Brandenburg University of Applied Sciences

IT Security
Computerscience
Prof. Dr. Oleg Lobachev
MSc. Florian Eich

Reliable Natural Language Interfaces using LLMs, Self-Correction and Incremental Schema Analysis

Bachelor Thesis

Winter Semester 2025

November 9, 2025

Mara Schulke – Matr-Nr. 20215853

Abstract

This thesis explores the integration of large language models (LLMs) into PostgreSQL database systems in order to make the database accessible via natural language instead of the postgres SQL dialect. The research focuses on implementation strategies, performance optimization, and practical applications of this concept.

Contents

List of Figures

List of Abbreviations

GPT	Generative Pretrained Transformer
SQL	Structured Query Language
API	Application Programming Interface
LLM	Large Language Model
DBMS	Database Management System
NL2SQL	Natural Language to SQL

1 Introduction

1.1 Problem Statement and Motivation

Database systems represent a backbone of modern computer science, allowing for rapid advancements whilst shielding us from the problem categories that come along with managing and querying large amounts of, usually structured, data efficiently. However, most Database Management Systems (DBMS) have traditionally required specialized knowledge, usually of the Structured Query Language (SQL), in order to become useable. Whilst this barrier may be percieved differently across diverse usergroups it represents a fundamental misalignment between end-user goals (e.g. analysts, researchers, domain experts etc.) and the underlying DBMS, thus often requiring software engineering efforts in order to reduce this friction.

This barrier is the reason entire classes of software projects exists (for example, admin / support panels), data analytics tools etc. which therefore introduce significant churn and delay between the implementation of a database system and reaching the desired end user impact. Often these projects span multiple years, require costly staffing and yield little to no novel technical value.

Emerging technologies such as Large Language Models (LLMs) have proven themselves as a sensible tool for bridging fuzzy user provided input into discrete, machine readable formats. Prominent models in this field have demostrated outstanding capabilities that enable computer scientists to tackle new problem classes, that used to be challenging / yielded unsatisfying results with logical programming approaches.

This thesis is exploring ways to overcome the above outlined barrier using natural language queries, so that domain experts, business owners, support staff etc. are able to seamlessly interact with their data, essentially eliminating the requirement of learning SQL (and its pitfalls). By translating natural language to SQL using Large Language Models this translation becomes very robust (e.g. against different kinds of phrasing) and enables novel applications in how businesses, researchers and professionals interact with their data — it represents a fundamental shift (ie. moving away from SQL) towards a more inclusive and data driven world.

1.2 Objectives of the Thesis

This thesis aims to address the aforementioned challanges when it comes to database accessibility. The following objectives are the core research area of this thesis:

1. Develop a database extension that can translate natural language queries into semantically accurate SQL queries using Large Language Models.
2. To evaluate the effectiveness and feasibility of different Models aswell as prompt engineering techniques in order to improve the performance of the system.
3. Identify and address issues when it comes to handling amibguous, complex and domain specific user input.
4. Benchmark the performance of the implementation against common natural language to SQL (NL2SQL) benchmarks.
5. Idenitfy potential use cases for real world scenarios that could deliver a noticable upsides to users.
6. Analyze the short commings and limitations of this approach and propose potential solutions to overcome them.

1.3 Research Questions

RQ1 — Are natural language database interfaces feasible for real world application?

The primary research questions when it comes to natural language database interfaces evolve around their semantic accuracy and reliability, therefore questioning their feasibility for real world usage. LLMs have notoriously been known for their ability to hallucinate / produce false, but promising outputs. This behaviour can be especially dangerous when opting for data driven decisions that rely on false data due to a mistranslation from natural language to SQL. LLMs could cause hard to understand and debug behaviour, like false computation of distributions when the intermediate format is not being shown to the user. This thesis tries to determine whether such hallucinations could be reasonably prevented and whether the associated performance and hardware requirements are suitable for a real world deployment, outside of research situations.

Specifically the two big underlying questions are:

1. Is the semantic accuracy of natural language database interfaces high enough to yield a noticable benefit to users?

2. Is it possible to run such an interface on reasonable, mass available hardware (e.g. excluding high end research GPUs).

RQ2 — What approaches are most effective in resolving ambiguity when translating natural language queries into SQL?

To provide semantically correct results ambiguity in the user-provided natural language queries must be adequately addressed. This thesis investigates various approaches to ambiguity management and resolution. Natural language queries can demonstrate ambiguity even at low levels of complexity — e.g. there are two different types of "sales" in a database schema, and the user asks to retireve "all sales".

Such situations present the second major challenge associated with the practical implementation of natural language database interfaces. The success of this concept will significantly depend on whether suitable designs and mitigation techniques can be implemented without creating problems with regards to the aforementioned performance and hardware requirements. The research focus lies on both preventative measures through optimized pre-processing stages and prompt engineering techniques as well as reactive strategies that post process LLM output, either on the basis of further user input or context inference.

RQ3 — Which strategies are increasing semantic accuracy of queries?

In order to enhance the semantic accuracy a series of improvements may be applied to the pipeline. Potential optimizations include supplying (parts of) the schema during LLM prompting, implementation of interactive contextual reasoning through a conversational interface which would allow for user refinement, the implementation of a robust SQL parsing and validation mechanism and a hybrid approach partly relying on traditional NLP preprocessing techniques. This research will quantify semantic accuracy using popular NL2SQL benchmarks and empirically evaluate the impact each approach has on the benchmark performance. Furthermore this research will take a look at the optimal combination of the aforementioned solutions in order to develop a system that strikes the right balance between accuracy and performance.

1.4 Structure of the Thesis

This thesis is following a research and development methodology in order to implement a natural language interface for databases, in particular postgres is used.

1. **Literature Review** — An analysis of the existing research in the fields of natural language interfaces (NLI) for databases, GPU integration for acceleration of database operations, and LLM/AI Model integration within database systems. This phase establishes the theoretical foundation for this research and identifies current state-of-the-art approaches, their benefits and shortcomings.

2. **Decomposition & Requirements** — Decomposing the problem statement into its fundamentals and deriving system requirements for the design phase from it. The goal of this section is to arrive at a list of functional and non-functional requirements that must be taken into account and fulfilled by the design and implementation phases respectively.

3. **System Design** — Design of a system architecture that can utilize GPU acceleration for LLM integration from within postgres. The primary goals of the system design phase are to arrive at an architecture that yields low latency natural language processing, schema-aware SQL query generation, ambiguity detection and resolution whilst maintaining a high semantic accuracy.

4. **Implementation** — The implementation of a PostgreSQL extension according to the above system design that relies on `rust` and `pgrx`. This extension will provide a GPU accelerated framework for executing LLMs, implement a natural language to query generation pipeline that relies on the SQL schema and create database functions and operators for both query generation and execution.

5. **Evaluation and Benchmarking** — An assesment framework and benchmark that introspects the implementations performance in multiple dimensions. Namely the most relevant dimensions for this thesis are:

   (a) Semantic Accuracy — Measuring the overall accuracy of results delivered for a given natural language input.

   (b) Ambiguity Resolution Capabilities — How well the system performs when confronted with ambiguous natural language input and database schemas.

   (c) Performance Metrics — Measuring the latency, throughput and resource utilization of the implementation.

6. **Discussion** — Analysis and interpretation of the evaluation phase results against the research goals of this thesis. Evaluating the performance and accuracy results recorded during the benchmarks against the question whether real world deployments of NILs are feasible. Furthermore the effectiveness of ambiguity resolution capabilities and semantic accuracy enhancement strategies are showing a statistically significant effect.

7. **Summary and Outlook** — Summarizes the contributions, addresses limitations of this thesis and the implementation, and proposes directions for future research alongside possible applications. Primary future research topics include advanced GPU optimization techniques (e.g. further quantization), accuracy and performance impact of model fine tuning, techniques, scalability of such a system in enterprise scenarios and the evaluation of security and privacy considerations (e.g. managing access control).

2 Literature Review

In this section a comprehensive literature review is performed to asses the research landscape on NL2SQL (sometimes also referred to as Text-to-SQL or T2SQL) and NLIDBs. From the time their development accelerated in the late 1990s and early 2000s (**????**) until now, observing multiple larger paradigm shifts happening over time (**?????**). In particular this research focuses on the recent advancements when it comes to language models and how they can be harnessed for effective NL2SQL systems (**??????**).

This literature review is covering the foundational concepts, challenges, key advancements and research gaps associated with using natural language instead of SQL. It lays the foundation for this thesis and helps to set the research questions introduced in the previous chapter in context.

2.1 Foundations of Natural Language Interfaces to Databases

Earlier papers in the research landscape on Natural Language Database Interfaces (NLIDBs) date over half a century back, into the early 1970s. Two decades after the first major research systems where developed in this domain, **?** have published an introduction and an overview over NLIDBs where an overview of state-of-the-art approaches were provided. (**?**) Their work outlined multiple key issues and challenges associated with NLIDBs, and compared them against existing / competing solutions like formal query languages, form-based interfaces and graphical interfaces. These challenges (like unobvious limits, linguistic ambiguities, semantic inaccuracy, tedious configuration etc.) have shaped this field of research and are still considered relevant metrics today.

Early NLIDBs primarily relied on traditional natural language processing (NLP) techniques in order to achieve natural language understanding capabilities. With Chill an inductive logic programming (ILP) approach was first introduced for NL2SQL systems, marking one of the key events when it comes to machine learning usage. (**?**) In **? ?** have extended the approach of ILP parsing for natural language database queries with multi clause construction, yielding promising results in the field of NLIDBs. (**?**)

Building on the systematic overview of **?** and the first machine learning approaches from **?** aswell as **?**, **?** have proposed a novel approach for implementing NLIDBs and outperformed at the time state-of-the-art solutions from **? ?** — achieving 80% semantic accuracy. (**?**) The novelty of the Percise system lies in its natural language processing approach, specifically its lexical mapping strategy, allowing Percise to identify questions it can, and can't answer (introducing the concept of *semantically tractable questions*) which therefore results in a better and interactive end user experience. Their experiements also showed that this approach is *transferrable* and *unbiased* — it is possible to feed in new, unknown questions into the system and maintain performance characteristics, whereas it was shown that **?** were suffering from a distribution drift of the questions asked. (**?**)

The theoretical foundations and research questions highlighted by the aforementioned works, shaped the research field and highlighted the following, ongoing research:

1. The trade-off characteristics derived from choosing a machine learning vs. traditional NLP approach (e.g. Chill versus Percise). E.g. coverage versus correctness. (**??**)
2. The linguistic challenges associated with bringing NLIDBs into use (e.g. semantic inaccuracy, linguistic ambiguity, unclear language coverage etc.) (**?**)
3. The value of systems and approaches which double down on reliability and semantic accuracy rather than giving promising but incorrect answers. (**??**)

Fundamentally this highlights the tension and mismatch between the characteristics of natural language, which is able to be ambiguous, *semantically untractable* or able to be incomplete in meaning and formal languages like SQL which always have on deterministic and *semantically tractable* meaning they convey in each statement. As Schneiderman and Norman have pointed out according to **?**, users are "unwilling to trade reliable and predictable user interfaces for intelligent but unreliable ones" which induces performance expectations on NLIDB implementations to be highly certain about the questions it can, and can't answer, whilst maintaing as high as possible natural language coverage. (**?**)

2.2 Traditional NL2SQL Approaches

Prior to the wide-spread dominance of machine learning approaches for natural language processing a variety of traditional, rather discrtete approaches have been explored in the field of NL2SQL / NLIDBs. These logical programming approaches have laid the foundations for transitioning towards the application of machine learning techniques for NL2SQL.

2.2.1 Rule-based and Grammar-based Systems

Foundational research of NL2SQL system mostly focused around applying rule engines that were tedious to set up and expensive to maintain / transfer across database systems. These rule engines mostly relied on the systematic identification of linguistic patterns / were trying to template SQL from information that was derived from processing the natural language query. (**???**) These approaches mostly tried to formalize natural language queries into formal grammers which could then be deterministically mapped into a valid SQL query. (**?**) These approaches have strong downsides when it comes to the variety of natural language constructs they can process, aswell as runtime adoption of new / unknown databases, query constructs etc. A potential upside of this class of NL2SQL systems is that they can confidently and reproducably identify questions they can, and can't answer — thus leading to very reliable and predictable user interfaces.

2.2.2 Semantic Parsing using String-Kernels

A significiant milestone in parsing techniques of natural language queries was reached by **?** in **?**. The introduction of string kernels for semantic parsing represented a novel achievement, when it comes to fusing logical programming approaches using a formal grammer like LSNLIS developed by **?** and learning / training approaches to understand unseen language patterns / unknown natural language query structures. This allowed for more flexible pattern recognition when compared to traditional rule-based systems.

The core innovative characteristic of this approach lies in its capability to understand similarities between natural language expressions based on subsequence patterns rather than relying on exact matches. This made Krisp, the research NLIDB system developed by **?** much more robust to language variations in phrasing and noise (e.g. spelling mistakes) in the input. As the **?** demonstrated through experiments on real-world datasets, this approach compared favorably to existing systems of the time like Chill, especially in handling noisy inputs — a frequent challenge rigid rule-based systems faced in real world scenarios (**??**).

2.2.3 Graph Matching Methods

? brought together several research threads and reapplied emerging graph matching research models to natural language processing, specifically to natural language queries. Graph matching was applied once the natural language query was parsed using a Combinatory Categorial Grammar (CCG) approach into a semantic graph which denotes the relationship between semantic

entities in it. This graph could then be matched against the actual graph derived from the database, since they share topological traits that can be used for matching (**?**). This approach allowed to apply querying systems without having any question-answer pairs or manual annotations for training the system, which implies easier scalability / transferability across domains, since the system does not require any additional tweaks.

Even though this approach was novel and showed improved the performance over existing state-of-the-art approaches, it was showing that graph matching quickly reaches its limitations. This approach relied heavily on the CCG parser's accuracy, with parsing errors accounting for 10-25% (depending on the dataset) of system failures (**?**). Furthermore it struggled with both ambiguous language constructs and potential mismatches between natural language representation of relationships and database layouts — more complicated database designs, which may not match the users intuitive understanding resulted in a different topology and hence could not be matched (**?**, p. 387).

2.2.4 Interactive Systems

In **?** **?** identified that perfect translation of natural language into SQL was challenging due to natural language not being made for query expressions as it heavily relies on contextual information and clarifying questions in order to disambiguate conversations (**?**). These learnings relate to early prior art from **?** and **?** which also made this observation — "natural language is not a natural query language." (**?**). The solution introduced by NaLIR further emphasized how important an interactive, conversational usage model is, when offering a natural language interface (**?**).

NaLIR could accept logically complex English language sentences as input and translate them into SQL queries with various complexities, including aggregation, nesting, and different types of joins etc. The key innovative characteristic of NaLIR lies in its interactive communication mechanism (much like Rendezvous) that could detect potential misinterpretations and engage users to resolve ambiguities present in their natural language query without forcing them to entirely rephrase their query **?**. This approach, while showing awareness for its limitations (with regards to entirely automating / deriving SQL generation from potentially ambiguous or faulty user input) showed that it was possible to overcome these limtations through choosing the right interaction model — "In our system, we generate multiple possible interpretations for a natural language query and translate all of them in natural language for the user to choose from" —, rather than optimizing the generation part of the system **?**.

2.2.5 Query Synthesis

? introduced SQLizer, which synthesizes SQL queries from natural language (**?**). This paper presents a novel approach when it comes to NL2SQL as it is merging prevalent semantic parsing techniques (outlined above) with an program synthesis (or query synthesis) approach. SQLizer makes use of a three stage processing model for natural language models: first generating a sketch of the query using semantic parsing, then using type-directed synthesis to complete the sketch and finally using automated repair, if required.

? show that alternating between repairing and synthesis yields results that beat state-of-the-art NL2SQL approaches like NaLIR. SQLizer is fully automated and database-agnostic, requiring no knowledge of the underlying schema. The authors evaluated SQLizer on 455 queries across three databases, where it ranked the correct query in the top 5 results for roughly 90% of the queries. This represents a significant improvement over NaLIR (**?**), the previous state-of-the-art system (**?**).

Potential short commings of this approch include queries which yield empty results, dealing with language variations as SQLizer is still using semantic parsing, and domain-specific terminology, all while still requiring users to select from multiple query options which reduces the overall usability of the system (**?**, p.22-23).

2.2.6 Limitations of Traditional Approaches to NL2SQL

Despite being innovative and achieving state-of-the-art results, many of the above outlined approaches face severe challenges when moving outside of an research environment. Many of these systems performed comparatively good on research benchmarks that were often composed of controlled question types and limited data variety. Ultimatively no standard benchmark existed for NL2SQL in this era, hence comparing different NL2SQL systems against each other is a problem on its own. Despite not having a standard benchmark that all approaches could be unifiably evaluated against, several fundamental challenges emerged / remained with these approaches:

1. **Limited linguistic coverage** — Prevalent rule-based and semantic-parsing based systems were only able to process the a small subset of the natural language they were programmed for. This severely limited their ability to handle different phrasings of the same end-user goal (**????**).

2. **Transferability** — Traditional approaches typically required extensive manual configuration or at least a training phase / adaption for each database they were deployed for, hindering cross domain usage through being expensive and time-consuming to adapt (**??**).

3. **Brittleness** — Many of the systems introduced in this subchapter did not handle synonyms, paraphrasing, or spelling errors well. Manual adaption / handling was needed in order to becomes resilient against each class of problems (**??**).

4. **Poor scalability** — With potentially more complex underlying databases, traditional solutions often showed to perform worse. **?** found, that with increasing schema complexity more compute was required to resolve the natural language query to a suitable query candidate making them less transferable and scalable than initially anticipated (**?**) — "Evaluating on all domains in Freebase would generate a very large number of queries for which denotations would have to be computed ... Our system loads Freebase using Virtuoso and queries it with SPARQL. Virtuoso is slow in dealing with millions of queries indexed on the entire Freebase, and is the only reason we did not work with the complete Freebase." which indicates underlying system design issues with runtime complexity.

These flaws of traditional NL2SQL approaches made it apparent, that a different class of approaches is needed, which increase transferability and reduce the brittleness since users are "unwilling to trade reliable and predictable user interfaces for intelligent but unreliable ones" according to **?**. Whilst many approaches outlined tractable ways to increase user satisfaction and accuracy (like **?** did in **?** with a conversational approach), NLIDBs were and are not considered to be a solved problem.

2.3 Neural NL2SQL Approaches

The previously outlined limitations of traditional approaches to solving NL2SQL / implementing NLIDBs pushed the research branch around neural network application forward to step in and propose new solutions which address the brittleness, transferability and scalability concerns addressed with logical programming approaches. Neural approaches showed to yield significant

improvements in terms of transferability and overall accuracy which led to a paradigm shift in this research field.

2.3.1 Early Neural Approaches

In **?** **?** released Seq2SQL which represents a significant breakthrough and leap in NLIDB research. Seq2SQL was an early research system that in the field of neural network application and as one of the first papers to frame the implementation of NLIDBs / NL2SQL Systems as a reinforement learning problem. The system utilized iterative query execution in the reward function to improve its accuracy (**?**). In the same paper **?** introduced WikiSQL, a training dataset, which enables large scale (in **?**) model training.

SQLNet (**?**) addressed primarily the order-sensitivitiy trait of Seq2SQL (**?**) that was prevalent due to being a derivative approach from sequence-to-sequence approaches. SQLNet diverges from sequence-to-sequence and joins multiple research threads, employing a sketch-based query generation. SQLNet breaks down complex queries into smaller (hence more manageable) subqueries which can then be individually sketched and refined, yielding a system that outperformed state-of-the-art by 9% to 13% (**?**).

? have introduced TypeSQL, a variation of the SQLNet-approach, in **?**. TypeSQL's primary difference to SQLNet is the encoding of type information for SQL generation. The approach scanned for entity references and values in natural language and was able to improve performance by 5.5% over SOTA-Models like SQLNet whilst requiring significantly less training time, indicating that type information was a useful information for deriving accurate SQL queries from user input (**?**).

2.3.2 Intermediate Neural Developments

Later in **?** **?** released SyntaxSQLNet, a followup research to TypeSQL, which represented a slight change in approach and research focus. In direct comparison SyntaxSQLNet focused primarily around complex query generation using a syntax tree decoder, allowing for longer and more cohesive query generation (**?**). This advancement over TypeSQL allowed more complex queries to be reliably generated, enabling multiple clauses aswell as nested queries. SyntaxSQLNet was one of the earlier research efforts which utilized Spider instead of WikiSQL (introduced by **?**), a large-scale NL2SQL dataset, incorperating 10.181 hand annotated natural language question and alongside 5.693 unique SQL examples that spread across 138 different domains (**?**). This research led the transition of comparatively simple, research-grade, neural systems for NLIDBs towards systems which are feasible in the real world.

Building on the above approaches, **?** have introduced IRNet, a neural network approach using intermediate representation as a bridge between natural language and SQL in which semantic queries could be expressed. The intermediate format SemQL (or semantic query language) was utilized to transform and synthesize queries on the actual database schema more accurately than Seq2SQL. IRNet followed a three phase approach: schema linking between the natural language query and database layout, synthesis of SemQL as intermediate representation and deterministic conversion of SemQL to SQL. This approach allowed IRNet to outperform state-of-the-art approaches on the Spider benchmark by 19.5%, placing IRNet at an overall accuracy of 46.7% (**?**).

Following IRNet, graph neural networks (GNN) have been explored as alternative architecture by **?**, representing the database schema as a graph and using message passing to model relationships between tables, columns and natural language input. This approach demonstrated the capability to improve reasoning and query generation capability. **?** showed that when evaluating against the Spider benchmark GNN outperforms both SyntaxSQLNet (and therefore SQLNet

and TypeSQL). Although presenting a signficiant advancement over previous state-of-the-art approaches, GNN falls behind in performance against IRNet by 6% (**??**).

2.3.3 Relation-Aware Transformer Approaches

The release of RAT-SQL (Relation-Aware Transformer for SQL) **?** represents the most significant leap in research of neural NL2SQL approaches. RAT-SQL diverged from earlier research through emphasizing the relationship between natural language and the database schema elements using relation-aware self-attention representing a novel approach for solving *schema linking* (**?**).

RAT-SQL's primary innovations was the ability to infer, understand and utilize the relationship between individual tokens in the natural language query and link it to the database schema. Thus allowing for reasoning capabilities on the actual database schema while generating the query.

This architecture yielded a 57.2% in exact match accuracy when being evaluated on the Spider benchmark, substantially outperforming comparative approaches like GNN, IRNet and IRNet V2 by 10.5%, 9.8% and 8.7% respectively. Although overall accuracy improved across all approaches when being paired with BERT (Bidirectional Encoder Representations from Transformers, a popular pre-trained lanugage model from Google) the δ between the indivudual approaches remained relatively steady, leaving RAT-SQL outperforming state-of-the-art approaches by 5% to 12.2% further demonstrating the capability advancement yielded by this system (**?**).

2.3.4 Comparative Analysis of Neural Approaches

The evolution from early neural approaches to RAT-SQL emphasized the rapid advancements that happened in the research field of neural NL2SQL approaches in different dimensions:

1. **Model Complexity** — Given the research progression from early sequence to sequence translation approaches (Seq2SQL) towards sketch based and type augumented and graph based approaches (TypeSQL, SQLNet, GNN) and syntax tree decoding emphasized by SyntaxSQLNet, neural approaches continously advanced in the complexity of approaches that is required to beat state-of-the-art approaches in contemprorary benchmarks like Spider. RAT- SQL presents one of the late and most complex advancements in the field of neural NL2SQL approaches with its adapted self attention mechanism (**?????**).

2. **Transferability** — Each of the approaches introduced above represents a succession in terms of their transferability. The field of neural NL2SQL approaches significantly improved the ability for NLIDBs to generalize over the underlying database schemas. RAT-SQL showed the strongest cross-domain accuracy (that is benchmarked by the Spider benchmark). With standard benchmarks emerging it became easier to verify and quantify which approach had the highest transferability as Spider specifically had independent development and test datasets, preventing approaches from over-optimizing on training data (**?**).

3. **Robustness** — As research systems advanced in complexity and shifted from raw input to output translation (Seq2SQL) their robustness steadily increased. Through more approaches like SyntaxSQLNet which utilized structured decoding, IRNet which relied on an intermediate representation and RAT-SQL the challenges around *schema linking* outlined by **?** have increasilingly led to more robust systems that can handle rephrasings, spelling mistakes and variations in natural language usage far beyond what traditional NL2SQL approaches could accomplish **???**.

4. **Query Complexity** — The performance on complex queries involving multiple tables, relying on complex aggregations, nested structures and joins dramatically improved over the course of the research that happened in this field. Whilst IRNet reresents one of the first singificant advancements when it comes to the ability of neural approaches to handle complex queries, RAT-SQL still showed to outperform the intermediate representation approach introduced by IRNet by up to 10.5% (**??**).

5. **Schema Understanding** — Whilst early approaches like Seq2SQL primarily applied reinforcement learning for end to end query generation (**?**), later approaches like TypeSQL, GNN and specifically RAT-SQL showed novel and state-of-the-art *schema understanding / schema linking* capabilities, yielding the ability to accurately reason about user intent and traverse the database schema while generating queries (**???**).

2.3.5 Limitations of Neural Approaches

Despite the dramatic *accuracy, transferability* and *robustness* improvements that could be observed with late neural approaches (**??**), neural approaches still suffered from serious shortcommings / unsolved challenges:

1. **Training Data** — Utilizing neural networks these approaches required susbtantially more training data (ie. natural language paired with output SQL queries) than traditional systems which required serious efforts of data collection (**?**).

2. **Correctness** — The inherent mismatch between neural networks and formal languages yielded cases where models produced invalid SQL code. Approaches like SyntaxSQLNet improved the tried to solve this circumstance by utilizing syntax trees during decoding but nonetheless syntactic correctness remained a challenge across future iterations of neural systems. (**?**)

3. **Domain Language** — Despite increased *transferability* characteristics neural approaches still suffered from a limited vocabulary and inter-domain understanding of terminology and relation between concepts which made highly domain specific natural language queries challenging.

4. **Observability** — The black-box nature of neural networks made approaches relying on them, particularly the ones with complex architectures, hard do understand / explain in case when neural systems yielded undesirable output.

The introduction and advancement of early neural NL2SQL approaches led to significant advancements in the research and feasibility of NLIDBs. The research shift started in this era established the foundations for further and more advanced machine learning approaches (specifically language model oriented approaches)being researched. Neural approaches showed to significant improvements in performance when being paired with pre-trained language models (**?**) which led to further research on their applicability.

2.4 Pre-trained Language Models

The advantages of combining specialized neural networks with general-purpose pre-trained language models led to a pivotal point in the NL2SQL research field towards focusing increasingly on the application of pre-trained language models for NLIDBs. Models like BERT or T5 offer noticable performance improvements (especially when it comes to language understanding) over specialized NL2SQL networks due to training happening on unrestrained amounts of natural

language data, instead of pure NL2SQL datasets which are often fairly limited in size and therfore natural language use — SPIDER2.0 which is a contemprary NL2SQL benchmark consists of just 632 real-world questions (**?**). Thus PLM-based (or at least augumented) NL2SQL systems can observe dramatic performance improvements through the language models's ability to understand patterns and identify semantic relationships of natural language query elements.

2.4.1 Early Pre-trained Language Model Adaptations

The above outlined benefits have led to concrete research efforts focusing on the question whether the sole application of pre-trained language models could outperform neural state-of-the-art approaches — which often implicitly require a far more sophisticate architecture when it comes to natural language analysis.

In the time of emerging PLM applicationm GRAPPA was introduced by **?** in **?** — a novel grammar-augmented pre-training approach built on $\text{RoBERTa}_{\text{LARGE}}$ (a derivative model from BERT). It generates synthetic training data (ie. natural language and sql pairs) using a synchronous context-free grammar (SCFG) which analyses and identifies patterns in natural language queries that can be used as templates for sythesizing training data. The specialized pretraining helps GRAPPA to establish a robust connection between natural-language and database schema elements, showing significant improvements on existing approaches on multiple contemporary benchmarks like SPIDER and WIKISQL (**?**).

Several NL2SQL approaches in this era focused on *schema understanding* and *schema linking* — the generalizability of PLMs required advanced techniques on ensuring that models both understand the semantic intent of users when querying and correctly identify database schema elements in natural language queries. Thus improving semantic accuracy of generated SQL queries. STRUG (Structure-Grounded-Pretraining) was introduced in **?** by **?** and presented a novel pretraining approach that improves model abilities when it comes to *schema linking*, it separates the problem in three facets: column grounding, value grounding and column-value mapping. In direct comparison with GRAPPA, STRUG achieves similar performance while being significantly cheaper to train (**?**).

In parallel, **?** released GAZP (Grounded Adaptation for Zero-shot Executable Semantic Parsing) in **?**. **?** specifically addressed the challenge of adapting semantic parsers across databases / domains which was a apparent problem with neural approaches which had a strong tendency to overfit on benchmark datasets. Its novel contribution was the combination of forward semantic parsing with a backward utterance generator which allowed for data synthesis in unseen environments which could then be used to adapt the semantic parser (**?**). This approach enables a improvement in robustness and accuracy in situations where training and inference environments differ without requiring manually annotated examples (**?**).

2.4.2 Advanced Pre-trained Language Model Approaches

Building on earlier foundational research on PLM application for NL2SQL tasks, researchers have developed increasingly complex systems that leveraged pre-trained language models whilst addressing their limitations when it comes to generating valid SQL.

? introduced RYANSQL (Recursively Yielding Annotation Network for SQL) in **?**, which implements a sketch-based approach for decomposing complex SQL generation into multiple smaller problems. RYANSQL transformed nested statements into a set of top-level statements using the Statement Position Code (SPC) technique. This flattening of structure allowed RYANSQL to limit the complexity of the query generation problem whilst maintaining its ability to answer complex questions by recomposing complex queries from their parts. This approach allowed

RYANSQL to achieve 58.2% accuracy on the SPIDER benchmark, representing a 3.2% improvement over contemporary state-of-the-art approaches at the time (**?**). The sketch-based approach makes RYANSQL a PLM-augumented successor of SQLNET which was a early neural approach to employ sketch-based query generation (**??**).

A significant advancement in terms of execution accuracy was reached with the application of T5-Models for NL2SQL tasks. T5 (Text-to-Text Transfer Transformer) Models have proven themselves as well-suited for for query generation — T5-3B for NL2SQL yielded 71.4% execution accuracy and thus presented a breakthrough in this domain of research (**?**). This established a new baseline for PLM-based approaches and demonstrated that general-purpose language models could not only compete but outperform specialized architectures by far when properly fine-tuned (**?**).

Following the adavancements through T5, **?** introduced GRAPHIX-T5 in **?**, which combined the T5 PLMs with a further graph-aware layers for NL2SQL tasks. This architecture could leverage both pre-trained knowledge of T5 models aswell as the database schema structure during inference. GRAPHIX-T5 constructs a schema graph where nodes represent tables and columns and edges represent relationships between them, such as foreign keys or columns association. This architecture allows the model to deeply understand relationships and the layout of the database schema (**?**). GRAPHIX-T5 outperformed standard T5 models significantly, with GRAPHIX-$T5_{LARGE}$ showing 6.6% increase in execution accuracy over $T5_{LARGE}$. When both GRAPHIX and the baseline T5 models were combined with PICARD (a novel constrained decoding mechanism) absolute δ between them jumped to 7.6% (81.0% in absolute numbers), evaluated on SPIDER-DEV (**?**).

In parallel, **?** proposed RESDSQL in **?**, which proposed to decouple *schema linking* and *skeleton parsing*. This addressed the typical challenges sequence-to-sequence models faced when simultaneously trying to link both schema elements and generate the query skeleton (e.g. `SELECT <columns> FROM <table>`). RESDSQL further employed a ranking approach to filter schema elements before passing them to the model for query generation, which reduced noise (when working with large database schemas) and enabled passing only the most relevant parts. This twofold approach allowed RESDSQL to achieve state-of-the-art performance when being evaluated on SPIDER, outperforming GRAPHIX-$T5_{3B}$-PICARD by 0.8% in execution accuracy. When combined with NATSQL (a contemporary intermediate representation approach introduced by **?**) absolute improvement over GRAPHIX-$T5_{3B}$-PICARD jumped to 3.1% emphasizing the robustness gain decoupeled architectures have over model-oriented approaches.

These advancement showed rapid improvements over earlier methods — far surpassing neural approaches — through advanced mechanisms when it comes to schema understanding and query generation. The wide language understanding inherited from PLM-basemodels further strengthens robustness and shows effectiveness through a large gain on the SPIDER benchmark. Collectively these approaches represent a leap in NL2SQL research, emphasizing their usability potential and real-world feasibility. This era primed the research field for the transition towards large language model adoption.

2.4.3 Constrained Decoding and Ranking Techniques

A major challenge in NL2SQL research is making sure that model generated queries are not just semantically accurate but also syntactically valid queries and thus executable. To address this issue **?** released PICARD (Parsing Incrementally for Constrained Auto-Regressive Decoding) in **?**, a constrained decoding mechanism for language models which utilizes the SQL grammar and constrained decoding mechanisms to incrementally parse the generate SQL, rejecting invalid tokens based on the grammar. PICARD showed to significantly improve the performance of pre-

trained language models (like T5 or BERT) when it comes to NL2SQL tasks, lifting them from mid-level to state-of-the-art solutions on the SPIDER benchmark (**?**).

PICARD operates as a incremental parser during model output decoding of pre-trained language models and continuously evaluates the probability of each token. Instead of just passing model outputs to a database for execution PICARD incrementally parses and validates the generated SQL, rejecting tokens if neeeded thus significantly improving the valid output accuracy (sometimes referred to as VA) of language models. This approach is addressing a significant issue associated with pre-trained language models — while they outperform in natural language understanding and reasoning, they often lack SQL grammer knowledge and tend to generate queries that are not executable due to ther unconstrained output space (**?**).

The above introduces RESDSQL built ontop of PICARD's foundations and used a ranking-enhanced framework for input encoding. These two approaches represent a unqiue class of approaches that utilize input and output constraining in order to increase the performance characteristics of pre-trained language models (**?**).

2.4.4 Advantages of PLM Approaches

PLM approaches to NL2SQL tasks have yielded significant performance improvements for the NL2SQL domain and represent a leap in NLIDB-research. They primed the research field towards using language models which led to a transition towards large language models in the following years. Namely PLM approaches brought a series of upsides with them:

1. **Compute Efficiency** — PLMs like RESDSQL achieve high accuracy (up to 84.1% on SPIDER depending on variants) wilst using far fewer parameters than conteporary LLMs, making them significantly more efficient and therefore reduce hardware requirements for their deployment (**?**).

2. **Transferability** — Approaches like GRAPPA and STRUG can incorporate domain-specific understanding of natural language, table structures and SQL syntax during pre-training which addressed one of the primary issues with neural approaches (**??**).

3. **Vocabulary** — PLMs offer a larger vocabulary due to the vast amounts of training data available. This enables them to handle a wide variety of natural language patterns which addresses the benchmark-overfitting tendency of neural approaches which primarily trained on the development sets of contemporary benchmarks.

2.4.5 Limitations of PLM Approaches

Although representing the state-of-the-art at the time, PLMs introduce a class of problems which are associated with their non-NL2SQL associated nature. There have been an array of approaches to mitigate these shortcommings but nonetheless they must be considered when using a PLM-based approach to NL2SQL:

1. **Fine-tuning Requirements** — Most PLMs require substantial domain-specific, or at least NL2SQL specific, fine-tuning, limiting a straight forward adaptation to new domains or databases. Although being significantly more efficient than LLM-based approaches the potential need for initial fine-tuning represent a significant computational resource burden. Furthermore when not using synthetic data generation (e.g. GRAPPA) annotated datasets of training data are needed to achieve appropiate performance characteristics (**?**).

2. **Wide Input & Output Space** — Due to the general nature of PLMs their input and output space is often far larger than needed NL2SQL tasks. "Large pre-trained language

models for textual data have an unconstrained output space; at each decoding step, they can produce any of 10,000s of sub-word tokens" (**?**). This applies to both the input and output token space, therefore multiple approaches have been researched which focus on constraining these to the subset needed for NL2SQL tasks. Namely GRAPHIX-T5 and PICARD have proposed potential (and promising) solutions to this issue (**??**).

3. **Limited Schema Awareness** — Due to being general purpose, and non-NL2SQL optimized, PLMs tend to incorporate limited amounts of schema awareness when being applied out of the box for NL2SQL tasks. Multiple research efforts focused on improving this situation, most notably RESDSQL and GRAPHIX-T5 tried to improve the schema linking & awareness of PLMs (**??**), nonetheless the non-specialized nature of PLMs prevents NL2SQL being part of the fundamental model architecture.

These characteristics positioned PLMs as powerful but comparatively resource-intensive solutions for NL2SQL (especially in direct comparison with neural approaches), ultimately yielding the research domain to transition toward exploring Large Language Model approaches that promise even greater flexibility in adaptation and potentially superior handling of complex queries through advanced in-context learning approaches.

2.4.6 Comparison with Large Language Models

The research on applying pre-trained language models for NL2SQL tasks primed the field for the transition towards LLM usage. While PLMs like T5 and BERT range from millions to a few billion parameters, prevalent LLMs such as GPT-3 and GPT-4 operate at significantly larger scales, ranging from a few billions to hundred of billions parameters. The scale of LLMs enables in-context learning techniques that enable significantly easier and cheaper transferability of NL2SQL systems across domains (**?**). The δ of deployment, inference and training requirements of these two approaches are significant due to the size difference in models, which transfer to hardware requirements and therefore cost. While PLMs can require extensive fine-tuning on domain-specific data which may aswell be resource intensive (**????**), LLMs transfer the cost to the inference environment, where model modificants are less impactful, due to the extensive pre-training that took place. Approaches like DINSQL show that with the application of LLMs the engineering challenges around model instruction gained relevance while model training became less of a central problem to solve (**?**).

2.5 Large Language Models

The emergence of LLMs such as GPT-3, GPT-4, and Claude fundamentally transformed the landscape of NL2SQL research. Early experiments with LLMs for NL2SQL tasks showed state-of-the-art capabilities in comparsion with contemporary PLM approaches (**?**). **?** demonstrated that CODEX (a contemporary model based on GPT-3), without any fine-tuning efforts, could achieve competitive performance on SPIDER, outperforming many state-of-the-art approaches that required extensive training. This breakthrough challenged the contemporary assumption that further specialization of model architectures would yield increases in NL2SQL performance (e.g. GRAPHIX-T5) (**?**).

2.5.1 In-Context Learning

In-Context Learning (ICL) is a foundational approach for leveraging the ability of LLMs to utilize larger context windows for inference than traditional PLMs. Typical context windows of state-of-the-art LLMs can reach up to hundred thousands of tokens. This characteristic of LLMs

enabled researchers to utilize this context window to provide examples of accurate NL2SQL translation instead of applying parameter updates. This paradigm shift has made developing NL2SQL systems significantly more accessible.

The fundamental principle of few-shot learning for NL2SQL involves providing the LLM with a small number of example pairs of natural language and their corresponding SQL representation. These examples can benefit the model's understanding of mapping between natural language and SQL syntax. This essentially builds on top of prior research like GRAPPA and STRUG, but applying these examples at inference time, rather than training time. Although this increases the inference cost of such a system, the upsides lie primarily in the flexibility of such an approach — database content / prior usage of the system can be dynamically utilized, rather than requiring retraining.

Example selection strategies showed to have a considerable impact on ICL performance. **?** evaluated various example selection methods like *Random*, *Question Similarity Selection (QTS)*, *Masked Question Similarity Selection (MQS)*, and *Query Similarity Selection (QRS)*. **?** propose a novel strategy to select, organize and present ICL examples to LLMs. DAIL-SQL utilizes both question and query similarity, masking domain-specific words and prioritizing examples that exceed a similarity threshold of τ (**?**, p. 5). DAIL-SQL encodes examples as question-SQL-pairs without the respective schema to improve token efficiency. Using a Code Representation Prompt (CR) for question and schema encoding yielded DAIL-SQL to achieve state-of-the-art 86.6% execution accuracy on SPIDER.

The comparison between zero-shot and few-shot performance reveals the accuracy gain potential through supplying examples to models in the inference context. While contemporary LLMs (such as GPT-4) have demonstrate impressive zero-shot performance (achieving 72.3% execution accuarcy on benchmarks like SPIDER) (**?**, Table 1, p. 8), few-shot learning still shows to substantially improve model performance. **?** shows that even one-shot learning boosts GPT-4's execution accuracy to 80.2%, representing a 7.9% increase, while five-shot learning reaches 82.4% (**?**, Table 2, p. 8).

Especially with complex queryies which can involve multiple tables, nested queries and complex joins, zero-shot approaches often dramatically underperform k-shot ones. NL2SQL approaches that dont supply examples to the model during inference time fail more frequently to generate semantically accurate SQL queries (**?**). Notable is the leap in exact match ratio measured by the SPIDER benchmark — jumping from 22.1% for GPT-4 using zero-shot to 71.9% with five-shot. The results presented by **?** show signficant correlation between k and the execution accuracy of k-shot approaches.

This effectiveness has established ICL approaches as a standard technique applied in LLM-based NL2SQL approaches. Contemporary approaches like XIYAN-SQL, CHASE-SQL AND DIN-SQL all utilize variations of ICL to achieve state-of-the-art results (**???**).

2.5.2 Self-Correction and Iterative Refinement

? proposed DIN-SQL as an innovative approach to NLIDBs that rely on LLMs. DIN-SQL decomposes complex queries into sub-parts and utilizes in-context-learning and self-correction during the generation phase. Compared to DAIL-SQL which relies on example selection during the in-context-learning phase, DIN-SQL focuses on a refinement loop that allows the model to self- correct errors it made during the initial generation phase — thus the model can repair schema linking, syntactic or semantic errors. By explicitly instructing the LLM to review its work against a specfic schema, the user input and potential database errors, DIN-SQL achieves a high execution accuracy on SPIDER with 85.3%. Therefore DIN-SQL outperforms contemporary approaches but is surpassed by by DAIL-SQL by 1.3% (**??**). Furthermore DIN-SQL makes

observations on the impact that the self-correction prompt can steer results significantly — **?** found that using *generic self-correction* (ie. assuming the query contains errors) lowers the execution accuracy by 4.2% on SPIDER compared to *gentle self-correction* (ie. assuming nothing about the validity of the query). It was noted that the impact of the self-correction mechanism relies on the model size, with smaller models performing better with *generic self-correction* and larger models performing better with *gentle self-correction* (**?**). The self-correcting nature of DIN-SQL represents a diversion from DAIL-SQL's emphasis on input optimization towards output refinement. **?** demonstrate how structured introspection can play a significant role in enhancing LLM performance for formal language generation tasks.

Building upon DIN-SQL's self-correction module, **?** proposed MAGIC (Multi-Agent Guideline for In-Context Text-to-SQL), which further advances the self-correction mechanism through harnessing a set of specialized agents to automate the self-correction prompt engineering (**?**). MAGIC consists of a manager agent, a correction agent and a feedback agent that collaboratively refine LLM instructions during the refinement loop. Further MAGIC derives common faliure patterns of the initial query generation phase from training data, allowing it to efficiently spot the most common mistakes that the model makes at generation time. This approach represent a further advancement on **?**'s DIN-SQL, effectively supersetting the *generic* and *gentle* correction mechanisms through an intelligent, self-adapting one (**?**). The autogenerated guidelines from MAGIC yield 85.6% execution accuracy on the SPIDER development set — representing a 5.31% improvement over DIN-SQL's human written correction guidelines. These results emphasize that optimized self-correction mechanisms have the ability to significantly drive up overall system performance of NLIDBs (**?**).

While DIN-SQL and MAGIC focus on automated self-correction in single-turn settings, **?** introduced the concept of Chain-of-Editions (CoE-SQL), which addresses the unique challenges of multi-turn conversational NLIDBs. Conversational interfaces for NL2SQL systems enable human-in-the-loop refinement. Interactive information seeking from the user has shown to be an effective way to drive overall accuracy of the system and improve user satisfaction (**?**). Rather than approaching each query independently, CoE-SQL recognizes that in a conversational context, successive SQL queries usuallly require only small and incremental modifications of the previous queries. Interactive user input is an effective measure for dealing with ambiguous natural language queries (**???**).

2.5.3 Candidate Selection Frameworks

Contemporary NL2SQL approaches have increasingly emphasized on the generation of query candidates and their selectiion as a promising architecture. Candidate selection strategies have shown significant performance improvements on challenging benchmarks. These approach acknowledge the inherent difficulty of generating perfect SQL queries in one attempt / using one generation mechanism, even with capable LLMs and modern self-correction mechanisms.

? introduced CHASE-SQL in **?**, a framework that leverages multiple reasoning paths to generate multiple query candidates. After the initial generation phase CHASE selects the most promising solution to the natural query input. CHASE-SQL harnesses three different generation strategies: A divide-and-conquer approach which breaks down complex natural language queries into multiple sub tasks that can be individually tackled, a chain-of-thought based generation approach which inspects execution plans of SQL queries and a schema-aware generation of synthetic examples that can be used for in-context learning (**?**). These different generation mechanisms produce a set of query candidates that each have different characteristics. For candidate selection CHASE harnesses a fine-tuned LLM that can do binary selection of candidates. **?** have demostrated that their query selection approach is more robust than apparent alternatives and

yields state-of-the-art performance with 73% execution accuracy on BIRD and 87.6% on SPIDER (**?**).

Conceptually similar work has been done by **?** with XIYAN-SQL which is architected as multi-generation ensemble strategy with better schema representation. XIYAN-SQL integrated in-context learning alongside supervised fine-tuning approaches to generate query candidates (**?**). A key contribution of **?** is their M-Schema representation of database schemas, which improves the models schema awareness and reduces frequent schema linking errors. XIYAN-SQL enhances accuracy by utilizing multiple different strategies that have complementary characteristics during query generation to enhance the robustness of the overall system. The query generation stage utilizes both a fine-tuned SQL generation model aswell as ICL strategies to achieve a breadth of candidate coverage. Following to the query generation stage a self-correction stage (referred to as *refinement* stage by **?**) is utilized to correct common errors. Lastly a selection model is used to choose the most accurate candidate that was produced during the generation stage (**?**). Through this sophisticated and diverse architecture XIYAN-SQL was able to achieve impressive results across contemporary NL2SQL benchmarks — achieving 89.65% execution accuracy on SPIDER and 73.34% on BIRD, which renders XIYAN-SQL state-of-the-art (**?**).

Both CHASE and XIYAN-SQL show that diversifing candidate generation and training specialized models for candidate selection yield state-of-the-art execution accuracy which significantly outperforms single-path generation approaches. The success of these two approaches indicates that for increasingly complex NL2SQL tasks (such as SPIDER2.0), the capability to generate multiple valid interpretations of the natural language query is an important stepping stone to achieving meaningful execution accuracy. Both CHASE and XIYAN-SQL rely on specialized candidate selection models which renders these approaches to combine the strengths of LLMs when it comes to language understanding and transferability with the robustness of specialized model architectures for candidate ranking and selection.

2.5.4 Retrieval-Augumented Generation

Retrieval-Augmented Generation (RAG) has emerged as a powerful paradigm for enhancing NL2SQL systems by integrating external knowledge retrieval with the generative capabilities of LLMs. This technique is seen in all above introduced papers in varying forms (**??????**). The most prevalent form of RAG in NL2SQL is the encoding of the database schema into the LLM prompt for in-context-learning. This allows LLMs to be aware of the table structures and names, foreign key relationships, primary keys etc. **?** proposed the serialization of database schemas in the M-SCHEMA format, a semi structured, text-based schema description. **?** proposed to encode the schema using a Code Representation Prompt (CR) which refers to the encoding of the raw SQL statements need to construct the schema. **?** provided an ablation study for the M-Schema which yielded questionable results on the optimization of this approach. While the M-Schema format yielded best results when XIYAN-SQL was combined with GPT-4o or Claude 3.5 Sonnet, it performed worse than alternatives on DeepSeek and Gemini models (**?**).

A relevant optimization technique for RAG is the selection of a schema subset before encoding the schema for the model. RESDSQL was one of the earlier approaches to explore subset-encoding of database schemas with **?** building on top of this (**??**). **?** introduced ASTRES which dynamically retrieves database schemas and uses abstract syntax trees (ASTs) to select optimal few-shot examples for ICL. By pruning the ASTs down to the most relevant subset, ASTRES achieved the highest at-the-time (**?**) execution accuracy on SPIDER with 86.6% indicating that subset-encoding is a sensible optimization mechanism. ASTRES was combined with GRAPHIX-T5 in order to achieve this result (**?**).

The impact of RAG when NL2SQL systems face large and complex database schemas has

been particularly significant. Traditional approaches struggle when database schemas contain hundreds or thousands of tables and columns, as the complete schema may not fit within model context windows depending on their size. RAG-based systems address this by dynamically retrieving only the most relevant portions of the schema based on the natural language query. The technique of subset-encoding becomes especially relevant in enterprise environments where database schemas can be extremely large and complex. Recent benchmarks like Spider2.0 emphasize enterprise environments and show that existing solutions underperform in those scenarios, often reaching single digit execution accuracy.

The work of **?** indicates that prefiltering of the environment of language models is an effective and promising technique that has the ability to reduce computational requirements of contemporary NL2SQL systems. As introduced above, recent state-of-the-art systems often utilize closed-source models like Gemini, GPT-4(o), Claude 3.5/3.7 Sonnet etc which often come with massive parameter sizes (reaching hundreds of billions of parameters). ASTReS demonstrated that efficient schema retrieval mechanisms enable smaller models (e.g. GRAPHIX-T5) to achieve state-of-the-art performance against LLM based approaches like DAIL-SQL (**??**).

2.5.5 Specialized LLMs and Fine-tuning

While general-purpose LLMs demonstrated state-of-the-art natural language understanding capabilities, the research domain of NLIDBs increasingly explored the potential of fine-tuned LLMs for NL2SQL tasks, which offer a promising tradeof between natural language understanding (ie. breadth of the model) and concrete SQL generation capabilities (ie. depth of the model). Multiple research works have been done on the fine-tuning of language models which yielded a series of dedicated models for optimized SQL generation.

A significant limitation of many LLM-based NL2SQL solutions is their dependency on proprietary and closed-source LLMs like GPT-4(o), Claude and Gemini. Whilst they are useful for initially proving the potential of LLMs on contemporary benchmarks, this dependency introduces significant concerns related to data-privacy, high-side use (e.g. in classified environments), transparency of data-flow and deployment costs. To address these challenges **?** have introduced CodeS in **?**, a series of of open-source language models dedicated for NL2SQL tasks with parameter sizes ranging from 1B to 15B. The CodeS models were evaluated against contemporary benchmarks like Spider and Bird and showed promising inference results when compared to their closed-source counterparts. **?** showed that CodeS 7B achieves 85.4% execution accuracy on the Spider development set, outperforming both fine-tuning approaches like RESDSQL and GRAPHIX-T5-PICARD aswell as prompting based methods DIN-SQL+GPT-4 and DAIL-SQL+GPT-4 (**?**). The same tendency was observed on Bird with CodeS-15B achieving 60.37% execution accuracy, compared to 57.41% for DAIL-SQL+GPT-4 and 55.90% for DIN-SQL+GPT4 (**?**). This marks a significant advancement in the open-source language model research area, with CodeS reaching new state-of-the-art performance in **?**. While more recent approaches like XiYan-SQL and CHASE both outperform CodeS, CHASE relies on proprietary models and XiYan-SQL doesnt provide any information on what base models where used.

? addressed serveral critical research challenges in the NL2SQL domain and proved that open-source models could perform competitively with proprietary models whilst maintaining a significantly smaller parameter footprint (ie. 7B and 15B) compared to GPT-4 which is a multi-hundred-billion parameter model. This makes it feasibile to deploy CodeS locally, instead of relying on an enterprise API like OpenAI's one (**?**), therefore making it highly practical for real-world deployments where computation resource are constrained.

? published a follow-up paper in **?** which introduced the next-generation OmniSQL models

with 7B, 14B and 32B sizes, trained using synthetic data generation. OmniSQL achieves state-of-the-art performance when compared to alternative LLMs - including both open-source and closed-source competitors. The models achieve significant execution accuracy improvements on both Spider and Bird, the 7B model reaches 88.9% on Spider and 66.1% execution accuracy on Bird which represents a significant (5%+) improvement over comparable alternatives (**?**). These results were achieved without combining OmniSQL with advanced in-context-learning, self-correction, retrieval-augumented-generation or candidate-selection techniques, which indicates that even higher accuracy scores are possible.

2.5.6 Limitations and Challenges

Despite the impressive advancements of LLM-based approaches for NL2SQL, several significant limitations and challenges are apparent that impact their practical implementation and real-world deployability.

1. **Hallucination and Accuracy Concerns** — Since LLMs demonstrate a tendency to hallucinate, LLM-based NL2SQL systems face the challenge of detecting when LLMs generate plausible but incorrect SQL queries. The breadth of possible errors ranges from detectable errors like invalid table or columns references, invalid SQL syntax etc. to undetectable, slight errors in semantics, like reversing the order of aggregations or using the wrong aggregation method. Contrary to traditional approaches which fail visibly with unknown structures, LLM-based approaches might produce semantically flawed queries that execute without errors and yet return semantically incorrect data. As noted by **?**, "achieving Enterprise-Grade NL2SQL is still far from being resolved" even with state-of-the-art models, particularly when handling complex real-world database schemas and ambiguous queries (**??**).

2. **Computational Resource Requriements** — The resource intensity of LLM-based systems presents barriers to widespread adoption. High-performance state-of-the-art models require substantial computational resources during inference, making them expensive to deploy at scale. While smaller models exist, they typically show reduced performance on complex queries, creating a challenging trade-off between accuracy and resource efficiency. Studies like DAIL-SQL demonstrate a direct correlation between model size and performance, where the most accurate systems are also the most challenging to deploy economically (**?**). Even state-of-the-art specialized LLMs like OmniSQL degrade in performance as parameter sizes shrink, although they maintain a significantly higher parameter efficiency, outperforming enterprise-level closed-source competitors.

3. **Data Privacy and Security Implications** — Using LLMs particularly through proprietary APIs brings along considerable concerns with regards to data privacy and security as sensitive data may be transferred to respective model vendors. Potential senstive data aswell as database schemas alongside user questions needs to be communicated to external parties in order to form a usable system if closed source models are used. Recent open-source model developments like CodeS and OmniSQL seem to mitigate this problem partially, but as mentioned above using local inference with LLMs brings along stark computational requirements.

4. **Ambiguity Handling in Complex Scenarios** — Following previous paradigms LLMs also continue to struggle with effective ambiguity resolution in natural language queries over complex database schemas. These challenges are particularly prominent in large scale enterprise environments where similar column names exist across multiple tables,

or domain-specific terminology has multiple potential interpretations that can result in different queries. Though approaches like multi-path reasoning and candidate selection show promising improvements in execution accuracy, they often increase inference time and complexity as they run multiple parallel inference steps and rank their results (**??**).

5. **Competitive PLM Approaches** — Prevalent PLM-based approaches like RESDSQL and GRAPHIX-T5 achieved impressive execution accuracy while using significantly fewer parameters than LLMs, making them more computationally efficient (**?**). Despite reducing inference-time requirements, these approaches typically utilize re-training and domain specific fine-tuning which both limits their flexibility and introduces training costs. Yet for certain scenarios PLM-based approaches offer an interesting tradeoffs: Frontloading the computational effort can be interesting in compute-constrained environments and help to reduce ongoing costs. LLMs dont require the upfront cost of fine-tuning and adaption but require significantly more resource for their deployment. This indicates that certain systems and environments might benefit more from PLM approaches than LLM ones. Contemporary research, which typically focuses around state-of-the-art-performance on benchmarks, shifted primarily towards LLMs due to their transferability and natural language coverage.

2.6 Benchmarking

In order to evaluate NL2SQL systems standardized benchmarks like Spider and Bird have emerged. These benchmarks can measure performance across different approaches and models, enable meaningful ablation studies and are a useful indicator for the state of the research field. In the past decade significant advancements have been made with Spider being released in **?** the first major, widely adopted, benchmark emerged in this field (**?**).

2.6.1 Spider

Spider, introduced by **?** in **?**, has become the de facto standard benchmark for evaluating complex and cross-domain Text-to-SQL systems. It consists of 10,181 questions and 5,693 unique SQL queries spanning 200 databases across 138 domains. Previous benchmarks like lacked complexity and cross-domain distrubtion of datapoints which prevented the *transferability* of approaches or models to be accounted for in benchmarks. With Spider the capability to be database agnostic was required to achieve meaningful accuracy scores. Furthermore Spider was split in training and test sets which contain different database in order to prevent overfitting models from succeeding. This design specifically tests a model's ability to handle schema linking and generalization challenges rather than memorizing specific database patterns. Spider evaluates both *exact matching accuracy* and *execution accuracy*, with contemporary state-of-the-art systems achieving approximately 85-90% *execution accuracy* (as of 2025) (**????**).

2.6.2 Bird

The Bird benchmark (BIg bench for large-scale database gRounded Text-to-SQLs), released in **?**, and addresses the gap between academic benchmarks and real-world applications by focusing on large-scale databases with actual data content (**?**). Bird contains 12,751 text-to-SQL pairs and 95 databases with a total size of 33.4 GB across 37 professional domains. Unlike Spider, which primarily evaluates against database schemas with minimal content, Bird emphasizes challenges related to dirty database contents, external knowledge between natural language questions and database values, and SQL efficiency in massive databases. This places Bird as a relevant benchmark for real world feasibility of approaches and models. Even state-of-the-art LLMs like GPT-4

achieve only 54.89% execution accuracy on BIRD, compared to human performance of 92.96%, highlighting the significant challenges posed by real-world database scenarios on NLIDBs (**?**).

2.6.3 Spider 2.0

SPIDER 2.0 which was introduced by **?** in **?** represents the most recent advancements of benchmarks for NL2SQL systems. It represents a significant evolution in NL2SQL benchmarking and focuses primarily on enterprise level database challenges. SPIDER 2.0 is much smaller with only 632 real-world problems which were derived from enterprise database usecases, but yet represents a meaningful indicator for the complexity of databases that approaches and models can handle. SPIDER 2.0 goes beyond simple query generation tasks and moves towards testing the deep understanding of the database, requiring models to understand metadata, SQL dialect documentation and project-level codebases (**?**). The tasks contained in SPIDER 2.0 often demand multiple complex SQL queries often exceeding the 100-line mark and require incorporating a diverse set of database operation from transformation to analytics. **?** further highlights the gap between academic research and enterprise-level environments with even advanced approaches achieving only 21.3% on SPIDER2.0 compared to 91.2% on SPIDER 1.0 (**??**).

2.7 Research Gaps

This literature review highlights that significant progress was made in the development of real world feasible NL2SQL systems and that the research domain has undergone multiple large paradigm shifts from rule-based to neural-network-based to PLM-based and most recently to LLM-based approaches. Despite these advancement several critical research gaps remained open or emerged which limit the practical deployment of NL2SQL systems and their widespread adoption as natural language database interfaces in real world scenarios.

2.7.1 Advanced Open-Source Approaches

While recent research papers have made remarkable progress on both open-source model development (CODES and OMNISQL) aswell as advanced architecture development like CHASE and XIYAN-SQL (which utilize multi-path generation and candidate selection, self-correction and RAG), a significant research gap evolved around state-of-the-art approaches that dont rely on proprietary LLMs and only utilize open-source models. Most contemporary research proposes solutions which (partially) rely on the baseline proprietary LLM in order to achieve state-of-the-art results. Whilst these efforts give meaningful signal to the effect that specific prompt engineering techniques, self-correction mechanisms and candidate selection models have, they are not yet feasible for real world scenarios due to the cost and data privacy concerns they introduce.

1. **Limited Exploration of Technique Synergies** — Contemporary research primarily focuses on techniques like candidate selection, RAG, and self-correction in isolation or with specific closed-source LLMs. A gap remains in understanding how these techniques can be optimally combined, especially with emerging open-source models. For instance, the potential of synergy of specialized open-source LLM like OMNISQL with candidate-selection, advanced self-correction and subset-encoding of database schemas has yet to be researched.

2. **Efficiency-Accuracy Tradeoffs** — Whilst recent research efforts focus primarily on achieving a new state-of-the-art execution accuracy metric on prevalent benchmarks, the relationship of computational requirements and performance gains achieved remains unclear. Some research papers present ablation studies indicating the relevance of certain

architecture components but the overall relationship between state-of-the-art performance and cost is still underexplored.

3. **Cross-Technique Optimization** — There is limited research on how to optimize the interplay between the various NL2SQL techniques. For example, it is yet unclear how subset-encoding of database schemas might impact the effectiveness of multi-path generation and candidate selection, or whether synthetic example generation for in-context-learning is effective when working with specialized models versus general-purpose LLMs.

2.7.2 Deployment and Performance Gaps

Despite impressive academic results, significant gaps remain in transitioning NL2SQL systems from research environments to real-world, enterprise-feasible, deployments:

1. **Database Integration** — While existing research has often focused on standalone systems, little attention has been given to the integration complexities between NL2SQL capabilities and database management systems like PostgreSQL. Having two standalone systems imposes a significant data transfer need between the two systems when infact, natural language queries can be treated as an extension to most existing databases. This implementation gap prevents seamless adoption into existing databases as it requires additional software layers which in turn increase the overall complexity and cost.

2. **Hardware Requirement Optimization** — As mentioned above, current LLM-based approaches often require significant compute resources in order to achieve state-of-the-art results. There is limited research available on performance optimization of NLIDBs in order to achieve practical and industry-viable hardware constraints.

3. **Real-time Performance Considerations** — Most research papers evaluate models based on their accuracy metrics alone without factoring in the latency and throughput characteristics of their solutions. This imposes a possibly significant δ between academic research and production environments. Responsiveness is an important metric for user experience and should therefore play a role in NLIDB research when it comes to evaluating different NL2SQL architectures.

4. **Privacy and Security** — While open-source models address some privacy concerns by enabling a local deployment of LLMs, research gaps remain in ensuring that NL2SQL systems respect database access controls and security policies. This is especially important for enterprise and government environments where data access is strictly regulated.

2.7.3 Ambiguity Resolution and Semantic Accuracy

Despite the long existance of the research field, fundamental questions remain open with regards to handling ambiguity effectively in natural language queries. Current, especially language model based, systems struggle to correctly identify when natural language queries contain ambiguities that they cant confidently resolve. Although some approaches like multi-path generation and candidate selection are a promising way to work around ambiguous language, early systems like NaLIR showed that the most effective way to deal with inherently ambiguous language is asking clarifying questions, instead of trying to interpret the query best-effort (**?**). Ambiguous language is an inherent source of inaccuracy and therefore a cause for misleading query results. Contemporary research and benchmarks dont focus ambiguity detection and strategies resolution which therefore leaves open questions to be further researched.

2.7.4 Evaluation and Benchmarking Gaps

1. **Enterprise-grade Evaluation** — As highlighted by SPIDER 2.0, there is a gap between academic benchmarks and enterprise realities. Further research is needed to create evaluation frameworks that better represent real-world enterprise environments with thousands of tables and complex relationships. **?** is a promising first step is this direction.

2. **Performance Metrics** — As mentioned above, current benchmarks often dont capture latency or throughput of NL2SQL systems at all, allowing for solutions to achieve state-of-the-art scores that require significantly more resources than their predecessors. Having meaningful performance metrics and benchmarking would allow to further analyze the proposed solutions for their real world feasibility.

3. **User Experience Metrics** — Contemporary benchmarks focus on execution accuracy without assessing potential user satisfaction, trust, and overall experience. Metrics that capture these aspects are necessary for understanding the actual utility of NL2SQL systems in practice. Although execution accuracy is a big factor for the trustworthiness of a NLIDB, it is not the only component as indicated by **?** in **?**.

2.7.5 Thesis Placement

This thesis addresses several of the above outlined research gaps. The primary focus of this work is the integration of open-source NL2SQL models with advanced techniques like candidate selection, subset-encoding of database schemas, and synthetic example generation for in-context learning. Through the implementation of a PostgreSQL extension this work will bridge multiple critical gaps between theoretical advancements and their practical deployability into real world systems whilst exploring performance characteristics.

3 Theoretical Foundations

This section serves as an introduction to the theoretical concepts utilized in this thesis. It is meant to establish a baseline of understanding for the design, implementation and evaluation phases. It briefly covers relational database theory (incl. SQL), machine learning fundamentals and their subsequent application for large language models.

3.1 Relational Database Theory

The relational database theory provides a mathematical framework for formalizing data layout in database management systems (DBMS), enabling efficient storage and retrieval mechanisms. This theory is helps to understand the way in which natural language queries need to be translated into database operations and what challenges arise during this translation.

3.1.1 Relational Model Fundamentals

The relational model was introduced by **?** in **?** and essentially forms the theoretical foundation for most modern database systems (**??**). A relational database organizes data in relations (ie. tables), where each relation consists of tuples (ie. rows) containing attributes (ie. columns) of specific domains (ie. data types and constraints). The mathematical origins of this model root in set theory and first-order predicate logic (**??**).

Furthermore the relational model abstracts data storage and respective physical implementation details by design, allowing interactions with data through respective logical structures (relations, tuples, attributes etc.) rather than exposing the underlying storage mechanism. This abstraction is fundamental for NL2SQL systems, as natural language queries can be mapped into queries on these structures regardless of underlying physical storage implementation, allowing for better transferability of NL2SQL systems (e.g. across different database deployments).

3.1.2 Core Concepts

The core concepts of the relational model that are relevant for this thesis include:

- **Relations** — Mathematical sets of tuples representing entities (e.g. employees, products) or relationships (e.g. enrollment, purchases). Each relation has a fixed structure.
- **Tuples** — Tuples represent rows of data, by grouping a several attributes into one logical unit (e.g. one specific purchase, containing the amount, date and shop). All tuples in the same relation share the same structure.
- **Attributes** — Attributes are frequently referred to as columns, representing a specific dimension of a tuple (e.g. customers have names). Attributes are represented in a fixed domain.
- **Domains** — Domains are sets of allowed values for attributes, including mathematically formalizing data types and constraints.
- **Schemas** — Structural definitions specifying relations, their attribute names, domains, and integrity constraints.
- **Primary Keys** — Attributes that uniquely identify tuples within a relation. These ensure entity integrity and ensure referential relationships.

- **Foreign Keys** — Attributes referencing primary keys in other relations, allowing databases to maintain referential integrity and enabling semantically correct complex queries across multiple tables through joins.

These concepts form the semantic foundation that natural language interfaces must navigate when translating user queries into formal database operations. As shown in the literature review *schema-linking* is a crucial problem of NLIDBs — establishing an understanding of which relations a natural language query refers to is the first step of formalizing a database operation for retrieval.

3.1.3 SQL as a Declarative Query Language

Structured Query Language (SQL) is a defacto standard for interfacing with relational databases. It is a declarative language to describe database operations like selection, insertion, updating and deletion of database conte/home/mara/Workspace/thb/thesis/main.texnts. SQL's design is strongly influenced by relational algebra and tuple relational calculus. It enables complex data retrieval through a readable query syntax incorporating relational algebra operations like:

- **Selection** (σ) — Filtering tuples based on specified conditions
- **Projection** (π) — Extracting specific attributes from relations
- **Joins** ($\bowtie$) — Combining data from multiple relations based on common attributes
- **Aggregation** — Computing summary statistics over groups of tuples

The basic syntax of SQL queries follows a logical pattern that reflects these theoretical operations:

```
SELECT attributes      -- Project attributes
FROM relations         -- Specify relations to use
WHERE conditions       -- Selection of tuples
GROUP BY attributes    -- Grouping for aggregation
HAVING conditions      -- Group-level filtering
ORDER BY attributes    -- Ordering of results
```

There are further operations like various types of `JOIN`s, `OFFSET`, `LIMIT`, `WITH` that are heavily used in SQL which are excluded for readability and simplicity.

The nature of SQL creates a fundamental challenge for NL2SQL systems; natural language queries express (often fuzzy) user intent in terms of desired output (e.g., "Give me 5 great movies") while SQL formalizes an explicit and discrete way of how to retrieve this information from the database:

```
SELECT m.title, AVG(r.score) as rating
FROM movies m
JOIN ratings r ON m.movie_id = r.movie_id
GROUP BY m.movie_id, m.title
HAVING AVG(r.score) >= 8
ORDER BY rating DESC
LIMIT 5
```

This semantic gap between natural language and formal queries represents the core challenge that this thesis addresses.

Furthermore, SQL's compositional nature allows complex queries to be built from simpler components through nesting and combination of operations. NL2SQL systems must therefore not only fill in the individual query components but also how to compose these to semantically accurate and syntactically correct query statements. This is particularly important when dealing with complex natural language queries that may require multi-clause SQL statements involving subqueries, multiple joins, and aggregation functions to answer.

3.1.4 Normalization and Schema Design

Database schemas typically follow normalization principles to eliminate redundancy and maintain data integrity. The normal forms (1NF, 2NF, 3NF, BCNF) provide a set of design guidelines and rules for relational database design to reduce duplication, inconsistencies and anomalies (**?**). Understanding normalization forms is crucial for NL2SQL systems to handle:

- **Schema complexity** — Normalized schemas often distribute logically related information across multiple tables, requiring natural language interfaces to understand implicit relationships and generate appropriate joins.
- **Semantic mapping** — Humans typically think about data in denormalized, conceptual terms, while relational databases store data in its normalized form. NL2SQL systems must overcome this layout difference (e.g. there is no 1:1 mapping of concepts to tables).
- **Query complexity** — Retrieving simple information from relational databases may require multiple joins in normalized schemas, challenging NL2SQL systems to generate potentially complex SQL statements from simple user requests.

The tension between normalized database design and intuitive natural language use represents a key challenge that influences the architecture and design decisions explored in subsequent sections of this thesis.

3.2 Machine Learning Fundamentals

Machine learning represents the algorithmic foundation for state-of-the-art NL2SQL systems, enabling systems to learn how natural language queries translate into SQL. Understanding these foundations is essential for implementing and optimizing large language model-based NL2SQL approaches.

3.2.1 Neural Network Architecture

Neural networks are the computational concept behind contemporary NL2SQL approaches, representing complex capabilities through compositions of simpler operations. A neural network consists of interconnected computational units (called neurons) organized in layers, where each connection has an associated (and learnable) weight and bias parameters.

The fundamental computation paradigm of neural network outputs is called forward propagation, which is applying the respective weights (W) and biases (b) to the input parameter (a) and transforming it using a (non-linear) activation function f:

$$a^{l+1} = f(Wa^l + b) \tag{1}$$

where $W \in \mathbb{R}^{m \times n}$ is the weight matrix, $a \in \mathbb{R}^n$ is the input vector, $b \in \mathbb{R}^m$ is the bias vector, and f is an activiation function (e.g. ReLU, sigmoid, or tanh). Using this function repeatedly, propagates information through the neural network.

The most important architectural components for language processing models include:

- **Feedforward Networks** — Which can process fixed-size inputs through successive linear transformations and activations, this architecture is particularly suitable for classification and regression problems within NL2SQL systems.
- **Recurrent Networks (RNNs/LSTMs)** — Which can handle sequential data of variable-length by maintaining hidden states that capture dependencies, enabling processing of natural language sequences of arbitrary length.
- **Embedding Layers** — Which map discrete tokens (e.g. words, characters, or subwords) to a dense vector representation, thus providing the foundation for subsequent neural language processing.

The ability of neural networks to learn hierarchical representations through multiple layers makes them particularly well-suited for the complex translation required in NL2SQL systems.

3.2.2 Sequence-to-Sequence Models

Sequence-to-sequence (Seq2Seq) models established the foundational architecture for neural NL2SQL approaches (**??**). These models which rely on RNNs and LSTMs addressed the fundamental challenge of mapping variable-length input sequences (natural language queries) to variable-length output sequences (SQL statements).

The Seq2Seq model architecture is composed out of three essential components:

1. **Encoder** — Process the input natural language sequence $x_1, x_2, \ldots, x_n$ into a sequence of hidden representations $h_1, h_2, \ldots, h_n$, capturing the semantic content of the query.
2. **Decoder** — Generates the output SQL sequence $y_1, y_2, \ldots, y_m$ autoregressively, where each token is predicted based on previous outputs and encoder representations.
3. **Attention** — Allows the decoder to selectively focus on relevant portions of the input sequence, addressing the information bottleneck of fixed-size context vectors.

3.3 Natural Language Processing

3.4 Large Language Models

3.5 Graph Theory Fundamentals

3.6 GPU Computing and Parallel Processing

4 System Design

This chapter describes the design of NATURAL, our proposed NL2SQL system, addressing limitations and research gaps identified in the literature review.

NATURAL is architected as a pipeline that transforms natural language questions into SQL queries using example selection (σ), schema subsetting (ϕ), and self refinement (ρ) and voting (ν). The proposed system consists of five components:

1. **Example Selection** σ – Identifies semantically and structurally similar examples using cosine similarity and schema distance.
2. **Schema Subsetting** ϕ – Reducing schema complexity by subsetting the schema to the relevant subset of tables and relationships.
3. **Query Projection** π – Few-shot learning with a finetuned model to project natural language queries to SQL statements.
4. **Self Refinement** ρ – Self refinement of generated SQL statements through execution feedback and error analysis.
5. **Voting** ν – Self consensus voting mechanism to choose the most likely result from multiple generation attempts.

The system processes queries using the following algorithm:

```
  Natural Language Query
→ Sketch Generation
→ Example Selection
→ k-times [
    → Schema Subsetting
    → Few-Shot Generation
    → Self-Correction
  ]
→ Consensus Voting
→ SQL Query
```

This design largely builds upon few-shot learning concepts from DAIL-SQL (**?**) and OmniSQL (**?**) but incorporates novel contributions in schema-aware example selection by harnessing a graph-based structural similarity metric.

4.1 Initialization

Before the NATURAL system can process queries, it requires initialization with historical data to construct a specific embedding space $v \in \mathcal{V}$ and schema distance index $d \in \mathcal{D}$. This preprocessing phase analyzes existing datasets and previous user interactions to enable semantically-aware example selection.

The initialization process consists of two main components: (1) embedding generation for semantic similarity computation, and (2) schema indexing for structural similarity measurement. These components work together to support the example selection function σ.

4.1.1 Embedding

Building upon DAIL-SQL's masked example selection approach, NATURAL constructs an embedding space v to enable semantic similarity search. The system uses cosine similarity to identify relevant examples based on masked representations of both natural language queries and SQL statements, allowing efficient retrieval from large collections of question-answer pairs.

We therefore embed a set of training samples $\mathcal{T} = \{(q_1, \omega_1), ..., (q_n, \omega_n)\}$ into an embedding space v, where each q_i is a natural language question and ω_i is the corresponding SQL query.

The embedding space v can be formally defined as:

$$v = \{\ (q, \iota(mask(q)), \omega, \iota(mask(\omega))) \mid (q, \omega) \in \mathcal{T}\ \}$$

where $mask$ represents the masking function described in sections 4.1.1.1 and 4.1.1.2, and ι is the embedding function that maps text to vector representations.

4.1.1.1 Natural Language Masking

Masking of natural language is the process of replacing all words which are not deemed *structurally relevant* with the `<mask>` token. All words w in a constant whitelist $\mathcal{W}$ are determined structurally relevant. The constant $\mathcal{W}$ contains common structural SQL keywords and is determined empirically.

Working with masked language enables the system to capture semantic patterns independently of domain terminology, improving the generalizability across domains.

Given $\mathcal{W} = \{\text{list}, \text{show}, \text{all}, \text{the}, \text{with}\}$ the natural language questions:

```
Show all customers with the firstname John
List all products with the name iPhone
```

Will get masked into the same sentence, such that the semantic words are maintained, but domain specific terminology is discarded:

```
Show all <mask> with the <mask> <mask>
```

4.1.1.2 SQL Masking

Masking SQL queries refers to transforming all identifiers contained in a SQL query to `<mask>` and all value literals to `<value>`.

Thus, the queries:

```
SELECT * FROM customers WHERE firstname = 'John'
SELECT * FROM products WHERE name = 'iPhone'
```

get masked to:

```
SELECT * FROM <mask> WHERE <mask> = <value>
```

4.1.2 Schema Indexing

To choose the most relevant subset of samples for few-shot learning, it is important that the SQL queries we choose as examples are written for structurally similar database schemas in order to minimize the structural difference between the selected samples and the ground truth query for a given natural language question.

```
CREATE TABLE users (
  id TEXT PRIMARY KEY
);

CREATE TABLE contacts (
  user_id TEXT NOT NULL,
  name TEXT NOT NULL,
  FOREIGN KEY (user_id)
   REFERENCES users(id),
  PRIMARY KEY (user_id, name)
);
```

Figure 1: Normalized schema

```
CREATE TABLE users (
  id TEXT PRIMARY KEY,
  contacts TEXT[] NOT NULL
);
```

Figure 2: Denormalized schema

For example ω_{ground} the question "Give me all contacts for the user with the id 10" might look different depending on the database schema at hand, thus only selecting samples based on the similarity of the natural language question will yield inferior sample quality.

Given the database schemas in figures 1 and 2 respective definitions of ω_{ground} would be 3 and 4.

```
SELECT contacts.name
FROM users
JOIN contacts
  ON contacts.user_id = users.id
WHERE users.id = 10;
```

Figure 3: SQL JOIN selection

```
SELECT contacts
FROM users
WHERE id = 10;
```

Figure 4: SQL Array selection

As shown in the figures 3 and 4 the structural similarity of the underlying database schema is a crucial component of the relevance of an example.

4.1.2.1 Graph Representation

To determine structural similarity of database schemas systematically, we propose using graph representation of database schemas as a data definition language (DDL) independent representation of the database structure. Choosing a graph representation allows us to leverage established methods for measuring graph similarity such as Wasserstein-Weisfeiler-Lehman graph kernels (**?**). Given a database schema s we define the corresponding graph $G_s = (V, E, \ell, w)$ as:

1. $V = V_t \cup V_c$ where V_t represents table nodes and V_c represents column nodes
2. $E = E_{tc} \cup E_{fk} \cup E_{ref}$ where:
   - E_{tc} are table-column edges

- E_{fk} are foreign key relationship edges between tables
- E_{ref} are reference edges between foreign key columns

3. Each node $v \in V$ has a semantic label $\ell(v) \in \{1, 2, \ldots, 9\}$ where:
   - $\ell(v) = 1$ for table nodes
   - $\ell(v) = 2$ for generic column nodes
   - $\ell(v) = 3$ for primary key columns
   - $\ell(v) = 4$ for foreign key columns
   - $\ell(v) = 5$ for text columns
   - $\ell(v) = 6$ for numeric columns
   - $\ell(v) = 7$ for datetime columns
   - $\ell(v) = 8$ for boolean columns
   - $\ell(v) = 9$ for other/unknown data types
4. Each edge $e \in E$ has a weight $w(e) \in \{0.5, 0.9, 1.0\}$ reflecting its structural importance:
   - $w(e) = 1.0$ for foreign key relationships (highest importance)
   - $w(e) = 0.9$ for column foreign key reference edges
   - $w(e) = 0.5$ for table-column edges

We define the structural similarity of two databases as the topological distance of the respective graphs. The graph representation captures essential schema structure through semantic node labeling that prioritizes constraints over data types: primary key columns receive label 3 regardless of their underlying data type, foreign key columns receive label 4, and only then are remaining columns categorized by data type (text=5, numeric=6, datetime=7, boolean=8, other=9). This hierarchical labeling ensures that structural relationships take precedence over exact content, while omitting table names, column names and domain terminology to achieve schema-agnostic comparison.

The graph representation of the database schemas introduced in 1 and 2 are therefore:

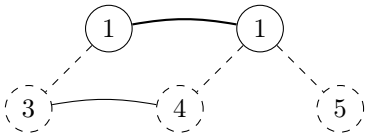

Figure 5: Normalized graph repr.

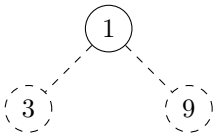

Figure 6: Denormalized graph repr.

Legend: ○ table ◌ column **-** foreign key – reference - - table-column

The δ of the order of the graphs 5 and 6 further highlight the structural difference between the two schemas.

4.1.2.2 Distance Measurement

To measure the distance between the graphs displayed in 5 and 6, we employ the Wasserstein Weisfeiler-Lehman (WWL) graph kernel method (**?**).

WWL kernels compute structural similarity by combining discrete Weisfeiler-Lehman graph features with continuous optimal transport theory. The method first extracts WL features from the labeled graph structure, then computes the Wasserstein distance between the resulting feature distributions. For two graphs G and G' with semantic node labels, the distance is computed as:

$$D_W^f(G, G') = W_1(f(G), f(G')) \tag{2}$$

as introduced by **?** in **?**.

For the database schemas above, the graph representation of the normalized schema G_{norm} (see figure 5) with 5 nodes and 5 edges and denormalized schema G_{denorm} (see figure 6) with 3 nodes and 2 edges show significant topological differences. The WWL distance captures both the reduction in graph complexity (fewer nodes and edges) and the loss of relational structure (elimination of foreign key relationships), resulting in a substantial distance value reflecting their structural dissimilarity despite representing equivalent logical data with $D_W^f(G_{norm}, G_{denorm}) \approx 0.78$.

4.1.2.3 Distance Index

The distance index d maintains precomputed distances between all observed database schemas, enabling efficient retrieval of structurally similar databases during sample selection. This index is constructed as a set of schema-distance tuples $\{(sm_1, di_1), \ldots, (sm_i, di_i)\}$ where sm_i is a database schema name and di_i denotes its WWL distance to the current schema.

4.2 Functions

The following sets are subsequently used to introduce the functions σ, ϕ, π, ρ and ν.

1. $\mathcal{Q}$ – The set of all possible natural language queries.
2. $\mathcal{S}$ – The set of all possible database schemas.
3. $\mathcal{E}$ – The set of all possible execution functions for SQL validation.
4. $\mathcal{V}$ – The set of all possible embedding spaces.
5. $\mathcal{D}$ – The set of all possible distance indices over historically observed databases.
6. $\mathcal{C}$ – The set of all possible candidate sets $\mathcal{C}'$.
7. $\mathcal{Q}_{\mathcal{S}}$ – The set of all valid queries over the database schema S.

4.2.1 Example Selection – σ

The selection function σ retrieves the most relevant examples from the historical embedding space v based on semantic similarity to the input query. This function implements the core example selection mechanism that subsequently enables few-shot learning for SQL generation.

$$\sigma : \mathcal{Q} \times \Omega_{\mathcal{S}} \times \mathcal{S} \times \mathcal{V} \times \mathcal{D} \to \mathcal{C}'$$

For a specific input instance, we write:

$$\sigma(q, z, s, v, d) = \{(q_1, s_1, \omega_1, d_1), \ldots, (q_k, s_k, \omega_k, d_k)\}$$

where $q \in \mathcal{Q}$ is the input natural language query, $z \in \Omega_{\mathcal{S}}$ is the zero-shot inference result, $s \in \mathcal{S}$ is the database schema, $v \in \mathcal{V}$ is the embedding space, $d \in \mathcal{D}$ is the distance index, and the result is a candidate set $\mathcal{C}' \in \mathcal{C}$ containing k tuples of natural language queries, schemas, corresponding SQL queries and their combined distance.

The function utilizes cosine similarity in the embedding space to identify examples that are semantically closest to the input query q, considering both masked natural language representations and schema distance as described in Section 4.1.1. This approach ensures that examples are weighted by the following properties:

1. **Question similarity** – Semantic similarity between the masked input query and historically observed natural language queries in the embedding space v, measured using cosine distance between their respective embeddings.

2. **SQL similarity** – Structural similarity between the zero-shot inference result z and candidate SQL queries using masked SQL representations, enabling pattern recognition across different database domains.

3. **Database similarity** – Schema compatibility measured through the distance index d, ensuring selected examples operate on analogous database structures with similar table relationships and column types.

The exact weighting of these can be adjusted through three constants, w_q, w_s, and w_d.

Algorithm 1 σ - Example Selection

Require: $q \in \mathcal{Q}$, $z \in \Omega_{\mathcal{S}}$, $s \in \mathcal{S}$, $v \in \mathcal{V}$, $d \in \mathcal{D}$
Require: $k \in \mathbb{N}$, $k \geq 1$ ▷ Number of examples to select
$q' \leftarrow mask(q)$ ▷ Mask input question
$z' \leftarrow mask(z)$ ▷ Mask zero-shot result
$candidates \leftarrow \emptyset$ ▷ Initialize candidate set
for $(q_i, s_i, \omega_i) \in v$ **do** ▷ For each sample
    $sim_q \leftarrow cosine(\iota(q'), \iota(mask(q_i)))$ ▷ Calculate question similarity
    $sim_s \leftarrow cosine(\iota(z'), \iota(mask(\omega_i)))$ ▷ Calculate SQL similarity
    $sim_d \leftarrow d(s, s_i)$ ▷ Calculate schema distance
    $score \leftarrow w_q \cdot sim_q + w_s \cdot sim_s + w_d \cdot sim_d$
    $candidates \leftarrow candidates \cup \{(q_i, s_i, \omega_i, score)\}$
end for
return $top_k(candidates)$ ▷ Return k highest scoring examples

4.2.2 Schema Subsetting – ϕ

The subsetting function ϕ reduces the database schema to only include tables, columns, and relationships that are relevant to the current set of candidates C. This schema pruning mechanism reduces the complexity of the SQL generation task and token efficiency by focusing on the most relevant schema elements.

$$\phi : \mathcal{C}' \times \mathcal{S} \to \mathcal{S}'$$

For a specific input instance, we write:

$$\phi(c, s) = s'$$

where $c \in \mathcal{C}'$ is a candidate set containing selected examples, $s \in \mathcal{S}$ is the full database schema, and $s' \subseteq s$ is the reduced schema subset containing only relevant tables and columns.

The function analyzes the selected examples in c to identify which schema elements are commonly referenced in similar queries. This analysis enables the system to steer the model's attention on the most relevant portions of potentially large and complex database schemas, thereby improving both accuracy and computational efficiency and prevents exceeding limited context windows.

Algorithm 2 ϕ - Schema Subsetting

Require: $c \in \mathcal{C}'$, $s \in \mathcal{S}$

$s' \leftarrow \emptyset$ ▷ Empty schema subset
for $table \in s$ **do**
  if $\exists\ r \in references(c, table)$ **then** ▷ Any candidates reference the table
    $s' \leftarrow s' \cup \{table\}$ ▷ Extend subsetted schema
  end if
end for
return s' ▷ Return subsetted schema

4.2.3 Query Projection – π

The projection function π represents the core translation mechanism that converts natural language queries into SQL statements using the selected examples and database schema. This function encapsulates the inference using LLMs that generates candidate SQL queries based on the provided context.

$$\pi : \mathcal{Q} \times \mathcal{S} \times \mathcal{C}' \to \Omega_{\mathcal{S}}$$

For a specific input instance, we write:

$$\pi(q, s, c) = \omega$$

where $q \in \mathcal{Q}$ is the input natural language query, $s \in \mathcal{S}$ is the database schema, $c \in \mathcal{C}'$ is the set of selected examples, and $\omega \in \Omega_{\mathcal{S}}$ is the generated SQL query.

The function operates by constructing a prompt that combines the natural language query, the relevant schema information, and the selected examples in a format optimized for large language model inference. The model then generates a SQL query that attempts to capture the semantic intent of the natural language input while adhering to the constraints imposed by the database schema.

4.2.4 Self Refinement – ρ

The refinement function ρ implements a self-correction mechanism that iteratively improves generated SQL queries by identifying and correcting syntax errors, semantic inconsistencies, and execution failures. This function enables the system to learn from its mistakes and produce higher-quality outputs through automated feedback loops.

Algorithm 3 π - Query Projection

Require: $q \in \mathcal{Q}$, $s \in \mathcal{S}$, $c \in \mathcal{C}'$
Require: $\tau \in \mathbb{R}$, $\tau > 0$ ▷ Relevance threshold
$c_{rel} \leftarrow \{ (q_i, s_i, \omega_i, d_i) \in c : d_i < \tau \}$ ▷ Filter by relevance
$ex \leftarrow \{ fmt(q_i, \omega_i, d_i) \mid (q_i, s_i, \omega_i, d_i) \in c_{rel} \}$ ▷ Format examples
$prompt \leftarrow prompt(q, s, ex)$ ▷ Construct prompt
$out \leftarrow model(prompt)$ ▷ Model inference
$\Omega_{cand} \leftarrow extract_{sql}(out)$ ▷ Extract SQL candidates
for $\omega_{raw} \in \Omega_{cand}$ **do** ▷ Validate candidates
    if $\omega_{raw} \in \Omega_{\mathcal{S}}$ **then** ▷ Check syntactic validity
        return ω_{raw} ▷ Return first valid query
    end if
end for

$$\rho : \mathcal{Q} \times \mathcal{S} \times \mathcal{E} \times \Omega_{\mathcal{S}} \to \Omega_{\mathcal{S}}$$

For a specific input instance, we write:

$$\rho(q, s, e, \omega_{raw}) = \omega_{refined}$$

where $q \in \mathcal{Q}$ is the original natural language query, $s \in \mathcal{S}$ is the database schema, $e \in \mathcal{E}$ is the execution function for validation, $\omega_{raw} \in \Omega_{\mathcal{S}}$ is the previously generated SQL query and $\omega_{refined} \in \Omega_{\mathcal{S}}$ is the improved SQL query.

The function operates by executing the candidate query against the database, analyzing any errors or unexpected results, and then prompting the language model to generate an improved version based on the identified issues. This iterative refinement process continues until a valid, executable query is produced or a maximum number of refinement attempts is reached.

Algorithm 4 ρ - Self Refinement

Require: $q \in \mathcal{Q}$, $s \in \mathcal{S}$, $e \in \mathcal{E}$, $\omega_{raw} \in \Omega_{\mathcal{S}}$
$exec \leftarrow e(s, \omega_{raw})$ ▷ Verify execution output
$prompt \leftarrow prompt_{refine}(q, s, \omega_{raw}, exec)$ ▷ Construct refinement prompt
$out \leftarrow model(prompt)$ ▷ Generate refinement output
$\Omega_{ref} \leftarrow extract_{sql}(out)$ ▷ Extract refined candidates
for $\omega_{ref} \in \Omega_{ref}$ **do** ▷ Validate refined queries
    if $\omega_{ref} \in \Omega_{\mathcal{S}}$ **then** ▷ Check syntactic validity
        return ω_{ref} ▷ Return first valid refinement
    end if
end for

4.2.5 Voting – ν

The voting function ν implements a consensus mechanism that selects the most reliable SQL query from multiple candidate solutions generated through the pipeline. This function enhances robustness by leveraging the result distribution of multiple generation attempts to identify the most likely correct answer.

$$\nu : \mathcal{C} \times \mathcal{E} \to \Omega_{\mathcal{S}}$$

For a specific input instance, we write:

$$\nu(C, e) = \omega_{consensus}$$

where $C \in \mathcal{C}$ is a set of candidate SQL queries, $e \in \mathcal{E}$ is the execution function for validation, and $\omega_{consensus} \in \Omega_{\mathcal{S}}$ is the selected consensus query.

The voting function ν implements a majority voting algorithm similar to that described by OmniSQL (**?**), where the result that appears most frequently across multiple generation attempts is deemed to be the most likely correct answer. The function applies frequency-based selection among the valid candidates. In cases where no clear majority exists, the function may apply additional heuristics such as query complexity or execution performance to determine the final proposed query candidate $\omega_{consensus}$.

Algorithm 5 ν - Consensus Voting

Require: $C \in \mathcal{C}$, $e \in \mathcal{E}$

$results \leftarrow \{\}$ ▷ Map from result sets to candidate lists
for $\omega \in C$ **do** ▷ Execute each candidate
  $result \leftarrow e(\omega)$ ▷ Execute query
  $results[result] \leftarrow results[result] \cup \{\omega\}$
end for
$dist \leftarrow \{results[r] : r \in results\}$ ▷ Get all result groups
$dist \leftarrow sort(dist, |group| \mapsto |group|)$ ▷ Sort by group size (largest first)
return $top_1(top_1(dist))$ ▷ Return largest group

4.3 Composition – *nq*

The *nq* function composes the system by first establishing a zero-shot baseline, then using the precomputed embedding space v and distance index d from the initialization phase to guide few-shot generation while self refining results through execution feedback and finally yield the most likely candidate ω through majority voting.

$$nq : \mathcal{Q} \times \mathcal{S} \times \mathcal{E} \times \mathcal{V} \times \mathcal{D} \to \Omega_{\mathcal{S}}$$

For a specific input instance, we write:

$$nq(q, s, e, v, d) = \omega$$

where $q \in \mathcal{Q}$ is the input natural language query, $s \in \mathcal{S}$ is the database schema, $e \in \mathcal{E}$ is the execution function for validation, $v \in \mathcal{V}$ is the embedding space, $d \in \mathcal{D}$ is the distance index, and $\omega \in \Omega_{\mathcal{S}}$ is the generated SQL query.

The *nq* function implements Algorithm 6.

Algorithm 6 *nq*

Require: $q \in \mathcal{Q}$, $s \in \mathcal{S}$, $e \in \mathcal{E}$, $v \in \mathcal{V}$, $d \in \mathcal{D}$
Require: $k \in \mathbb{N}$, $k \geq 1$ ▷ Number of candidates to generate
$z \leftarrow \pi(\emptyset, q, s)$ ▷ Generate zero-shot baseline
$\mathcal{C}' \leftarrow \sigma(q, z, s, v, d)$ ▷ Select relevant examples
$C \leftarrow \emptyset$ ▷ Initialize candidate set
while $|C| < k$ **do** ▷ Generate k candidates
    $s' \leftarrow \phi(\mathcal{C}', s)$ ▷ Subset schema to relevant elements
    $\omega \leftarrow \pi(\mathcal{C}', q, s')$ ▷ Generate candidate query
    if $e(\omega, s')$ **then** ▷ Validate syntactic correctness
        $C \leftarrow C \cup \{\omega\}$ ▷ Add valid candidate
    end if
    $\omega' \leftarrow \rho(q, s', e, \omega)$ ▷ Attempt self-correction
    if $e(\omega', s')$ **then** ▷ Validate corrected query
        $C \leftarrow C \cup \{\omega'\}$ ▷ Add corrected candidate
    end if
end while
return $\nu(C, e)$ ▷ Select consensus result

5 Implementation

This chapter presents the practical implementation of the NATURAL system, translating the theoretical framework outlined in Chapter 4 into a working NL2SQL system.

Primary attention is given to the engineering challenges encountered, performance bottlenecks identified, and optimization strategies implemented to achieve practical deployment viability on consumer hardware. The chapter is comprised of a software architecture and infrastructure discussion as well as implementation outlines of each pipeline component.

5.1 Architecture and Infrastructure

The implementation of the system design phase is split into inference, sampling and evaluation code. The runtime code focuses on the actual algorithm implementations outlined in section 4, sampling code focuses on indexing samples and computing distance indices and the evaluation code runs the NATURAL pipeline on prevalent benchmarks.

This subsection focuses on outlining the scope of each technological component, the design rationale behind them and discusses the technology stack decisions.

5.1.1 Software Architecture

NATURAL consists of five software components:

1. `natural-models` – For handling the actual model execution on GPUs for inference and embedding.
2. `natural-graphs` – Graph library for representing database schemas in graphs as well as computing graph similarities and distances.
3. `natural-inference` – The core pipeline implementation using other software components during inference time.
4. `natural-sampling` – The sampling setup that focuses on indexing samples and computing graph distance indices.
5. `natural-benchmark` – The benchmarking setup used to run experiments, continuously evaluate the accuracy of the system and compile statistics.

```
THIS NEEDS A DIAGRAM TO UNDERSTAND THE SOFTWARE ARCH.
```

This separation of concerns emerged as the split between inference, sampling and evaluation code required shared fundamentals like model execution and graph representation which in turn improves testability and maintainability. Furthermore separating the inference, sampling and evaluation code yields a smaller footprint when embedding the inference code into databases.

Challenges associated with this multi component architecture are mostly tied to dependency management and increased build complexity although these can be mitigated through `cargo`'s workspace support.

5.1.2 Resource Management Strategy

Given the constrained 24 gigabytes of VRAM capacity of the RTX 3090 not all available model sizes can be used. While 14B model variants theoretically work, the need for an embedding model typically exhausts the VRAM unless using steep quantization formats. The OMNISQL

7B models were shown to have an average performance degradation of only 0.3% compared to their larger 14B counterparts by **?** in **?**. On the spider test dataset the 7B model surprisingly outperformed its 14B counterpart by 0.6%. Due to the unclear performance gains of the 14B model, the significant increase in inference time and the limited VRAM available, this thesis focuses on the 7B variant of OmniSQL.

As an embedding model is required for doing semantic search of samples in the pipeline a small companion model is required to embed both the sample datasets (SynSQL, Bird and Spider) during sampling time and the user provided natural language question during inference. The Qwen3 series embedding models ranked first place on the MTEB multilingual leaderboard. With the Q8_0 quantized version of the 8B model consumes 8 gigabytes of VRAM which is not possible to fit into 24 gigabytes of VRAM when accounting for KV-Cache requirements. Therefore, instead of choosing a heavily quantized version of the 8B variant (ie, Q4_K_M or below), the 4B model with Q8_0 shows similar performance characteristics on simple use cases while offering a significantly smaller memory footprint at 2 gigabytes. https://huggingface.co/Qwen/Qwen3-Embedding-8B cite.

As models have significant loading times, the system loads models globally and at startup and hands around references using atomic reference counting (`std::sync::Arc`) to ensure that models are not loaded twice, can be reused between different inference calls and GPU memory is not exhausted which would lead to a program crash.

5.1.3 Technology Stack Decisions

The choice of programming language, inference frameworks, and supporting libraries implies the system's performance and deployment characteristics, as well as development velocity. This section analyzes the key decisions made for Natural, discussing their trade-offs between research flexibility and production readiness, performance optimization and development speed, and ecosystem maturity versus cutting-edge capabilities.

The critical decisions are the programming language, the model inference framework, and the vector similarity search solution. Each decision was evaluated against the constraints of limited hardware resources (24GB VRAM), the need for database integration capabilities, and the requirement for reproducible research outcomes.

5.1.3.1 Language and Ecosystem

The most apparent and impactful decision is likely the language and ecosystem choice made. Viable languages for implementing natural language processing and machine learning heavy systems are Python, R, Julia, Rust, C / C++ and Java as well as other general purpose languages.

While interpreted languages like Python, R and Julia tend to be significantly easier to use for rapid prototyping and approach validation due to their loose type systems and great scientific ecosystem, they come with serious drawbacks with regards to deployability, speed and robustness compared to compiled and strongly typed languages.

Languages like Java, C / C++ and Rust offer greater stability at runtime, better interoperability into other programs (eg, database extensions) and better resource utilization they represent an interesting trade-off between performance optimization and portability vs. speed of iteration and research ecosystems.

R and Julia are inferior to Python when it comes to adoption, machine learning frameworks and natural language processing. Java requires a Java Virtual Machine (JVM) at runtime which yields worse portability than C / C++ and Rust while offering little advantage over interpreted languages like Python. Thus Python, C / C++ and Rust emerge as strong contenders for the implementation of Natural. C and C++ have the primary downside that they are prone to

program crashes and memory safety issues whereas Rust resolves most of these downsides while maintaining similar performance, memory management and portability characteristics.

Given that performance plays a critical role when developing machine learning systems with limited access to hardware, the path to a potential production deployments of NL2SQL is easier and the language ecosystem offers bindings for the most notable scientific libraries, Rust offers a great value proposition for ML systems at the cost of development speed.

5.1.3.2 Inference Framework

For local model inference, two primary frameworks emerged: llama.cpp (FFI), Candle (Rust-native). While Rust-native frameworks generally offer better portability, llama.cpp provided a better balance between performance, ecosystem maturity, and implementation simplicity.

llama.cpp provides out-of-the-box optimizations with comprehensive GGUF quantization support, yielding 25-30 tokens per second on consumer grade RTX 3090 hardware. It offers high-level abstractions for model loading and tokenization and sampling. The mature GGML ecosystem and advanced quantization schemes (like `Q4_K_M`, `Q8_0`) justified the FFI complexity.

Candle, even though it offered comparable raw inference speeds, required rather extensive low-level implementation work and proved incompatibility with scenarios where the resulting binary is embedded into a database (e.g. PostgreSQL). The combination of manual tensor management, more convoluted quantization support, and deployment constraints made llama.cpp a better alternative for production-ready systems.

5.1.3.3 Similarity Search Framework

For lightweight similarity search SQLite in combination with it's `sqlite-vec` extension is a sensible option. Especially for smaller data loads introducing the complexity of `faiss` and comparable vector search solutions like `qdrant` outweighs their speed benefits.

The `sqlite-vec` extension advertises to be a "fast enough" vector search solution, allowing for reduced complexity and compatibility with graphical database interfaces that support SQLite to inspect the embedding space.

5.1.3.4 Trade-offs

Overall the development overhead of choosing Rust over Python for the implementation phase was noticeable; Rust's machine learning and natural language processing frameworks are less advanced, porting research code written in python from other papers (like **?**) turned out to be non trivial. This decision likely doubled the time needed for the implementation, but in turn provides a clear path for the algorithms in this thesis to be productionized. The outcome of the implementation has significantly better performance and portability characteristics than using Python would have allowed for.

5.2 Pipeline Implementation

This subsection describes the system design realisation of each core pipeline component (σ, ϕ, π, ρ, ν) into working code, their algorithmic details, performance characteristics and trade-offs made.

5.2.1 Example Selection Engine (σ)

The example selection system is implemented using a multi-dimensional similarity scoring system that identifies the most relevant samples that were previously indexed for subsequent in-context

learning. The component addresses the fundamental challenge of selecting contextually appropriate examples from large training corpora whilst balancing semantic relevance and structural compatibility.

5.2.1.1 Similarity Computation

The selection mechanism combines three distinct similarity measures implemented in the selection module in `natural-sampling::selection`. The `selection` algorithm computes semantic similarity through cosine distance of the masked question embedding. Structural SQL similarity is computed via measuring the cosine distance of the masked SQL embedding, and schema compatibility using WWL kernel distance from the `natural-graphs` component. The embeddings are computed using the Qwen3-Embedding model (4B) using the query masking algorithm in `natural-sampling::masking`.

5.2.1.2 Weighting Strategy and Performance Characteristics

As this algorithm is aware of three distinct ways to measure sample similarity (masked question, masked query, structural). The implementation employs empirically optimized weights defined as constants: 70% for semantic question similarity and 30% for SQL structural similarity. Additionally, the final scoring combines 70% sample-level similarity with 30% schema-level compatibility.

The selection algorithm yields a maximum of 32 candidates (`TOP_K = 32`) to limit computational overhead whilst maintaining selection quality. Vector similarity queries are performed using SQLite with the `sqlite-vec` extension, achieving sub-second retrieval times for vector databases exceeding 2,000,000 samples.

5.2.1.3 Implementation Architecture

Using the `Selector` struct the embedding computation and representation is encapsulated from the actual selection algorithm. Thus via `Selector::new` the consuming code can compute the masked embeddings and use them subsequently to run example selection.

The `selection` algorithm implementation follows a two-stage approach, split into initial candidate retrieval through vector search (using the `Vector` struct and SQLite) and subsequent reweighting using the above described weights and a precomputed WWL distance index from `natural-graphs`.

5.2.2 Schema Subsetting System (ϕ)

The schema subsetting system is implemented in `natural-inference::pipeline::subsetting` as `SchemaSubsetter` struct. The core algorithm is the `optimize` method (lines 21-47) which performs query validation to determine which tables are crucial for the query candidates provided.

5.2.2.1 Query Validation

Query validation employs a trial-and-error approach where for each query candidate and table in the schema, an in-memory SQLite database is created using `Connection::open_in_memory()`. Subsequently the connection is initialized using the schema with all tables in the schema except the current one.

Once the connection is ready, the `SchemaSubsetter` prepares every query candidate against this reduced schema. If preparation fails, this indicates the table removed is crucial for the query, thus it gets added to the list of crucial tables. This process is repeated for all query-table

combinations to build a minimal schema containing only essential tables for the execution of the set of query candidates provided to the subsetting algorithm.

5.2.2.2 Performance and Trade-off Characteristics

As a new in-memory database for each table-query combination is created, this leads to $O(queries \times tables)$ complexity. For schemas with XXX+ tables and multiple query candidates, this results in XXX-XXXms processing time compared to 10-15ms for heuristic approaches.

However, this execution-based approach provides superior correctness guarantees since heuristic approaches cannot determine whether candidates will execute successfully in real database environments. Using a real SQLite in memory instance both the correctness of query candidates and the actually referenced set of tables can be known prior to actual query execution.

5.2.2.3 Pipeline Integration

The `SchemaSubsetter` is used prior to prompting the model using ICL to ensure that the model context is used efficiently and attention is given to the relevant parts of the schema. The `optimize` method returns a `Schema` object containing only the tables identified as crucial through execution testing which is in turn processed by the subsequent pipeline stages like generation and refinement.

5.2.3 Query Projection (π)

The ICL module is implementing the query projection algorithm described in section 4.2.3. It is implemented in the `ICLGenerator` struct and wraps the `SqlModel` struct from `natural-models`. Using llama.cpp it runs model inference using a prompt optimized for in-context-learning with OmniSQL.

5.2.3.1 In-Context Learning

The `ICLGenerator::generate` method implements few-shot prompting with relevance filtering. Relevance filtering refers to removing all selected samples with a similarity of less than 0.5 (this value was emperically derived from evaluations). Thus only semantically or structurally similar samples are actually provided to the model.

The prompt is constructed based on an adapted version of the OmniSQL format (**?**): task overview, sql schema, filtered examples with similarity scores, explicit instructions, and the target question. The biggest differentiation to the prompt of **?** is the example section including similarity. For the actual SQL query presentation the code representation prompt format is used, inspired by DAIL-SQL (**?**). Every example includes the masked question, similarity score, and formatted SQL query for clarity.

5.2.3.2 Prompt Engineering Strategy

As outlined above the OmniSQL prompt was used as base for Natural's prompt together with a code representation prompt. Natural uses more explicit instructions (see A.4) compared to DAIL-SQL. Key differentiations include the inclusion of similarity scores to give the model the ability to weight the samples itself.

This prompt steers the model towards precision and chain-of-thought reasoning. It addresses apparent LLM issues like verbosity, over complexity or missing query constraints, as well as the hallucination and accuracy concerns identified in literature review section where LLMs generate plausible but incorrect SQL queries.

5.2.3.3 Model Integration and Performance

The `SqlModel` is wrapped around `llama-cpp-2` bindings and loaded globally at startup to avoid a 30-60 second initialization times per query. Using the `prompt` method from `natural-models` tokenization, inference, and decoding with configurable PromptParams for context size and generation limits is handled automatically.

5.2.3.4 Output Processing and Validation

As OmniSQL was finetuned to output its thoughts and predictions using markdown a markdown postprocessing module is needed, as well as a module to identify whether a SQL query is syntactically valid.

To extract possible candidates all generated responses are post-processed through a markdown parser (`pulldown-cmark`) which parses the model output and looks for the code-block fence characters ‘‘‘. Subsequently all candidates are processed in reversed order and the first full valid query is returned as potential candidate. The reversing of processing order is needed as the model outputs it's thoughts top-to-bottom as a markdown document with the most likely answer usually being output at the end. This approach ensures that a returned query candidates are executable and ensured to contain valid SQL. Thus Natural can aid the difficulty of generating perfect SQL queries by acknowledging limitations from large language models and implementing recovery and refinement mechanisms in the subsequent pipeline flow.

5.2.4 Self-Refinement Mechanism (ρ)

The self-refinement algorithm described in 4.2.4 is implemented in `natural-inference::pipeline::refinement`. This implementation corresponds to the ρ function, providing automated error correction through execution feedback and iterative improvement of generated SQL queries.

5.2.4.1 Error Correction Through Execution Feedback

The `Refinement::optimize` method takes a `RawQueryCandidate` and attempts to improve it through targeted prompting. The refinement prompt explicitly instructs the model to “spot any errors in the SQL query, correct them” and provides the original question, schema context, and the candidate query that needs improvement.

Contrary to the ICL generation which uses few-shot examples, the refinement process focuses on single-query self-correction with explicit error-fixing instructions.

5.2.4.2 Prompt Engineering for Correction

The refinement prompt uses a structured format similar to ICL generation but with key differences: it includes the original question as context, provides the candidate query that needs fixing, and uses explicit correction language (see A.5).

5.2.4.3 Model Resource Management

As both the ICL implementation and the refinement are ultimately generating SQL, they reuse the same underlying `SqlModel` from the shared pipeline context rather than loading separate models. This design choice significantly reduces memory requirements but potentially impacts refinement quality compared to having dedicated refinement models with different prompt conditioning. Due to the limited available hardware the effects of having a separate, distinc refinement model could not be verified. **?** have implemented multi model generation pipelines in **?** and achieved promising results which.

5.2.4.4 Output Processing and Validation

Similar to ICL generation, refined responses are processed through the same `Markdown` parser to extract the predicted SQL. The system validates each extracted candidate by trying to parse it using `QueryCandidate::try_from` and returns the first syntactically valid refinement starting from the bottom. This ensures that refinement produces executable SQL whilst falling back gracefully if refinement fails.

5.2.4.5 Integration with Pipeline

The refinement module is integrated into the main pipeline after the ICL generator. It processes the `RawQueryCandidate` and provides a self-correction mechanism that improves overall pipeline robustness. Notably both the unrefined and the refined query candidates are kept in the candidate set for majority voting.

5.2.5 Consensus Voting System (ν)

The `natural-inference::pipeline::voting` implements the majority function ν – this implementation closely follows the design and algorithm outlined in the section 4.2.5. The function `voting` providing a result-based self-consensus mechanism which takes in the set of candidates that were predicted during the generation phase and returns the most likely query.

5.2.5.1 Result-Based Self-Consensus

? have shown in **?** that self-consensus can significantly improve the accuracy of models that are already showing state-of-the-art performance. The result-based self-consensus mechanism is executing every query candidate, verifies that it works on the database, and loads it's results. By partitioning the available candidates into buckets based on their hashed result, queries can be deemed semantically equivalent if they end up in the same bucket. After every query has been exeuted and partitioned, the algorithm groups the buckets in a hash map `buckets` with the result hash as key and bucket as value.

5.2.5.2 Bucket Selection Heuristic

The heuristic method employed by the voting algorithm is steering the pipeline to agree with itself – if multiple generation attempts yielded the same result, these generation attempts are more likely to be accurate the others.

5.2.5.3 Error Handling and Fallback

Queries that fail execution are contained in an `errors` hash set rather than being included in voting buckets. If all candidates fail execution, the voting function fails with an error that contains all collected execution errors to provide diagnostic information to the calling side.

5.2.5.4 Limitations and Trade-offs

The result-based partitioning approach has limitations with regards to subtle errors that differ by few rows, as these would be treated as completely different result buckets. Furthermore executing every query candidate against the real database comes along with a significant performance penalty for expensive queries where the voting step might incur load onto the database, use significant amounts of memory for loading in all data into memory and slow down the voting as

the entire result needs to be hashed. This system lacks model uncertainty calibration for SQL confidence scoring which could be used as another dimension for partitioning.

Another more advanced optimization could be to group by result-schema and row count, which still needs to execute the candidate queries partially but does not load the actual data from the database into memory. For the scope of this thesis and small real-world application scenarios the cost of this stage is negligible compared to the llm inference done in previous steps.

5.2.5.5 Integration with Pipeline

The voting mechanism represents the last step of the pipeline function and selects the final consensus from all generated and refined candidates, and ultimately returns the output of the NATURAL system.

5.3 Supporting Systems and Optimizations

The supporting systems of NATURAL are important tools to simplify the actual pipeline development, employing performance optimizations and enable rapid development of new approaches and hypotheses. The primary support systems are `Vector` – a vector database abstraction on top of SQLite, `wwl` – a library that implements Wasserstein-Weisfeiler-Leman kernels (**?**) and the `embedding` and `masking` modules to compare and semantically search the embedding space.

5.3.1 Vector Database

The vector database implementation `Vector` is used both during sampling to construct the vector db and runtime to run the sample selection algorithm. Using SQLite and `sqlite-vec` to implement cosine search in text corpora of up to a few million samples dramatically simplified the development of NATURAL as no manual performance optimizations had to be implemnented. SQLite has a second architectural benefit besides relatively good performance, which is the portability characteristics of having all samples indexed in a single file on disk. This enables swapping out the respectively used samples by using another `.vector` file on disk. See A.7 for reference.

5.3.1.1 Database Schema

As the `.vector` file is still a regular SQLite database, alongside indexing tables using embeddings (`float[4096]` or `float[2048]`) regular database tables can be maintained to save data derived at sampling time. `Vector` hosts the database schema described in A.7.1.

Which maintains the word whitelist for query masking, the database graphs of the sample databases and the precomputed database indices. Thus at pipeline runtime the database indices and whitelists can be reconstructed through a cheap selection from `Vector` even though they are not indexed through embeddings.

The schema of the `samples` table largely drives the capabilities for example selection at runtime, it maintains embeddings of the masked sql query and natural language question respectively. Thus after cosine similarity search the original query or question can be reconstructed as well as the graph layout of the underlying database can be accessed through the `database` table.

The `database` and `database_indices` table contain `JSON` columns respectively for serializing complex graph and graph index structures to the database.

5.3.2 Embedding and Masking

The embedding and masking functionalities in `natural-sampling` abstract batch embedding of masked question and SQL queries using the GPU so that the runtime and sampling code can efficiently compute masked embeddings.

5.3.2.1 Architecture

The two primary types for embedding plain and masked text are the `SemanticString` and `MaskedSemanticString` structs which encapsulate strings alognside their embedding vectors. The `MaskedSemanticString` version additionally contains the masked and unmasked versions of the embedded string. Furthermore they offer `embed_seq` and `embed_chunked` methods which can be used for batch computation of embeddings.

5.3.2.2 Batch Computation

During sampling, offloading data to the GPUs memory is an expensive operation. When sampling hundred thousands or millions of samples, incurring the IO roundtrip from CPU to GPU for computing a single embedding is a performance bottleneck that increases the sampling time to an order of days (eg, when sampling the SynSQL) dataset. Thus batch processing can help to minimise the IO roundtrips needed between the CPU and GPU before computing embeddings and better utilise the available computing power.

Using the `embed_seq` method, a sequence of strings can be embedded and a sequence of `SemanticStrings` is returned. Furthermore through the `embed_chunked` method, chunks of unrelated data can be embedded in a single operation. The method maintains the original chunk layout which makes it possible to reconstruct the input semantic seamlessly. This optimization is used to embed the questions and sql queries alongside while maintaining clear separation in the embedding output.

```
INSERT DIAGRAM
```

5.3.2.3 Integration

The `masking` module tightly integrates wiuth the embedding pipeline. During sampling, questions and queries are first masked, then embedded together using `embed_chunked` and finally combined into `MaskedSemanticString` instances for storage in `Vector`.

5.3.3 Wasserstein-Weisfeiler-Leman Kernels

WWL kernels are used to construct the distance index $d \in \mathcal{D}$ for the σ function (see section 4.2.1) by computing schema distance through graph-based distance metrics. The underlying WWL implementation used by Natural is the reference implementation of WWL kernels from **?**, which is implemented in Python. Thus Natural can't directly use the WWL implementation as Rust and Python FFI is not directly possible. In order to buy into the Python libraries and ecosystem (eg, optimal transport) and utilize the existing implementation of the WWL kernels, writing a thin layer of Rust bindings around **?**'s implementation was deemed most sensible. Using `pyo3` calling Python code is made possible and let's Natural hook into existing ecosystems where needed. Due to the design laid out in section 4 the WWL implementation is only needed during sampling time and not required at runtime as all distances are precomputed and stored in a distance index, keeping the runtime portability characteristics of using Rust that were discussed above.

5.3.3.1 Rust-Python Integration Architecture

The WWL kernel library used by NATURAL is implemented using `pyo3` around the existing Python `wwl` library. The Rust bindings are usable through the `wwl` crate. The `WWLKernel` struct encapsulates the Python module reference and provides type-safe interfaces for both categorical and continuous propagation modes through respective methods. Using an existing graph library (`petgraph`) the complexity of the Rust bindings could be kept minimal, but required conversion from Rust's `petgraph` graph representation to Python's `igraph` graph representation.

5.3.3.2 Schema Graph Representation

The `natural-graphs` library implements the schema graph representation discussed in section 4.1.2.1 by parsing SQL statements using the `sqlparser` library and constructing an undirected `petgraph` graph instance. Nodes represent tables and columns and edges capture foreign key relationships and column ownership. Node labeling is used to encode schema constraints hierarchically as outlined in section 4.1.2.1.

5.3.3.3 Propagation Mechanisms

The `wwl` crate supports both categorical and continuous Weisfeiler-Leman propagation schemes. Categorical propagation operates on discrete node labels through iterative label refinement, suitable for constraint-based schema comparison. Continuous propagation utilizes node features encoded in matrices, enabling similarity computation based on quantitative schema properties like column cardinalities or data type frequencies (**?**). As NATURAL projects the database schema into a graph representation with categorical node labels, the categorical propagation scheme is used in `natural-graphs` to determine schema distance. The `wwl` bindings allow configuration of the kernel through the `KernelConfig` and `DistanceConfig` structs, which provide control over the iteration count (which defaults to 3), sinkhorn approximation settings etc.

5.3.3.4 Distance Computation and Caching

The pairwise Wasserstein distance is computed between the current database schema and each sample database during sampling phase. The distance index is a `HashMap` of a sample database name to it's distance to the current database NATURAL is running on. Caching of this computation is an efficient strategy to minimise runtime inference time as the distance index is static as long as database schemas are not mutated. NATURAL is storing JSON-serialized distance indices in `Vector` for fast distance lookups at runtime. This design prevents the computational cost of WWL distance computations from affecting inference, reducing runtime schema comparison to constant-time lookups for all previously indexed database combinations that were known during sampling time. Given that the set of sample database and target databases are usually fixed, the distance lookup becomes negligible in terms of computational cost.

5.3.3.5 Integration with Example Selection

The WWL kernel is integrated in the sampling phase, where distance indices are computed and cached. Furthermore during example selection (σ) the cached distance index is loaded from `Vector` for distance lookups.

The distance index is used for weighting the schema compatibility in the selection algorithm (see section 4.2.1) to balance structural similarity against question-level semantic relevance for improved in-context learning effectiveness.

5.4 Benchmarking Infrastructure

NATURAL's benchmarking infrastructure is implemented in `natural-benchmark`. This benchmarking infrastructure is enabling the development, verification and ultimately deployment of NATURAL pipeline. This section focuses on the respective benchmarking infrastructure development, performance optimizations that were required and engineering challenges encountered.

Whilst NATURAL's benchmarking infrastructure is not part of the core pipeline, it helps to develop and test hypotheses for extensions or design changes confidently, compare performance across pipeline versions, models and understand the benchmark performance.

5.4.1 Execution-Based Evaluation System

The `natural-benchmark` CLI evovles around the concept of executions which are single benchmark runs against a specific benchmark dataset (defaulting to SPIDER). An execution is the set of tests that have been executed using a version of NATURAL against a benchmark. The interface of `natural-benchmark` offers multiple options to create executions and subsequentlty understand NATURAL's performance:

1. Running new benchmarks via:
   `natural-benchmark new`
2. Continuing a halted execution via:
   `natural-benchmark continue -execution <id>`
3. Comparing performance on previous (partial) executions via:
   `natural-benchmark compare-to -previous <id>`
4. Computing statistics on past executions via:
   `natural-benchmark stats -execution <id>`

While benchmarking `natural-benchmark` maintains a local SQLite database maintaining a history of past performance, pipeline failures etc. for future introspection as well as comparison of approaches.

5.4.2 Cross-Dataset Validation

In order to measure the performance of NATURAL against multiple benchmarking datasets, the benchmarking and execution system in `natural-benchmark` must be generalize across one dataset. The benchmarking setup achieves this through a set of rust traits (eg, type characteristics) to model benchmarking datasets.

The traits `Benchmark`, `BenchmarkDatabase` and `BenchmarkTest` allow to abstract the file system layout of different evaluation datasets. Thus when implementing benchmark support against SPIDER the implementation absatracts the fact that SPIDER is using SQLite for test databases, and uses a JSON file to store the questions. The file system layout of SPIDER and BIRD is largely similar.

This abstracted benchmarking system (see A.7.2) allows integrating further benchmarks in the future (eg, SPIDER2 which relies on cloud databases like Snowflake) while maintaining the same ergonomics and tooling across benchmark datasets (eg, recording and indexing all test executions in a local database, recording pipeline logs etc).

5.4.3 Metric Computation

The benchmarking infrastructure computes the two metrics Execution Accuracy and Exact Match found in prevalent benchmark leaderboards (eg, Spider and Bird) through a Rust port of the Python reference implementation found in Spider. Spider determines the Execution Accuracy by comparing the result sets of two queries based on possible row and column permutations as well as optional order sensitivity. The Exact Match metric is determined by exact equivalence of the ground truth query ω_{ground} and the candidate query ω.

5.4.4 Benchmarking Challenges

Two primary issues emerged while developing the benchmarking infrastructure for Natural:

While loading result sets from the provided test database, a faulty code path failed to parse query result with types other than `Text`. This casued empty result sets to be returned as query results from both the ground truth query and the candidate query, resulting in a false positive, skewing the EA metric significantly upwards.

Furthermore frequent pipeline failures of Natural made it hard to get compute reliable performance metrics as context window constraints, out of memory issues and CUDA issues caused a signficiant number of test cases to fail, making subsequently computed metrics unreliable as only a subset of the dataset was included in the results.

5.5 Engineering Challenges and Lessons Learned

The implementation approach of Natural highlighted significant challenges in terms of practicability that were independent of the algorithmic design and theoretical problems. While Natural turned out to demonstrate the approaches recommended in the system design section, the development process was significantly slowed down due to constrained hardware availability, technology stack decisions and research methodology. The overall system architecture turned out positive, but the timeline was noticably expanded.

This section reflects on the most impactful challenges that were encountered during the implementation of Natural and analyzes how hardware limitations influenced design decisions, confidence in the performance of Natural and the overall impact on development velocity. These insights are transferrable beyond Natural and are applicable to any production-grade implementation of research algorithms.

5.5.1 Hardware Constraints and Development Velocity

Likely the most significant circumstance that affected development velocity was the constrained access to high performance hardware. The hardware available during development time was the RTX 3090 with 24GB of VRAM. This limitation fundamentally shaped the development process and research methodology as the iteration speed was severly slowed down.

5.5.1.1 Benchmarking and Evaluation Bottlenecks

A full benchmark execution of Natural against the Spider test dataset required around 12-36 hours depending on the number of candidates generated and refinement algorithm used during the benchmark. This made continuous validation during development a non trivial task, requiring extensive evaluation phases between times of active development.

Testing new hypotheses, validating algorithmic changes, experimenting with the implementation approaches, comparing different configurations and sampling strategies therefore became a multi-week processes rather than the rapid iteration typically desired in research environments.

The prolonged evaluation cycles created a cascading effect on development velocity. Rather simple adjustments that would normally be validated within hours required days of computational time, effectively preventing what should have been normal experimentation. The hardware constraints forced a rather conservative approach to experimentation, where each change needed to be carefully considered before committing to multiple days of benchmarking.

5.5.1.2 Memory Management and CUDA Issues

Frequent CUDA out-of-memory (OOM) errors made the evaluation interpretation non trivial as accuracy scores became inaccurate due to pipeline failures. This persistet as a

second development challenge throughout the implementation phase. These memory exhaustion issues occurred rather unpredictable during both development and benchmarking, depending on the database schema and question asked. As Natural uses a code representation prompt, the database schema is inlined into the prompt during inference. If Natural is executed against large databases, even with schema subsetting, the context window can be exceeded or the KV-Cache can exhaust the rest of the available GPU memory (typically around 2-4 gigabytes) after models have been loaded. This required extensive trial-and-error debugging to identify the root causes and implement workarounds.

5.5.2 Technology Stack Trade-offs

During the implementation of Natural, several critical technology stack decisions fundamentally impacted both development velocity and the final system characteristics. The two most consequential decisions were the choice of programming language and the model deployment strategy, each presenting distinct trade-offs between research efficiency and production readiness.

5.5.2.1 Programming Language: Rust vs Python

The decision to implement Natural in Rust rather than Python represents perhaps the most impactful architectural choice made during the implementation phase. Choosing Rust presented the opportunity to port the research system into production environments soon after, but came at the cost of having to port existing benchmarking infrastructure and not being able to exactly reuse the inference code used by OmniSQL. This decision significantly increased implementation time, likely doubling the development effort compared to what would have been required in Python.

Using Python would have provided immediate access to existing machine learning infrastructure and could have leveraged existing implementations by previous researchers directly in the implementation of Natural.

The rather mature Python ecosystem for NLP and ML would have eliminated the need to develop abstractions for model inference, vector database operations, and benchmark evaluation.

However, the Rust implementation as is provides significant benefits for production deployment and system integration. The resulting system can be embedded directly into database systems such as PostgreSQL through extensions, providing a path toward true production deployment that would be impractical with Python. The ability to compile an entire NL2SQL system into a single, standalone binary offers superior portability characteristics.

5.5.2.2 Model Deployment: Local vs Cloud-Based Inference

Using local and open source models reemphasized the limited local hardware access as using a cloud provider like OpenAI or Anthropic for inference would have singificantly speeded up

the benchmarking time. As a primary research goal of this thesis is to explore the open source capabilities of NL2SQL systems, choosing a proprietary inference service was deeemed unviable.

5.5.2.3 Integration Complexity and Ecosystem Maturity

The machine learning ecosystem of Rust proved to be singificantly less mature than anticipated, requiring extensive custom development and research for functionality that would be available off-the-shelf in prevalent Python packages. The integration with Python libraries for specialized components like the Wasserstein-Weisfeiler-Leman kernels required rather complex FFI code using `pyo3`, further adding architectural complexity and potential stability concerns.

Despite these challenges, the resulting architecture demonstrates that multi language approaches can be viable when different components have distinct requirements. The Python integration was isolated to the sampling phase, preserving the runtime portability characteristics of the core Rust implementation while still leveraging existing, specialized libraries where appropriate.

6 Evaluation

6.1 Test Environment and Methodology

6.2 Performance Tests

6.2.1 Latency

6.2.2 Throughput

6.2.3 Scalability

6.3 Use Cases

6.3.1 Natural Language Queries

6.3.2 Text Generation Within the Database

6.3.3 Semantic Search and Text Classification

6.4 Ablation Study

6.5 Comparison with Alternative Approaches

7 Discussion

7.1 Interpretation of Results

7.2 Limitations of the Implementation

7.3 Ethical and Data Privacy Considerations

7.4 Potential Future Developments

8 Summary and Outlook

8.1 Summary of Results

8.2 Addressing the Research Questions

8.3 Outlook for Future Research and Development

A Appendix

A.1 Installation Guide

A.2 API Documentation

A.3 Prompts

A.4 Inference Prompt

```
EXAMPLE OF THE PROMPT
```

A.5 Refinement Prompt

```
EXAMPLE OF THE PROMPT
```

A.6 Code Examples

A.7 Vector Database API

```
let question = "How many ..";

// Initialize vector database for similarity search
let vector = Vector::infer();
let model = EmbeddingModel::from_env().expect("failed to load embedding model");

// Query distance indices
let index = DistanceIndexRow::get("test_db", &vector)?;

// Embed question and perform similarity search
let selector = Selector::new(&question, None, &model, &vector)?;
let samples = selection::<4>(
    &Selector::new(
        question.as_ref(),
        None,
        &model,
        &vector,
    )?,
    &index.into(),
    &vector,
)?;
```

A.7.1 Vector Database Schema

```
CREATE TABLE IF NOT EXISTS whitelist (
  word             text not null primary key,
  frequency        integer not null,
  domain_count     integer not null
);

CREATE TABLE IF NOT EXISTS databases (
  id               text not null,
```

```
  name             text not null,
  graph            text not null
);

CREATE TABLE IF NOT EXISTS distance_indices (
  db_id            text not null,
  distances        text not null
);

CREATE VIRTUAL TABLE IF NOT EXISTS samples USING vec0 (
  id                 text not null,
  db_id              text not null,

  question           text not null,
  question_masked    text not null,
  question_embedding float[2048],

  sql                text not null,
  sql_masked         text not null,
  sql_embedding      float[2048]
);
```

A.7.2 Benchmark Traits

```
pub trait Benchmark {
    type Error: From<eyre::Report> + Debug;
    type Database: BenchmarkDatabase<Error = Self::Error>;

    fn name() -> &'static str;

    fn databases(&self) -> impl Iterator<Item = &Self::Database>;
    fn database(&self, id: impl AsRef<str>) -> Option<&Self::Database>;

    fn tests(&self) -> impl Iterator<
      Item = &<Self::Database as BenchmarkDatabase>::Test
    >;

    fn test(&self, id: impl AsRef<str>) -> Option<
      &<Self::Database as BenchmarkDatabase>::Test
    >;
}

pub trait BenchmarkDatabase {
    type Error: From<eyre::Report> + Debug;
    type Test: BenchmarkTest;

    fn id(&self) -> &str;
    fn connect(&self) -> Result<Database, Self::Error>;
    fn tests(&self) -> impl Iterator<Item = &Self::Test>;
```

```
    fn test(&self, id: impl AsRef<str>) -> Option<&Self::Test>;
}

pub trait BenchmarkTest {
    fn id(&self) -> &str;
    fn db_id(&self) -> &str;

    fn name(&self) -> &str;
    fn question(&self) -> &str;
    fn query(&self) -> &str;
}
```

A.8 Test Data and Results

References

Androutsopoulos, I., Ritchie, G. D., & Thanisch, P. (1995). Natural language interfaces to databases - an introduction. *CoRR*, *cmp-lg/9503016*. Retrieved from http://arxiv.org/abs/cmp-lg/9503016

Askari, A., Poelitz, C., & Tang, X. (2024). *Magic: Generating self-correction guideline for in-context text-to-sql.* Retrieved from https://arxiv.org/abs/2406.12692

Bogin, B., Berant, J., & Gardner, M. (2019, July). Representing schema structure with graph neural networks for text-to-SQL parsing. In A. Korhonen, D. Traum, & L. Màrquez (Eds.), *Proceedings of the 57th annual meeting of the association for computational linguistics* (pp. 4560–4565). Florence, Italy: Association for Computational Linguistics. Retrieved from https://aclanthology.org/P19-1448/ doi: 10.18653/v1/P19-1448

Choi, D., Shin, M. C., Kim, E., & Shin, D. R. (2020). *Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases.* Retrieved from https://arxiv.org/abs/2004.03125

Codd, E. F. (1970, June). A relational model of data for large shared data banks. *Commun. ACM*, *13*(6), 377–387. Retrieved from https://doi.org/10.1145/362384.362685 doi: 10.1145/362384.362685

Codd, E. F. (1974, January). Seven steps to rendezvous with the casual user. In J. W. Klimbie & K. L. Koffeman (Eds.), *Ifip working conference data base management* (p. 179-200). North-Holland. Retrieved from http://dblp.uni-trier.de/db/conf/ds/dbm74.html#Codd74 (IBM Research Report RJ 1333, San Jose, California)

Date, C. (2003). *An introduction to database systems* (8th ed.). USA: Addison-Wesley Longman Publishing Co., Inc.

Deng, X., Awadallah, A. H., Meek, C., Polozov, O., Sun, H., & Richardson, M. (2020). Structure-grounded pretraining for text-to-sql. *CoRR*, *abs/2010.12773*. Retrieved from https://arxiv.org/abs/2010.12773

Floratou, A., Psallidas, F., Zhao, F., Deep, S., Hagleither, G., Tan, W., ... Curino, C. (2024). Nl2sql is a solved problem... not! In *Cidr*. Retrieved from https://www.cidrdb.org/cidr2024/papers/p74-floratou.pdf

Gan, Y., Chen, X., Xie, J., Purver, M., Woodward, J. R., Drake, J. H., & Zhang, Q. (2021). Natural SQL: making SQL easier to infer from natural language specifications. *CoRR*, *abs/2109.05153*. Retrieved from https://arxiv.org/abs/2109.05153

Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Ding, B., & Zhou, J. (2023). *Text-to-sql empowered by large language models: A benchmark evaluation.* Retrieved from https://arxiv.org/

abs/2308.15363

Gao, Y., Liu, Y., Li, X., Shi, X., Zhu, Y., Wang, Y., ... Li, Y. (2025). *A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql.* Retrieved from https://arxiv.org/abs/2411.08599

Guo, J., Zhan, Z., Gao, Y., Xiao, Y., Lou, J.-G., Liu, T., & Zhang, D. (2019, July). Towards complex text-to-SQL in cross-domain database with intermediate representation. In A. Korhonen, D. Traum, & L. Màrquez (Eds.), *Proceedings of the 57th annual meeting of the association for computational linguistics* (pp. 4524–4535). Florence, Italy: Association for Computational Linguistics. Retrieved from https://aclanthology.org/P19-1444/ doi: 10.18653/v1/P19-1444

Hendrix, G. G., Sacerdoti, E. D., Sagalowicz, D., & Slocum, J. (1978, June). Developing a natural language interface to complex data. *ACM Trans. Database Syst.*, *3*(2), 105–147. Retrieved from https://doi.org/10.1145/320251.320253 doi: 10.1145/320251.320253

Kate, R. J., & Mooney, R. J. (2006, July). Using string-kernels for learning semantic parsers. In N. Calzolari, C. Cardie, & P. Isabelle (Eds.), *Proceedings of the 21st international conference on computational linguistics and 44th annual meeting of the association for computational linguistics* (pp. 913–920). Sydney, Australia: Association for Computational Linguistics. Retrieved from https://aclanthology.org/P06-1115/ doi: 10.3115/1220175.1220290

Lei, F., Chen, J., Ye, Y., Cao, R., Shin, D., Su, H., ... Yu, T. (2025). *Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows.* Retrieved from https://arxiv.org/abs/2411.07763

Li, F., & Jagadish, H. V. (2014). Nalir: an interactive natural language interface for querying relational databases. In *Proceedings of the 2014 acm sigmod international conference on management of data* (p. 709–712). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2588555.2594519 doi: 10.1145/2588555.2594519

Li, H., Wu, S., Zhang, X., Huang, X., Zhang, J., Jiang, F., ... Li, C. (2025). *Omnisql: Synthesizing high-quality text-to-sql data at scale.* Retrieved from https://arxiv.org/abs/2503.02240

Li, H., Zhang, J., Li, C., & Chen, H. (2023). *Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql.* Retrieved from https://arxiv.org/abs/2302.05965

Li, H., Zhang, J., Liu, H., Fan, J., Zhang, X., Zhu, J., ... Chen, H. (2024). *Codes: Towards building open-source language models for text-to-sql.* Retrieved from https://arxiv.org/abs/2402.16347

Li, J., Hui, B., Cheng, R., Qin, B., Ma, C., Huo, N., ... Li, Y. (2023). *Graphix-t5: Mixing pre-trained transformers with graph-aware layers for text-to-sql parsing.* Retrieved from https://arxiv.org/abs/2301.07507

Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., ... Li, Y. (2023). *Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls.* Retrieved from https://arxiv.org/abs/2305.03111

Montgomery, C. A. (1972). Is natural language an unnatural query language? In *Proceedings of the acm annual conference - volume 2* (p. 1075–1078). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/800194.805902 doi: 10.1145/800194.805902

Popescu, A.-M., Etzioni, O., & Kautz, H. (2003). Towards a theory of natural language interfaces to databases. In *Proceedings of the 8th international conference on intelligent user interfaces* (p. 149–157). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/604045.604070 doi: 10.1145/604045.604070

Pourreza, M., Li, H., Sun, R., Chung, Y., Talaei, S., Kakkar, G. T., ... Arik, S. O. (2024). *Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql.* Retrieved from https://arxiv.org/abs/2410.01943

Pourreza, M., & Rafiei, D. (2023). Din-sql: decomposed in-context learning of text-to-sql with self-correction. In *Proceedings of the 37th international conference on neural information processing systems.* Red Hook, NY, USA: Curran Associates Inc.

Rahaman, A., Zheng, A., Milani, M., Chiang, F., & Pottinger, R. (2024). *Evaluating sql understanding in large language models.* Retrieved from https://arxiv.org/abs/2410.10680

Rajkumar, N., Li, R., & Bahdanau, D. (2022). *Evaluating the text-to-sql capabilities of large language models.* Retrieved from https://arxiv.org/abs/2204.00498

Reddy, S., Lapata, M., & Steedman, M. (2014). Large-scale semantic parsing without question-answer pairs. *Transactions of the Association for Computational Linguistics*, *2*, 377–392. Retrieved from https://aclanthology.org/Q14-1030/ doi: 10.1162/tacl_a_00190

Scholak, T., Schucher, N., & Bahdanau, D. (2021). PICARD: parsing incrementally for constrained auto-regressive decoding from language models. *CoRR*, *abs/2109.05093*. Retrieved from https://arxiv.org/abs/2109.05093

Shen, Z., Vougiouklis, P., Diao, C., Vyas, K., Ji, Y., & Pan, J. Z. (2024). *Improving retrieval-augmented text-to-sql with ast-based ranking and schema pruning.* Retrieved from https://arxiv.org/abs/2407.03227

Tang, L. R., & Mooney, R. J. (2001). Using multiple clause constructors in inductive logic programming for semantic parsing. In *Proceedings of the 12th european conference on machine learning* (p. 466–477). Berlin, Heidelberg: Springer-Verlag.

Togninalli, M., Ghisu, M. E., Llinares-López, F., Rieck, B., & Borgwardt, K. M. (2019). Wasserstein weisfeiler-lehman graph kernels. *CoRR*, *abs/1906.01277*. Retrieved from http://arxiv.org/abs/1906.01277

Wang, B., Shin, R., Liu, X., Polozov, O., & Richardson, M. (2020, July). RAT-SQL: Relation-aware schema encoding and linking for text-to-SQL parsers. In D. Jurafsky, J. Chai, N. Schluter, & J. Tetreault (Eds.), *Proceedings of the 58th annual meeting of the association for computational linguistics* (pp. 7567–7578). Online: Association for Computational Linguistics. Retrieved from https://aclanthology.org/2020.acl-main.677/ doi: 10.18653/v1/2020.acl-main.677

Woods, W. A., Kaplan, R., & Nash-Webber, B. (1972). *The lunar sciences natural language information system: Final report.* Cambridge, Massachusetts: Bolt, Beranek and Newman, Inc.

Xu, X., Liu, C., & Song, D. (2017). Sqlnet: Generating structured queries from natural language without reinforcement learning. *CoRR*, *abs/1711.04436*. Retrieved from http://arxiv.org/abs/1711.04436

Yaghmazadeh, N., Wang, Y., Dillig, I., & Dillig, T. (2017, October). Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, *1*(OOPSLA). Retrieved from https://doi.org/10.1145/3133887 doi: 10.1145/3133887

Yu, T., Li, Z., Zhang, Z., Zhang, R., & Radev, D. (2018, June). TypeSQL: Knowledge-based type-aware neural text-to-SQL generation. In M. Walker, H. Ji, & A. Stent (Eds.), *Proceedings of the 2018 conference of the north American chapter of the association for computational linguistics: Human language technologies, volume 2 (short papers)* (pp. 588–594). New Orleans, Louisiana: Association for Computational Linguistics. Retrieved from https://aclanthology.org/N18-2093/ doi: 10.18653/v1/N18-2093

Yu, T., Wu, C., Lin, X. V., Wang, B., Tan, Y. C., Yang, X., ... Xiong, C. (2020). Grappa: Grammar-augmented pre-training for table semantic parsing. *CoRR*, *abs/2009.13845*. Re-

trieved from https://arxiv.org/abs/2009.13845

Yu, T., Yasunaga, M., Yang, K., Zhang, R., Wang, D., Li, Z., & Radev, D. (2018, October-November). SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task. In E. Riloff, D. Chiang, J. Hockenmaier, & J. Tsujii (Eds.), *Proceedings of the 2018 conference on empirical methods in natural language processing* (pp. 1653–1663). Brussels, Belgium: Association for Computational Linguistics. Retrieved from https://aclanthology.org/D18-1193/ doi: 10.18653/v1/D18-1193

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., ... Radev, D. R. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *CoRR*, *abs/1809.08887*. Retrieved from http://arxiv.org/abs/1809.08887

Zelle, J. M., & Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *Proceedings of the thirteenth national conference on artificial intelligence - volume 2* (p. 1050–1055). AAAI Press.

Zhang, B., Ye, Y., Du, G., Hu, X., Li, Z., Yang, S., ... Mao, H. (2024). *Benchmarking the text-to-sql capability of large language models: A comprehensive evaluation.* Retrieved from https://arxiv.org/abs/2403.02951

Zhang, H., Cao, R., Xu, H., Chen, L., & Yu, K. (2024). *Coe-sql: In-context learning for multi-turn text-to-sql with chain-of-editions.* Retrieved from https://arxiv.org/abs/2405.02712

Zhong, V., Lewis, M., Wang, S. I., & Zettlemoyer, L. (2020, November). Grounded adaptation for zero-shot executable semantic parsing. In B. Webber, T. Cohn, Y. He, & Y. Liu (Eds.), *Proceedings of the 2020 conference on empirical methods in natural language processing (emnlp)* (pp. 6869–6882). Online: Association for Computational Linguistics. Retrieved from https://aclanthology.org/2020.emnlp-main.558/ doi: 10.18653/v1/2020.emnlp-main.558

Zhong, V., Xiong, C., & Socher, R. (2017). Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, *abs/1709.00103*. Retrieved from http://arxiv.org/abs/1709.00103