14           conditions.append(token.left, token.comparison_operator, token, token.right)
15     return conditions #structure: {('SELECT category FROM  animal', "=",'WHERE animal.category='dog'', 'dog'), (...)}
16
17  #Execute SQL queries inside conditions against the database
18  query_results = execute_queries_on_conditions(conditions):
19     for i in  conditions: # For the whole conditions list {(...),(...),(...)}
20        for l in i: # Inside a comparison e.g. ('SELECT category FROM  animal', "=",'WHERE animal.category='dog'', 'dog')
21           if l is SQL_query: # Check if the element is a SQL query
22              l = query_database(l) # Substitute SQL query with result from database of that query e.g 'SELECT category FROM
                ↪ animal;' ---> ('chien','perro','chat','dog')
23     return conditions #Structure is the following {(('chien','perro','chat','dog'), "=",'WHERE animal.category='dog'', 'dog'),
       ↪ (...)}
24
25
26  # Main Soft Binding Procedure using the LLM
27  semantic_list = compare_semantics_in_list(query_results):
28    semantic_list = [] #Initialize empty list for storing the bindings
29
30    #Iterate for each sublist e.g. (('chien','perro','chat','dog'), "=",'WHERE animal.category='dog'', 'dog')
31    for each sublist in query_results:
32
33          #Compare a list e.g. ('chien','perro','chat','dog') with the fixed binding e.g. 'dog'
34          temp_string, temp_list = separate_binding_and_list(sublist) #Generate the temp_string e.g 'dog' and the temp_list
            ↪ e.g ('chien','perro','chat','dog')
35
36          #Abstracts meaning of comparison operator in natural language
37          # e.g " A = B" ---> " A has the same semantic meaning as B"'
38          phrase = ask_LLM("Get semantic phrase for: " + sublist[1]) #sublist[1] contains comparison operator e.g. "=", "<",
            ↪ ">"
39
40
41          soft_binding_list = [] #Construct a list of expressions to be included
42          prompt="" #Construct an initial prompt to feed LLM
43          for i in  temp_list #Iterate over temp_list e.g.('chien','perro','chat','dog')
44            prompt +=  build_comparison_prompt(temp_string, i, phrase)
45          #Construct final prompt for each individual comparison e.g
46          ["Does 'dog' and 'chien' have the same meaning?",
47          "Does 'dog' and 'perro' have the same meaning?",
48          "Does 'dog' and 'chat' have the same meaning?",
49          "Does 'dog' and 'dog' have the same meaning?"]
50
51          #Return a boolean list e.g [True, True, False, True]
52          boolean_results = gemini_json(prompt, response_type = list[boolean])
53
54          #Append ('chien','dog','perro'). These are all the values where LLM said True.
55          soft_binding_list.append(temp_list if boolean_result is true)
```

```
56
57          #Append (('chien','dog','perro'), 'WHERE animal.category='dog'') for final result_list
58          semantic_list.append(soft_binding_list, sublist.where_Clause)
59
60      return semantic_list
61
62  #Modified query Construction
63
64  #Construct the modified query based on the semantics list e.g "WHERE animal.category='dog'" --->
65  ---> "WHERE animal.category='dog' OR animal.category='perro' OR animal.category='chien'"
66  query = ask_LLM(f"Based on semantic list {semantic_list} and the intital query {sql_query} generate  a new query")
67
68
69  result=query_database(query) #Query database to get result of the modified query
70  return result #Return the results of the query to the user
```

JOIN Pipeline

The following provides the pseudo code for the *join pipeline*.

```
1   query = INPUT  #Get predicate calculus expression as input
2
3
4   #Extract all JOIN comparisons (e.g. JOIN animal.category=owner.animal) using a SQL parser and divide it into  the different
    ↪  parts
5   join_conditions = extract_join_conditions_sqlparse(initial_sql_query):
6      join_conditions= [] # List to fill with divided JOIN statments, divided into left part, right part and compariosn
       ↪  operator
7      for token in sqlparse(sql_query): # Iteration over all tokens in SQL query
8         if isinstance(token, sqlparse.sql.Comparison): # If is part of a JOIN clause
9
10           left = str(token.left).strip() #Identiy attributes on left part of the JOIN condition
11           right = str(token.right).strip() #Identiy attributes on rigth part of the JOIN condition
12           operator = str(token.token_next(0)).strip # Identify comparison operator e.g. '=', '>', '<'
13
14           #List to mark the order of the different attributes
15           order = [copy.deepcopy(left), copy.deepcopy(right)]
16
17           # Use column to generate a SQL query e.g. 'animal.category' -> 'SELECT category FROM animal;'
18           left <- Convert_to_SQL(left)
19           right <- Convert_to_SQL(right)
20
21           # Append to conditions, structure: {('SELECT category FROM  animal', "=",'JOIN animal.category=owner.animal'',
             ↪  SELECT animal from owner), (...)}
22           join_conditions.append([left, operator, token.normalized,  right])
23           order_list.append(order) #[animal.category, owner.animal]
24     return join_conditions, order_list
25
26  #Execute SQL queries inside conditions against the database
27  new_list = execute_queries_on_conditions(join_conditions):
28     for i in  join_conditions: # For the whole conditions list {(...),(...),(...)}
29        for l in i: # Inside a comparison e.g. ('SELECT category FROM  animal', "=",'JOIN animal.category=owner.animal'',
          ↪  'SELECT animal from owner')
30           if l is SQL_query: # Check if the element is a SQL query
31              l = query_database(l) # Substitute SQL query with result from database of that query e.g 'SELECT category FROM
                ↪  animal;' ---> ('chien','perro','chat','dog')
32     return conditions #Structure is the following {{('chien','perro','chat','dog'), "=",'JOIN animal.category=owner.animal'',
       ↪  ('cat','snake', 'dog'))), (...)}
33
34
35  # Main Soft Binding Procedure using the LLM
36  semantic_dic = compare_semantics_in_list(new_list):
37
38
39    dict_list = [] = [] #Initialize empty list for storing the bindings
40
41    #Iterate for each sublist e.g. ('chien','perro','chat','dog'), "=",'JOIN animal.category=owner.animal'', ('cat','snake',
      ↪  'dog'))
```

```
    for each outer_list in new_list:

        temp_list1 = outer_list[0] #Get left part
        temp_list2 = outer_list[-1] #Get right part

        #Check if a LLM comparison is necessary or not. It would not be necessary if the type of both columns is int or date for example.
        necessary = gemini(f"Based on these list {temp_list1} and {temp_list2} and especially the type of lists, is there a possbility for residual noise")

        #Only proceed if it is necessary
        if necessary:
          #Compare a list e.g. ('chien','perro','chat','dog') with the fixed binding e.g. 'dog'
          temp_string, temp_list = separate_binding_and_list(sublist) #Generate the temp_string e.g 'dog' and the temp_list e.g ('chien','perro','chat','dog')

          #Abstracts meaning of comparison operator in natural language
          # e.g " A = B" ---> " A has the same semantic meaning as B"'
          phrase = ask_LLM("Get semantic phrase for: " + sublist[1]) #sublist[1] contains comparison operator e.g. "=", "<", ">"

          for item in temp_list1:

            #Create total prompt
            total_prompt = f"Answer the following questions with True or False.\n"

              for other_item in temp_list2:
                #Add instances to the total prompt
                total_prompt+= f"'{item_str}' {phrase} '{other_item_str}' \n"

            #Add the instances which are relevant dic['dog']=['dog', 'chien', 'perro']
            response = gemini_json(total_prompt, response_type=list[bool])
            dict[item]= [temp_list2[i] for i, is_relevant in enumerate(response) if is_relevant]

        #Append dictionary to total dictionary
        dict_list.append(dict)


    return dict_list

#Modified query Construction

#Construct the modified query based on the semantics list e.g "WHERE animal.category='dog'" --->
#---> "WHERE animal.category='dog' OR animal.category='perro' OR animal.category='chien'"
query = ask_LLM(f"Based on semantic dic {semantic_dic} and the intital query {sql_query} generate  a new query")


result=query_database(query) #Query database to get result of the modified query
return result #Return the results of the query to the user
```