UNIVERSITY OF CALIFORNIA SAN DIEGO

Making Database Interactions More Natural Through Speech-Focused Query Syntax and Improved Natural Languate to SQL Translation

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Kyle Richard Luoma

Committee in charge:

Professor Arun Kumar, Chair
Professor Alin Deutsch
Professor Zhiting Hu
Professor Jingo Shang

2025

The Dissertation of Kyle Richard Luoma is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2025

DEDICATION

I dedicate this dissertation to my loving wife Michelle who has shown nothing but love and support throughout my academic journey.

To my sons Constantine and Conrad, may this research in some small way improve the future that belongs to you.

and to my parents *Debbie* the English teacher, and *Jack* the Engineer whos examples and willingness to spend long days and nights teaching and shaping me directly contributed to an excellent life;

and to the memory of my good friend *Ethan Clark*, whose passion prompted my first steps toward a lifetime of exploration in the field of computing. No matter how hard I study, I will never reach the level of skill that Ethan always demonstrated so effortlessly.

EPIGRAPH

True ease in writing comes from art, not chance,
As those move easiest who have learn'd to dance.
'T is not enough to no harshness gives offence,—
The sound must seem an echo to the sense.

Alexander Pope

You write with ease to show your breeding,
But easy writing's curst hard reading.

Richard Brinsley Sheridan

Writing, at its best, is a lonely life. Organizations for writers palliate the writer's loneliness, but I doubt if they improve his writing. He grows in public stature as he sheds his loneliness and often his work deteriorates. For he does his work alone and if he is a good enough writer he must face eternity, or the lack of it, each day.

Ernest Hemingway

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures ix

List of Tables xvi

Preface xviii

Acknowledgements xix

Vita xx

Abstract of the Dissertation xxi

Introduction 1

Chapter 1 Introduction 2

Chapter 2 Design and Evaluation of an SQL-Based Dialect for Spoken Querying 3
2.1 Introduction 3
2.2 Background 8
2.2.1 The Structured Query Language (SQL) 8
2.2.2 Natural and Controlled Natural Languages 10
2.2.3 SpeakQL 11
2.3 Our SpeakQL Dialect 11
2.3.1 Keyword Synonyms and Optional Syntax 12
2.3.2 Natural Functions 14
2.3.3 Query Clause Ordering 15
2.3.4 Query Unbundling 17
2.4 Implementation 23
2.4.1 Language Specification 23
2.4.2 SpeakQL to SQL Translation 32
2.5 User Study 44
2.5.1 Study Objectives 44
2.5.2 Study Protocol and Design 45
2.6 Results and Discussion 48
2.6.1 Quantitative Results and Hypotheses Tests 48
2.6.2 Feature Usage and Usefulness 51

2.6.3 Failure Analysis . . . 52
2.6.4 Learning Effect Impacts on Results . . . 53
2.6.5 Qualitative Survey Feedback . . . 55
2.6.6 Discussion of Results and Implications . . . 57
2.7 Conclusions and Future Work . . . 59

Chapter 3 SNAILS: Schema Naming Assessments for Improved LLM-based SQL Inference . . . 61
3.1 Introduction . . . 62
3.1.1 Preliminaries and Setup . . . 63
3.1.2 Our Benchmark Artifacts and Analyses . . . 66
3.2 Background . . . 69
3.3 Schema Identifier Naturalness . . . 69
3.3.1 Naturalness Categories . . . 70
3.3.2 Naturalness Classification . . . 71
3.3.3 Identifier Schema Naturalness Mapping . . . 74
3.3.4 Heuristics-based scoring . . . 75
3.3.5 Dataset Naturalness Classifications . . . 76
3.3.6 Training Data Collections . . . 76
3.3.7 ML Classifier-based scoring . . . 77
3.3.8 Character Tagging Feature . . . 78
3.3.9 GPT 3.5/4 Turbo Few-Shot-Based Scoring . . . 79
3.3.10 GPT Davinci Fine Tuning . . . 81
3.3.11 CANINE Fine Tuning . . . 81
3.3.12 Tokenizers . . . 82
3.4 Naturalness-modified Identifiers . . . 84
3.4.1 Decreasing Naturalness (Abbreviation) . . . 84
3.4.2 Increasing Naturalness (Expansion) . . . 86
3.5 Base Collections . . . 89
3.5.1 Datasets . . . 90
3.5.2 Data Sources . . . 94
3.5.3 NL Questions and Gold Queries . . . 105
3.5.4 Benchmark Naturalness Comparisons . . . 107
3.5.5 NL Question - SQL Query Pairs . . . 107
3.6 NL-to-SQL Benchmarking Setup . . . 110
3.6.1 Prompt Generation . . . 111
3.6.2 Prompt Naturalness Modification . . . 112
3.6.3 NL-to-SQL Inference . . . 113
3.6.4 Query Naturalness Modification . . . 115
3.7 Performance Evaluation . . . 117
3.7.1 Query Execution . . . 117
3.7.2 Execution Result Set-Subset Matching . . . 117
3.7.3 Human Evaluation . . . 118
3.7.4 Schema Linking End-to-end Example . . . 118

3.8 NL-to-SQL Benchmarking Results . . . 123
3.8.1 Execution Accuracy . . . 124
3.8.2 Schema Linking Evaluation . . . 126
3.9 Discussion and Limitations . . . 132
3.10 Practical Applications . . . 137
3.10.1 For *New* Databases . . . 137
3.10.2 For *Existing* Databases . . . 137
3.11 Additional Tables and Figures . . . 141

Chapter 4 SKALPEL: Schema Knowledge Adjustments for LLM NL-to-SQL Performance Enhancements in Large Databases . . . 156
4.1 Introduction . . . 157
4.2 Methodology . . . 160
4.2.1 Benchmarks . . . 160
4.2.2 Schema Subsetting . . . 161
4.2.3 NL-to-SQL Generation . . . 162
4.2.4 The SKALPEL Subsetter . . . 162
4.3 Schema Subsetting . . . 163
4.3.1 Survey of Subsetting Approaches . . . 164
4.3.2 Subsetting Effectiveness . . . 167
4.3.3 Definitions . . . 168
4.3.4 Subsetting Challenges . . . 170
4.4 Experiments . . . 171
4.4.1 Discovering Schema Size Limits . . . 171
4.4.2 Experiment 1: Subsetting Performance . . . 172
4.4.3 Experiment 2: NL-to-SQL Performance . . . 176
4.5 Results . . . 177
4.5.1 Experiment 1: Subsetting Performance . . . 177
4.5.2 Experiment 2: NL-to-SQL Performance . . . 186
4.6 Discussion and Limitations . . . 187
4.6.1 Discussion . . . 187
4.6.2 Limitations . . . 188

Chapter 5 Related Work . . . 190
5.1 Related Work for Speakql . . . 190
5.2 Related Work for Snails . . . 193
5.3 Related Work for Skalpel . . . 195

Chapter 6 Conclusion and Future Work . . . 197

Bibliography . . . 198

LIST OF FIGURES

Figure 2.1. Bridging the gap between naturalness and determinism in programming languages; extension of figure from [8]. 4

Figure 2.2. SpeakQL Synonyms 13

Figure 2.3. Natural Functions 14

Figure 2.4. Alternate Ordering 15

Figure 2.5. Query Unbundling 17

Figure 2.6. SpeakQL Unbundled Query Grammar Excerpt 25

Figure 2.7. SpeakQL Synonym Keywords Grammar Excerpt 26

Figure 2.8. SpeakQL Where Expression Grammar Excerpt 27

Figure 2.9. SpeakQL Table (From) Expression Grammar Excerpt 28

Figure 2.10. SpeakQL Select Modifier Expression Grammar Excerpt 29

Figure 2.11. SpeakQL Natural Function Expression Grammar Excerpt 30

Figure 2.12. SQL Select Statement Grammar Excerpt 31

Figure 2.13. SpeakQL Select Statement Grammar Excerpt 31

Figure 2.14. SpeakQL Select Expression Grammar Excerpt 32

Figure 2.15. ASR to SpeakQL to SQL translation workflow. 33

Figure 2.16. Grammar Parser Performance Comparison 43

Figure 2.18. Number of Attempts By Individual Query 49

Figure 2.19. Feature Usage Observations and Avoidance Reasons 50

Figure 2.20. SpeakQL feature usefulness compared to SQL. 51

Figure 2.21. Attempt error categories by dialect 52

Figure 2.22. Dialect performance measured by total time and separated by group dialect order. 53

Figure 2.23. Dialect performance measured by first attempt planning time and separated by group dialect order. 54

Figure 3.1. Databases with poorly named, or less natural, schema identifiers perform poorly in LLM-based NL-to-SQL interfaces, and this project exposes the need for more natural schemas. We offer approaches and artifacts, including a naturalness classification and modification workflow, that can aid in the naturalness assessment and modification processes required to create a performance-enhancing natural view. In this way, the native schema remains as-is so that existing tools can continue talking to it without modification, while an LLM-based NLI can be integrated into the existing stack via a natural view. 62

Figure 3.2. *Mean Token in Dictionary*, the proportion of tokens in an identifier that match a word in an English dictionary, generally aligns with the SNAILS 3-class naturalness categorization approach. 70

Figure 3.3. Comparison of the SNAILS database collection (Artifact 1) described in Section 3.5.1 to other real-world and benchmark schema collections. SNAILS naturalness proportions are generally biased toward less natural identifiers and is more consistent with the real-world SchemaPile collection than other existing benchmarks including Spider and Spider Realistic. 72

Figure 3.4. Schema identifiers are classified (Artifact 2) and modified to increase or decrease naturalness as appropriate. Modified identifiers comprise the schema crosswalks used for schema modification during NL-to-SQL experimentation (Artifact 3). 73

Figure 3.5. Schema identifiers in our benchmark dataset are classified into a naturalness category and modified to increase or decrease naturalness as appropriate. Modified identifiers comprise the schema crosswalks used for schema modification during experiment query inference. 77

Figure 3.6. Cumulative distribution of schema identifier character counts by naturalness level. More natural (less abbreviated) identifiers logically have more characters. 83

Figure 3.7. Token count CDF, by naturalness level, for each language model. 83

Figure 3.8. Token counts to character count ratio, by naturalness level, for each language model. More natural identifiers generally contain fewer tokens-per-character than less natural identifiers, suggesting a higher presence of in-vocabulary keywords for more natural identifiers. 84

Figure 3.9. Proportion of identifiers in each naturalness category within the SNAILS real-world database collection (Artifact 1). Horizontal line markers indicate calculated combined naturalness as described in the appendix 92

Figure 3.10. Gold query clause composition - ASIS database 94

Figure 3.11. Gold query clause composition - ATBI database 95

Figure 3.12. Gold query clause composition - KIS database 96

Figure 3.13. Gold query clause composition - PILB database 97

Figure 3.14. Gold query clause composition - CWO database 98

Figure 3.15. Gold query clause composition - NPFM database 100

Figure 3.16. Gold query clause composition - NTSB database 101

Figure 3.17. Gold query clause composition - NYSED database 102

Figure 3.18. Gold query clause composition - SBOD database module example 103

Figure 3.19. Spider [119] and Bird [55] benchmarks classified with the Davinci-based classifier reveals that both benchmark databases have highly natural identifiers even compared to the most natural of the databases in our proposed benchmark. Our benchmark more closely aligns with the naturalness of the real-world schema collection SchemaPile [20] 108

Figure 3.20. At the individual database schema level, the SNAILS database collection has a diverse arrangement of naturalness levels. 109

Figure 3.21. Experiment setup workflow from NL question and schema as input to predicted query as output. 111

Figure 3.22. Screenshot of the query manual validation tool. This example depicts a query that passed set-subset matching, and is currently classified as ungraded. Helper information indicates that although the results matched, the incorrect table was selected during inference (AHEM instead of OHEM). This example was classified as incorrect. 119

Figure 3.23. Benchmark results evaluation includes generated and gold query execution on target schemas, parser-based analysis, and identifier set comparisons. We evaluate performance in terms of execution accuracy and schema linking (precision, recall, and F1). 123

Figure 3.24. Execution accuracy (proportion of correct queries) by model. There is slight accuracy improvement from native schemas to schemas modified to regular naturalness. Accuracy drops significantly for schemas modified to low naturalness. 124

Figure 3.25. Native identifier recall scores by model and naturalness level. Error bars set with confidence interval of 0.95. For all models, identifiers in lower naturalness categories yield lower recall scores. 127

Figure 3.26. Schema linking performance across database schema naturalness levels generally yields equal or better performance for higher levels of naturalness, with open source models Phind-CodeLlama2 (Ph-CdLlm2-ZS) and CodeS as well as OpenAI's GPT-3.5 (GPT-3.5-ZS) exhibiting higher sensitivity to changes in naturalness. Zero-shot prompting NL-to-SQL methods are denoted as (ZS). 128

Figure 3.27. Schema linking performance (QueryRecall score) changes across 3 example databases' native and virtual schemas. We selected these 3 examples to showcase the diversity of the databases in our collection. PILB Native is a more natural schema with 65 percent Regular, 22 percent Low, and 13 percent Least; NTSB Native contains 42 percent Regular, 34 percent Low, and 24 percent Least; and SBOD Native is the lowest naturalness schema with 24 percent Regular, 49 percent Low, and 27 percent Least. 130

Figure 3.28. Schema subsetting performance, measured with recall, precision, and f1 score, varies by naturalness levels for both DIN SQL and CodeS. Measurement Score is Recall, Precision, or f1 respectively. 131

Figure 3.29. QueryRecall and Execution Accuracy differences over the Spider [119] dev set modified using SNAILS renaming artifacts. 133

Figure 3.30. Schema linking performance (Recall Score) in terms of naturalness (Combined) by model on native schema identifiers. Histograms indicate non-parametric distributions of naturalness and Recall scores. 142

Figure 3.31. Schema linking performance (Recall Score) in terms of naturalness (Combined) by model on all schema identifiers (native and modified). Histograms indicate non-parametric distributions of naturalness and Recall scores with naturalness concentrations around modified schema identifier levels. 143

Figure 3.32. Schema linking performance (F1 Score) in terms of naturalness (Combined) by model on native schema identifiers. Histograms indicate non-parametric distributions of naturalness and F1 scores. 144

Figure 3.33. Schema linking performance (F1 Score) in terms of naturalness (Combined) by model on all schema identifiers (native and modified). Histograms indicate non-parametric distributions of naturalness and F1 scores with naturalness concentrations around modified schema identifier levels. 145

Figure 3.34. Kendall-Tau Correlations between the *Mean Token-to-Character Ratio* and *Query Recall*. 146

Figure 3.35. Kendall-Tau Correlations between *Query Combined Naturalness* and *Query Recall*. 146

Figure 3.36. Kendall-Tau Correlations between *Query Combined Naturalness* and *Query f1*. 146

Figure 3.37. Kendall-Tau Correlations between *Query Combined Naturalness* and *Query Precision*. 147

Figure 3.38. Kendall-Tau Correlations between *Regular Identifier Proportion* and *Query Recall*. 147

Figure 3.39. Kendall-Tau Correlations between *Low Identifier Proportion* and *Query Recall*. 147

Figure 3.40. Kendall-Tau Correlations between *Least Identifier Proportion* and *Query Recall*. 148

Figure 3.41. Kendall-Tau Correlations between *Regular Identifier Proportion* and *Query f1*. 148

Figure 3.42. Kendall-Tau Correlations between *Low Identifier Proportion* and *Query f1*. 148

Figure 3.43. Kendall-Tau Correlations between *Least Identifier Proportion* and *Query f1*. 149

Figure 3.44. Kendall-Tau Correlations between *Regular Identifier Proportion* and *Query Precision*. 149

Figure 3.45. Kendall-Tau Correlations between *Low Identifier Proportion* and *Query Precision*. 149

Figure 3.46. Kendall-Tau Correlations between *Least Identifier Proportion* and *Query Precision*. 150

Figure 3.47. Kendall-Tau Correlations between *Regular Identifier Proportion* and *Execution Accuracy*. 150

Figure 3.48. Kendall-Tau Correlations between *Low Identifier Proportion* and *Execution Accuracy*. 150

Figure 3.49. Kendall-Tau Correlations between *Least Identifier Proportion* and *Execution Accuracy*. 151

Figure 3.50. Kendall-Tau Correlations between *Query Combined Naturalness* and *Execution Accuracy*. 151

Figure 3.51. Schema linking performance (F1 score) changes across database naturalness levels. 152

Figure 3.52. Schema linking performance (F1 score) changes across database naturalness levels. 153

Figure 3.53. Schema linking performance (Recall score) changes across database naturalness levels. 154

Figure 3.54. Schema linking performance (Recall score) changes across database naturalness levels. 155

Figure 3.88. Schema linking performance (Recall score) changes across database naturalness levels. 150

Figure 4.1. The prototype SKALPEL subsetter combines LLM-based question decomposition and embedding vector generation to perform table retrieval from a vector store. SKALPEL measures cosine distance between the embeddings of 2-3 sentence table descriptions and a set of object descriptions decomposed from an NLQ and returns tables and all of their columns whose distance are below a set threshold. 163

Figure 4.2. Schema subsetting methods use a variety of techniques and resources including frontier (API-based) LLMs, smaller locally-hosted and finetuned models (both LLM and SLM), and vector search. Some methods use more than one method such as CHESS which uses multiple LLM agents and vector search. Underlined methods indicate that a method is included in our experiments. 165

Figure 4.3. Execution accuracy (y-axis) over the medium- and large-sized schemas in the Snails benchmark generally exhibits a downward trend as the proportion of unneeded identifiers increases. 167

Figure 4.4. Mean total precision (x-axis) and mean total recall (y-axis) for each subsetting method for each database size. Besides the obvious drop in performance as database size increases, the data hint at some Pareto-like tradeoffs between recall and precision for most subsetting methods. 177

Figure 4.5. Inference time and Prompt tokens are log scale values with higher scores indicating better performance (e.g., lower token counts and inference time). Schema coverage indicates the proportion of the schemas in the combined benchmarks that the subsetting method can process. 178

Figure 4.6. These radar charts display the tradeoffs between performance (measured by precision, f1, and recall) and time and resource usage (measured by inference time, prompt tokens, and pre-processing). Sensitivity to database size (S, M, L, XL, XXL) varies by both subsetting method and measure, and generally performance across all measures decreases as database size increases. Metrics where lower is better (schema proportion, inference time, token usage, preprocessing rate) are inverted (1 - x). Inference time, prompt tokens, and preprocessing rate are fit to the range [0, 1] via natural log functions. 182

Figure 4.7. This chart displays total token (subsetting prompt tokens + NL-to-SQL prompt tokens) usage along the Y-axis (log_{10} of prompt tokens) and database size as defined in Table 4.1. Except for Crush and SKALPEL, the subsetting task dominates token usage to the point where NL-to-SQL prompt tokens are not visible. The filled area between the Crush and SKALPEL lines represent the additional tokens required for NL-to-SQL prompts, where the lower line represents subsetting tokens and the space between the lower and upper line represents the NL-to-SQL prompt tokens. 184

Figure 4.8. Without considering schema size, in aggregate all subsetting methods underperform vs. full schemas on NL-to-SQL execution accuracy. We discover that subsetting effects on execution accuracy (y-axis) are schema size-dependent (x-axis). Here, we bin schema sizes into two categories (s-m, and l+ which includes l, xl, and XXL categories), and we categorize flagship models (GPT-4.1, Gemini-2.5-Pro), economy models (GPT-4.1-nano, Gemini-2.5-Flash/Flash-Lite), and open source models (Llama 3.3, GPT-OSS). 185

LIST OF TABLES

Table 2.1. Synonyms in SpeakQL for SQL keywords. 13

Table 2.2. Thematic Category and Code Frequencies 58

Table 3.1. Example identifiers and their naturalness levels, from the SNAILS naturalness labeled dataset (Artifact 2). 69

Table 3.2. Performance comparison of different language models for classifying a database identifier's naturalness 78

Table 3.3. SNAILS Real-World Database Schemas 90

Table 3.4. Gold query clause counts for each SNAILS database. Columns represent a count of gold queries that contain the listed clause types. Qs is the count of question-query pairs for each database. C-Join is the subset of joins that require a composite key. Ex indicates the use of an exists clause. SQ indicates a subquery. Neg, Grp, Ord, and Hvg represent negation, group by, order by, and having. Note: MS SQL Server dialect replaces the common LIMIT clause with an equivalent TOP clause that precedes select items in the SELECT clause. 93

Table 3.5. SBO Demo Module Schemas 104

Table 4.1. Schema size distribution for each benchmark dataset. 160

Table 4.2. Percent of each schema size category that each subsetting method is capable of processing. Only 3 of the 7 evaluated methods are capable of processing all schemas. 171

Table 4.3. Experiment 1 Results: The mean of subsetting performance metrics for all $\mathbb{S}'$ by schema size (Sz) and subsetting and subsetting function Ψ. Metrics include Inference Time (T (s)), Prompt Token Count (Tokens), Perfect Recall (PRe), Schema Recall (SchRe), Schema Precision (SchPr), Schema f1 (Schf1), Relation Recall (RelRe), Relation Precision (RelPr), Relation f1 (Relf1), Attribute Recall (AtrRe), Attribute Precision (AtrPr), Attribute f1 (Atrf1), Relation Proportion (RelPn), and Attribute Proportion (AtrPn). Section 4.4.2 provides definitions for each listed metric. 181

Table 4.4. This table shows the Value Reference Problem (VRP) and Hidden Relation Problem (HRP) mean occurence percentages and sensitivities. Sensitivity (Sens.) is the proportion of a given problem occurence percentage (defined as realized problem over all instances where the problem might occur) to the percentage of all missing relations. 182

Table 4.5. Logistic regression coefficients, standard error, and significance (p value) for non-colinear parameters with execution accuracy as the dependent variable. n = 10,241, Pseudo R-squared = 0.099. Execution accuracy is binary (1, 0) where 1 indicates a correct NL-to-SQL generation given a schema subset. . 187

PREFACE

Almost nothing is said in the manual about the preface. There is no indication about how it is to be typeset. Given that, one is forced to simply typeset it and hope it is accepted. It is, however, optional and may be omitted.

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Arun Kumar for his support as the chair of my committee. Through multiple drafts and many long nights, his guidance has proved to be invaluable.

I would also like to acknowledge the "Smith Clan" of lab 28, without whom my research would have no doubt taken fives times as long. It is their support that helped me in an immeasureable way.

Chapter 2, contains material from "SNAILS: Schema Naming Assessments for Improved LLM-based SQL Inference" by Kyle Luoma, and Arun Kumar, which appears in Proceedings of the 2025 International Conference on Management of Data (SIGMOD'25). The dissertation author was the primary investigator and author of this paper.

Chapter 3, contains material from "SKALPEL: Schema Knowledge Adjustments for LLM NL-to-SQL Performance Enhancements for Large Databases" by Kyle Luoma, and Arun Kumar, which has been submitted for publication of the material. The dissertation author was the primary investigator and author of this material.

VITA

2005	B.S. in Business Management, Biola University, La Mirada, California
2017	M.S. in Management - Manpower Systems Analysis, Naval Postgraduate School, Monterey California
2018	B.S. in Computer Science, California State University - Monterey Bay, Seaside California
2025	Ph.D. in Computer Science, University of California San Diego

PUBLICATIONS

Kyle Luoma and Arun Kumar. "SKALPEL: Schema Knowledge Adjustments for LLM-based NL-to-SQL Performance Enhancements in Large Databases," Under Submission.

Kyle Luoma and Arun Kumar. "SNAILS: Schema Naming Assessments for Improved LLM-Based SQL Inference," Proceedings of the 2025 International Conference on Management of Data (SIGMOD'25), 1875–1900, 2025.

ABSTRACT OF THE DISSERTATION

Making Database Interactions More Natural Through Speech-Focused Query Syntax and Improved Natural Languate to SQL Translation

by

Kyle Richard Luoma

Doctor of Philosophy in Computer Science

University of California San Diego, 2025

Professor Arun Kumar, Chair

The Abstract begins here. The abstract is limited to 350 words for a doctoral dissertation. It should consist of a short statement of the problem, a brief explanation of the methods and procedures employed in generating the data, and a condensed summary of the findings of the study. The abstract may continute onto a second page if necessary. The text of the abstract must be double spaced.

Introduction

This optional section is barely described in the OGS manual other than saying it is optional and that it appears in the table of contents between the Abstract and the first chapter.

No formatting guidelines appear so presumably, it should be formatted like an ordinary chapter. It should appear after the `\mainmatter` macro because it should start on page 1.

Chapter 1

Introduction

Chapter 2

Design and Evaluation of an SQL-Based Dialect for Spoken Querying

2.1 Introduction

The database community has long explored new interfaces to make databases easier to access and query in various kinds of settings for both data professionals and lay users. Typing SQL remains the main form of usage for data professionals but many lines of research have explored other new modalities: visually-oriented, touchscreen-oriented (e.g., [40]), speech-oriented (e.g., [8]), and natural language interfaces or NLIs (e.g., [45, 27, 94, 116]). In particular, a recent paper describing a SQL dictation system [8] showed that modern automatic speech recognition (ASR) tools have matured enough to combine the benefits of dictating regular SQL in conjunction with touchscreen capabilities. Such speech-driven querying was shown to make query specification significantly faster in tablet environments, which could offer more flexibility for anywhere-anytime querying for data analysts and other data professionals.

In this paper, we go further to ask a more radical exploratory research question: *If we are to design a structured query language for the speech-first era, how should it look?* At first blush, it may sound odd to still study structured querying in the era of ChatGPT [11, 88, 71] and advances in NLIs [45]. Why bother with anything more than mere English "prompting"? First off, there is a difference in motivation: NLIs mainly target lay users, while our focus is on the swathe of data professionals who are *already familiar* with SQL and use it regularly. Second, AI models

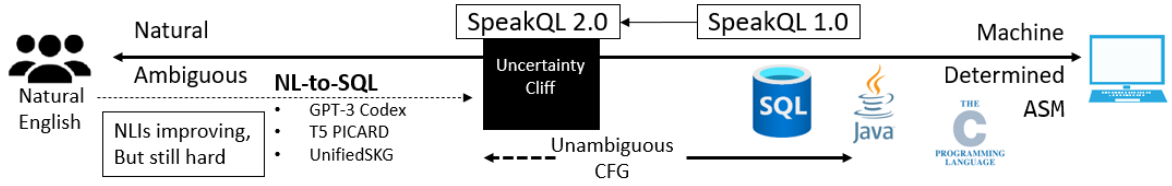


Figure 2.1. Bridging the gap between naturalness and determinism in programming languages; extension of figure from [8].

still do not offer the strong guarantee of "correct-by-construction" that SQL-like languages do–we know *exactly what* we get in response to a query. In contrast, AI models suffer the "hallucination" problem [38] that can cause insidious errors that may be hard to catch, especially on arbitrary database schemas and more complex queries. Overall, we believe exploiting ASR to make SQL-style querying easier could supplement existing and emergent NL-to-SQL systems by contributing natural structure to an otherwise ambiguous interaction modality. To explore the validity of this belief, we designed and executed a within-subjects A/B user study to compare the ease of use of a new speech-first dialect of SQL, SpeakQL, against SQL. We exclude ChatGPT and other AI models from our study because we are interested in the ease of use of a *structured* query language, not a natural language interface.

Desiderata

We start by describing some desirable properties of a spoken structured query language motivated by balancing *usability for specification* and *practicality in design*. (1) Minimal deviation from SQL to ensure it is easy to pick up for people who know SQL already. (2) Less rigid than SQL and more natural English-style flow in structure for the speech modality. (3) Unambiguous context free grammar (CFG) to ensure a valid spoken query can be translated to a *semantically equivalent* SQL query to enable use of an RDBMS as is for query execution.

Our Approach

In this paper, we explore the potential benefits of natural syntax by designing and evaluating a *new dialect of SQL* for spoken querying in response to the above desiderata. We call

our dialect SpeakQL 2.0 or just *the SpeakQL dialect*. Figure 2.1 illustrates where SpeakQL 2.0 falls in the spectrum of naturalness and determinism. Unlike NLIs that do not offer correctness guarantees, SpeakQL 2.0 has an unambiguous CFG. But it is less rigid than regular SQL although it is not multimodal (no touchscreen component) like what SpeakQL 1.0 proposed [8]. We design the SpeakQL dialect as an extension of the ANSI SQL grammar. It has several features to help increase the "naturalness" of speaking queries in the style of "stream of thought" instead of specifying everything all at once.

Motivating Applications

Before explaining the dialect's features, we describe some motivating application scenarios for spoken structured querying with a dialect such as SpeakQL.

- *On-the-go ad hoc database access.* Data-driven operations are now common in many domains. Organizations with field-based assets who operate in remote or austere environments such as the military [19] and the oil-and-gas industry [64] may face barriers to purely typing-based or touch-based data access on the field. Such barriers could be due to lack of workspace for keyboards, personal protective equipment requirements impeding typing/touching, or on-the-go working conditions such as a mobile headquarters. Spoken querying boosts capabilities in such settings for on-the-go ad hoc database access. As an example, suppose a military cyber-defense team in the field that must wear protective equipment detects anomalous behavior on some client devices. Pre-built (canned) analytics dashboards may not address all their unexpected querying needs. NLIs run the risk of erroneous query translation. But dictating a precise structured query can empower the team to access and analyze relevant network traffic data without compromising team security, agility, and query fidelity.

- *Assistive technology for people with motor impairment.* The U.S. Bureau of Labor Statistics data shows that in 2020 in the IT and engineering sectors, 18% of nonfatal injuries/illness

involving days away from work were in the upper extremities, viz., shoulders, arms, hands and wrists [3]. For such people, as well as for many people with arm disabilities, typing, clicking, or touchscreens may not be viable as a modality but speech can be a powerful modality. Thus, spoken structured querying can help data professionals with such disabilities or injuries.

User Study-based Evaluation

In this paper, we focus primarily on evaluating the *usability* of the SpeakQL dialect's features designed for spoken querying compared to regular SQL. We leave more extensive comparisons with other querying modalities or integration with multimodal interfaces to future work. We implemented the SpeakQL dialect and conducted a within-subjects A/B user study. We had 22 participants, all students familiar with SQL and relational databases. They were given a 6-table university database schema and asked to speak 12 queries, half designed to be simple and half, complex. The user study was conducted over a span of 4 months. Performance was measured in terms of both the time required to plan and specify a full query in response to an English prompt posed and the number of attempts till a fully correct query.

Overall, we find no statistically significant differences in the total query specification time for SQL vs. the SpeakQL dialect. This suggests that SpeakQL's extra verbosity was compensated for by lower thinking effort/time. Indeed, on average SpeakQL sees slightly lower median specification times for planning the complex queries. We also find that participants get better at speaking SpeakQL as they become more familiar with its features. The mass of qualitative textual feedback from the post-participation surveys also offer numerous interesting insights into the strengths and current weaknesses of SpeakQL. We see both positive and negative feedback on both the dialect and its individual features. But in aggregate, participants reported that SpeakQL made it "much easier" or "somewhat easier" to use than SQL between half to four-fifths of the time depending on the feature. Natural functions and unbundled queries were the most highly liked and used features, while synonyms were (perhaps surprisingly) deemed not that useful. In all, our user study results and surveys suggest that the SpeakQL dialect is indeed user-friendly.

We hope it spurs more research conversations in the community on more design iterations, additional features, and ultimately, fully fledged spoken querying for databases in more contexts.

SpeakQL Design and Features.

The SpeakQL dialect has 4 new features with increasing sophistication, grouped into 2 categories: smaller local changes to the grammar's production rules and deeper structural changes with more complex rules. Each feature is *optional*, which means regular SQL syntax is also valid in SpeakQL. Section 3 dives into their details with examples but we summarize their rationale here. The first category has the following two features. (1) *English synonyms* for some SQL keywords such as SELECT and FROM. (2) *Natural functions* to omit speaking of special character symbols such as commas and parentheses in some contexts. These two extensions make SpeakQL sound less like code and more like English (compared to SQL). So, they enhance the "naturalness" of the spoken query.

The second category has the following two features. (3) *Clause reordering*. Specifically, the SELECT, FROM, and WHERE clauses can be spoken in any order. Query modifier clauses such as GROUP BY and ORDER BY too can be reordered. (4) *Unbundling* of complex queries into per-table decomposed queries. This allows users to reason about one table at a time instead of "all at once" like in SQL. This is inspired by function-stitching style programming seen for Python Pandas and Spark DataFrame. It can reduce the amount of schema context to keep in mind when speaking, albeit at the cost of raising verbosity and query token lengths. Overall, these two features reduce query rigidity and offer more freedom for "stream of thought" querying.

To summarize, this paper makes the following technical contributions:

- To the best of our knowledge, this is the first paper to systematically study and evaluate an extension to SQL tailored for spoken querying.
- We present the SpeakQL dialect with four new features that raise naturalness and reduce rigidity compared to SQL, while preserving correct-by-construction guarantees with a context free grammar.

- We describe the implementation of the SpeakQL dialect, including its grammar rules and our translator to convert any SpeakQL query to regular SQL to ensure it can be used for existing RDBMSs.
- We present an extensive user study-based evaluation of SpeakQL vs. SQL for spoken querying. Our empirical findings, both quantitative and qualitative, suggest the usability of such a dialect and also offer avenues for more research to improve it.

2.2 Background

2.2.1 The Structured Query Language (SQL)

Origins and Purpose

Introduced in 1974 [15], SQL is nearing its 50th birthday. Despite (or perhaps because of) its age, it remains the de facto standard language for database quyering. SQL (originally named SEQUEL) was intended for both application programmers and a non-programmer target audience of business professionals and other laypersons requiring data access from relational databases [14]. Its initial design and following updates were informed by a user-centric approach, and it is perhaps one of the first programming languages for which human-computer interaction considerations were deliberately studied [91, 90]. Due to its high popularity, declarative nature, targeted purpose, structured syntax, and human-centric design, SQL is a natural starting point for a spoken query language.

Syntax Grammar and SQL Variants

All SQL grammars include syntax rules for both *data definition language* (DDL) and *data manipulation language* (DML) statements. DDL statements are intended to enable specification of data structure; and DML statements enable data access and update functions [6]. Data analysts, informaticists, and other data consumers generally make use of DML statements to fulfill data requirements. DDL statements are generally expressed by database administrators and software developers responsible for designing, implementing, and maintaining data models within database management systems.

Numerous variants of SQL syntax exist from multiple vendors and open source projects. While each variant tends to contain implementation-specific features targeted at specific RDBMSs, most (if not all) generally adhere to the ISO/IEC 9075-1:2016 information technology standard for SQL [44]. Seven SQL grammars are available under various open source licenses on the ANTLR parser Github repository including: Hive, MySQL, PL-SQL, PostgreSQL, SQLite, Trino, and T-SQL [4].

Human Factors

Human factors evaluations were conducted as part of the SEQUEL development effort. Usability experiments comprised of teaching SEQUEL to programmer and non-programmer college students. The study yielded several results, including the recommendation to make SEQUEL a layered system consisting of three layers representing increasing levels of complexity, and the recommendation to replace complicated correlation and computed variable syntax with the join feature, which most SQL users are familiar with now.

Reisner also discovered that sources of minor errors when converting English statements into SEQUEL queries included ending errors, spelling errors, and synonym errors. These discoveries resulted in the recommendation to incorporate spelling correction, introduce a synonym dictionary to the language syntax, and a create stem-matching procedure as user aids that would enable users to use keywords with various forms of conjugation. [90] In a later study, Reisner also confirmed that query complexity has a directly proportional effect on the likelihood of error occurrence during query formulation [91].

A more recent SQL usability study revealed that user tendency toward invalid syntax synonyms, omission of punctuation, and the NL-like nature of some SQL keywords can be sources of programming errors among novice users. Table joins, aliases, and subqueries were also identified as significant sources of programming errors [62].

2.2.2 Natural and Controlled Natural Languages

Natural Language Interfaces

Natural language interfaces (NLIs), such as the recently popular ChatGPT [88, 11, 71], show that NL-based chatbots can be a viable tool for many purposes, including NL-to-SQL querying, wherein users express their query intent in regular English. NL-to-SQL is still an active area of research [45, 11, 94, 116, 111], and it has strong potential to lower the barrier to entry to lay users (people without SQL knowledge). But NLIs still suffer from three issues in technical applications such as database querying that data professionals may be wary of: *ambiguities*, which can confound user goals; *out-of-vocabulary terms*, common in database schemas and predicate content, can hinder accurate translation; and *lack of correctness guarantees*, compounded by the "hallucination" problem of generative NLP models. Additionally, users may tend to seek out a *latent syntax* when performing coding tasks with a NL interface to ground their uncertainty about language model capabilities [39]. That said, recent research suggests that NLI with more restricted grammar and/or structure can improve user experience for technically complex tasks [65].

Controlled Natural Languages

Controlled or restricted NL are based on an NL such as English but more restrictive in their lexicon, syntax, and semantics. They retain a majority of its base NL properties and are defined explicitly [48]. Most PLs (including SQL) are not controlled NLs because their syntax deviates too much from the NL and have many statements that do not exist in the NL. Controlled NLs have been evaluated against linear keyword languages such as SQL and found to be easier for novice users for performing data retrieval tasks [100]. Early research on controlled query languages suggests a continuum between formal and natural language exists with a mid-point serving as an optimal combination of structure and flexibility [70]. In contrast to NL-to-SQL described previously, a controlled NL does offer the benefit of being "correct-by-construction" at the cost of being less flexible than a natural language interface.

Naturalness

Naturalness of a controlled NL can be evaluated by how close an expression in it is to its base NL. This is evaluated in terms of both *readability* and *understandability*. These criteria can range from completely unnatural, where the controlled NL uses symbols, characters, and unnatural keywords, to languages with natural sentences where the controlled NL can yield expressions that appear as if they were written in the base languages [48].

2.2.3 SpeakQL

Prior work on SpeakQL 1.0, a speech+touch multimodal querying interface [8], was aimed at data professionals such as data analysts, nurse informaticists, and DBAs who desired ad-hoc on-the-go querying in settings without a regular computer but with mobile devices such as a tablet. That paper's user study showed that the SpeakQL interface reduced query specification times vs. typing SQL in such settings by 2.7x on average. But conversations with such data professionals in that work revealed a key functionality gap: people with SQL knowledge may want to do a quick record retrieval or analytics query in an ad-hoc setting where even touchscreens are unviable for query specification, let alone keyboards, but voice is feasible. Speech-driven querying can also help people with disabilities or temporary injuries and perhaps also augment conversational assistants such as Alexa, Siri, etc. to aid in database querying. That provided the basis for this exploratory work on a spoken SQL dialect.

2.3 Our SpeakQL Dialect

We now present our prototype speech-first dialect extension of SQL. We overload the prior art interface name to call our dialect SpeakQL. It has four new features, all optional for usage:

- Keyword Synonyms and Optional Syntax
- Natural Functions

- Query Clause Ordering
- Complex Query Unbundling

SpeakQL can be considered a controlled NL based on English that extends SQL. Note that SQL is a constructed language rather than a controlled NL. This distinction arises because the objective of the SpeakQL dialect is to increase naturalness of specifying queries, achieved via the introduction of English grammar features and the reduction of special character (non-alphabet) symbol usage in SpeakQL queries.

We defined the SpeakQL grammar by extending a big part of the MySQL grammar from the Antlr4 repository [4]. We added additional production rules within existing SQL rules to realize our features. This means that SpeakQL is a *superset* of that chosen SQL subset. That is, a query constructed using the regular SQL syntax and keywords is a valid SpeakQL query too. So, users can "fall back" on regular SQL if they desire to.

2.3.1 Keyword Synonyms and Optional Syntax

This is a simple feature designed to increase the naturalness of an SQL query by enabling more sentence-like expressions. The intuition driving the development of these features is that speech patterns may be more amenable to use NL-like behavior, e.g., omitting syntax such as symbols and punctuation but including concepts such as determinatives (e.g., THE) and prepositions (e.g., OF).

Synonyms

This feature is motivated in-part by early human-computer interface research performed on SQL users [90] that recognized the benefit of syntax synonyms and optional word stemming, as well as more recent observations on tendencies toward synonyms [62]. We introduce SQL keyword-equivalent synonyms for the most common DML syntax keywords: SELECT, FROM, and JOIN, as well as for the comma as a column- or table-delimiter within the SELECT and FROM clauses, respectively.

Table 2.1. Synonyms in SpeakQL for SQL keywords.

SQL Keyword	SpeakQL Synonyms
SELECT	Select, Find, Retrieve, Get, Show Me, Display, Present, What Is, What Is The, What Are, What Are The
FROM	From, From table, From Tables, In Table, In Tables
' , ' (Comma)	' , ' (Comma), And
JOIN	Join, Join Table, Join With Table, By Joining, By Joining Table, By Joining With Table, Joined With, Joined With Table

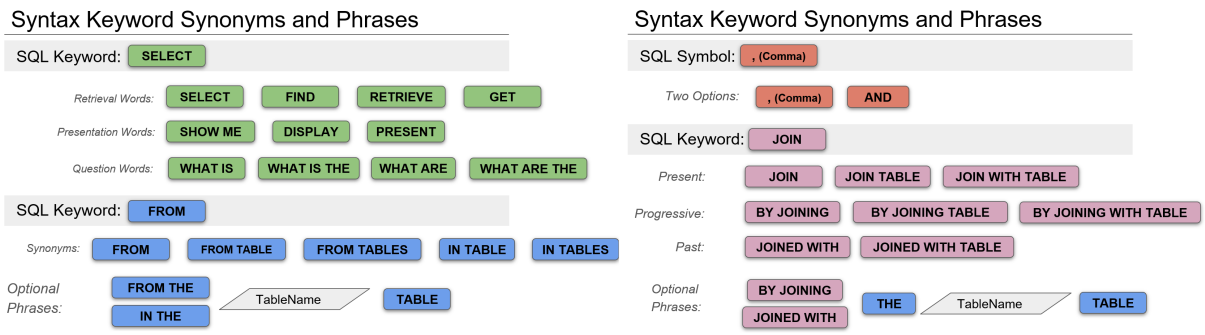


Figure 2.2. SpeakQL Synonyms

Keyword synonyms are listed in Figure 2.2.

Optional Syntax

SpeakQL allows for the use of optional THE and TABLE keywords when dictating a table expression. This permits expressions such as *SELECT everything FROM THE courseoffering TABLE*, which can be more natural for speaking than the SQL equivalent *SELECT everything FROM courseoffering*. Introducing the THE keyword as a determinant clarifies the context of the subject of the SpeakQL sentence, which is the *courseoffering* table. Appending the TABLE keyword to the expression provides further context and clarity that the referenced table is a tangible object within the database. While neither keyword changes the expression's meaning, their usage improves the naturalness of the SELECT statement, thus potentially improving the dictation experience.

Examples

```
SQL: SELECT area, wheelchairspaces
     FROM room WHERE floor = 2;
```

SpeakQL

Show me *area and wheelchairspaces* ***in the*** *room* ***table*** *where floor equals 2*

```
SQL: SELECT COUNT(id) FROM course;
```

SpeakQL:

What is the *count parenthesis id parenthesis* ***in the*** *course table*

2.3.2 Natural Functions

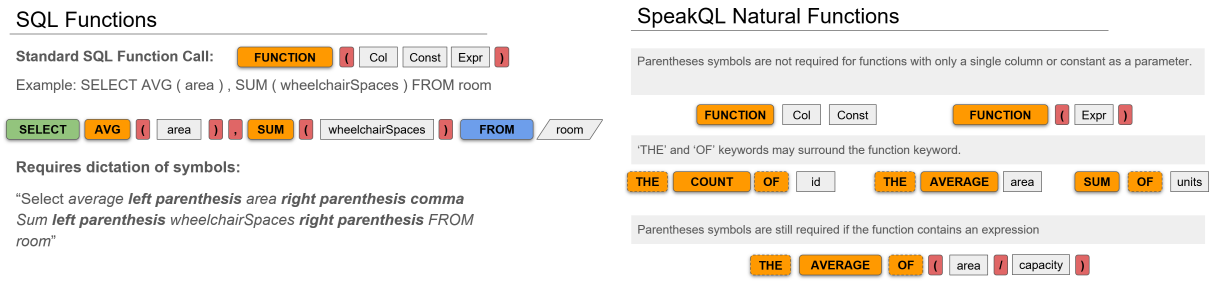


Figure 2.3. Natural Functions

The SQL subset we support includes aggregator functions such as SUM, AVG, and COUNT. In the prior work on the SpeakQL interface, users had to verbalize the parentheses symbols, which reduces the naturalness of dictation. While parentheses are often essential for disambiguation, for the SpeakQL dialect we identified a set of function references in which parentheses could be omitted safely without affecting the query's semantic meaning. Specifically, our SpeakQL dialect permits the expression of functions naturally, that is without verbalizing parenthesis, for functions that have a single constant or column as an argument. This feature also permits the optional syntax keywords THE and OF to surround the function name, resulting in more NL-like sentence expressions.

However, if the query intent involves the inclusion of an expression as a function argument, the verbalization of parenthesis remains a requirement. This allows the SpeakQL dialect to retain SQL's capability to pass mathematical, comparative, and subquery expressions as function arguments within the boundaries of dictated parentheses, ensuring we avoid ambiguity such as cases in which neighboring SELECT clause elements get misinterpreted as function arguments by the translator.

Natural function syntax and examples are portrayed in Figure 2.3.

Examples

```
SQL: SELECT COUNT(id) FROM course;
```

SpeakQL

Get the count of *id* ***from the*** *course* ***table***

```
SQL: SELECT AVG(units), COUNT(title)
     FROM course
```

SpeakQL:

Find the average *units* ***and the count of*** *title* ***in the*** *course* ***table***

2.3.3 Query Clause Ordering

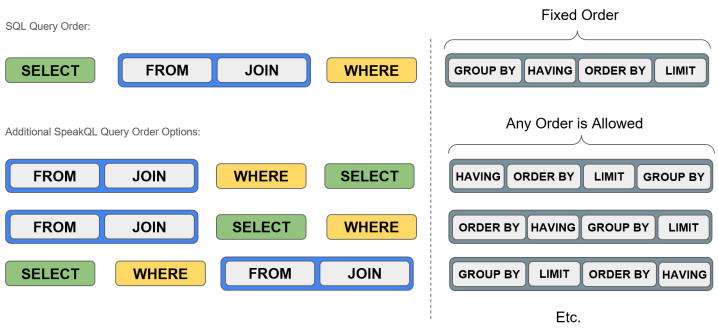


Figure 2.4. Alternate Ordering

This feature allows for optionally reordering the SELECT, FROM/JOIN, and WHERE clauses, as well as the GROUP BY, HAVING, ORDER BY, and LIMIT clauses. That is, these clauses may appear in any order within a SpeakQL statement.

SELECT-FROM-WHERE Ordering

Intuitively, we recognize that there are alternate paths to forming a SQL query. In some instances, the SELECT, FROM, WHERE ordering of SQL syntax is the order a user may find most-useful when dictating a query. In other cases, users may wish to reason about table sources and joins prior to defining columns and functions; alternatively, users may wish to establish restrictions using where predicates before defining other aspects of a query. Additionally, alternate ordering provides "second chances" for query recovery. For example, if a user is dictating a query and defines the SELECT and WHERE clauses, forgetting to state the table source, they may resolve this omission by dictating the FROM clause at the end of the query rather than starting over from the beginning.

Modifier Ordering

The *modifier* optional ordering feature (in this paper modifiers refer to the GROUP BY, HAVING, ORDER BY, and LIMIT expressions) is partially motivated by the observation that queries that require multiple modifier statements such as GROUP BY and HAVING tend to be more complex than simple single table queries or queries with a single modifier such as GROUP BY [7]. SQL syntax requires that these clauses occur in the strict order GROUP BY, HAVING, ORDER BY, and LIMIT. If these clauses appear out of order within a query, it is invalid and requires correction. While this is not a significant problem for typed queries, as they can easily be rearranged in a text editor, if such an error is introduced during the spoken querying process, more sophisticaed error correction is required and the query speaker must likely re-dictate the entire query.

Examples

```
SQL: SELECT DISTINCT termperiod FROM term
     WHERE year = 2022;
```

SpeakQL

From the *term* ***table show me distinct*** *termperiod* ***where*** *year equals 2022*

```
SQL: SELECT facultyname, ondays
     FROM courseoffering
     WHERE capacity > 20
     ORDER BY facultyname LIMIT 10;
```

SpeakQL:

In the *courseoffering* ***table where*** *capacity is greater than 20* ***find*** *facultyname* ***and*** *ondays* ***limit*** *10* ***order by*** *facultyname*

2.3.4 Query Unbundling

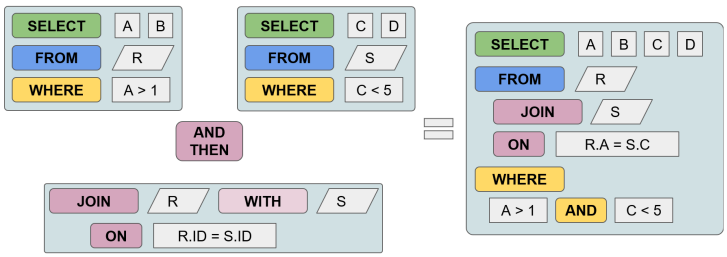


Figure 2.5. Query Unbundling

This features aims to make it easier to formulate more complex queries joining multiple tables. In SQL, the SELECT clause requires the user to specify *all* columns, scalars, and functions from across all tables in one go, followed by naming *all* table sources, subqueries, and joins in one go, followed by expressing *all* constraints in the form of WHERE predicates. Only after all that can the user add modifiers such as LIMIT, ORDER BY, HAVING, and GROUP BY.

Basically, it is a "global" approach to using the full database's schema in query construction. It forces the retention of a lot of schema details in the speaker's working memory for the entire duration of the query dictation, e.g., all non-aggregate columns that appeared in the SELECT clause must reappear in the GROUP BY clause at the end. While this may not be a big deal for typing, it can be a hindrance for ease of spoken querying.

Query unbundling aims to directly reduce this cognitive load based on the "stream of thought" philosophy inspired by functional programming APIs such as Python Pandas and Spark DataFrames. Basically, this is a "local-first" approach to using the database schema in query construction. It is closer to the logical query plan produced behind the scenes for SQL queries. We extend the grammar to permit the expression of *unbundled SELECT* queries that specify columns, table source, and WHERE predicates for *one relation at a time*. These unbundled relation can then be joined together using separate *join-with* clauses where the speaker defines the join predicate(s). Each unbundled query is delimited by the AND THEN, THEN, or NEXT keywords. Modifier clauses can be specified together in a single expression or separately using multiple modifier clauses. The order of the *unbundled SELECT*, *JOIN-WITH*, and *modifier* clauses remains optional, retaining the additional flexibility for dictation that SpeakQL offers.

Query Prompt

Find the titles of all courses offered in terms with the year 2022.

SpeakQL

join the *term* ***table with the*** *room* ***table on*** *term dot id equals courseoffering dot termid*
and then join the *courseoffering* ***table with the*** *course* ***table on*** *courseoffering dot courseid equals course dot id*
and then show me *title* ***in the*** *course* ***table***
and then from *room* ***where*** *floor equals 3* ***select average*** *capacity*
and then get *nothing* ***from*** *term* ***where*** *year equals 2022*

```
SQL: Select title from course
```

```
join courseoffering
   on course.id = courseoffering.courseid
join term
   on courseoffering.termid = term.id
where year = 2022;
```

Unbundled Query Parts

There are 3 types of unbundled query parts: a single-relation select-project, a join clause that specifies the join criteria between two single-relation queries, and a modifier clause that enables specification of GROUP BY, HAVING, LIMIT, and ORDER BY.

Within a single-relation select-project clause, a relation can be defined as a table reference or a subquery. Projections are expressed within the SELECT clause; and selections relating to the query's relation are defined within the WHERE clause as usual. As with non-unbundled SpeakQL queries, the table expression, SELECT expression, and WHERE expression may be dictated in any order. Table items for unbundled query parts that contain joins are consolidated in the output SQL query as a combination of a FROM clause for a single table and join expressions for the remaining tables. Otherwise, table items for unbundled queries where join conditions between tables are defined within a single-table part's WHERE clause are consolidated in the output SQL query's WHERE clause in the same fashion. We present two examples.

Unbundled SpeakQL query:

SELECT a and b FROM table R AND THEN GET c and d FROM S WHERE R.id = S.id

```
SQL: SELECT R.a, R.b, S.c, S.d FROM S, R WHERE R.id=S.id;
```

Unbundled SpeakQL query:

SELECT a and b FROM table R AND THEN GET c and d FROM S AND THEN JOIN R WITH S on R.id=S.id

```
SQL: SELECT R.a, R.b, S.c, S.d FROM S JOIN R on R.id=S.id;
```

All 4 queries above are logically equivalent, representable using the relational algebra expression shown below. The first line lists the individual unbundled queries in SpeakQL that perform the individual projections on R and S first before adding the equi-join part. (NB: We overload π for non-deduplicating project in bag semantics for brevity sake.)

$$\begin{aligned}&[\pi_{a,b}(R)]\&\&[\pi_{c,d}(S)]\&\&[R \bowtie_{P.id=S.id} S]\\ &\mapsto \pi_{R.a,R.b,S.c,S.d}(R \bowtie_{R.id=S.id} S)\end{aligned} \tag{2.1}$$

The above expression introduces two non-standard symbols to represent unbundled queries: square brackets [and] encase individual unbundled query parts, while the double ampersand && is the delimiter of the unbundled query parts, representing the AND THEN keyword or equivalent synonyms in SpeakQL. No other operations, say, relational algebra operations, are permitted in between the individual unbundled parts.

Translating an Unbundled Query to SQL

Our SpeakQL-to-SQL translator consolidates the unbundled query parts to produce a holistic valid SQL query.

First, column projections, function calls, and function arguments from all single-relation select-project unbundled parts are consolidated together into a single SELECT clause for the output SQL query. To avoid ambiguity in column names from different tables that are identical strings, every column reference in either a SELECT clause or WHERE clause that is not already in table-prefixed format (i.e., table.column) is prepended with its source table during the translation. This is shown in the examples above in which the translated SQL converts reference of column, say, “a” in SpeakQL to “R.a” in SQL. If the column reference already prefixes the table, our translator leaves it as is.

Second, selection predicates in the WHERE clauses of individual single-relation select-project query parts are consolidated into a single WHERE clause in the output SQL query using a boolean AND. A complex situation arises for a cross-table selection predicate within a boolean expression. In such cases, we only support *conjunctive* relationships between single-relation components of the overall boolean expression. That is, if a single-relation select-project query part contains multiple selections, these selections are encapsulated in parentheses prior to consolidation within the output SQL query. We currently do not support cross-table disjunctive predicates due to the additional complexity they add for speaking and anecdotally such cases being rarer in practice. As such, users always have the option of falling back on regular SQL syntax in SpeakSQL for such cases.

Detailed Example

Figure 2.5 illustrates how an unbundled SpeakQL query with two separate single-table WHERE clauses gets consolidated into a single SQL query. The relational algebra representation for both queries is given below.

$$[\pi_{a,b}(\sigma_{a>1}(R))]\&\&[\pi_{c,d}(\sigma_{c<5}(S))]\&\&[R \bowtie_{R.id=S.id} S]$$
$$\mapsto \pi_{R.a,R.b,S.c,S.d}(\sigma_{((a>1)\wedge(c<5))}(R \bowtie_{R.id=S.id} S)) \tag{2.2}$$

Selections Without Projections

One tricky situation in unbundling arises when a selection predicate (WHERE clause) is applied to a relation from which no columns are returned, i.e., it has no SELECT clause. This can lead to an "odd" impulse for a query speaker when nothing is retrieved from a table despite it having its own unbundled query part. To handle this situation, we introduce the NOTHING keyword as a special token within the *selectExpression* parser rule in the SpeakQL grammar.

SpeakQL:

FROM TABLE R SHOW ME a AND THEN GET NOTHING FROM S WHERE b = 2 and R.id=S.id

```
SQL: SELECT R.a FROM R, S WHERE R.b=2 and R.id=S.id
```

Automatic Group By Aggregation

To help reduce possible errors from incorrect GROUP BY clauses, we introduce the AUTOMATIC or AUTOMATICALLY keywords. The expression GROUP BY AUTOMATICALLY lets the translator infer the output SQL query's GROUP BY clause directly from the SELECT clauses of the unbundled query parts.

SpeakQL:

SELECT a AND THE SUM OF b FROM TABLE R AND THEN GET c and d FROM S AND THEN JOIN R WITH S on R.id=S.id AND THEN GROUP BY AUTOMATICALLY

```
SQL: SELECT R.a, SUM(R.b), S.c, S.d FROM R JOIN S
on R.id=S.id GROUP BY R.a, S.c, S.d
```

Verbosity vs. Brevity

In general, unbundled queries may contain more tokens than their corresponding SQL query, which means they may take slightly more time than reading the pure SQL translation. So, the higher naturalness comes at the cost of higher verbosity. This is an explicit tradeoff we made based on the rationale that perhaps less "think time" may be needed for the "local-first" unbundled SpeakQL query than the "global" SQL query. We view this tradeoff as reasonable to both reduce the chance of semantic errors and raise the user's overall query dictation experience.

Longer Detailed Example

The English prompt is as follows: "What is the average room seating capacity of rooms in buildings where the course with id 'CSE 232' has ever been offered?".

SpeakQL

Get *buildingname* ***from*** *building*

and then from the *room* ***table where*** *floor equals 3* ***show me the average*** *capacity*

and then get *nothing* ***from*** *courseoffering* ***where*** *courseid equals quote CSE232 quote*

then join *courseoffering* ***with*** *room* ***on*** *courseoffering dot roomid equals room dot id*

next join *room* ***with*** *building* ***on*** *room dot buildingid equals building dot id*

and then group by *automatically*

```
SQL: SELECT buildingname, AVG(capacity)
     FROM courseoffering
     JOIN room ON courseoffering.roomid = room.id
     JOIN building ON room.buildingid = building.id
     WHERE (courseoffering.courseid = 'CSE232')
        AND (room.floor = 3)
     GROUP BY buildingname;
```

2.4 Implementation

2.4.1 Language Specification

SpeakQL's grammar is defined by extending the modified Backus-Naur Form (BNF) of the MySQL grammar on the ANTLR GitHub repository [4, 81]. We use a subset of rules nested beneath the *querySpecification* parser rule, which are rules within the *dmlStatement* rule set. This is similar in size to the SQL subset supported in SpeakQL 1.0 [8]. While the majority of the MySQL grammar in this subset is reused as is, our major changes are within the *selectStatement* and *querySpecification* parser rules and their descendants.

Grammar Extension Strategy.

Our general strategy is to add intermediate rules between terminal token nodes and their parent SQL parser rules in order to modularize the grammar in a way that enables the SpeakQL-to-SQL translator to manipulate the abstract syntax trees (ASTs). This results in deeper ASTs and may result in a tradeoff between modularity and performance. Since SpeakQL is meant for spoken querying, near-realtime performance is an important design factor. To combat performance degradation due to deeper ASTs, we pruned the source MySQL grammar to include only DML parse rules. Within the DML parse rules, we further pruned out rules that were unlikely to be used during speech dictation.

Grammar.

We present an excerpt of the grammar for the most complex feature of SpeakQL, query unbundling, in Figure 2.6. Symbols or tokens in double quotes mean the content enclosed is a terminal symbol. The grammar contains a subset of child and descendant rules relevant to the unbundling feature. The root query specification rule, abbreviated as *querySpec*, is the point at which queries that use the unbundling feature diverge.

Unbundled queries may lead with any of the three unbundled query part type rules– *multiJoinExpr*, *selModExpr*, or *unbdQryOrdSpc*–followed by the *exprDelim* rule that contains the *AND THEN*, *THEN*, and *NEXT* terminal tokens. Although unbundled and non-unbundled SpeakQL queries may share the parser rules that describe SELECT clauses and WHERE clauses, unbundled queries have distinct FROM clause rules. The *tabExprNoJoin* and *fromClsNoJoin*, as their names suggest, omit any possible join expressions. That ensures that individual unbundled query parts reference only one table and leaves the join operations to the *multiJoinExpr* rule and its descendants. The following Figures 2.6 through 2.14 contain the grammar for all of the SpeakQL features described in this section.

<qrySpec> ::=	<qryOrdSpec> <selModExpr>
	| (<multiJoinExpr exprDelim)? (<selModExpr> <exprDelim>)? <ubndQryOrdSpc> (<exprDelim> (<ubndQryOrdSpc> | <multiJoinExpr)) (<exprDelim> <selModExpr)?
<ubndQryOrdSpc> ::=	<selExpr> <whereExpr>? <tabExprNoJoin>
	| <selExpr> <tabExprNoJoin> <whereExpr>?
	| <tabExprNoJoin> <selExpr> <whereExpr>?
	| <tabExprNoJoin> <whereExpr>? <selExpr>
<tabExprNoJoin> ::=	<frmClsNoJoin>
<frmClsNoJoin> ::=	<frmKW> <theKW>? <tblSrcNoJoin> <tblKW>?
<tblSrcNoJoin> ::=	<tblSrcItm> | "(" <tblSrcItm> ")"
<mltiJnExpr> ::=	<mltiJnPrt> (<exprDelim> <mltiJnPrt>)*
<mltiJnPrt> ::=	<mltiInnrJn> | <mltiOutJn> | <mltiNatJn>
<exprDelim> ::=	"and then" | "then" | "next"
<mltiInnrJn> ::=	<inJoinKW>? <joinKW> <tblSrcItm> <withKW> <tblSrcItm> (<onKW> <expr> | "using" "(" <uidLst> ")")?
<mltiOutJn> ::=	<joinDir> <outJoinKW>? <joinKW> <tblSrcItm> <withKW> <tblSrcItm> (<onKW> <expr> | "using" "(" <uidLst> ")")
<mltiNatJn> ::=	<natJoinKW> (<joinDir> <outJoinKW>)? <joinKW> <tblSrcItm> <withKW> <tblSrcItm>
<withKW> ::=	"with" | "with table" | "and"

Figure 2.6. SpeakQL Unbundled Query Grammar Excerpt

<selExpr> ::=	<selKW> <selSpec>* <selElmts>
<selKW> ::=	"select" | "find" | "retrieve" | "get" | "show me" | "display" | "present" | "what is" | "what is the" | "what are" | "what are the"
<tabExpr> ::=	<frmKW> <tblSrcs>
<frmKW> ::=	"from" | "from table" | "from tables" | "in table" | "in tables"
<joinKW> ::=	"join" | "join table" | "by joining" | "by joining table" | "joined with" | "join with" | "joined with table" | "join with table" | "by joining with table"
<onKW> ::=	"on"
<theKW> ::=	"the"
<tblKW> ::=	"table"

Figure 2.7. SpeakQL Synonym Keywords Grammar Excerpt

<whereExpr> ::=	<whereKW> <expr>
<whereKW> ::=	"where"
<expr> ::=	("not" | "!") <expr>
	| <expr> <logicOp> <expr>
	| <predicate> "is" "not"? ("true" | "false" | "unknown")
	| <predicate>
<logicOp> ::=	"and" | "xor" | "or"
<predicate> ::=	<predicate> "not"? "is" "in" <leftParen> <selStmt> <rightParen>
	| <predicate> "is" "not" "null"
	| <predicate> <compareOp> <predicate>
	| <predicate> <compareOp> ("all" | "any" | "some") <leftParen> <selStmt> <rightParen>
	| <predicate> "not"? "between" <predicate> "and" <predicate>
	| <predicate> "not"? "like" <predicate>
	| (expressionAtom)

Figure 2.8. SpeakQL Where Expression Grammar Excerpt

<tabExpr> ::=	<frmKW> <tblSrcs>
<frmKW> ::=	"from" | "from table" | "from tables" | "in table" | "in tables"
<tblSrcs> ::=	<theKW>? <tblSrc> <tabKW>? (<delim> <theKW>? <tblSrc> <tblKW>?)*
<tblSrc> ::=	<tblSrcItm> <joinPart>* | "(" <tblSrcItm> <joinPart> ")"
<tblSrcItm> ::=	"(" <slctStmt> ")" | <tblName> <tblAlias>? | "(" <tblSrcs> ")"
<joinPart> ::=	<inJoin> | <outJoin> | <natJoin>
<inJoin> ::=	<inJoinKW>? <joinKW> <tblSrcItm> (<onKW> <expr>)
<inJoinKW> ::=	"inner" | "cross"
<outJoin> ::=	<joinDir> <outJoinKW>? <joinKW> <tblSrcItm> (<onKW> <expr>)
<joinDir> ::=	"left" | right
<outJoinKW> ::=	"outer"
<natJoin> ::=	<natJoinKW> (<joinDir> <outJoinKW>)? <joinKW> <tblSrcItm>
<natJoinKW> ::=	"natural"
<joinKW> ::=	"join" | "join table" | "by joining" | "by joining table" | "joined with" | "join with" | "joined with table" | "join with table" | "by joining with table"
<onKW> ::=	"on"
<theKW> ::=	"the"
<tblKW> ::=	"table"

Figure 2.9. SpeakQL Table (From) Expression Grammar Excerpt

<selModExpr> ::=	<selModItm>? <selModItm>? <selModItm>? <selModItm>?
<selModItm> ::=	<grpByCls> | <havingCls> | <orderByCls> | <limitCls>
<grpByCls> ::=	<grpByKW> <grpByItm> (<delim> <grpByItm>)* ("with" "rollup")?
<grpByKW> ::=	"group by" | "group"
<grpByItm> ::=	<grpByExpr> <order> | <autoMtcKW>
<autoMtcKW> ::=	"automatic" | "automatically"
<grpByExpr> ::=	"not" <grpByExpr> | <predicate> "is" "not"? ("true" | "false" | "unknown") | "predicate"
<delim> ::=	"," | "and"
<havingCls> ::=	<havingKW> <expr>
<havingKW> ::=	"having"
<ordrByCls> ::=	"order by" <ordrByExpr> (<delim> <ordrByExpr>)
<ordrByExpr> ::=	<expr> <order>?
<order> ::=	<ascKW> | <descKW>
<ascKW> ::=	"asc" | "ascending"
<descKW> ::=	'desc' | 'descending'
<limitCls> ::=	'limit' <limitClsAtm>
<limitClsAtm> ::=	(decimalLiteral | simpleId)

Figure 2.10. SpeakQL Select Modifier Expression Grammar Excerpt

<funCall> ::=	<aggrFun> | <noParAggrFun> | <nonAggrWinFun> | <sclr-FunNam>
<aggrFun> ::=	<theKW>? ("avg" | "average" | "max" | "min" | "sum") <ofKW>? "(" ("all" | "distinct")? **<funArg>** ")" | <theKW>? "count" <ofKW>? "(" ("*" | "all"? **<funArg>** | "distinct" **<funArg>**) ")" ("std" | "std dev" | "std dev pop" | "std dev samp" | "var pop" | "var sample" | "variance") "(" "all"? **<funArg>** ")" "group concat" "(" "distinct"? **<funArg>** ("order by" <ordByExp> (<delim> <ordByExp>)*) ")"
<funArg> ::=	(<constant> | <colName> | <funCall> | <expr>) (<delim> (<constant> | <colName> | <funCall> | <expr>))*
<noParAggrFun> ::=	<theKW>? ("avg" | "average" | "max" | "min" | "sum") <ofKW>? ("all" | "distinct")? **(<constant> | <colName>)** | <theKW>? "count" <ofKW>? ("*" | "all"? <funArg> | "distinct" **(<constant> | <colName>)**)

Figure 2.11. SpeakQL Natural Function Expression Grammar Excerpt

<selStmt> ::=	<qrySpec>
<qrySpec> ::=	"select" <selSpec>* <selElmts> <fromCls>? <grpBy-Cls>? <havingCls>? <ordrByCls>? <limitCls>?
<fromCls> ::=	("from" <tblSources>?) ("where" <expression>)?
<grpByCls> ::=	"group by" <grpByItm> ("," <grpByItm>)?
<havingCls> ::=	"having" <expression>
<ordrByCls> ::=	"order by" <ordrByExpr> ("," ordrByExpr)
<ordrByExpr> ::=	<expression> ("asc" | "desc")
<limitCls> ::=	"limit" (<limitClsAtom> ",")? <limitClsAtom> | "limit" <limitClsAtom> "offset" <limitClsAtom>

Figure 2.12. SQL Select Statement Grammar Excerpt

<selStmt> ::=	<qrySpec>
<qrySpec> ::=	<qryOrdSpec> <selModExpr> | (<multiJoinExpr exprDelim)? (<selModExpr> <exprDe-lim>)? <ubndQryOrdSpc> (<exprDelim> (<ubndQryOrd-Spc> | <multiJoinExpr)) (<exprDelim> <selModExpr)?
<qryOrdSpec> ::=	<selExpr> <whereExpr>? <tabExpr> | <selExpr> <tabExpr> <whereExpr>? | <tabExpr> <selExpr> <whereExpr>? | <tabExpr> <whereExpr>? <selExpr>
<ubndQryOrdSpc> ::=	<selExpr> <whereExpr>? <tabExprNoJoin> | <selExpr> <tabExprNoJoin> <whereExpr>? | <tabExprNoJoin> <selExpr> <whereExpr>? | <tabExprNoJoin> <whereExpr>? <selExpr>

Figure 2.13. SpeakQL Select Statement Grammar Excerpt

<selExpr> ::=	<selKW> <selSpec>* <selElmts> | <selKW> <nothingElmt>
<selKW> ::=	"select" | "find" | "retrieve" | "get" | "show me" | "display" | "present" | "what is" | "what is the" | "what are" | "what are the"
<selSpec> ::=	"all" | "distinct" | "distinctrow
<selElmts> ::=	("*" | <selElmt>) (<delim> <selElmt>)*
<delim> ::=	"," | "and"
<selElmt> ::=	″*″ | <colName> | <funCall>
<nothingElmt> ::=	<nothingKW>
<nothingKW> ::=	"nothing"

Figure 2.14. SpeakQL Select Expression Grammar Excerpt

Figures 2.13 and 2.12 are excerpts of SpeakQL and SQL select statement grammar in a modified BNF format. The *'?'* symbol appended to a rule indicates that the rule is optional. The *'*'* symbol indicates that 0 to many occurences of a rule are allowed. Symbols or tokens surrounded by double quotes indicate that the content enclosed within the quotes is a terminal symbol. Both figures contain only a subset of child and descendant rules and both are limited to two levels of depth below the *selectStatement* rule, and certain SQL rules including *union*, *window*, and *into* are omitted for the sake of brevity. Figure 2.14 is an example of a child expression to the SpeakQL select statement grammar defined in figure 2.13 and is provided as an example implementation of the extension strategy that facilitates SpeakQL to SQL translation.

2.4.2 SpeakQL to SQL Translation

Our translator takes a valid SpeakQL query as input and returns a semantically equivalent SQL query. Figure 2.15 shows its place in the overall workflow. The translator performs a series of steps that we summarize next.

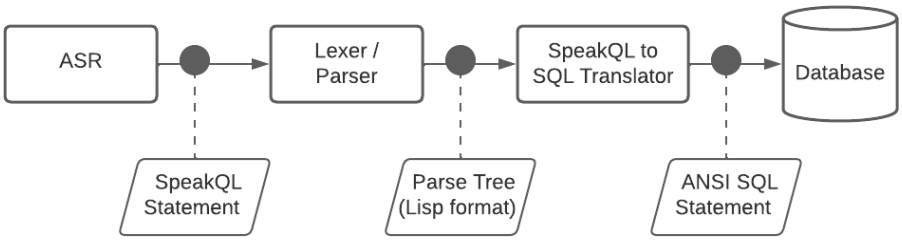


Figure 2.15. ASR to SpeakQL to SQL translation workflow.

The translator performs translation through a series of steps. The first step takes a valid SpeakQL query as input, where correctness is determined upstream of this task, and sends it to the Java-based SpeakQL parser via an HTTP post request. The parser responds with a Lisp-formatted abstract syntax tree AST that the Python-based translator transforms into an editable AST. The translator performs operations on the SpeakQL AST in order to transform it into a valid SQL query which include finding and replacing SpeakQL synonyms with SQL syntax, reordering expressions, reordering modifier items, and transforming unbundled queries into a single SQL select statement. The unbundling transformation step includes additional substeps including inference of a group by expression, consolidation of select elements (i.e. columns, functions, and scalars), consolidation of where statements, consolidation of join expressions, and finally consolidation of any table sources not included within join expressions.

Synonym Replacement

In this step performs a scan search of all active nodes in the AST for rules that are a member of the keyword and delimiter categories for which SpeakQL synonyms exist. This includes *selectKeyword*, *fromKeyword*, *selectElementDelimiter*, and more. When a keyword or delimiter rule is identified during the search, the translator performs an update on the rule node that replaces the terminal token representative of a SpeakQL synonym with its corresponding SQL keyword. Optional keywords such as THE and TABLE are identified in the same manner and removed from the AST.

Algorithm 1. Find and Replace Synonyms

$speakQlAst \leftarrow parseSpeakQL(query)$
$synonymKwRuleset \leftarrow \{selectKW, fromKW, joinKW, ...\}$
$syntaxSugarRs \leftarrow \{theKW, tableKW, ofKW, isKW\}$
$synonymHashMap \leftarrow \{(selectKW : SELECT), (fromKW : FROM), ...\}$
for all *nodes* such that $node \in speakQlAst$ **do**
  if $node.rule \in synonymKwRuleset$ **then**
    $node.replaceKW(synonymHashMap[node.rule])$
  end if
  if $node.rule \in syntaxSugarRs$ **then**
    $speakQlAst.removeNode(node)$
  end if
end for

Clause Reordering

This step includes reordering SELECT, FROM, WHERE, GROUP BY, ORDER BY, HAVING, and LIMIT clauses. The translator takes advantage of AST structures that guarantee that all children of the *queryOrderSpecification*, *unbundledQueryOrderSpecification*, and *selectModifierExpression* parser rule nodes are rules that require evaluation and reordering. Specifically, it is guaranteed that the query order clause parser rules will only contain the children *selectExpression*, *whereExpression*, and *tableExpression*. We also know that the *selectModifierExpression* has no more than four children of type *selectModifierItem*, each of which may have a single child of type *groupByClause*, *havingClause*, *orderByClause*, or *limitClause*. With these guarantees, the translator simply collects and reorders the AST's parser rules' children into the correct SQL clause order.

Natural Function Transformation

Translating natural functions to valid SQL function clauses involves locating and removing the optional keyword parser rules and inserting parentheses for all occurrences of the *noParenAggregateWindowedFunction* parse rule in the AST.

Algorithm 2. Reorder Select, From and Where Expressions

```
speakQlAst ← parseSpeakQL(query)
for all ruleNodes such that ruleNode ∈ speakQlAst do
  if ruleNode == qryOrdSpec then
    expr ← ruleNode.children
    reorderedExpr ← [∅]
    reorderedExpr.append(expr.selExpr)
    reorderedExpr.append(expr.tabExpr)
    reorderedExpr.append(expr.whereExpr)
    ruleNode.children ← reorderedExpr
  end if
end for
```

Algorithm 3. Reorder Select Modifier Items

```
speakQlAst ← parseSpeakQL(query)
for all ruleNodes such that ruleNode ∈ speakQlAst do
  if ruleNode == selModExpr then
    reorderedItms ← [∅]
    grpByCls ← ruleNode.getChildByRule(grpByCls)
    havingCls ← ruleNode.getChildByRule(hvingCls)
    ordrByCls ← ruleNode.getChildByRule(ordrByCls)
    limitCls ← ruleNode.getChildByRule(limitCls)
    reorderedItms.append(grpByCls)
    reorderedItms.append(havingCls)
    reorderedItms.append(ordrByCls)
    reorderedItms.append(limitCls)
    selModExpr.children ← reorderedItms
  end if
end for
```

Algorithm 4. Natural Function Transformation

```
speakQlAst ← parseSpeakQL(query)
syntaxSugarRs ← {theKW, ofKW}
for all ruleNode ∈ speakQlAst.nodesWithName(natFun) do
  for all child ∈ ruleNode.children do
    if child.rule ∈ syntaxSugarRs then
      speakQlAst.removenode(child)
    end if
  end for
  functionArgs ← ruleNode.getChildWithName(funArgs)
  speakQlAst.surroundNodeWithParens(functionArgs)
end for
```

Query Bundling

This is a series of up to 5 steps to consolidate a set of unbundled query parts into a valid SQL query. The number of steps varies by query and is dependent on the presence of WHERE clauses, JOIN clauses, and aggregate function calls. The steps in order are:

1. Infer GROUP BY clause, if any.
2. Bundle SELECT clauses.
3. Bundle WHERE clauses, if any.
4. Bundle JOIN clauses, if any.
5. Bundle all tables, if more than one.

Translating Unbundled Queries

The translator uses the same approach for each bundling step, which is to identify the first expression parser rule node in the AST for a given expression and use it as a migration target for additional expression parser rule nodes that may exist in the AST. For example, if a query contains two separate unbundled select-from-where queries connected with a join query, the translator will designate the first of the two *selectExpression* parser rules in the AST as *selectExpression'* and will migrate the *selectElements* that are children of the second *selectExpression* rule node to become children of *selectExpression'*. Parser rule node migration is accomplished by appending the migrated parser rule node's ID to the migration target rule node's child list and removing the migrated parser rule node's ID from the migration source rule node's child list.

Inferring the group by expression

A feature of the SpeakQL translator is its ability to infer a group by expression using the presence of aggregator functions and column references within multiple *selectExpression* parser rules. A user can instruct the translator to perform group by expression inference using the *group*

Algorithm 5. Transform Unbundled Query (High Level)

```
speakQlAst ← parseSpeakQL(query)
Require: ∃unbndQryOrdSpc ∈ speakQlAst
speakQlAst.inferGroupBy()
speakQlAst.bundleSelectElements()
speakQlAst.bundleWhereStatements()
speakQlAst.bundleJoinParts()
speakQlAst.bundleTables()
for all ruleNodes such that ruleNode ∈ speakQlAst do
  if ruleNode ∈ {qryOrdSpec, multQryOrdSpec, exprDelim} then
    speakQlAst.removeNode(ruleNode)
  end if
end for
```

Algorithm 6. Unbundled Function: Infer GroupBy

```
speakQlAst ← parseSpeakQL(query)
Require: ∃grpByCls ∈ speakQlAst ∧ ∃autoMtcKW ∈ speakQlAst
autoKws ← speakQlAst.nodesWithName(autoMtcKW)
for all autoKw ∈ autoKws do
  speakQlAst.removeNode(autoKw)
end for
gbNode ← speakQlAst.nodeWithName(grpByCls)
elmtsByTbl ← speakQlAst.getAllTablesAndElements()
gbItms ← [∅]
for all selElmt ∈ elmtsBtTbl do
  if selElmt is column then
    gbItems.append(selElmt)
    gbItems.append(delim)
  end if
end for
gbNode.children ← gbItms
```

by automatically keyword within either a single-table SpeakQL query or a multi-table unbundled query.

When the translator encounters the *group by automatically* instruction, it proceeds as depicted in algorithm 6. It makes use of AST helper functions *nodeWithName* which returns a list of all nodes with a given rule name, and *getAllTablesAndElements* which returns a Python dictionary ({ *table name* : [*selectElements*]}) that contains a list of select elements and function calls for each table referenced within the SpeakQL query. The translator uses the existing *groupByExpression* parser rule node that contains the *auomaticGroupByKeyword* parser rule and replaces the *automaticGroupbyKeyword* rule with a *groupByItems* parse rule. The *groupByItems* parse rule node serves as the parent node for individual *groupByItem* and *groupByItemDelimiter* nodes generated from non-function *selectElement* nodes registered in the table-element python dictionary. Because the *groupByItems* parse rule already exists beneath the *selectModifierExpression* parse rule, no further AST transformations are required; and the group by inference operation is complete after the generation of the *groupByItems* parse rule and its children *groupByItem* and *groupByItemDelimiter* parse rules.

Bundling select elements

Select element bundling is a required step for any SpeakQL query that makes use of the unbundling feature. Because any SpeakQL query that uses unbundling must have at least two *selectExpression* parse rules, select element bundling is a mandatory step of the bundling process.

As shown in algorithm 7 as *selElmtsRlNd'*, which abbreviates *selectElementsRuleNode*, the translator uses the first *selectElements* parser rule as a migration target for all other select elements within the SpeakQL query. Also depicted in the same algorithm is a preemptive method for avoiding ambiguity where the translator prepends each migrated select element with its associated table, resulting in a dotted id in the format *tableName.columnName*. This step is performed for both standalone column references and column references as arguments within function expressions. After iterating through each *selectElementExpression* parser rule

Algorithm 7. Unbundled Function: bundleSelectElements

```
speakQlAst ← parseSpeakQL(query)
for all ruleNode ∈ speakQlAst.nodesWithName(nothingElmt) do
  ruleNode.rename(selElmt)
end for
elmtsByTbl ← speakQlAst.getAllTablesAndElements()
selElmtsRlNd′ ← speakQlAst.nodesWithName(selElmts)[0]
newChildren ← [∅]
for all tblItm ∈ elmtsByTbl do
  for all selectElmt ∈ elmtsByTbl[tblItm] do
    newChildren.append(tblItm.selectElmt)
  end for
end for
selElmtsRlNd′.children ← newChildren
for all ruleNode ∈ speakQlAst.nodesWithName(selElmts) such that ruleNode! =
selElmtsRlNd′ do
  speakQlAst.removeNode(ruleNode)
end for
```

and migrating their associated *selectElements* children to *selectElementsRuleNode'*, a cleanup operation removes them from the AST leaving a single *selectElementsExpression* in the AST that contains all column and function references present in the original SpeakQL unbundled query.

Bundling where expressions

The where expression bundling step is optional, and is only invoked if at least one bundled query expression contains a *whereExpression* parse rule. When performing where expression bundling, the SpeakQL translator makes the following assumptions:

- All cross-table predicates are conjunctive (and)
- Multiple predicate expressions within a single unbundled query should be encased in parentheses before consolidation

Given these assumptions, where expression bundling proceeds as depicted in algorithm 8.

Similar to the select element bundling step, the first occurence of a *whereExpression* parse rule, *whereExpression'*, within the query AST and designates it as the target for additional *whereExpression* parse rule migration. Because where expressions are optional within

Algorithm 8. Unbundled Function: bundleWhereStatements

$speakQlAst \leftarrow parseSpeakQL(query)$
Require: $\exists whereExpr \in speakQlAst$
  $unbdlExprs \leftarrow speakQlAst.nodesWithName(unbdlQryOrdSpc)$
  $unbdlExpr' \leftarrow unbdlExprs[0]$
  $whereExpr' \leftarrow unbdlExpr'.nodeWithName(whereExpr)$
  for all $exprNode \in unbdlExprs[1:]$ **do**
    $tblName \leftarrow exprNode.nodeWithName(tblName)$
    $exprWhereExpr \leftarrow exprNode.nodeWithName(whereExpr)$
    $exprWhereKw \leftarrow exprWhereExpr.nodeWithName(whereKW)$
    $exprWhereExpr.removeNode(exprWhereKw)$
  end for

selectStatement parser rules, it is possible that *whereExpression'* is not a member of the first *selectStatement* parse rule, *selectStatement'*, in the query. Because of this possibility, the translator may perform an additional migration step where it migrates *whereExpression'* to become a child of *selectStatement'*.

The translator takes advantage of the recursive nature of the where expression grammar (see figure **??**) by designating where expressions that exist in *selectStatement* parse rules that are not *selectStatement'* as children of *whereExpression'*. During this migration process, the translator employs helper functions to surround migrated expressions with parentheses and deliminating the migrated expressions with *andKeyword* parser rules. The end result of this process is a single cross-table conjunctive *whereExpression* parser rule that contains all other *whereExpression* parser rules specified in other unbundled queries within the SpeakQL query.

Bundling join parts

Join part bundling collects all *multiJoinExpression* parse rules and aggregates them in an arbitrary order to form a valid SQL join expression. Join part bundling is an optional feature because an alternate form of relation joining where all table sources are specified within *fromExpression* parse rules, and join conditions between each table source are specified within *whereExpression* predicates (e.g. *from table one, table two where one.id = two.id*), is also possible. Currently, because join order in the resulting SQL query is arbitrary within the bounds

of its implementation rules, join part bundling is limited to allowing only inner joins. Queries that require outer join expressions may still be expressed using other SpeakQL features; but may not employ the unbundling feature. Outer join capability is a future SpeakQL feature development objective.

The join part bundling process begins when the translator creates a list of all *multiJoinExpession* parse rule nodes within the AST. Given this list, it performs iterative analysis on each expression to determine if a subquery exists as a table item within any of the *multiJoinExpression* nodes, and if so, substitutes the subquery with a subquery masking rule and alias in the join expression. The translator then checks the left and right table reference parse rules within the join expression to determine if an alias exists for each table, or if the table is referenced in the join expression by its alias. If an alias association is encountered, it creates *asKeyword* and *multiJoinTableAlias* parse rules and adds them as children to the target *joinExpression* parser rule that represents the objective SQL-correct join expression.

After analyzing all *multiJoinExpression* and making modifications toward SQL-correct syntax, the translator begins the join consolidation process by designating the first table referenced in the queries first *selectStatement'* as the base table upon which the SQL-correct join statement will be built. In order to ensure valid join expression chaining, the translator determines a table join order that ensures that each subsequent table added to a join expression references a table in its join condition that already exists within the objective join expression. In other words, while building the SQL-correct join expression, a *multiJoinExpression* will not be added to the target *joinExpression* until all tables referenced in its join condition have been added to the target *joinExpression*. The process will continue iterating over remaining *multiJoinExpressions* until the table existence constraint can be satisfied and all expressions have been added to the objective SQL-correct expression, or it is determined that *multiJoinExpressions* exist in the SpeakQL query that cannot be chained and the translator responds with an error condition. The translator then performs an additional safety check to ensure that all tables referenced in the query's *unbundledQueryOrderSpecification* parser rules are also referenced in a corresponding

multiJoinExpression parser rule, and throws an error if a violation is encountered.

After the join bundling process is completed, all join expressions consolidated within a single *multiJoinExpression* are migrated to the first *tableExpressionNoJoin* parse rule. The *tableExpressionNoJoin* parse rule name is then updated to *tableExpression* and becomes a valid SQL from + join expression; and join bundling is complete.

Bundling table expressions

The final step of the bundling process involves collection of all *tableExpressionNoJoin* parse rule nodes within the AST. This is only required in cases where an unbundled query contains multiple table references but has no associated *multiJoinExpression* parser rules. In this case the first *tableExpressionNoJoin*, designated as *tableExpressionNoJoin'*, is established as the expression migration target, and additional *tableExpressionNoJoin* rules that exist as children of additional *selectStatement* rules are added to *tableExpressionNoJoin'*. If a where expression exists in any (selectStatement) that defines a join condition between two tables within the query, the expression consolidation occurs during the previously described where expression bundling step. If no where condition is defined between two tables within the query, the translator still performs table expression bundling, and the result is a SQL statement that produces a cartesian product of the two tables.

Syntax tree cleanup

After all relevant bundling steps are complete, the translator performs syntax tree cleanup by removing parse rules from which their child sub expressions have been migrated. Upon completion of tree cleanup, the SpeakQL to SQL translation process is complete, and serialization of the abstract syntax tree's terminal nodes results in a valid SQL statement equivalent to the input SpeakQL query.

SpeakQL Parser Performance Comparisons

A comparison of the SQL and SpeakQL grammar implementations reveals that SpeakQL feature implementation results in deeper trees with additional branches. Naturally, this increase

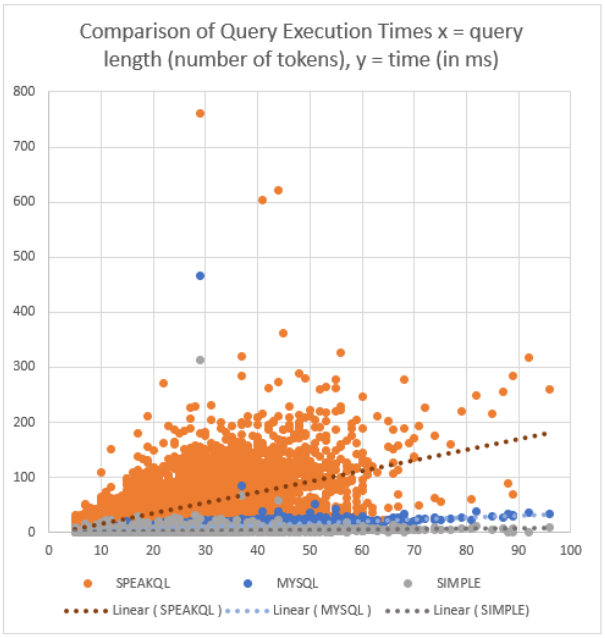


Figure 2.16. Grammar Parser Performance Comparison

in complexity will result in a corresponding increase in time required to parse a query. A performance comparison between three parsers, a MySQL grammar-based parser, a full SpeakQL parser that extends the entire MySQL grammar, and a simple SpeakQL parser that extends only a DML subset of the MySQL grammar was conducted to validate assumptions about parser performance improvements for smaller grammars.

We measured parse time for each parser using a set of 8,946 SQL queries. The full SpeakQL parser, on average, exhibited a 5.9x *slowdown* compared to the MySQL parser, suggesting that the modularity-based rule extensions negatively impact parser performance. On the other hand, the simple SpeakQL parser extending only a DML subset of MySQL rules exhibited a 2.6x *speedup* compared to the MySQL parser and a 15.1x *speedup* compared to the full SpeakQL parser. This performance increase is achieved by sacrificing certain SQL features including *window* and *union*. This rule omission is mitigated using a fallback strategy where SpeakQL queries identified by the translator as containing syntax not covered by the Simple SpeakQL parser are passed back to the full SpeakQL parser for processing.

2.5 User Study

To evaluate the utility of the SpeakQL dialect, we perform an apples-to-apples comparative A/B user study of *dictating SpeakQL vs. dictating regular SQL*. Our goal is to understand the role of our *dialect's features* on how efficiently one can dictate a syntactically valid query. So, we use a "Wizard of Oz" strategy to simulate a speech-based interface to allow us to focus only on the dialect's role. We do *not* want to confound this evaluation with orthogonal factors such as interface specifics, auxiliary additional modalities such as touch, etc. Thus, our goal here is different from the user study conducted for the SpeakQL 1.0 system [8], which compared typing SQL on a tablet against speech+touch modality on their multimodal tablet interface. We leave it to future work to study how to integrate the SpeakQL dialect into such multimodal interfaces.

2.5.1 Study Objectives

Research Questions and Hypotheses

The user study is motivated by three main research questions. We also posit our hypothesis alongside each question.

Q1: Effectiveness of alternate syntax. To what extent, if any, do syntax synonyms and symbol reduction improve user experience during spoken querying?

H1: Synonyms and reduction in special characters improve user experience. We expect that syntax synonyms and avoiding the need to dictate special characters such as comma, parentheses, or asterisk can make the process feel more natural and reduce number of errors and time taken to dictate simple queries.

Q2: Effectiveness of alternate ordering. To what extent, if any, does relaxing structure through alternate clause ordering reduce chances of errors during spoken querying?

H2: Alternate ordering reduces errors. We expect that relaxing the ordering requirement among clauses can reduce number of ordering-related syntax errors and reduce the amount of

time and number of attempts required to dictate a simple or complex query.

Q3: Effectiveness of unbundling. To what extent, if any, does unbundling a complex query into smaller single-relation parts reduce chances of errors during spoken querying?

H3: Unbundling reduces burden on working memory. We expect that unbundling a complex multi-table query into single-table query parts can reduce the speaker's working memory burden, reduce chances of errors when speaking the whole query, and reduce the number of attempts to craft a fully correct query.

2.5.2 Study Protocol and Design

Participants

We recruited participants from academic programs that teach SQL and data analytics. Since the SpeakQL dialect is primarily aimed at data professionals who already know SQL (not lay users), we also required participants to be familiar with SQL. They had to complete a short SQL screening test. Those who passed the test were invited to join the user study and offered up to $30 as compensation. It was structured as a flat rate of $6 for joining and $2 per query prompt completed (both SQL and SpeakQL conditions) for up to 12 query prompts. We also attempted to recruit participants from online database- and SQL-focused communities; but no prospective candidates passed the SQL screening questionnaire. This resulted in a set of participants sourced from a single university which, though the university celebrates a very diverse student body, may limit the scope of the study to Anglo-centric and English-only contexts.

Managing the Learning Effect

We applied a latin squares approach using counterbalancing to attempt to counteract the learning effect that is known to be inherent in within-subjects studies of two or more treatments [61]. Specifically, participants are divided into two groups: a SpeakQL-to-SQL group and a SQL-to-SpeakQL group. All participants in one group answered all 12 questions in one dialect first and then switch to the other dialect.

Database Schema and Queries

We use a 6-table university course database schema for our user study. It is a snowflake schema with a course offerings table at its center with three foreign keys referencing tables on courses, rooms, and terms. The course and room tables have foreign keys referencing tables on departments and buildings, respectively. For practice we created 3 realistic queries of increasing complexity: a single-table project, a single-table aggregate, and a 3-table join. The study itself uses 6 simple and 6 complex queries, with all the simple queries being single-table queries, while the complex ones join between 2 and 5 tables each.

Logistics

Study sessions were conducted over Zoom with a simple browser-based web interface. The interface enables the speaker to see the database schema, receive query prompts, dictate queries through their device's microphone, and see the live ASR transcription of their dictation. We use the state-of-the-art Whisper model for ASR [87]. The study administrator employed a separate "Wizard of Oz" control panel to evaluate correctness of the spoken query, to offer feedback in realtime (i.e., identify errors and ask for re-dictation if needed), to answer general questions on SQL or SpeakQL, and to manage the overall progression of the study session.

Quantitative Analyses

We evaluate performance using the following dependent variables: planning time (time from seeing query prompt to starting recording which accounts for schema review, note taking, questions, and verbal rehearsals), number of attempts till a fully correct query, completion time per attempt, and total completion time for all attempts. Independent variables are based on query prompt attribute and feature usage. Prompt attribute is the complexity of the correct SQL query, binarized as simple or complex (more details below). Feature usage determination is based on post-participation transcription of recordings and further analyses.

Measuring Query Complexity

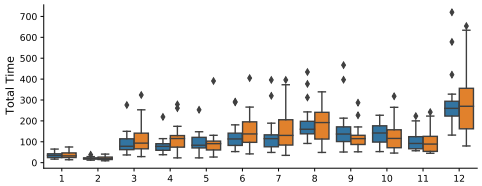
We use a series of weighted criteria: number of relations, number of projection terms (columns, functions and constants in SELECT clause), number of functions, number of predicates (in WHERE clause), number of joins, and number of modifiers (GROUP BY, HAVING, ORDER BY, and LIMIT). We use these to derive both raw and standardized query complexity scores.

Determining Feature Usage

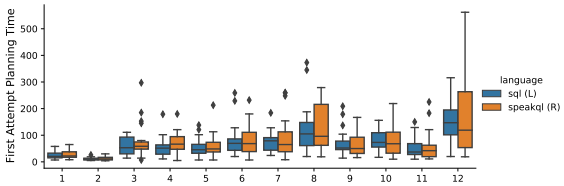
For each query attempt, we analyze feature usage post-hoc based on both the ASR transcription and the raw audio recordings. We used syntax-focused heuristics on the ASR output text to make this process easier for us, e.g., to check if unbundling was used, we performed a search for at least one "AND THEN" in the transcript and coded the attempt based on the presence or absence of that keyword.

Study Session Overview

Each session was for 90 minutes, and began with a 22 minute training video that provided an overview of SpeakQL features and usage examples. Following the video was a brief question and answer session and user interface demonstration. Participants then answered up to 15 prompts using both dialects (3 practice and 12 measured). They dictated in one dialect (SpeakQL or SQL) and then repeated the same queries in the same order for the other dialect. They were encouraged to use as many SpeakQL features as possible. The study administrator was available to answer questions about the schema or language syntax. The online context made a note taking prohibition unenforceable. As such, we deemed that a complete restriction would result in a temptation to take notes covertly. Participants were advised that they *should* try to perform without using notes, but if they did, they were required to discard them after the first dialect to mitigate learning effect bias.



(a) Total Time, All Attempts, By Individual Query



(b) Planning Time, First Attempts, By Individual Query

2.6 Results and Discussion

Our recruiting efforts elicited 35 prospective participants, of which only 30 completed the SQL filtering test. Of those, 29 were invited to participate and 23 ultimately did. Data from one participant was omitted from analysis due to a violation of the note transfer policy. 19 participants completed all 12 prompts; 3 answered at least 7 prompts in both dialects before opting to end the session.

2.6.1 Quantitative Results and Hypotheses Tests

Feature Usage Impact

Hypotheses H1 and H2 are feature-focused tests to analyze the impact of specific SpeakQL features. But since SpeakQL feature usage was optional, it turned out that participants did not consistently use many SpeakQL features across many prompts. Feature usage was also not consistent between participant groups (SpeakQL-first vs. SQL-first) and participants who had SQL as their first condition were less likely to use SpeakQL features as often as participants who had SpeakQL as their first condition. Due to this unexpected imbalanced voluntary usage rate for some features, we are unable to make any significant observations of feature usage effects on dependent variables.

Planning Time

Planning time is a part of H1 and H2. We analyzed total number of attempts to reach a correct query, as well as the planning time for the first attempt. We measure first attempt planning time as opposed to the planning time for the final attempt because our observation

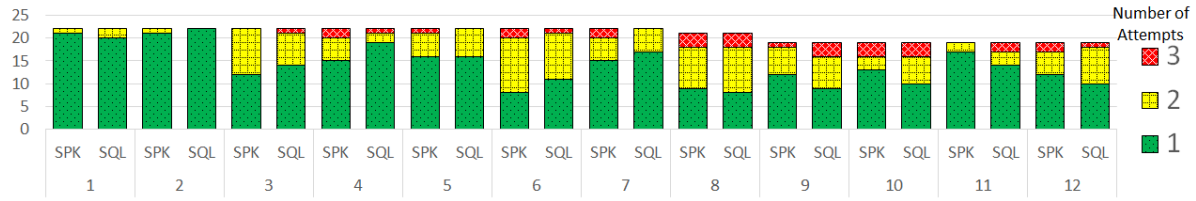
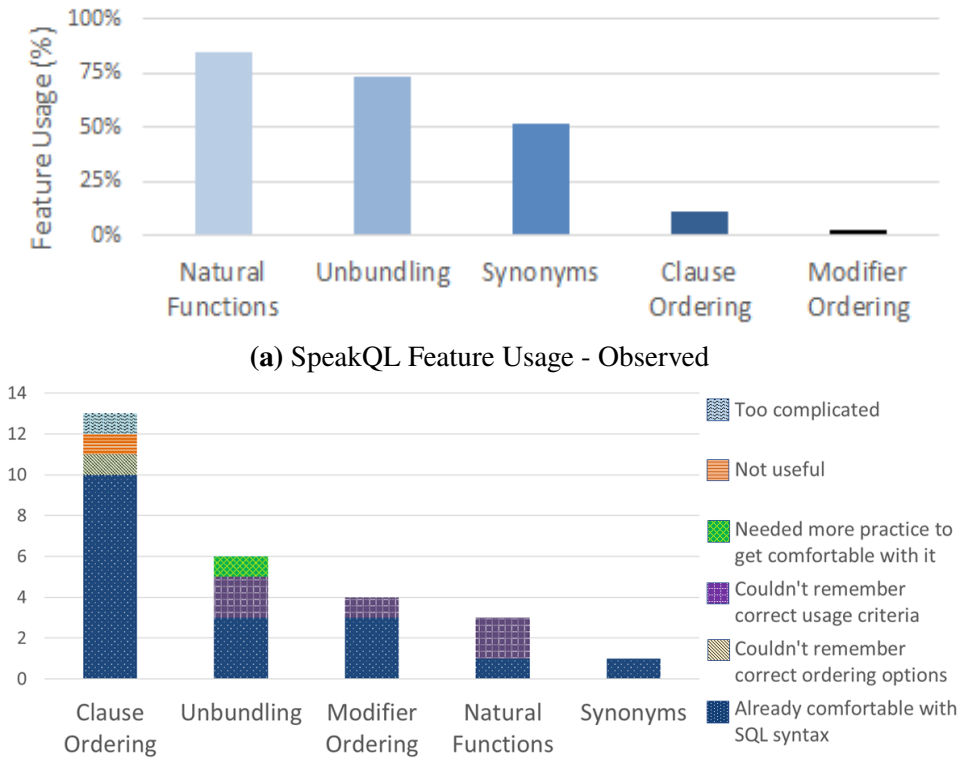


Figure 2.18. Number of Attempts By Individual Query

was that for second and third attempts, participants generally performed no additional planning. Thus, the time participants take to digest the prompt, analyze the schema, and plan the query is reflected within the first attempt planning time. Figure 2.17b shows the distribution for each query. The median planning time for simple queries ended up *longer* for SpeakQL than SQL: 38.5s vs. 31.5s, although this difference is not statistically significant (p = 0.14). (The p-values are derived using the Mann-Whitney U Test.) For complex queries, the median planning time is *shorter* for SpeakQL than SQL: 66.0s vs. 72.0s, but again this difference is not statistically significant (p = 0.295). We also compared these results for each individual query and find no statistically significant difference in median planning times for SpeakQL vs. SQL for any query (p-values between 0.07 and 0.48).

Number of Attempts

This is a part of all three hypotheses. Figure 2.18 shows the distributions. Overall, we do not see any statistically significant differences in either mean or median numbers of attempts between SQL and SpeakQL. But we observed that as queries become more complex, more second and third attempts are required. However, as the session progressed, first attempt correct answers increased, suggesting that participants gained a familiarity with the query dictation process and that counteracted the increasing query complexity. Additionally, we observe a slightly higher improvement advantage for SpeakQL over SQL from queries Q9 to Q12, i.e., the frequency of first attempt correct answers is higher for SpeakQL and that keeps going up.



(a) SpeakQL Feature Usage - Observed

(b) Participant Reasons for Not Using a Feature

Figure 2.19. Feature Usage Observations and Avoidance Reasons

2.6.2 Feature Usage and Usefulness

Analysis of feature usage shows varying levels of popularity for SpeakQL features. Since feature usage was optional, some participants relied more heavily on regular SQL syntax than others even for the SpeakQL condition. Figure 2.19a shows a significant disparity in popularity of the four main features. Natural functions are the most popular, followed by unbundling and synonyms; clause and modifier reordering are the least popular. Figure 2.19b shows the frequency of the self-reported reasons for not using a given feature. As expected, SQL familiarity is a top reason that dissuaded some participants from using SpeakQL features. In particular, clause/modifier reordering were not used due to that reason the most. Some also could not remember how to use unbundling or natural functions, perhaps due to their novelty.

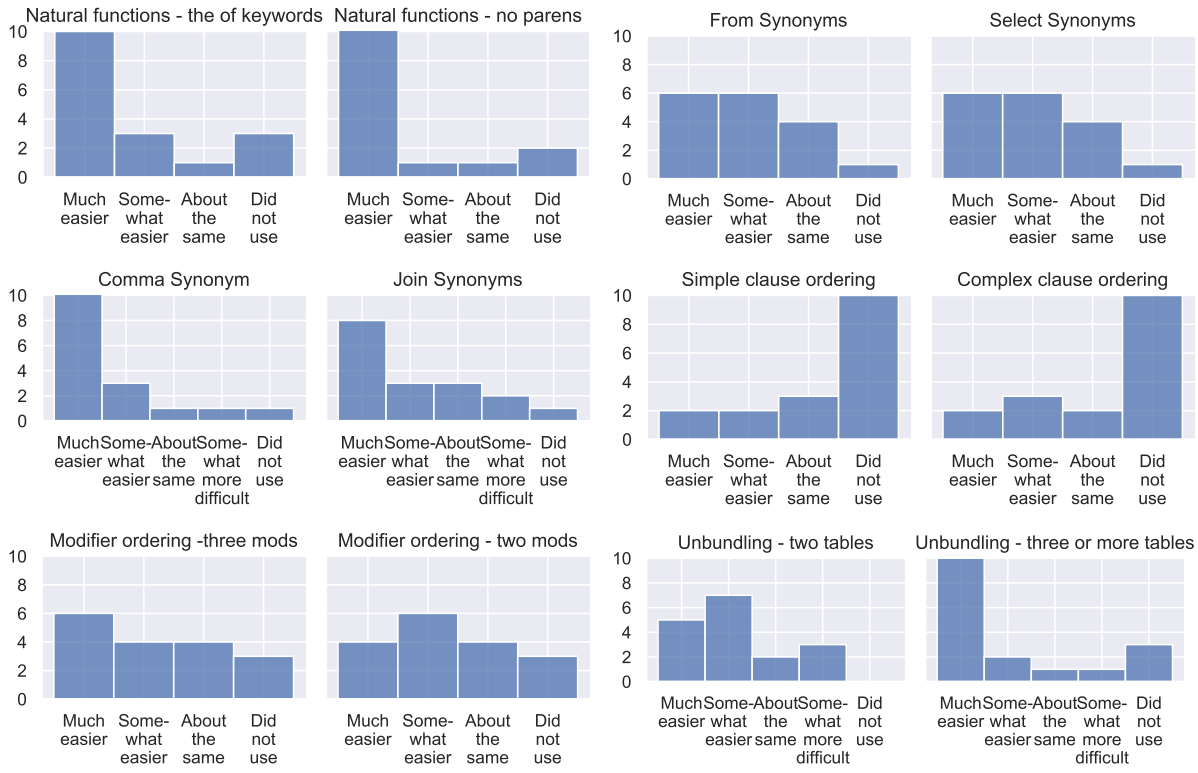


Figure 2.20. SpeakQL feature usefulness compared to SQL.

We also asked participants who used a feature to rate its usefulness. Figure 2.20 shows the results for all features. These results reveal interesting nuances on the low-popularity synonym

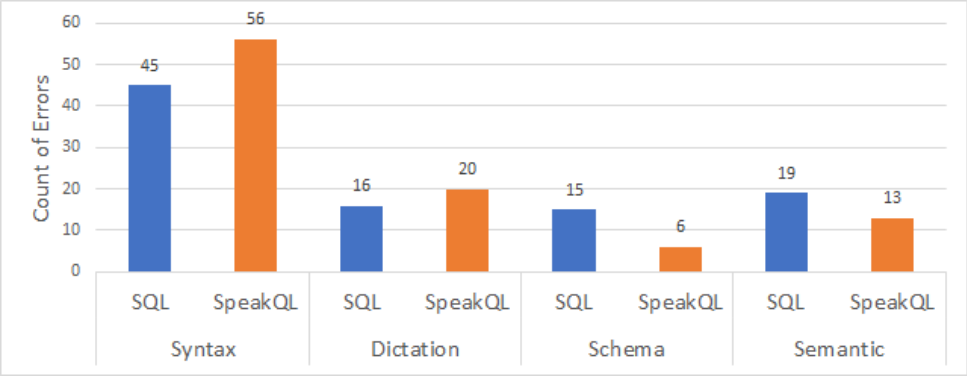


Figure 2.21. Attempt error categories by dialect

feature: participants preferred punctuation and join synonyms but they did not care for SELECT or FROM synonyms. The other ratings are consistent with the feature usage observations.

2.6.3 Failure Analysis

Failure Categories

We analyzed the transcripts and audio files of incorrect attempts in order to organize failures into the following categories: 1) Syntax, where the participant's query had at least one syntax error (e.g. missing a delimiter or parenthesis, using an invalid keyword, or omitting a clause such as group by); 2) Dictation, where the participant appeared to lose their train of though and terminated their attempt voluntarily; 3) Schema, where the participant referenced a table or column that either did not exist in the schema or was not valid in the context of the clause in which it appeared (e.g. referencing a table instead of a column); 4) Semantic, where the participant's query was syntactically correct but did not return the correct answer.

Observations

Figure 2.21 shows the breakdown of error types by dialect and failure category. The total number of failures analyzed was 190, 95 SpeakQL and 95 SQL. All failure categories occured for both dialects; but syntax and dictation failures were more commmon for SpeakQL than SQL. Conversely, schema and semantic failures were more common for SQL than SpeakQL. The higher rate of syntax and dictation errors for SpeakQL is likely due to the fact that SpeakQL is a

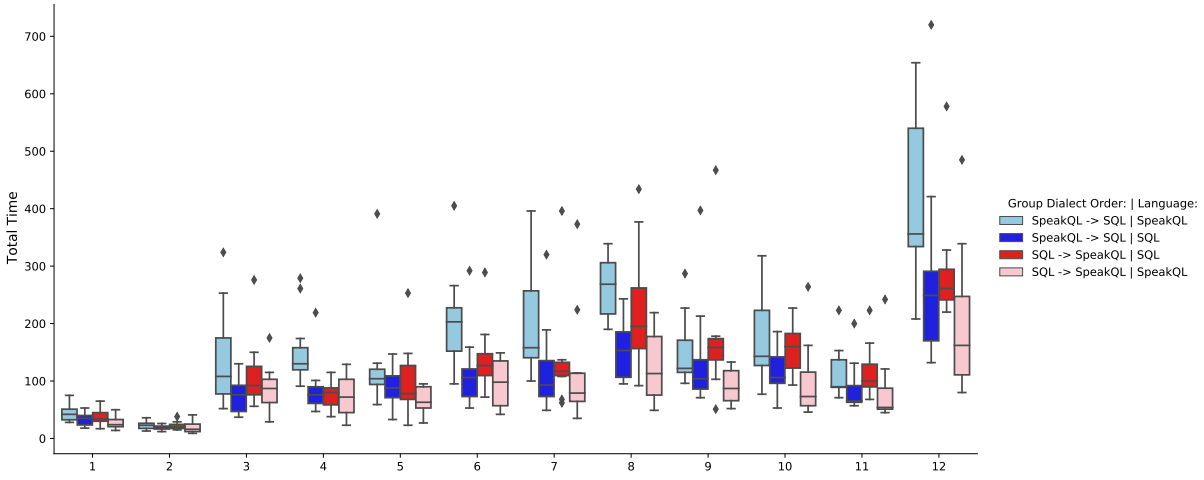


Figure 2.22. Dialect performance measured by total time and separated by group dialect order.

new language and participants were not as familiar with it, though we did observe a much higher rate of SpeakQL syntax errors where participants used incorrect keywords or invalid synonyms. The lower rate of schema and semantic error hints at the possibility that some aspect of SpeakQL may have made it easier for participants to reason about the schema and the query prompt.

2.6.4 Learning Effect Impacts on Results

During the study design process, we elected to use a within-subjects design in order to reduce the number of participants needed to achieve statistical significance. The drawback of this approach is that it introduces a learning effect bias. We attempted to mitigate this effect through counterbalancing the dialect order between participants so that half of our participants would have SpeakQL as their first dialect and half would have SQL as their first dialect.

Despite our learning effect mitigation strategy, we observed evidence of asymmetric learning effects between the two dialects. Specifically, members of group 2 (those who used SQL first) exhibited higher improvement using the second dialect relative to their first-dialect performance than members of group 1 (those who used SpeakQL first). Figures 2.22 and 2.23 replicate figures 2.17a and 2.17b but split by participant group and provide an impression of the assymetry between groups.

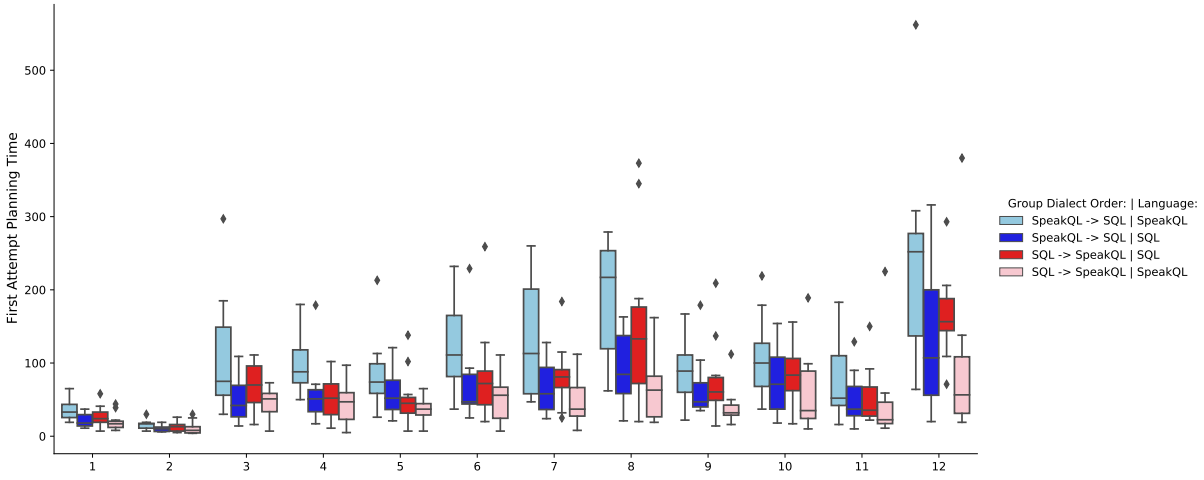


Figure 2.23. Dialect performance measured by first attempt planning time and separated by group dialect order.

Post-hoc analysis of participant data, comparing only first dialect performance between groups, provides an approximate view of the results of what would have been a between-subjects study design, though it is important to note that the study protocol was setup to be within-subjects. The results of this analysis are not necessarily a valid comparison of the two groups because within-subjects bias reducing design techniques such as participant stratification based on skill levels or other attributes were not implemented.

With the limitations of the post-hoc between-subjects approximation in mind, we observe that the median first attempt planning time and total attempt time for both simple and complex query types were lower for SQL than for SpeakQL (p value < 0.05 for all observations). There were no apparent advantages for either dialect in terms of number of attempts. The conclusion we draw from this observation is that should we pursue a more robust between-subjects study design, we would increase the amount of practice and training time for the SpeakQL dialect to help participants become more comfortable with SpeakQL syntax prior to performing measured attempts. We also consider that the naturalness of the SpeakQL syntax is not inherrently easy-to-learn and use, and thus does not necessarily translate to an immediate performance advantage over SQL for users who are already SQL-savvy.

Table 2.2. Thematic Category and Code Frequencies

	# Participants
– Positive –	**13**
General Positive Impressions	8
Positive Unbundling Impressions	7
Positive Natural Function Impressions	4
Positive Ordering Impressions	1
– Negative –	**9**
Syntax Difficulty	6
Unfamiliar Language, Needs Practice	7
Faux Natural Language	5
– Improvement Ideas –	**5**
Minimize Punctuation	4
Syntax Improvements	3

2.6.5 Qualitative Survey Feedback

Thematic Analysis

We categorized the survey feedback and sentiment into three thematic categories: positive, negative, and improvement suggestion. 13 participants provided at least one positive feedback. 9 provided at least one negative feedback. 5 provided at least one improvement suggestion. Table 2.2 lists the number of participants who gave each type of feedback, including the key content within each category.

Positive Comments

8 participants made positive comments that were not feature-specific. These included general impressions of their experience using SpeakQL, a comment that using SpeakQL was easier than using SQL for dictation, and 4 comments that SpeakQL had a natural feel and was easy to use. We present some quote verbatim.

"SpeakQL definitely makes dictation of queries more natural without worrying a lot about the syntax (which would include the orders) and even the parentheses." - Participant 10.

What I like the most is that it is almost like thinking out loud. You just think about what you want to do, and say the query, which makes it way more convenient. - Participant 16.

Unbundling received the most feature-specific positive feedback, from 7 participants. They reported that unbundling required less planning time, feels faster, enables focus on one table at a time, makes complex queries easier, and is generally easy or useful.

"Since I did not have to worry about ambiguous column names, I could write the bundles separately faster and join them all later on." - Participant 19.

4 participants reported that natural functions were useful and easier to use than speaking SQL functions with parentheses. One participant said the modifier reordering was useful.

Negative Comments

The most common ones related to syntax difficulty, with 9 participants reporting dislike of different aspects of SpeakQL syntax. 4 participants reported difficulty with unbundling for complex queries. 3 reported that using the unbundling join syntax or non-unbundled queries that used join synonyms was difficult. One participant said that the GROUP BY AUTOMATICALLY feature was not intuitive and felt less natural. 5 participants had negative comments on the naturalness of some SpeakQL features. One said that SpeakQL actually gave them a false impression of naturalness that made it difficult to discern when a natural language-like statement was a valid SpeakQL query. Some participants specifically noted that there were too many synonyms. They said they were unsure of how expressive SpeakQL was and that the language had too much flexibility, with synonyms being unbalanced between expression types, or that it was too nuanced.

"[I] Was afraid of saying the incorrect synonym. Do we really need all these synonyms? How many do we need? Too many synonyms might create the perception of natural language which will cause them to create incorrect queries." - Participant 1.

7 participants said that they did not have enough time to gain enough familiarity with all the SpeakQL features and wanted more practice.

"Though complex query bundling was easier, it takes a lot of time to go from creating individual table queries, and then joins and then grouping them, and so because I was querying

the columns from all the tables together first and then making all the joins in one go, I felt it a little inconvenient. However, I missed the fact that I can simply query 2 tables and join and then go to the next table would have been easier." - Participant 2.

Improvement Suggestions

Most comments focused on simple syntax improvements, including allowing "THE" before TABLE (e.g., *FROM THE TABLE building GET buildingname*), making generation of the GROUP BY expression fully automatic (i.e., without needing to speak *GROUP BY AUTOMATICALLY*), and adding *RETRIEVE* as an additional SELECT synonym. 4 participants also said they disliked dictating special characters such as quotes, commas, and parentheses and suggested a reduction of the need to speak these symbols, especially quotes.

2.6.6 Discussion of Results and Implications

Quantitative analysis of the within-subjects study revealed no statistically significant differences in planning time, number of attempts, or number of errors between the two dialects. Our results are inconclusive in this regard, and we cannot conclude that SpeakQL is faster to plan than SQL. However, we do find several encouraging pieces of evidence that SpeakQL, despite being a new and slightly more verbose dialect of SQL, takes comparable time to dictate but with a higher ease of use as per self-reported feedback.

Participant feedback on the survey indicates that more than half of participants seemed to have a positive experience using SpeakQL, with some indicating that using SpeakQL for dication was easier than using SQL. We find this interesting considering that SpeakQL performance was not measurably better than SQL in terms of time and number of attempts. Our post-hoc failure analysis provides some clues as to what may be driving this sentiment–that is, it appears that participants may have had fewer challenges reasoning about the schema and forming semantically correct queries when using SpeakQL.

Synonyms. Although the simplest feature, they were only the third-most popular feature

in the actual usage data. They also attracted the most negative feedback, with the crux being they caused more uncertainty about the valid keywords. We observed that 12 of the SpeakQL attempt failures were due to incorrect synonym usage. We plan to drop some synonyms in future implementations of SpeakQL.

Clause reordering. This ended up the most non-used feature, mainly because participants had high enouguh familiarity with SQL to not need such reordering. But we plan to retain this feature as it did not elicit negative feedback.

Natural functions. This was the most popular feature and universally liked by participants. We plan to expand this feature to reduce the need to speak other special characters such as quotes around string literals in predicates.

Unbundling. This was the second most popular feature and it received majority positive feedback. Many participants were particularly enthusiastic about how unbundling enabled them to approach query formulation in a "stream of thought" manner that made speaking queries easier. We plan to retain this feature as is.

Syntax vs. Structure. A lack of significant performance differences between SQL and SpeakQL syntax leads us to believe that the structure of the query is more important than the syntax. That is, that participants spent more time and effort thinking about which elements of the schema belonged in the query and less time thinking about the syntax of the query. The naturalness of the syntax does not appear to be a significant factor in improving measurable outcomes of the query dictation process, though many participants indicated that they found the syntax easier to use than SQL.

We believe that pursuing modifications to the SpeakQL dialect based on user study results may still be worthwhile. However, given the recent rapid progress of NL-to-SQL capabilities, we also condede that perhaps the most important factor in improving the ease of use of spoken querying is to improve the NL-to-SQL capabilities of existing NLIs. The SpeakQL dialect may benefit some aspects if these workflows, perhaps as a post-dictation error correction dialect within a stateful query system or as a prompt engineering language tool for NL-to-SQL systems.

Stateful Dialogue for Query Correction Attempt failures for SpeakQL queries were more likely to be syntax- or dictation-based (Figure 2.21). In SpeakQL's current implementation, such errors require the user to re-state the entire query from the beginning. Many of these errors would not require a full re-dictation if the user could simply correct the error in the query through a simple edit. Full implementation of a stateful system was outside of the scope of this paper's evaluation of a natural syntax. However, we believe dialogue system that provides verbal and/or visual error feedback and suggested corrections for a user to accept or modify would be a useful addition to the SpeakQL system in future iterations.

Within-Subjects Study Limitations The learning effect resulting from the within-subjects study limited our ability to draw conclusions about the relative performance of SpeakQL and SQL. Given our post-hoc analysis of an approximation of a between-subjects study, we consider that the naturalness of the SpeakQL syntax is not inherrently easy-to-learn and use, and thus does not necessarily translate to an immediate performance advantage over SQL for users who are already SQL-savvy. Should we pursue a more robust between-subjects study design to negate learning effects, we would likely increase the amount of practice and training time for the SpeakQL dialect to help participants become more comfortable with SpeakQL syntax.

2.7 Conclusions and Future Work

Motivated by the growing success of ASR-based interactions, this work considers the usefulness of a more natural spoken structured querying for databases. We design and evaluate a prototype dialect of SQL we call SpeakQL that we believe represents a potential solution for improving speech-based access that preserves correct-by-constructions guarantees of SQL. Our user study suggests the utility of some of our features, while also offering avenues for refinement of other features. As for future work, we consider a pivot toward natural language-based interfaces for databases, and envision integrating elements of the SpeakQL dialect into a stateful system to allow users to conversationally clarify and refine their query as part of the error correction and

intent clarification process.

Chapter 3

SNAILS: Schema Naming Assessments for Improved LLM-based SQL Inference

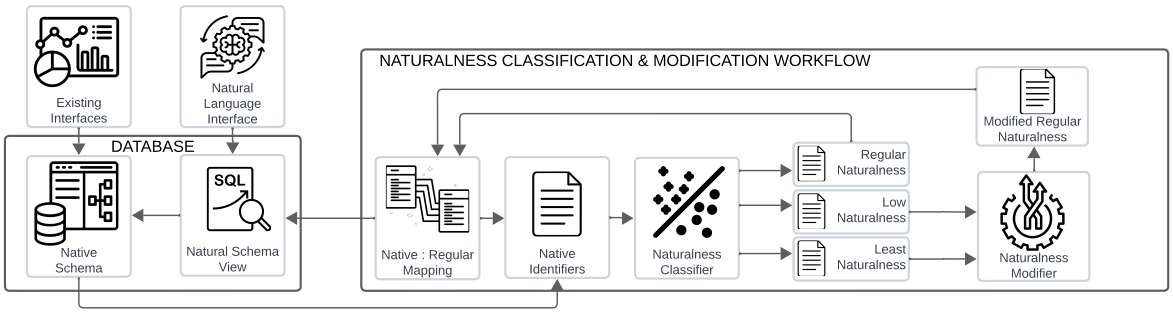


Figure 3.1. Databases with poorly named, or less natural, schema identifiers perform poorly in LLM-based NL-to-SQL interfaces, and this project exposes the need for more natural schemas. We offer approaches and artifacts, including a naturalness classification and modification workflow, that can aid in the naturalness assessment and modification processes required to create a performance-enhancing natural view. In this way, the native schema remains as-is so that existing tools can continue talking to it without modification, while an LLM-based NLI can be integrated into the existing stack via a natural view.

3.1 Introduction

Natural language-to-SQL (NL-to-SQL) query generation capability has been revolutionized by foundational large language models (LLMs) [71, 92, 106]. This has made the integration of LLM-based query tools into relational database workflows more viable, with both established DBMS vendors and startups beginning to offer commercial NL-to-SQL interfaces. However, challenges in the NL-to-SQL space remain that can degrade the effectiveness of an LLM-enabled data retrieval workflow in real-world databases [25]. Principal among such challenges is *schema linking*, which is the association of entities in NL utterances to elements in the database schema.

While much work has studied making LLMs larger or more sophisticated, a more basic issue often underlies this challenge: lexical mismatches between natural language and poorly-named tables and columns in a schema. Intuitively, schema elements that are "better named" could raise the accuracy of schema linking within the NL-to-SQL setup. In this paper, we unpack and dive deeper into this intuition to study how exactly the "naturalness" of schema elements matters for NL-to-SQL by instituting a new benchmark and performing extensive empirical analysis using that. One might ask: *Why bother formalizing a concept that seems obvious and*

intuitive? We believe this is important for 2 reasons. First, without a more formalized–or at least automated way–to define, verify, and compare "naturalness" researchers and practitioners alike will be forced to grapple with ad hoc and inconsistent approaches. In turn, this can lead to confounded conclusions by researchers on how different LLMs behave on different schemas and mislead practitioners comparing different NLIs. This points to the need for a new benchmark labeled dataset for this problem.

Second, practitioners need a way to efficiently and accurately operationalize any insights about the impact of naturalness on their schema elements for LLM-based NLIs. This points to the need for a systematic evaluation of how naturalness affects different databases, queries, and LLMs used for NL-to-SQL.

Our Focus

In this paper, we take the first steps toward deeper understanding on this seemingly obvious, but hitherto underexplored and important, relationship between schema identifier naturalness and LLM-based NL-to-SQL. Specifically, we ask the following three interconnected questions. (1) How do we quantify "naturalness" of schema identifiers? (2) Does it really impact schema linking accuracy in LLM-based NL-to-SQL and if so, by how much? (3) How does that impact vary by complexity of the database and query, as well as across different popular LLMs?

To answer the above questions, we create a novel integrated benchmark suite we call SNAILS with new collections of real-world databases and query pairs, a new labeled dataset of schema identifiers, a set of evaluation metrics, and LLM prompting and other AI artifacts.

3.1.1 Preliminaries and Setup

LLM-based NL-to-SQL

The most obvious way to seek LLM performance improvements would be by increasing the power of the language models themselves. But the cost of training and deploying LLMs continues to increase in concert with their complexities. Additionally, many practitioners seek "plug and play" solutions by employing already-available LLMs. Model training and finetuning

impose access barriers that may render such a pursuit untenable for organizations that use databases but lack the requisite talent such as data science and machine learning expertise.

The practice of prompt engineering can also help improve NL-to-SQL performance, though dealing with schema complexity and schema representations in LLM prompting is an ongoing challenge in enterprise-level NL-to-SQL applications [25]. The majority of leading submissions on the popular Spider NL-to-SQL benchmark leaderboard are LLM-based solutions [28, 84, 21] that employ a variety of prompting strategies, some of which require multiple successive API requests containing schema context and instructions. These approaches can be costly and unintuitive for NLIDB end users, and can incur excessive costs and overhead when deployed at scale.

A complementary line of work on realistic NL-to-SQL benchmarking uses structural schema modification such as normalization, flattening, and replacement to evaluate effects on LLM performance. Making such structural changes to target schemas challenges model robustness and increases error rates in NL-to-SQL performance [59], and this recent work indicates that schema design is a viable target for LLM-based NL-to-SQL accuracy improvements.

Schema Linking

Schema linking remains as a persistent challenge for LLMs. With the availability of capable LLMs that consistently generate valid SQL statements, a larger proportion of NL-to-SQL generation errors are now associated with incorrect or ambiguous database identifier selection as opposed to incorrect syntax [101]. Schema linking performance has been improved using lexical matching heuristics [121, 33], joint relationally aware embeddings with attention [113, 12], the use of pre-trained language models to perform schema probing [115], and multimodel pipelines with ML models for pruning schema knowledge [51]. Some NL-to-SQL methods address schema linking challenges by adding additional context such as sample values or metadata [84] to schema knowledge representations. These methods can improve performance in some cases [67], and can be useful for schemas with obscurely-named tables and columns, though they do so at the

cost of much larger schema knowledge representations.

Schema linking still often fails, even with the most capable LLMs due to poorly-aligned schema identifier names with natural language question contents, that could be due to the use of synonyms or the obscurity of a database identifier. In the latter case, it can be challenging for even a sophisticated linking solution to match natural language words to schema elements that yield minimal semantic meaning.

Schema Naming Conventions

The majority of database schema naming best practices originate from *practitioners* and are generally published as software documentation, organization policies, tutorials, etc. We find that there is a gap in database and data integration *academic* literature evaluating schema identifier naming practices for any purpose. While the semantics of schema identifiers may not have been considered as a necessary subject of database research in the past, the increasing integration of natural language interfaces to databases has elevated its importance.

Naming conventions for database schema identifiers vary by organization, database vendor, application, and purpose. A web search for database table and column naming guidelines yields multiple resources ranging from blog posts [16], StackOverflow responses [98], DBMS vendor documentation [77], and tutorials [31]. Poor schema identifier naming practices is considered a database code smell [96] where meaningless identifier names should be avoided. Generally, the most consistent best practices include selecting descriptive and concise names that contain only commonly-understood abbreviations and acronyms, though some conventions suggest the use of abbreviated prefix and suffix modifiers that describe application associations, or entity purpose [78].

In our research, we identified several databases containing schemas with varying levels of human-readability and understandability (what we will call naturalness) which suggests that there can be a tendency for database schema designers to choose shorter and less descriptive identifier naming conventions. As we will see, such naming shortcuts can negatively affect NL-to-SQL

performance.

3.1.2 Our Benchmark Artifacts and Analyses

Given the above context of our benchmarking setup, we now explain the new artifacts in SNAILS, followed by a summary of our empirical analysis.

Artifact 1: Real-World Database Schemas

The SNAILS benchmark contains several new *real-world database schemas* that are not part of existing NL-to-SQL benchmarks (Artifact 1). Our focus on schema naming motivates the creation of a new novel benchmark dataset, because existing benchmark naturalness levels are higher than those of many real-world schemas, and other real-world schema collections including SchemaPile [20] lack the necessary database instances to enable NL-to-SQL evaluation. In our analysis of these real-world schemas, we discover that identifier naming variances generally appear in the form of abbreviations and expansions; we refer to these variances as identifier *naturalness*.

Artifact 2: Identifier Naturalness Classifications

Our analysis reveals that naturalness can be formalized categorically with the help of finetuned language models and feature engineering. We then hand-label the schema identifiers, with some ML assistance, to classify their naturalness level and produce a new golden labeled dataset. We classify identifiers into one of 3 naturalness levels (Regular, Low, and Least) (Artifact 2). This dataset, consisting of over 17,000 labeled identifiers, serves as the training data for the naturalness classifiers described next.

Artifact 3: Naturalness Classifiers

We experiment with various classification approaches, and make available the models trained to classify the naturalness of a database schema identifier (Artifact 3).

Artifact 4: Naturalness-Modified Identifiers

To better understand the effect of schema identifier naturalness, and to enable within-database experiments, we create alternate versions of each real-world schema identifier at each naturalness level (Artifact 4). This dataset serves two purposes: 1) Training data for ML-based naturalness modifiers, and 2) Generation of schemas with varying naturalness levels to analyze the impact of naturalness on NL-to-SQL performance. We modify the identifiers with the assistance of LLM prompting, finetuned models, and database metadata.

Artifact 5: Naturalness Modifier

We offer an in-context learning-based prompting strategy for identifier naturalness reduction (or abbreviation). We also provide an identifier naturalness increaser (or expander) that leverages retrieval augmented generation, interactive few-shot example building, and database metadata parsing methods to streamline the database naturalness improvement process.

Artifact 6: NL-to-SQL Question Query Pairs

The SNAILS benchmark contains 503 NL question-SQL query pairs which we use for NL-to-SQL performance analysis of 4 LLMs. We created this new collection as another hand-labeled golden dataset without the use of AI-based workflows (Artifact 6).

Experimental Evaluation

Using the SNAILS benchmark artifacts, we analyze and experiment with the effects of schema identifier naturalness on LLM NL-to-SQL performance. We select 5 publicly-available LLMs: OpenAI's GPT-3.5, GPT-4o, a finetuned variant of Meta's Code-Llama, Google's newest Gemini 1.5, and CodeS finetuned for NL-to-SQL. We evaluate them using both execution result set matching and a novel identifier set comparison approach that pinpoints schema linking performance.

In this paper we focus primarily on a simple zero-shot prompting of the LLM for our experiments. We recognize that this may not be the best for overall execution accuracy, but it helps us isolate the impact of schema identifier naturalness in this first work on this problem. As such,

more complex workflows will create confounding effects while not necessarily providing more insights into schema linking performance. However, for completeness sake, we also compare two illustrative complex workflows: DIN SQL for task-specific prompt chaining [84], and CodeS [52] for NL-to-SQL finetuning.

We find that schema identifier naturalness by and large does have a meaningful effect on NL-to-SQL accuracy and schema linking performance. Specifically, identifier naturalness is moderately and positively correlated with both schema linking and execution accuracy. Identifiers of low naturalness yield lower performing NL-to-SQL inferences in terms of both schema linking (identifier recall) and execution accuracy. These findings have implications for practitioners who are either designing new databases intended for LLM-based applications, or seeking to augment existing RDBMSs with an LLM-based NL-to-SQL interface.

In summary, this paper makes the following contributions:

- We propose a novel measure of *naturalness* of a database schema identifier and demonstrate through extensive experiments that naturalness has a significant effect on LLM schema linking performance in the context of NL-to-SQL.
- We provide a hybrid LLM-generated and human-curated training dataset (Artifact 2) and language model (Artifact 3) for schema naturalness classification.
- We offer a new multi-domain NL-to-SQL evaluation benchmark collection consisting of 9 real-world relational databases (Artifact 1) and 503 unpublished NL-to-SQL query pairs (Artifact 6) that do not exist in any LLM training corpora.
- We create a novel labeled dataset of alternate naturalness levels that map the identifiers from Artifact 1 to hybrid LLM-human curated identifiers of different naturalness levels (Artifact 4), and methods for expanding and abbreviating identifiers to change their naturalness (Artifact 5).

Regular	Low	Least
airbag	AccountChk	AdCtTxIRWT
AdaptiveCruiseControl	IsueFrDate	COGM_Act
ModelYear	RecvAsst	DfltSlp
service_name	UsrQuery	FNDAbs
Research_Staff	ValueOfT	CSI22

Table 3.1. Example identifiers and their naturalness levels, from the SNAILS naturalness labeled dataset
(Artifact 2).

- We conduct an extensive empirical analysis of the performance of 5 popular foundational LLMs over our benchmark using a novel schema linking metric for NL-to-SQL.
- We propose a realistic workflow that enables the preservation of existing database integrations while offering LLM-based NLIs a natural view of a target schema.

3.2 Background

3.3 Schema Identifier Naturalness

Intuitively, naturalness can be thought of as the degree to which a phrase, or word, resembles natural language. Naturalness is a concept and target of research in field of controlled natural languages [48], where controlled language syntax is evaluated in terms of naturalness levels. Recent NL-to-SQL research also defines and measures naturalness [59] for the purpose of evaluating the naturalness of natural language question utterances, but avoids measuring the naturalness of schema elements.

To the best of our knowledge, no prior attempts have been made to definitively measure the naturalness of a database schema's identifiers. In order to achieve this goal, we propose a three-category naturalness classification scheme in order to measure the effects of naturalness on NL-to-SQL performance.

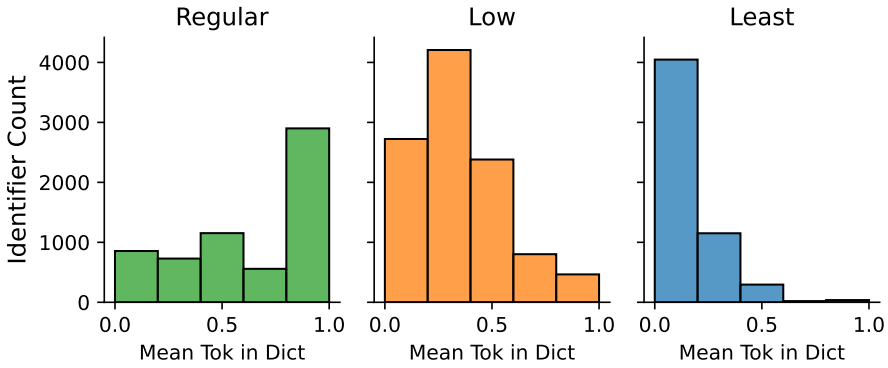


Figure 3.2. *Mean Token in Dictionary*, the proportion of tokens in an identifier that match a word in an English dictionary, generally aligns with the SNAILS 3-class naturalness categorization approach.

3.3.1 Naturalness Categories

As the first work on this topic of how schema identifier naturalness affects LLMs, we seek to define a preliminary metric–one that is consistent and descriptive enough to differentiate between naturalness levels and to measure their effects.

To gain insights into naturalness-related trends in the SNAILS datasets, we create a *mean token-in-dictionary* measurement that describes the proportion of tokens in an identifier that exactly match a word in a comprehensive English word list. Figure 3.2 reveals differences between each naturalness category where Least naturalness identifiers contain fewer in-dictionary tokens, and Regular naturalness identifiers are more likely to consist of in-dictionary tokens. This distribution suggests that because the bulk of the training corpora of LLMs is human-generated natural language text, what humans consider "natural" for such identifiers generally aligns with how LLMs react to them.

Examples of schema identifiers and their naturalness categories are displayed in Table 3.1. We define these categories with the underlying assumption that the identifiers are named as some semantic representation of the data, and that naming-related problems of interest are related how an identifier is codified. That is, identifiers are assumed to not be random character sequences or random words that do not correspond to the content of the database entities they represent. With

this assumption in mind, we categorize naturalness into 3 discrete levels as follows:

- Regular: The identifier contains complete English words with no abbreviations or acronyms, or contains only acronyms in common usage (e.g., ID or GPS).
- Low: The identifier contains abbreviated English words and less common acronyms that are usually recognizable by non-domain experts (e.g., UTM or CPI). The meaning of the identifier can be inferred without consulting external documentation.
- Least: The identifier's meaning cannot be inferred by non-experts due to indecipherable acronyms or abbreviations, and external metadata or other documentation must be consulted in order to determine its purpose.

While we recognize that naturalness can also be treated as a continuous spectrum, between the choices of continuous scoring and discrete categories, we select the latter as an initial approach to naturalness evaluation. The primary factors underlying this choice are the level of effort required to conduct human-based scoring of a large set of database identifiers, and the difficulty of consistently scoring naturalness on a continuous range over a large set of data. Therefore, we use an intuitive and easily-verifiable discrete 3-class taxonomy in the first work on this topic.

3.3.2 Naturalness Classification

To consider naturalness as a factor in NL-to-SQL performance, we derive naturalness scores of the target schemas' identifiers. We use this score to consider effects of individual identifier naturalness, schema naturalness, and query identifier naturalness. Because manual naturalness classification can be a time consuming task for large schemas, we automate the process by training a machine learning-based classifier. This effort is beneficial in multiple situations. First, it can ease some manual effort of the labeling process and make the process of scaling to more databases in the future less labor intensive. Second, it can help practitioners efficiently and consistently evaluate the naturalness of their own database schema identifiers prior to NLI integration.

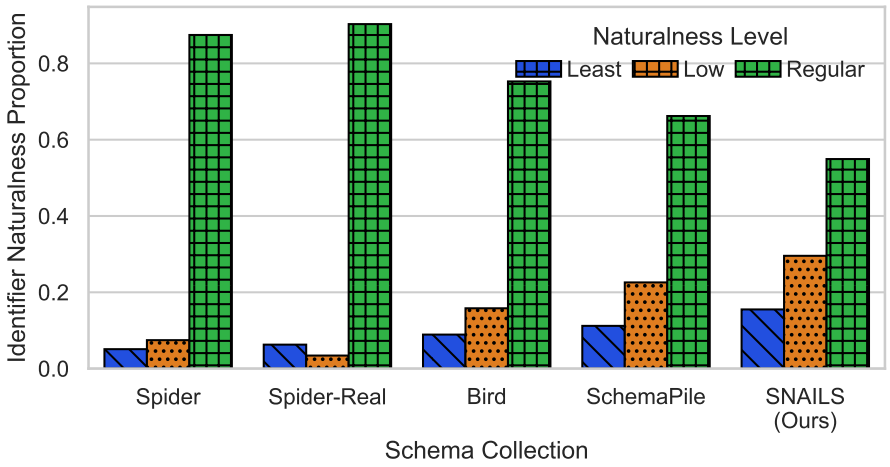


Figure 3.3. Comparison of the SNAILS database collection (Artifact 1) described in Section 3.5.1 to other real-world and benchmark schema collections. SNAILS naturalness proportions are generally biased toward less natural identifiers and is more consistent with the real-world SchemaPile collection than other existing benchmarks including Spider and Spider Realistic.

To train a classifier to perform identifier naturalness scoring, we employ the 3-class set of naturalness categories described in Section 3.3.1, and a list of database identifiers drawn from the SNAILS real-world database schemas (Artifact 1). We categorize the naturalness of each identifier to generate the SNAILS *identifier naturalness classification* labeled data (Artifact 2) which we use for ML-based naturalness classifier training, evaluation and testing.

We evaluate multiple classification approaches including heuristic-based word matching, few-shot LLM prompting with GPT-3.5 and GPT-4, and LLM finetuning. The GPT-4 few-shot approach achieves 74 percent accuracy and an f1 score of 0.77. We experiment with multiple finetuning collections, first using a hand-labeled collection of 1,648 naturalness classifications and then leveraging the initial classifier along with weak supervision to generate a larger collection of 17,226 labeled identifiers. Finetuning using the second collection outperforms all few-shot approaches, with the two best-performing classifiers fine-tuned GPT 3.5 and BERT-based CANINE [17] models performing at 89 percent accuracy, and 0.89 f1 score.

Figure 3.3 provides a visual comparison between the SNAILS schema collection and common NL-to-SQL benchmarks including Spider, Spider Realistic, and BIRD. Additionally, we

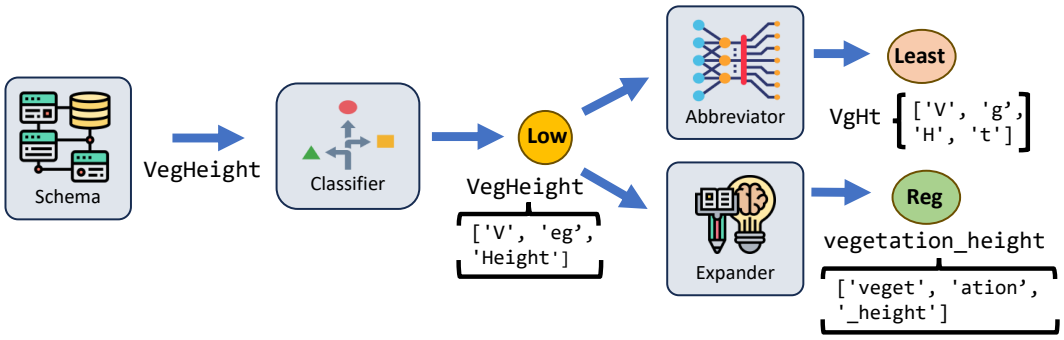


Figure 3.4. Schema identifiers are classified (Artifact 2) and modified to increase or decrease naturalness as appropriate. Modified identifiers comprise the schema crosswalks used for schema modification during NL-to-SQL experimentation (Artifact 3).

compare the SNAILS collection to the real-world SchemaPile collection and find that SNAILS collection proportions generally align to SchemaPile naturalness, more so than other existing benchmarks, which creates a more realistic and challenging benchmark in terms of schema naturalness.

To better understand the magnitude of naming practices in real-world schemas, we use the CANINE-based classifier to classify the naturalness of the SchemaPile collection: a large volume of real-world database schemas [20] that contains over 22,000 database schemas, 198,000 tables, and 1 million columns. We find that in over 7,500 schemas (32 percent of the collection) Least natural identifiers make up at least 10 percent of the schema identifier names. Additionally, over 5,000 schemas register a combined naturalness of 0.7 or below–an indicator that the schema contains a high level of Low and Least naturalness identifiers. We examined the naturalness category distribution for these 5,000 schemas, and found that for this subset of schemas Low and Least naturalness identifiers outnumber Regular naturalness identifiers. These findings reinforce the importance of the naturalness problem by confirming that, although a reasonable majority of schemas are already natural, there still exist many schemas with lower naturalness levels in the real-world–enough to motivate the formalization of schema naming quality measures.

3.3.3 Identifier Schema Naturalness Mapping

In addition to measuring the effects of identifier naturalness in existing schemas, we also seek to evaluate the effects of modifying schema naturalness. For this purpose, we create Artifact 4, naturalness-modified identifiers. This artifact enables schema modification during prompt generation and query inference, which provides a within-schema assessment of naturalness level effects on NL-to-SQL accuracy.

Identifier Mapping

In addition to the ground truth, or Native, naturalness of the 9 schemas in the SNAILS real-world database collection, the naturalness-modified identifier collection contains 3 additional sets of identifiers: Regular, Low, and Least. That is, each native identifier is mapped to 2 additional, semantically equivalent, identifiers of higher or lower naturalness, and mapped to itself for its own naturalness level (i.e., we do not generate new identifiers of the same naturalness as its native form).

Figure 3.4 provides a visual example of the Native identifier *VegHeight* which is classified as Low naturalness. With this naturalness classification as a starting point, we abbreviate it further to generate a corresponding Least naturalness identifier *VgHt*. We also expand it to generate the corresponding Regular naturalness version *vegetation_height*. We map the Native *VegHeight* identifier to itself in the Low naturalness category.

Naturalness Modification

For *more natural to less natural* modifications (the abbreviator in Figure 3.4), we employ in-context learning (few-shot) prompt strategies with GPT-3.5 turbo to generate naturalness-modified identifiers (e.g., Regular to Low, Low to Least, and Regular to Least). We favor this approach over model finetuning, as simple instructions to abbreviate the identifier coupled with several examples prove more effective and less prone to poor results (e.g., presence of unwanted characters in the modified identifier).

Automating the reverse *less natural to more natural* naturalness modification (the expander

in Figure 3.4) requires additional context and external knowledge from data description sources. Though a recent project describes a promising identifier expansion strategy [123] without external knowledge, it requires finetuning over a large dataset, and is likely susceptible to overfitting; therefore we opt for our own approach that incorporates the use of an LLM augmented with schema metadata lookup capability. To accomplish this, we create a Python program with GPT interaction that takes as input metadata describing a schema's native identifiers, and outputs an identifier with regular naturalness.

3.3.4 Heuristics-based scoring

Prior to experimenting with ML classifiers, we used a set of heuristics to score the naturalness of each identifier. Comparisons between the heuristics-based scoring approach and ML classification reveals that ML is superior in terms of recall, precision, and F1. We include a description of the heuristics here for completeness, but exclude them from the main body of the report.

- Vectorize an English word vocabulary as frequency counts of letters in the word.
- With a given database identifier, vectorize the identifier as frequency counts of letters in the identifier and downsample to the English word vocabulary to words that have a superset of the letters in the identifier.
- Further downsample the candidate words to words where the letters appear in the same order as the words in the identifier.
- For each word in the downsampled vocabulary, compute the Levenshtein distance between the word and the identifier. This number is called the edit distance.
- For each word, count the number of possible word candidates within 1 and 2 Levenshtein distance from the word. We call this number candidate ambiguity.

- The distribution of candidate ambiguity across our vocabulary is highly skewed, so we take the log of the candidate ambiguity to normalize the distribution.
- We then calculate the naturalness score as the weighted mean of the inverse of the edit distance and the inverse of the log of the candidate ambiguity. This yields values ranging from 0 to 1, where 0 is least natural and 1 is most natural.

3.3.5 Dataset Naturalness Classifications

Identifier naturalness within each dataset is categorized using the $N1$ (Regular), $N2$ (Low), and $N3$ (Least) categories. Naturalness of table and column identifiers are cataloged both separately, and in consolidated form (i.e. tables and columns together). Additionally, we calculate a combined naturalness score for consolidated identifiers using category weights.

CombinedNaturalness = 1.0*Regular + 0.5*Low + 0.0*Least

(3.1)

where $Regular$, Low, and $Least$ are proportions of schema identifiers in each respective category within the total count of identifiers in the identifier's source schema.

3.3.6 Training Data Collections

For finetuning tasks, we train language models using database identifiers extracted from the schemas in the SNAILS real-world database collection. We begin with a human-classified collection (Collection 1); then we employ classifier models trained on Collection 1 to generate a larger set (Collection 2) of machine-classified and human-curated identifier classifications.

Collection 1 The full dataset contains 1,648 manually classified unique schema identifiers. The identifiers are hand labeled as one of 3 naturalness levels (Regular, Low, Least). We randomly

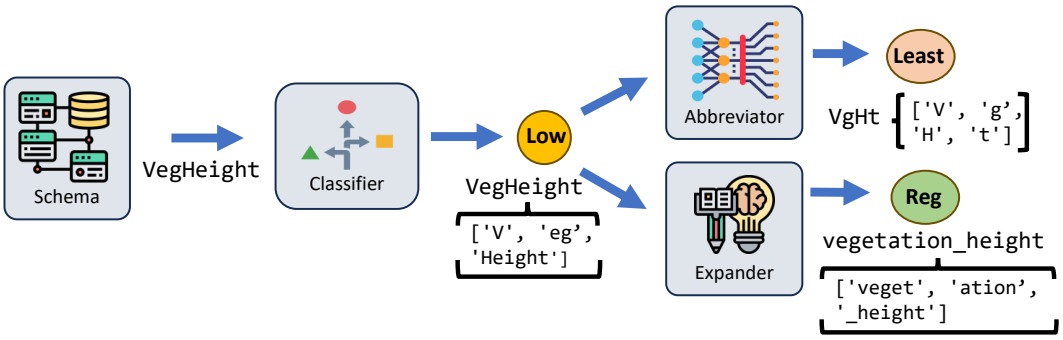


Figure 3.5. Schema identifiers in our benchmark dataset are classified into a naturalness category and modified to increase or decrease naturalness as appropriate. Modified identifiers comprise the schema crosswalks used for schema modification during experiment query inference.

divide the data into a training, validation, and test set. This resulted in a distribution of 959 identifiers used for training, 356 for validation, and 333 for testing.

Collection 2 The labeled data set contains 13,722 distinct column identifiers and 3,504 distinct table identifiers for a total size n = 17,226. We employ GPT's Davinci model finetuned on Collection 1 to generate the preliminary naturalness scores. The authors reviewed, and where necessary, modified the model-generated identifier scores to affirm the accuracy of the naturalness classifications. Of the 17,223 identifiers reviewed, 15,527 naturalness scores, or 90.1 percent, were accurately predicted by the Davinci-based model. We manually scored the incorrectly-predicted 1,696 identifiers. For model finetuning, we randomly split the resulting data into training (n = 10,327), validation (n = 3,457), and test (n = 3,457).

3.3.7 ML Classifier-based scoring

The use of pre-trained language models is a SoTA approach for classification problems [63], and we experiment with various attempts at model-based scoring, including few-shot learning via the GPT API, and finetuning several BERT-like language models on our dataset of database identifiers to create a second larger collection of identifier naturalness scores. Since the presence of acronyms and abbreviations is a significant determining factor of identifier naturalness, a primary consideration for our naturalness scoring task is the granularity of the tokenizer

Model	Accuracy	Precision	Recall	F1
GPT-3.5-FewShot	0.646	0.623	0.638	0.630
CANINE-Seq C1	0.719	0.699	0.727	0.712
GPT-4-FewShot	0.742	0.742	0.792	0.766
CANINE-Seq+TG C1	0.829	0.829	0.838	0.833
GPT-3.5-FineTune	0.899	0.878	0.877	0.878
GPT-3.5-FineTune+TG	0.896	0.896	0.897	0.896
CANINE-Seq+TG C2	0.896	0.896	0.898	0.897

Table 3.2. Performance comparison of different language models for classifying a database identifier's naturalness

output. For this reason, we use models that employ either character-level tokenization, word part tokenization, or byte pair tokenization techniques. We select 2 approaches: 1) Use of a foundational LLM in various capacities; and 2) Finetuning of a character-level token language model.

3.3.8 Character Tagging Feature

We include a pre-processing step that generates a sequence of special characters that correspond to the type of characters of the identifier to be classified. The sequence is then concatenated with the identifier and passed to the language models during training and inference. We refer to this approach as *character tagging*, and models employing tagging are labeled with TG in Table 3.2. Both GPT- and CANINE-based models exhibit improvement in F1 scores using character tagging.

There are intuitive structural differences between abbreviated words and their complete counterparts. Specifically, we observe that word abbreviations generally contain more consonants than vowels, as vowels seem more likely to be removed during abbreviation. We are unsure of

a language model's ability to make use of this observation, so we offer some assistance in the form of a pre-processing step that generates a sequence of special characters that correspond to the type of characters of the identifier to be classified. We concatenate the tag sequence to the identifier and pass it to the language models for training and inference. Special characters include:

- `^`: Vowels
- `+`: Consonants
- `#`: Numbers
- `$`: Special characters
- $*$: Any character not in the above categories

We refer to this approach as *character tagging*, and models employing tagging are labeled with TG in Table 3.2.

For example, the identifier `AuthorID_5` would be pre-processed as follows:

```
AuthorID_5 ^^++^+^+$#
```

Both GPT- and CANINE-based models exhibit improvement in F1 scores when using the character tagging feature.

3.3.9 GPT 3.5/4 Turbo Few-Shot-Based Scoring

We experiment with the effectiveness of few-shot prompting to classify identifier naturalness. We opt to provide one set of instructions followed by a series of 25 randomly selected human-validated examples of naturalness levels. We perform text replacement on the trailing row by replacing the _IDENTIFIER_ text with the identifier to be classified. This approach does not require model pre-training; but this convenience is paid for in terms of the number of tokens

in the prompt, and classifying a large schema with this method can incur rather high LLM usage costs.

```
The following is a list of database identifiers and
labels that indicate how closely they resemble
natural english words:
N1: most natural english words
N2: second most natural english words
(e.g. abbreviations or combinations of
natural words and acronyms)
N3: third most natural english words
(e.g. very short abbreviations with obscured
meaning or acronyms)

identifier: CASENO Label: N1
identifier: BENTHOS_TotalAreaSampled_m2 Label: N2
identifier: CAUSE3 Label: N1
identifier: MT_RIVPACS_2011_OTU Label: N3
identifier: ACTIVATE Label: N1
identifier: MotorcycleChassisTypeId Label: N1
identifier: First_Name Label: N1
identifier: IPCAREA_2ND Label: N2
identifier: INJNO Label: N2
identifier: tbl_MicroHabitat Label: N2
identifier: EMSGCSEYE Label: N3
identifier: HEADRESTDAM Label: N2
identifier: AutoPedestrianAlertingSound Label: N1
```

```
identifier: ModelTest Label: N1
identifier: tlu_topo_position Label: N2
identifier: Understory_Comp Label: N1
identifier: BAGDAMAGE Label: N1
identifier: HARNESSDESIGN Label: N1
identifier: Coord_Syst Label: N2
identifier: CINJSEV Label: N2
identifier: JKWGT12 Label: N3
identifier: _IDENTIFIER_ Label:
```

3.3.10 GPT Davinci Fine Tuning

We train the Davinci-based completion models using the OpenAI command line API. We generated models with character tagging, as well as models without tagging. Below is an excerpt from the tagging-based training data.

```
{"prompt":"ADDRESS ^+++^++ ->","completion":" N1"}
{"prompt":"AIS ^^+ ->","completion":" N3"}
{"prompt":"AISCODE ^^++^+^ ->","completion":" N3"}
{"prompt":"BACKBPILL +^++++^++ ->","completion":" N2"}
{"prompt":"ALIGNMENT ^+^+++^++ ->","completion":" N1"}
{"prompt":"ARRMEDICAL ^+++^+^+^+ ->","completion":" N2"}
```

Inference using charager tagged models requires appending the tag to the identifier in the same format as the training data.

3.3.11 CANINE Fine Tuning

CANINE [17] is a BERT-based language model that tokenizes inputs at the character level. We trained a sequence classification head using both generation 1 and generation 2 data

sets using a single NVIDIA GTX 1080 GPU. We employed the HuggingFace Transformers library, CUDA 12.1, and Torch 2.0.1 to fine tune the 'google/canine-s' model. Hyperparameter tuning was conducted using the optuna library, which resulted in the parameter settings:

- Optimizer: adamw hf
- Learning rate: 4.910828967396573e-05
- Per device training batch size: 24
- Per device evaluation batch size: 12
- Number of epochs: 15
- Weight decay: 0.04168784348465411

Models were trained both with, and without, the character tagging feature. We offer the evaluated models in our project repository. The *snails_naturalness_classifier.py* Python file contains the *CanineIdentifierClassifier* class. This class provides a simple *classify_identifier* method for using the CANINE model to classify identifier naturalness with or without character tagging.

3.3.12 Tokenizers

We examine the relationship between tokenization and naturalness by generating token counts, character counts, and a character-to-token ratio of each identifier. As expected, due to the unabbreviated nature of the identifiers, more natural identifiers have more characters (see Figure 3.6). Perhaps more surprising, token count is *not* very sensitive to naturalness levels, mainly due to the general behavior where more abbreviated identifiers will have character sequences not found in the LLM tokenizer's vocabulary. When a character sequence is not present in the vocabulary, the tokenizer will split the sequence into multiple subtokens.

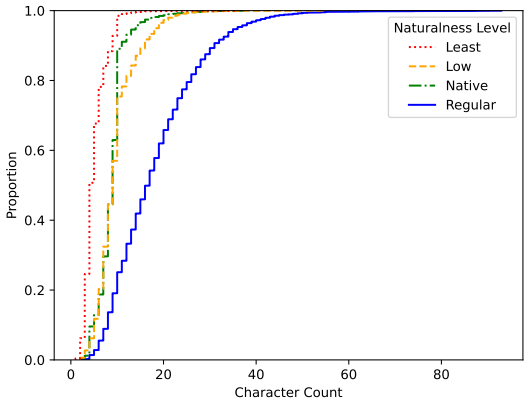


Figure 3.6. Cumulative distribution of schema identifier character counts by naturalness level. More natural (less abbreviated) identifiers logically have more characters.

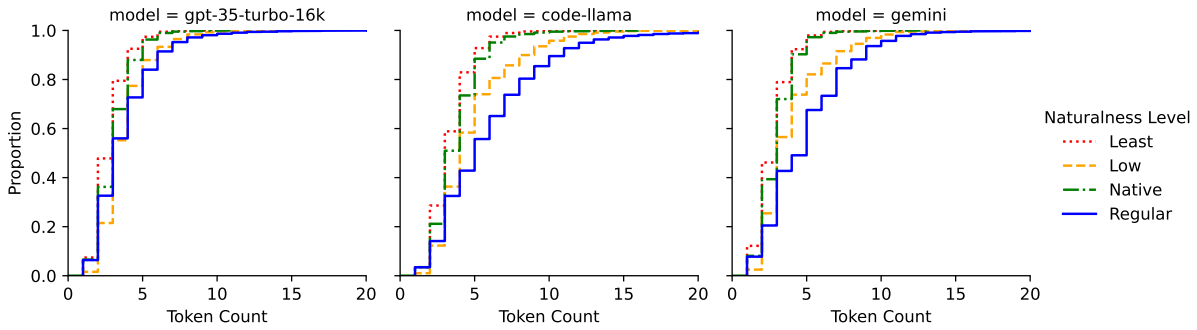


Figure 3.7. Token count CDF, by naturalness level, for each language model.

$$TCR = \frac{|I_{tokens}|}{|I_{characters}|} \tag{3.2}$$

Because there is not a clear relationship between token count and naturalness, we derive a token-to-character ratio metric (see 3.2) which is the count of identifier tokens I_{tokens} divided by the count of identifier characters $I_{characters}$. What we see in Figure 3.8 is a fairly strong differentiation between naturalness levels and TCR, where more natural identifiers have lower TCR than less natural identifiers. We believe that this hints at the effects of in- vs. out-of-vocabulary character sequences and the strength of their semantic meaning in latent space. However, the relationship is not so strong that TCR alone can serve as a useful classification method for identifier naturalness. We leave additional exploration of this topic to future research.

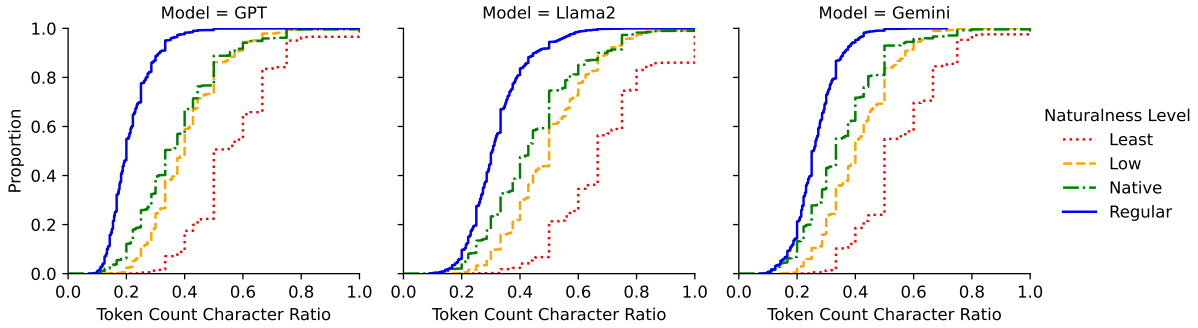


Figure 3.8. Token counts to character count ratio, by naturalness level, for each language model. More natural identifiers generally contain fewer tokens-per-character than less natural identifiers, suggesting a higher presence of in-vocabulary keywords for more natural identifiers.

3.4 Naturalness-modified Identifiers

Naturalness modification is the process of changing an identifier in such a way that it assumes a naturalness category that is not its original classification (see Figure 3.5). Modifying an identifier to become less natural is useful for creating benchmark schemas of varying naturalness levels. The same benefit applies to the process of modifying less natural identifiers to become more natural; this direction of modification also generally yields improved NL-to-SQL performance, as is demonstrated in the experiment and evaluation sections of this report.

Naturalness-modified identifiers generated by the ML-based approaches described next are human-validated and, when necessary, modified. Once validated, the identifiers are added to our ground truth dataset and used for the prompt and query naturalness modification processes described elsewhere in this report.

3.4.1 Decreasing Naturalness (Abbreviation)

Decreasing naturalness generally involves the removal of characters from an identifier in a manner that shortens the length while retaining some structure that still allows for some measure of readability. This cannot be achieved by randomly removing characters from an identifier; so we elect to use machine learning-based approaches to decrease identifier naturalness. As with our classification approaches, we experiment with both engineered few shot prompts targeted at a

general purpose foundational LLM (GPT), and a fine tuning approach (GPT Davinci).

FPT Davinci Fine Tuning Abbreviation

Seperate models are trained for converstion tasks from one naturalness level to a model-specific alternative naturalness level. This resulted in the following fine tuned models:

- Regular to Low
- Regular to Least
- Low to Least

Each fine tune dataset consists of 176 randomly selected identifiers and human-created naturalness modifications. Below is an example of Regular to Least model training data:

```
{"prompt":"Plot ->","completion":" p\n"}
{"prompt":"Metals ->","completion":" mt\n"}
{"prompt":"Station_ID ->","completion":" S_ID\n"}
{"prompt":"FUELEAK ->","completion":" F_Lk\n"}
```

The outputs of these finetuned models require significant adjustment by human researchers; so we elect to employ an alternative approach described next.

GPT Few Shot Abbreviation

GPT 3.5-based few shot prompting (see example below) resulted in the most consistent outputs, with a reasonably low prompt token count. Rather than explaining the different categories followed by an instruction to convert an identifier to a specific category, we find that providing a simple instruction to *abbreviate the database schema identifier to make it slightly shorter* followed by several examples is more effective.

```
Abbreviate the database schema identifier
to make it slightly shorter:
```

```
Protocol_Name -> Protcl_Nm


Abbreviate the database schema
identifier to make it slightly shorter:
WaterTemperature -> WaterTemp


Abbreviate the database schema
identifier to make it slightly shorter:
Customer -> Custmr


Abbreviate the database schema
identifier to make it slightly shorter:
_IDENTIFIER_ ->
```

3.4.2 Increasing Naturalness (Expansion)

Increasing naturalness requires the expansion of an abbreviated identifier. A recent attempt at performing this task with model fine tuning [123] was made during our research; and it appears to be a promising direction for research. However, we elect to enrich our process with external database metadata.

Expansion Process

In order to accomplish this, we engineered a solution that employs a database metadata reader capable of reading .pdf, .xml, and .csv formatted metadata. Metadata is read and indexed at the word level, where an array of file locations (page and line numbers for pdf, line numbers for xml and csv) where words occur are mapped to each word. When a user keys in an identifier to modify, file locations where the identifier exists in the document are returned via index lookup. These index locations are used as the centerpoints of context windows that retrieve the surrounding content. This content is added to a fewshot prompt to provide the language model

with document content that is likely to contain references to, and definitions of, the provided identifier.

The fewshot prompt for generating an expanded identifier adheres to the template:

```
Using the following text extracted from a
data dictionary:
__CONTEXT__

In the response, provide only the old identifier
and new identifier (e.g. "old_identifier,
new_identifier").
Create a meaningful and concise database identifier
using SQL compatible complete words to represent
abbreviations and acronyms for only
the identifier __IDENTIFIER__:
```

The __CONTEXT__ placeholder is replaced with up to ten context window-length excerpts from the database metadata. This is an example of a completed prompt using the NYSED .pdf based data manual with context window of 200 characters:

```
Using the following text extracted from a data dictionary:
r school  Text  255
YEAR  Reporting Year (2021 for 2020 -21; 2022 for 2021 -22)
Number  4
NUM_TEACH  Number of teachers as reported
in the Student Information Repository System
(SIRS)  Number  12
NUM_TEACH_INEXP  Number of teachers with fewer
than four years of experience in their positions
```

```
Number  12
PER_TEACH_INEXP  Percent of teachers with
fewer than four years of experience in their posi

In the response, provide only the old identifier
and new identifier (e.g. "old_identifier, new_identifier").
Create a meaningful and concise database identifier
using SQL compatible complete words to represent
abbreviations and acronyms for
only the identifier num_teach_inexp:

num_teach_inexp, number_of_teachers_inexperienced
```

In this successful example, we see that the identifier *num_teach_inexp* has been expanded to a more natural *number_of_teachers_inexperienced*. This is despite the observation that the data retrieved from the .pdf file is quite unstructured and contains document artifacts. A sufficiently wide context window coupled with the retrieval of multiple occurences of the identifier in the document generally results in valid expansions.

Prompt Building

In order to generate few shot prompts over an arbitrary metadata source, some prompt engineering is necessary. Generally, hand-crafted prompt building is suitable approach; but it does not scale nor does it lend itself to an automated solution that can be deployed beyond a research lab. To make this process more portable, we introduce a command line-based subroutine that enables the automatic build of a five example few shot prompt. In this process:

1. User enters an identifier

2. Zero shot prompt -> expanded identifier

3. User reviews and validates identifier
4. Correct: identifier added to example list
5. Incorrect: User tries again with different identifier
6. User enters another identifier
7. Few shot prompt (with prior successes as examples) -> expanded identifier
8. Correct: identifier added to example list
9. Incorrect: User tries again with different identifier
10. Process repeats until five successful examples are generated

Once a fewshot prompt has been created for a given database's metadata, the prompt is stored for any future program runs. This particular aspect of our project was built to support our research efforts; and we did not perform any experiments to evaluate its overall accuracy and usability. We leave these tasks as future research opportunities.

3.5 Base Collections

Given the recency of the LLMs selected for evaluation in this project, and the relative maturity of existing NL-to-SQL benchmarks, we believe that foundational LLMs have been exposed to existing benchmark training and development NL questions and queries in their training corpora. NL-to-SQL performance differences between queries over seen vs. unseen schema are significant [101], and we seek to avoid as much bias as possible due to intentional or unintentional pre-training on existing benchmark datasets.

We also find that existing benchmarks including Spider [119], and BIRD [55], do not match the identifier naturalness distribution of real-world schema collections such as SchemaPile [20]. Although SchemaPile is a very large representation of real-world schemas, it does not contain

Database	Tables	Columns	Questions	Org
ASIS	36	245	40	NPS
ATBI	28	192	40	NPS
CWO	13	71	40	NPS
KIS	18	157	40	NPS
NPFM	27	190	40	NPS
NTSB	40	1611	100	NHTSA
NYSED	27	423	63	NYSED
PILB	21	196	40	NPS
SBOD	2588	90,477	100	SAP

Table 3.3. SNAILS Real-World Database Schemas

database instances necessary for benchmark performance evaluations; so, we are not able to leverage its dataset in the creation of a new benchmark. To reduce bias due to benchmark data exposure, and to create a benchmark more representative of real-world schema naming, SNAILS contains two artifacts for NL-to-SQL benchmarking: Artifact 1, which is a collection of 9 publicly-available database schemas and data; and Artifact 6, a human-generated set of 503 NL question - gold query pairs.

3.5.1 Datasets

Native Schemas

The SNAILS real-world database schema collection (Artifact 1) consists of 9 databases sourced from multiple locations. We refer to the schema identifier names as they exist in the source databases as *Native*, and we classify each schemas' Native naturalness level (see Figure 3.9). Domain diversity facilitates a more thorough evaluation [24]; so, SNAILS database collections span multiple domains. Domain coverage includes scientific nature observation records, vehicle safety statistics, primary school performance data, and business resource planning.

The U.S. National Parks Service's IRMA Portal [1] is the source of the scientific observation databases which include the Field Data for the Inventory of Amphibians and Reptiles of Assateague Island National Seashore (**ASIS**) [18], Great Smoky Mountains All Taxa Biodiversity Inventory (**ATBI**) Plot Vegetation Monitoring Database [23], Wildlife Observations

Database: Craters of the Moon National Monument and Preserve 1921-2021 (**CWO**) [99], Exotic and Invasive Plants Monitoring Database (**KIS**) [37], Northern Plains Fire Management (**NPFM**) [66] and Pacific Island Network Landbird Monitoring Dataset (**PILB**) [42].

The National Transportation Safety Bureaus 2021 safety sampling dataset [68, 89] is the source of SNAILS **NTSB** safety statistics database. We source school performance data (**NYSED**) from the New York State Education Department [5].

The business resource planning database **SBOD** is a training example of the popular SAP Business One system, and is publicly available in MS SQL server backup format [83]. The SBOD schema consists of an extremely large number of tables and columns; so pruning is required to fit the schema within the context window of the LLMs we compared. We reduce the schema knowledge token requirements by segmenting the SBOD schema into submodules and further reducing tables through data profiling. Additional information on the SBOD schema knowledge management is available in the appendix.

Each database was migrated from its source format into an MS SQL Server database. Several databases contained identifiers with whitespace characters, which is uncommon in most schemas. To mitigate whitespace-related inference failures as a confounder, we modify the native identifiers by replacing whitespace characters with underscore characters. In total, 148 out of over 19,000 total identifiers (less than .01 percent) contained at least 1 whitespace character.

Native Schema Naturalness Levels

Since the intent of this project is to measure the effect of schema naturalness, we first check if there is sufficient distribution of naturalness levels across the collection. We employ the GPT-3.5-based classifier described in Section 3.3.2 to evaluate the naturalness of the native schema identifiers.

In addition to measuring the proportion of identifiers in each naturalness category, we also derive a combined naturalness score. Combined naturalness is the weighted average of category proportion values, where scores range from 0.0 to 1.0 with 1.0 representing a schema

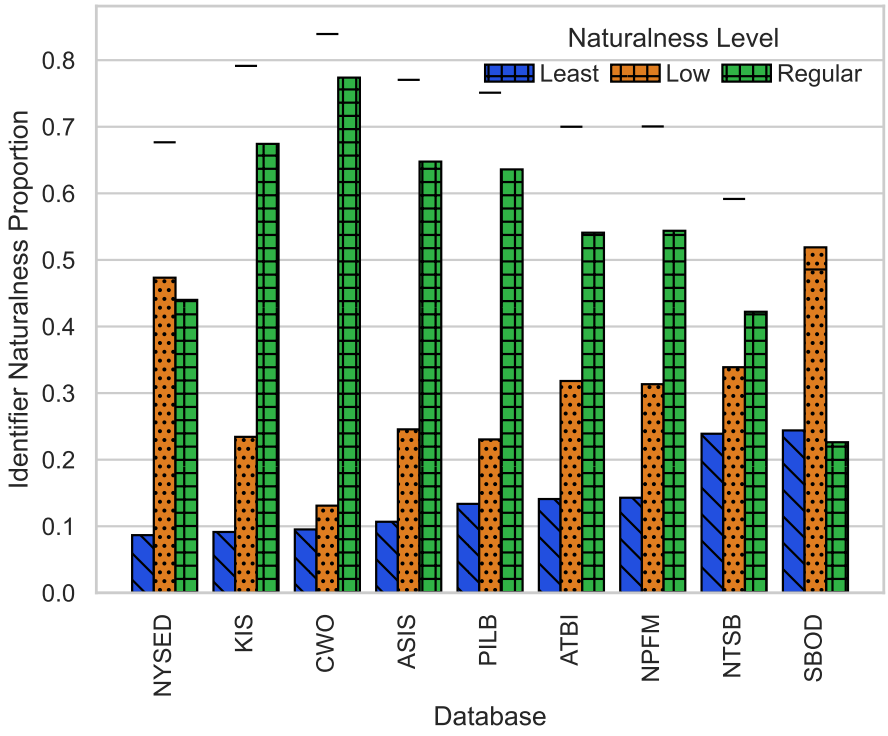


Figure 3.9. Proportion of identifiers in each naturalness category within the SNAILS real-world database collection (Artifact 1). Horizontal line markers indicate calculated combined naturalness as described in the appendix

containing only Regular naturalness identifiers. A more detailed description of its calculation is available in the appendix.

Figure 3.9 displays the proportions of identifiers in each naturalness category, as well as the combined naturalness, in each native schema. From the chart, we can see that the schemas in the SNAILS collection described in Section 3.5.1 represent a heterogeneous selection of naturalness combinations.

Modified (Virtual) Schemas

To control for confounding factors such as schema structure, normalization levels, and constraint variances between native schemas, we perform within-database evaluations of naturalness. To accomplish this, we generate 3 additional *virtual* schemas using the naturalness-modified identifiers (Artifact 4) described in Section 3.3.3. Each virtual schema is representative

Database	Qs	Top	Funct.	Join	C-Join	Ex	SQ	Where	Neg	Grp	Ord	Hvg
ASIS	40	1	24	13	1	0	2	18	0	17	1	0
ATBI	40	5	20	18	0	1	7	21	2	16	7	1
CWO	40	2	18	5	1	5	10	34	7	12	2	1
KIS	40	8	26	15	0	0	2	25	1	11	8	0
NPFM	40	5	27	21	0	0	1	29	0	16	5	0
NTSB	100	8	82	23	21	0	6	62	4	42	23	4
NYSED	63	10	36	10	4	1	21	55	1	16	10	1
PILB	40	6	25	23	0	0	3	20	0	16	11	2
SBOD	100	2	33	44	0	0	0	82	0	17	2	1

Table 3.4. Gold query clause counts for each SNAILS database. Columns represent a count of gold queries that contain the listed clause types. Qs is the count of question-query pairs for each database. C-Join is the subset of joins that require a composite key. Ex indicates the use of an exists clause. SQ indicates a subquery. Neg, Grp, Ord, and Hvg represent negation, group by, order by, and having. Note: MS SQL Server dialect replaces the common LIMIT clause with an equivalent TOP clause that precedes select items in the SELECT clause.

of a naturalness category, where schema identifiers are replaced with a semantically equivalent identifier of a different naturalness level. This results in 4 schema versions per database in the base collection: Native, Regular, Low, and Least.

The modified schemas are virtual because we do not create database instances that can be queried directly. Rather, we query virtual schemas via identifier replacement in prompts and generated queries using processes described in Section 3.6. This approach reduces storage overhead. It also enables possible future schema variations of different naturalness proportions without the need to instantiate additional database instances.

SNAILS Database Selection and Extension Processes

The initial 9 datasets and schemas are included because they were (1) publicly available, (2) not included in any prior NL-to-SQL benchmarks, (3) contained relational tables with dependencies and database instances with values, (4) had available table and column metadata, (5) represented a diversity of application domains, and (6) contain data potentially useful for real-world data analysis or data science applications. Databases are not selected or pre-screened using perceived naturalness as criteria.

We view the initial 9 schemas as a starting point from which the SNAILS dataset can grow. Researchers who wish to extend the SNAILS collection should use the same selection criteria. In addition, the extension process must ensure that new databases: (1) can be represented as MS SQL Server instances, (2) each native identifier's naturalness is classified according to defined criteria using the SNAILS naturalness classifier, and (3) that native identifiers are modified using the SNAILS modification artifacts to create alternate naturalness levels.

3.5.2 Data Sources

Field Data for Assateague Island National Seashore Amphibian and Reptile Inventory (ASIS)

Data Description

The ASIS database [18] is sourced from the National Parks Service (NPS) Irma portal [1] and contains scientific observation data of wildlife in the Assateague Island National Seashore preserve.

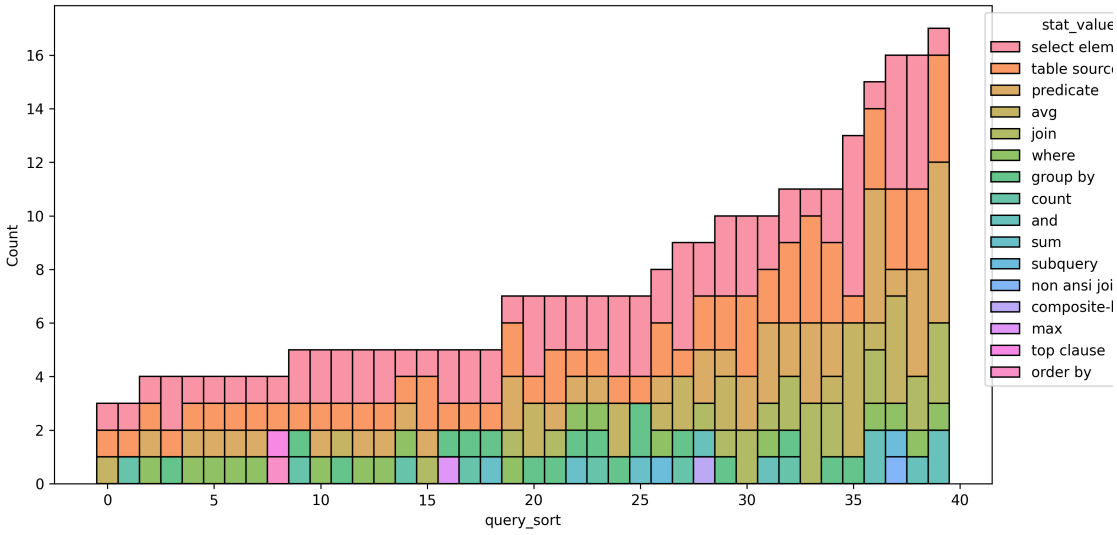


Figure 3.10. Gold query clause composition - ASIS database

ASIS Database Technical Details

- Data source format: Microsoft Access

- Migration method: SQL Server Migration Assistant
- Table count: 36
- Column count: 245
- Mean columns per table: 6.125
- NL Questions: 40
- Combined naturalness level: 0.77

Great Smoky Mountains All Taxa Biodiversity Inventory (ATBI) Plot Vegetation Monitoring Database

Data Description

The ATBI database [23] contains scientific observations of vegetation within the Great Smoky Mountains national park.

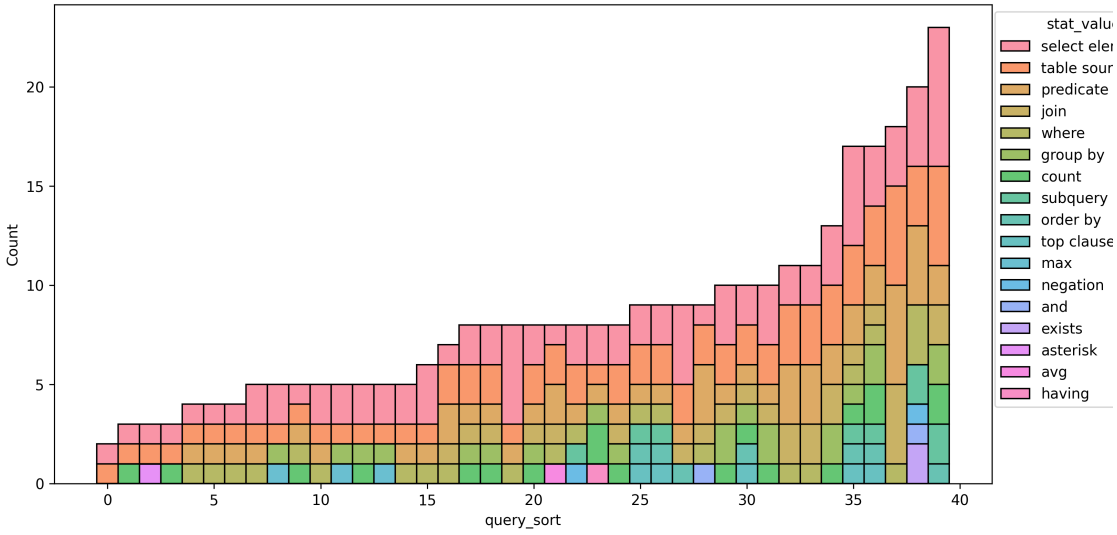


Figure 3.11. Gold query clause composition - ATBI database

ATBI Database Technical Details

- Data source format: Microsoft Access
- Migration method: SQL Server Migration Assistant
- Table count: 28
- Column count: 192
- Mean columns per table: 6.857
- NL Questions: 40
- Combined naturalness level: 0.70

Klamath Inventory and Monitoring Network (KIS)

Data Description

The Klamath Invasive Species (KIS) database [37] contains scientific observations of exoctic and invasive plants observed in Klamath Falls, Oregon.

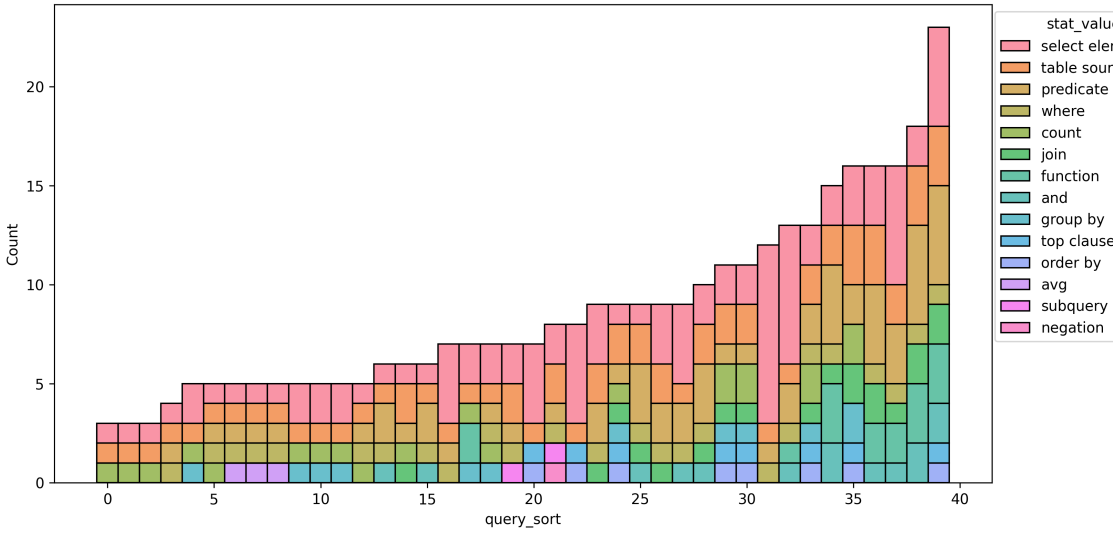


Figure 3.12. Gold query clause composition - KIS database

KIS Database Technical Details

- Data source format: Microsoft Access
- Migration method: SQL Server Migration Assistant
- Table count: 18
- Column count: 157
- Mean columns per table: 8.72
- NL Questions: 40
- Combined naturalness level: 0.79

Pacific Island Network Landbird Monitoring Dataset

Data Description

Pacific island landbirds (PILB) database [42] contains scientific observations of bird observations in various Pacific islands within the US states and territories.

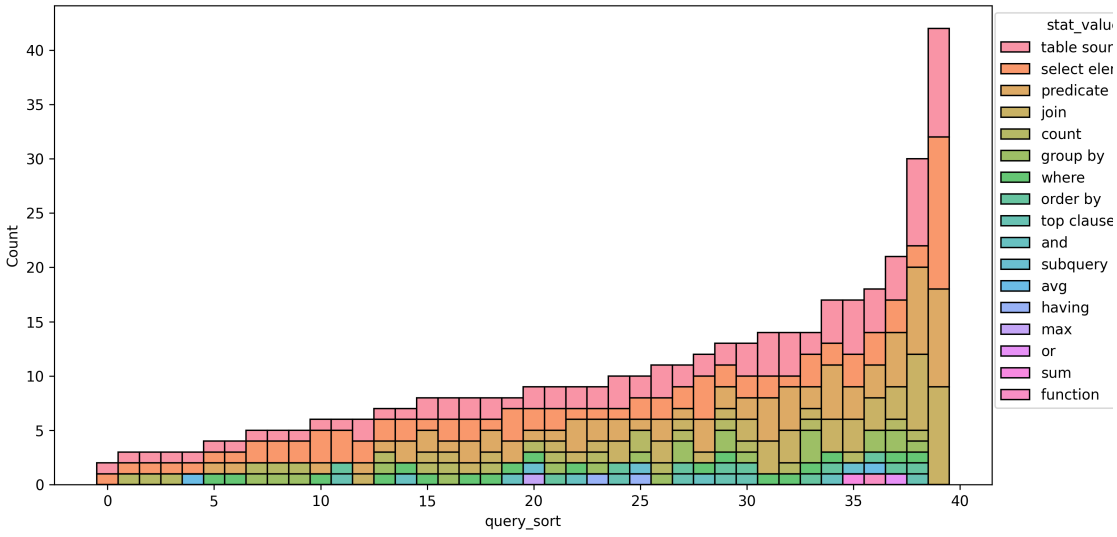


Figure 3.13. Gold query clause composition - PILB database

PILB Database Technical Details

- Data source format: Microsoft Access
- Migration method: SQL Server Migration Assistant
- Table count: 21
- Column count: 196
- Mean columns per table: 9.33
- NL Questions: 40
- Combined naturalness level: 0.75

Wildlife Observations Database: Craters of the Moon National Monument and Preserve 1921-2021

Data Description

The Craters Wildlife Observation (CWO) database [99] contains observations of wildlife spotted at the Craters of the Moon national monument and preserve. It is the smallest and most natural database in the benchmark data set.

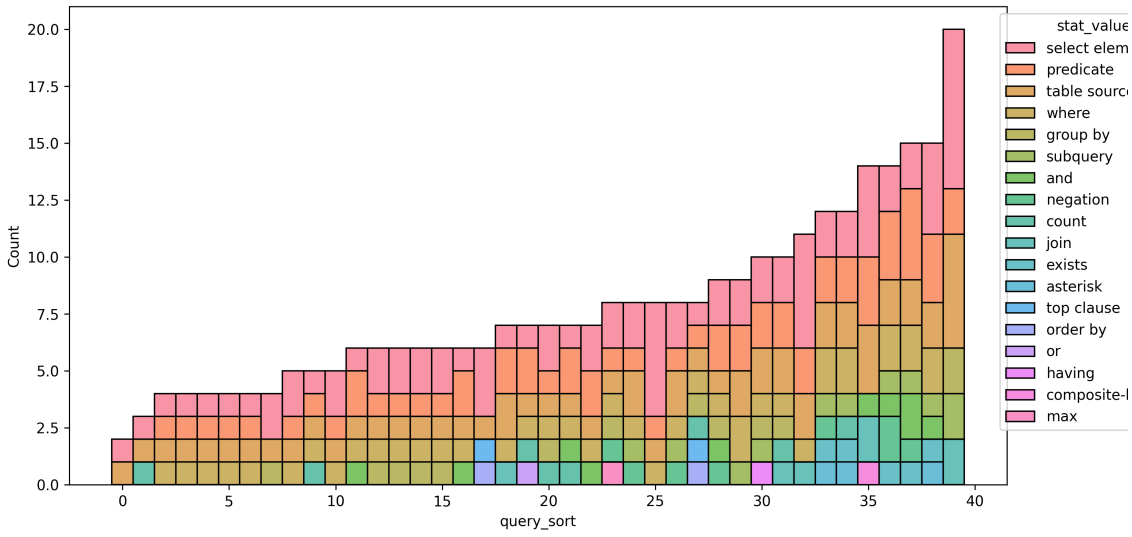


Figure 3.14. Gold query clause composition - CWO database

CWO Database Technical Details

- Data source format: Microsoft Access
- Migration method: SQL Server Migration Assistant
- Table count: 13
- Column count: 71
- Mean columns per table: 5.461
- NL Questions: 40
- Combined naturalness level: 0.84

Northern Great Plains Fire Management: FFI Database

Data Description

The NPFM database [66] contains observations of various overstory and other flora within the Northern Plains region of the Nationa Parks Service.

Code-Bison Evidence of Familiarity

With this dataset, we observe some indications of exposure to the Code-Bison language model. We note that we no longer report performance results of Code-Bison inference in our main report.

When prompted with the NL question "How many overstory's have a codominant canopy position?", it generated the query:

```
SELECT COUNT(*)
FROM tbl_Overstory
WHERE CanPos = 2;
```

Which is a correct reference to the canopy position (CanPos) lookup code of 2, which corresponds to the codominant canopy position. The LLM was not provided code lookup

information within the prompt, which suggests that some reference to the NPFM schema was included in its training data.

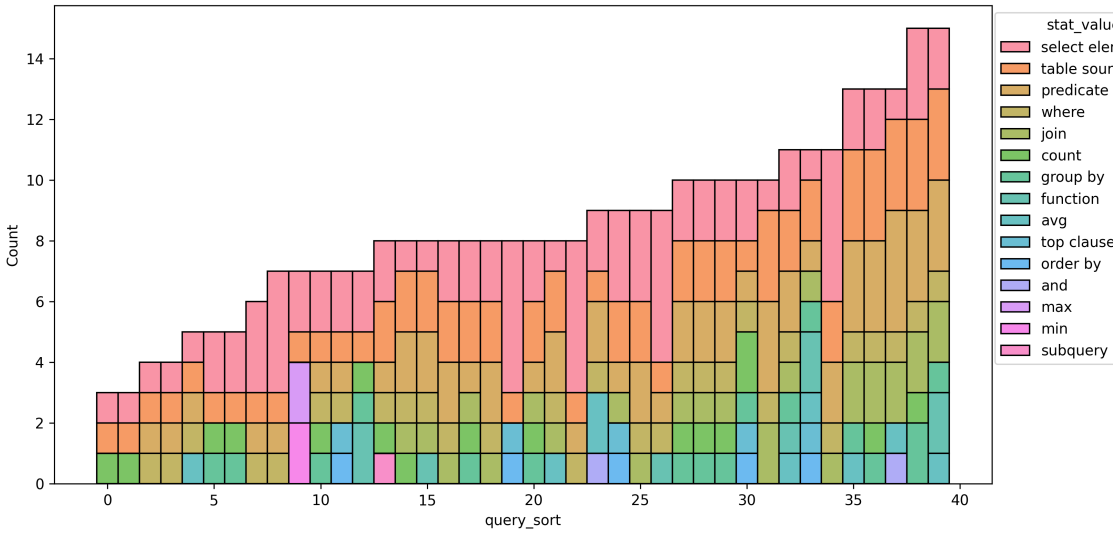


Figure 3.15. Gold query clause composition - NPFM database

NPFM Database Technical Details

- Data source format: Microsoft Access
- Migration method: SQL Server Migration Assistant
- Table count: 27
- Column count: 190
- Mean columns per table: 7.037
- NL Questions: 40
- Combined naturalness level: 0.70

2021 Crash Investigation Sampling System

Data Description

The crash investigation sampling system [89] is sourced from the National Transportation Safety Board, and referred to as NTSB in this paper. It contains sampled data of crash and vehicle statistics from 2021. The data is organized such that composite key joins are required for most multi-relation queries.

Additional Implementation Details

This is the only database in our collection that required deliberate migration from a non-database format to the target MS SQL Server environment. We acquired the data in .csv form, with a single .csv per table. Analysis of the files confirmed that although not in database form, the data was relational in nature, and migration involved SQL-based ingestion from .csv files into the target schema. The .sql scripts used to generate the database schema and insert table values are available in the project repository.

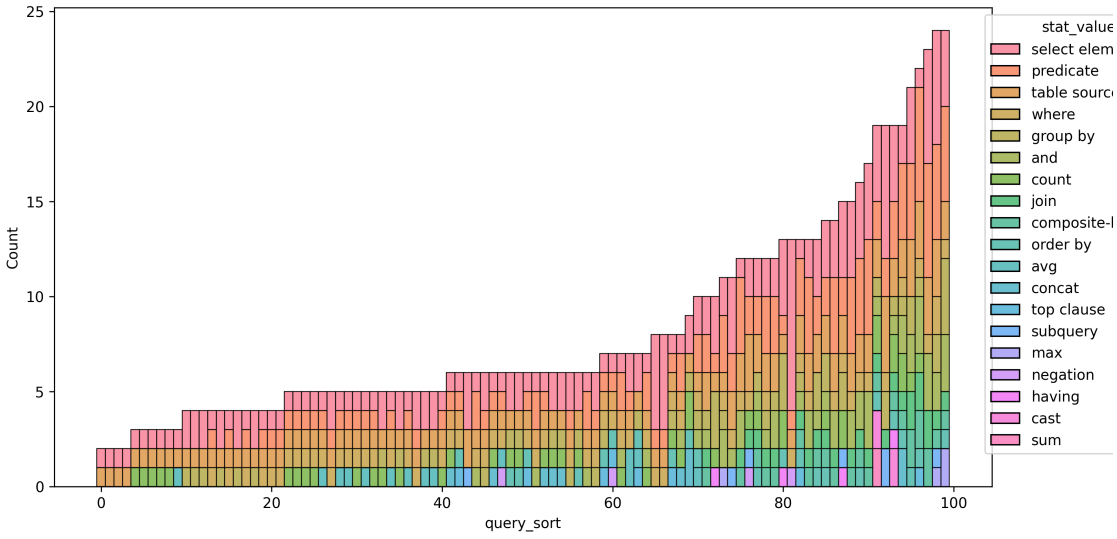


Figure 3.16. Gold query clause composition - NTSB database

NTSB Database Technical Details

- Data source format: Comma Separated Value (CSV) files

- Migration method: SQL database creation and Python-based ETL scripting
- Table count: 40
- Column count: 1,611
- Mean columns per table: 40.275
- NL Questions: 100
- Combined naturalness level: 0.59

New York State Education Department Report Card Database 2021-22

Data Description

The NYSED database [5] is sourced from the New York State Education Department. It contains standardized testing and demographic data for all public elementary, middle, and high schools in New York State.

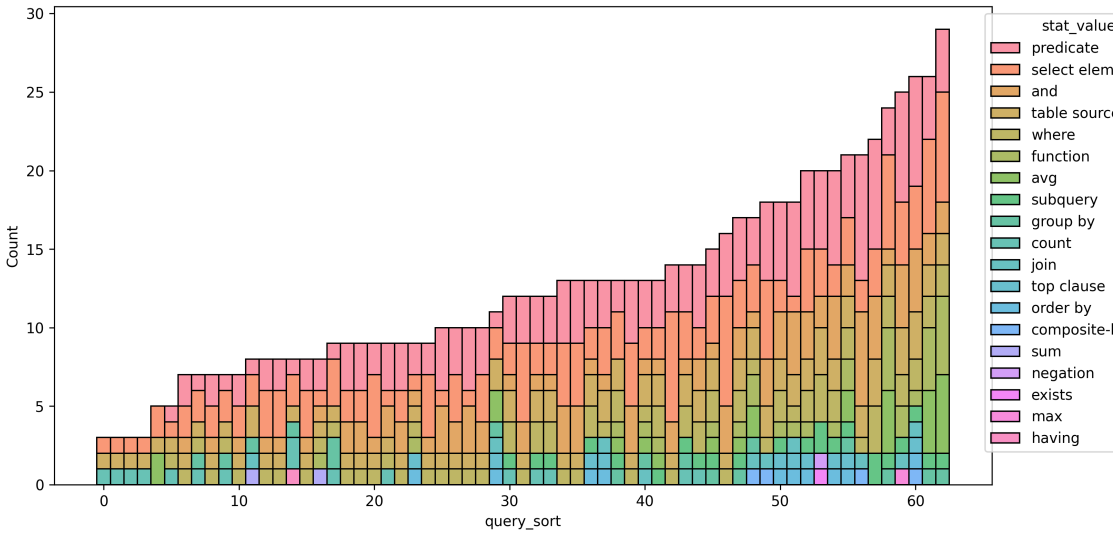


Figure 3.17. Gold query clause composition - NYSED database

NTSB Database Technical Details

- Data source format: Microsoft Access
- Migration method: SQL Server Migration Assistant
- Table count: 27
- Column count: 423
- Mean columns per table: 15.67
- NL Questions: 63
- Combined naturalness level: 0.68

Localized Demo Databases Now Available for SAP Business One

Data Description

The SBOD database [83] is sourced from a publically available SAP demonstration and training database. It is the largest, and least natural, database within our dataset. Given its schema size, we divided it based on SAP module, and further reduced the schemas used in our benchmark based on the training database cardinality (e.g. we removed most tables containing 0 tuples).

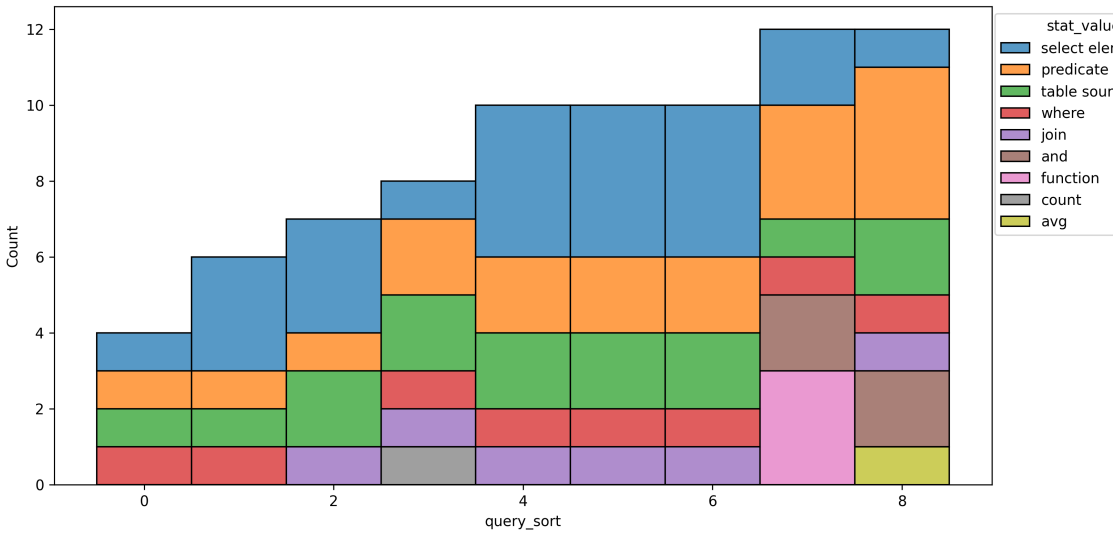


Figure 3.18. Gold query clause composition - SBOD database module example

SBOD Database Technical Details

- Data source format: MS SQL Server backup (.bak) file
- Migration method: MS SQL Server backup recovery
- Table count: 2,588
- Column count: 90,477
- Mean columns per table: 34.96
- NL Questions: 100
- Combined naturalness level: 0.49

Module	Tables	Columns	Questions
Banking	40	1720	10
Business Partners	40	1443	10
Finance	61	1988	10
General	71	1035	10
Human Resources	28	440	20
Inventory and Prod.	65	1942	10
Reports	40	734	10
Sales Opportunities	20	283	10
Service	40	875	10

Table 3.5. SBO Demo Module Schemas

SAP Business One Additional Details

Business One is an enterprise resource planning (ERP) system created by the German software systems developer SAP. It is a common platform in government and commercial domains where large-scale business management solutions are required. The SBOD database contains a significant number of tables and columns. Such a schema size poses a problem for generating schema knowledge representations in zero shot prompts, even for large context window variations of evaluated LLMs. To overcome this constraint, we divide the SBOD schema into 9 sub-modules

based on schema descriptions published by an online community of SAP practicioners [2]. We further prune the SBOD schemas using the cardinality of the training database, where tables without data entries were excluded from NL questions and prompt schema knowledge.

3.5.3 NL Questions and Gold Queries

The NL question - SQL query pair artifact consists of 9 .sql files containing between 40 and 100 entries each. Question and query pairs are written in executable .sql files. Natural language questions are written as SQL comments; and SQL is written in the T-SQL dialect employed in MS SQL Server. For public repositories storing the questions, we store them in .zip files in order to reduce the possibility of inclusion in language model training material. Each file is associated with a database in the SNAILS schema collection. Some NL questions contain hint and note entries annotated as HINT and NOTE in lines that follow the NL query. We used neither the hints nor columns in any of the experiments described in this paper, but retain them for possible use in future research.

While we store the data in .sql file format for readability and ease of use, we also offer a NL question loading script (load_nl_questions.py) in our repository. This script performs rudimentary parsing of the .sql files and returns a Pandas DataFrame and optional .xlsx formatted spreadsheet.

NL Question - Query Example 1, ASIS Database Question 8

The focus of this benchmark dataset is on the evaluation of schema linking. As such, we were generous with value descriptions, providing literal value strings (e.g. ASIS_HERPS_20H location code in example 1) in the prompt.

```
-- 8: show how many minnows of each stage were counted
      at the location ASIS_HERPS_20H
SELECT stage, sum(count) minnowCountSum
```

```
FROM tblFieldDataMinnowTrapSurveys
WHERE locationID = 'ASIS_HERPS_20H'
GROUP BY stage
;
```

NL Question - Query Example 2, NTSB Database Question 13

Example 2 shows additional code value hints provided in the NL question. In order to enable the recall evaluation statistic, we limited the use of columns and tables in gold queries to the minimum necessary to form a correct query. In the cases where any arbitrary column as an argument in the count function will yield the same result as the $*$, we use only the $*$ symbol. This eliminates the recall penalty for models selecting an arbitrary column within the count function.

```
-- 13: How many vehicles are there where drugs were present
       (presence code value is 1) and the vehicle was towed
       for a reason not due to disabling damage (towed code is 3)
SELECT COUNT(*) VEHCOUNT
FROM GV
WHERE PARDRUG = 1 AND TOWED = 3
;
```

NL Question - Query Example 3, SBOD Database Human Resources Module Question 18

Questions vary in their complexity. This example shows one of the more complex questions that require multiple projections and joins as well as a selection.

```
-- 18: Show the professional status and educational
       statuses as well as the home and work street
       numbers of employees on the purchasing team.
```

```
SELECT StatusOfP, StatusOfE, StreetNoW, StreetNoH
FROM OHEM employees
JOIN HTM1 teamMembers
    ON employees.empId = teamMembers.empID
JOIN OHTM emplTeams
    ON teamMembers.teamID = emplTeams.teamID
WHERE emplTeams.name = 'Purchasing'
;
```

3.5.4 Benchmark Naturalness Comparisons

Though we believe the quality of existing benchmarks is excellent, and the hard work of researchers associated with those projects has resulted in significant improvements in NL-to-SQL system design, we find that a large proportion of these benchmark schemas are canonical, small in composition, and highly natural compared to databases and data sets often encountered in real world scenarios. Using our naturalness classifier described in the main report and the appendix section **??**, we determine the naturalness levels of the Spider [119] and Bird-SQL [55]. Our classifier model indicates that both Spider and Bird database schemas are highly natural, moreso than any real-world schema we acquired for our benchmark (see Figure 3.19 for a visual comparison). Additionally, we evaluate the naturalness of the real-world schema identifiers in the SchemaPile [20] and find that the SNAILS naturalness better-aligns with SchemaPile than the previously-mentioned benchmarks. Figure 3.9

3.5.5 NL Question - SQL Query Pairs

To evaluate SQL inference performance over the Native and modified schemas in the SNAILS real-world database collection, we create a new set of 503 NL-question and SQL gold query pairs (Artifact 6). Schema identifier naturalness are the primary considerations for NL question and gold query composition. During question and query formulation we track schema

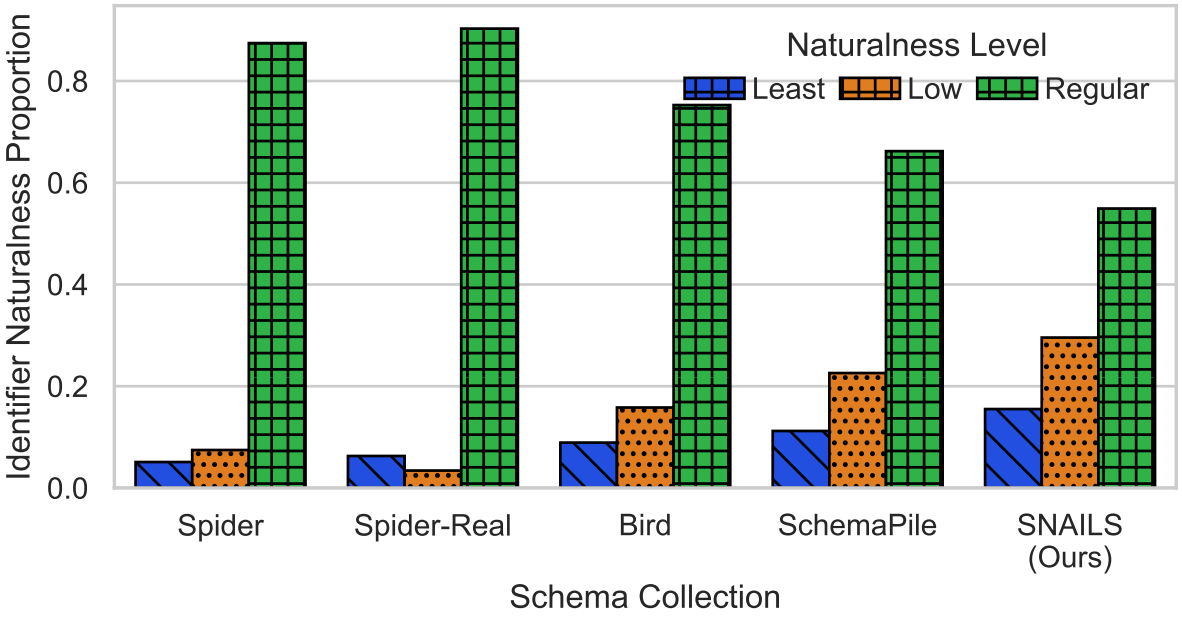


Figure 3.19. Spider [119] and Bird [55] benchmarks classified with the Davinci-based classifier reveals that both benchmark databases have highly natural identifiers even compared to the most natural of the databases in our proposed benchmark. Our benchmark more closely aligns with the naturalness of the real-world schema collection SchemaPile [20]

.

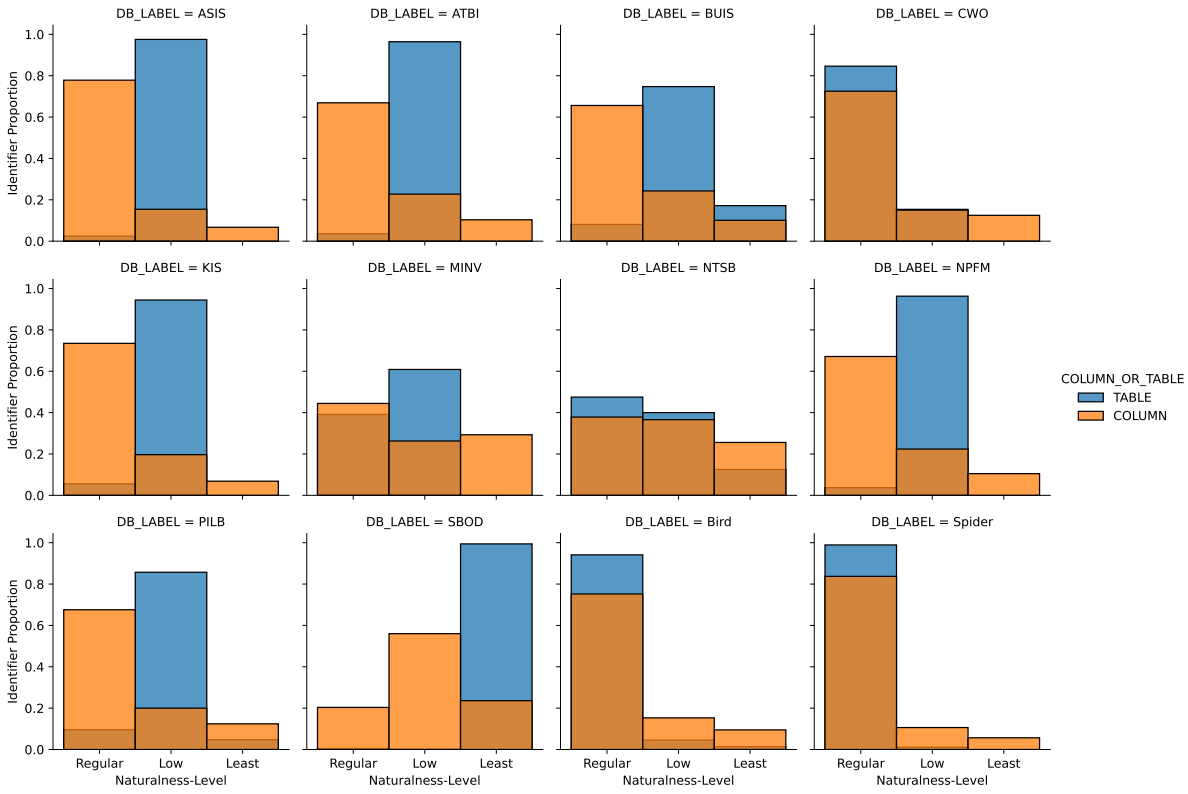


Figure 3.20. At the individual database schema level, the SNAILS database collection has a diverse arrangement of naturalness levels.

coverage to ensure that the distribution of identifier naturalness within a set of gold queries generally matches the naturalness distribution of target schemas.

To enable accuracy measurements at the identifier level, gold queries contain the minimum identifiers (tables and columns) required to answer its corresponding question. For this reason, for questions that require the count aggregation function, where appropriate, we use the COUNT(*) clause (as opposed to selecting an arbitrary column). This approach eliminates incorrect penalties to recall if a generated query fails to project an arbitrary column as a function argument.

Gold queries contain only native identifiers, such that all gold queries return valid non-null results from target databases in the real-world database collection (Artifact 1). We measure query complexity as a count of its clauses and identifiers. Gold queries span a range of complexities, from very simple single table projections, to multi-table joins and nested subqueries (see Table 3.4).

Adding New NL-SQL Pairs to the SNAILS Collection

For researchers interested in extending the SNAILS collection, it is necessary to create new ground truth NL-SQL pairs for evaluation. While we employed a fully manual approach for question writing, and this approach may be used for future additions, they may also consider the use of new approaches such as using a template-based approach for generating question-query pairs with relational data as input [80]. Regardless of NL-SQL pair creation method, researchers should ensure adequate schema coverage and minimum essential identifier selection as described in the preceding section.

3.6 NL-to-SQL Benchmarking Setup

To evaluate the impact of naturalness on NL-to-SQL accuracy, we build a benchmarking setup pipeline as shown in Figure 3.21. NL question and gold query pairs, database schemas, and naturalness crosswalk mappings are inputs into subprocesses. The subprocesses include prompt generation, schema identifier naturalness modification, identifier naturalness classification,

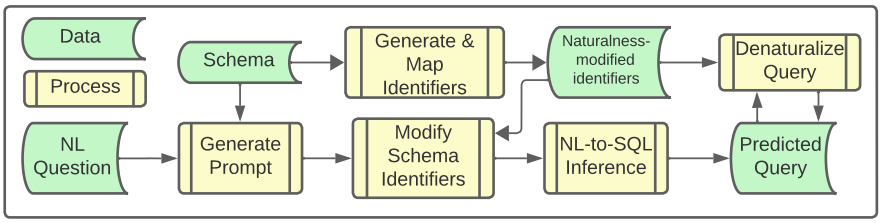


Figure 3.21. Experiment setup workflow from NL question and schema as input to predicted query as output.

LLM-based NL-to-SQL inference, and predicted query "denaturalization" (i.e., converting table and column identifiers to native schema identifiers prior to query execution). The output of the experiment setup is a predicted query, which along with its gold query counterpart, is executed against a target database. This predicted query is passed into a parser analysis tool as initial steps of the *Performance Evaluation and Results Classification* phase of the experiment described in Section 3.8.

3.6.1 Prompt Generation

The design space for LLM-based NL-to-SQL prompting is quite large, with options ranging from zero-shot instructions to sequential prompting broken into discrete tasks such as schema subsetting and error handling. Although we evaluate 2 complex NL-to-SQL workflows, to maintain consistency across the LLMs compared in this study, our performance comparisons focus on a single prompting strategy: zero-shot prompting with schema knowledge.

Prompting Strategy

SNAILS prompts consist of zero-shot instructions with schema knowledge (denoted as ZS in results figures) in a format similar to OpenAI's Text-to-SQL demonstration prompt [28] for completions. The prompt begins with task instructions and database information:

```
For the database described next, provide only a sql query.
do not include any text that is not valid SQL.
#Database: NTSB
```

```
#MS SQL Server tables, with their properties:
```

Target database system tables provide schema knowledge, which is represented as a list of tables and their column names with data types in the format:

```
#TableName (Col1Name Type, Col2Name Type, ...)
```

The prompt ends with the instruction:

```
### a sql query, written in the MS SQL Server dialect,
    to answer the question: <Question>
```

Where <Question> is replaced with an NL question directed at the given schema.

To evaluate naturalness effects on more complex NL-to-SQL prompting workflows, we also implement DIN SQL [84] which uses prompt chaining with GPT-4, and CodeS [52]–a multi-step NL-to-SQL system (schema filtering and SQL inference) based on StarCoder [56] and finetuned for the NL-to-SQL translation task.

Prompt Schema Identifier Modification

For inference on virtual schemas with modified naturalness levels, we replace Native identifiers with corresponding identifiers of the target virtual schema's naturalness level. We accomplish this step using the naturalness-modified identifier collection (Artifact 4) described in Section 3.3.3. We use a SQL parser to encase identifiers within tags to improve identifier replacement accuracy and eliminate errors due to substring matching between identifiers.

3.6.2 Prompt Naturalness Modification

Prompt naturalness modification is necessary when generating SQL queries over schemas with modified identifiers. In order to prevent producing additional database instances with renamed identifiers, we employ a middleware approach where modified identifiers are retrieved from a mapping of native identifiers to the target naturalness level. The prompt is generated using

native schema identifiers, and table and column names are encased in XML-like opening and closing <TABLE_NAME> and <COLUMN_NAME> tags. Native identifiers and their tags are replaced with modified identifiers using standard Python string replacement (e.g. str.replace(target, value)).

Prompt naturalness conversion example

The objective is to modify the naturalness of the Klamath Invasive Species (KIS) schema to the least natural form. The first step is to generate a tagged prompt which is formed using the database metadata accessed via system tables. A tagged prompt table with columns and datatypes takes on the form:

```
#<TABLE_NAME>tlu_Species_WHIS</TABLE_NAME>
(
  <COLUMN_NAME>Species</COLUMN_NAME> nvarchar,
  <COLUMN_NAME>SampleYear</COLUMN_NAME> nvarchar,
  <COLUMN_NAME>Park</COLUMN_NAME> nvarchar
)
```

Each table and its columns occupies a single line within the prompt. The resulting prompt after string replacement appears as the following in the final prompt:

```
#TSW( Sp nvarchar, S_Yr nvarchar, Pk nvarchar)
```

The modified schema knowledge is presented to the LLM as though it is the native schema.

3.6.3 NL-to-SQL Inference

Language Models

Foundational LLMs continue to grow in capability at a rapid pace. Despite this growth, not all NLI implementations can avail of the most-capable LLMs, often due to organizational policy

constraints (e.g., organizational security concerns [30]). Additionally, we seek to understand if schema naming effects generalize across model architectures and sizes. Thus, we consider several LLMs, both open and closed source, to capture as many use profiles as possible including OpenAI's GPT-3.5 Turbo and GPT-4o [71, 72]; Google's Gemini 1.5 Ultra [104, 103]; and Phind-CodeLlama-34B-v2 [82] which is a finetuned variant of Meta's CodeLlama 2 [92].

GPT 3.5 Turbo and GPT 4o

GPT-based [71] generations use the gpt-3.5-turbo-16k model accessed using OpenAI's API services [72]. GPT-based models employ BPE using tiktoken [73] to tokenize inputs and decode model outputs.

Due to the size of larger schema knowledge prompts, we make use of the GPT 3.5 model [71] with a context window of 16,000 tokens *gpt-3.5-turbo-16k*. For consistency, all queries were generated with a 0 temperature, 1 top p, 0 frequency penalty, 0 presence pentalty. Responses are fetched from the OpenAI Python *ChatCompletion* class' *create* method with a single message (prompt) passed in the *user* role.

Phind Code-Llama

Phind Code-Llama [82] is a fine-tuned version of the 34b parameter Code Llama model. We used the TogetherAI API to access this model. We find that the finetuning appears to have improved its performance as compared to the baseline Code Llama 34b version, and as such we replace our prior analysis using Code Llama 34b with the results generated using the Phind model.

Gemini 1.5 Pro

During the course of our research, Google released Gemini 1.5 Pro [103], and we quickly integrated it into our existing workflows using the Google generative AI Python library. The remarkable faced of the Gemini 1.5 model is its context window size if 1 million tokens, which negates the need to reduce schema knowledge representations in order to meet context window constraints.

CodeS and DIN SQL Implementation

For the more complex DIN SQL and CodeS NL-to-SQL workflows, we provide additional versions of the SNAILS schema artifacts to conform to the input requirements of the target systems. Additionally, we add data logging between agents to document the schema filtering step for additional analysis. For consistency between approaches, we use GPT-4o for all steps in the prompting chain. For CodeS inference, we execute the schema filtering and NL-to-SQL inference using the CodeS codebase and finetuned models.

Generated Query Denaturalization

For queries targeted at virtual schemas and generated using modified schema identifiers, we perform reverse modifications prior to query execution on the native database schema. Using a purpose-built Antlr [81]-based parser, we extract table and column identifiers, and generate a tagged query with identifier tags encasing table and column names. The tags guide the replacement algorithm, ensuring accurate replacement of naturalness-modified identifiers with their Native naturalness counterparts.

3.6.4 Query Naturalness Modification

When a query is formed against a schema with naturalness-modified identifiers, it is necessary to replace the modified identifiers with the native identifiers prior to executing the query over the target database. Simple string replacement is not sufficient in this case, because some identifier names may sometimes be substrings of other identifiers within a query; and replacing one may corrupt the other. We employ a Java-based parser and AST generator [81, 4] to build a parser system for tagging table and column identifiers in a query.

Query naturalness modification example

To answer the question *For each location type, show a count of locations in shasta county*, GPT 3.5 generated the query with lowest naturalness schema knowledge:

```
SELECT LcTp, COUNT(*) AS LocationCount
```

```
FROM Locs
WHERE Cty = 'Shasta County'
GROUP BY LcTp
```

This query is converted to all capital characters and passed to the query parser tagger via API. The tagger traverses the AST using a listener class to encase tables and columns with identifier opening and closing tags. The parser also returns a list of aliases generated within the query. This allows consuming systems to ignore tagged aliases within the tagged query:

```
@BEGINTAGGEDQUERY
SELECT
  <COLUMN_NAME> LCTP </COLUMN_NAME> ,
  COUNT ( * ) AS LOCATIONCOUNT
FROM <TABLE_NAME> LOCS </TABLE_NAME>
WHERE
  <COLUMN_NAME> CTY </COLUMN_NAME>
    = 'SHASTA COUNTY'
GROUP BY
  <COLUMN_NAME> LCTP </COLUMN_NAME>
@ENDTAGGEDQUERY
@BEGINALIASES
<COLUMN_ALIASES>LOCATIONCOUNT</COLUMN_ALIASES>
@ENDALIASES
```

With the tagged query and aliases, the query is converted into a form that is compatible with the target native schema. This is accomplished using the schema identifier mapping dataset and standard Python string operations (e.g. str.replace("<TABLE_NAME> LOCS </TABLE_NAME>", "TBL_LOCATIONS")).

```
@DENATURALIZED RESPONSE:
SELECT
  [LOC_TYPE],
  COUNT (*) AS LOCATIONCOUNT
FROM [TBL_LOCATIONS]
WHERE [COUNTY] = 'SHASTA COUNTY'
GROUP BY [LOC_TYPE]
```

This completes the query modification step. The modified query is then used to extract results from the target database for result set matching evaluation.

3.7 Performance Evaluation

3.7.1 Query Execution

Predicted and gold queries are executed over target databases with native schemas using the PyOdbc Python library. Valid result sets are stored as Pandas Dataframes for comparison.

3.7.2 Execution Result Set-Subset Matching

Result set and subset comparison consisted of a comparison of the columns and rows returned by the two queries. Because it is possible for semantically equivalent query results to differ in terms of column ordering, column and aggregate function aliasing, row ordering (when an order is not specified in the question), and even the number of columns returned (as long as the predicted column set is a superset of the gold column set), result set comparison was performed using a series of rules implemented in Python.

Result cardinality The number of tuples, denoted as $|C_G|$ and $|C_C|$ in the predicted result R_P and the gold result R_G must be equal, and must be greater than 0:

$$\forall C_G \in R_G, C_P \in R_P \, (|C_G| = |C_P|) \wedge (|C_G| > 0 \wedge |C_P| > 0)$$

Empty sets are tagged as undetermined and retained for syntactic comparison. Non-empty, non-equal-size sets are tagged as non-matches and witheld from further analysis.

Projection completeness The columns in the predicted result R_P must be a superset of the columns in the gold result R_G. Columns C_P and C_G equivalence is determined by value comparisons between tuples T_G and T_P of the columns' contents:

$$\forall C_G \in R_G, \exists C_P \in R_P \text{ such that } \forall T_G \in C_G, T_G \in C_P$$

Column match candidates are determined via pairwise comparison of the sorted values in each column in R_G to each column in R_P. Full result sets R_G and R_P are sorted by corresponding column match candidates, with columns containing the most unique values serving as the primary sort key. With both R_G and R_P sorted by column match candidates, the two sets are compared row-wise for all columns in the set of column match candidates. If the two sets are not equal, the result sets are considered semantically non-equivalent.

3.7.3 Human Evaluation

Predicted queries that pass execution result set-subset matching are further evaluated by a human researcher to rule out false positives. Predictions that fail result set comparison are pre-classified as also failing manual matching. Predictions that pass set comparison are classified as ungraded until reviewed by a human researcher. Once reviewed using the Python-based GUI evaluation tool (see Figure 3.22), the predicted query receives its final matching score. Human evaluation resulted in scoring as incorrect 41 predictions that passed result set matching which is approximately two percent of all result set matched queries.

3.7.4 Schema Linking End-to-end Example

To better pinpoint schema linking performance, we devise a new approach for evaluating NL-to-SQL generation. In this approach, performance is measured using set comparisons

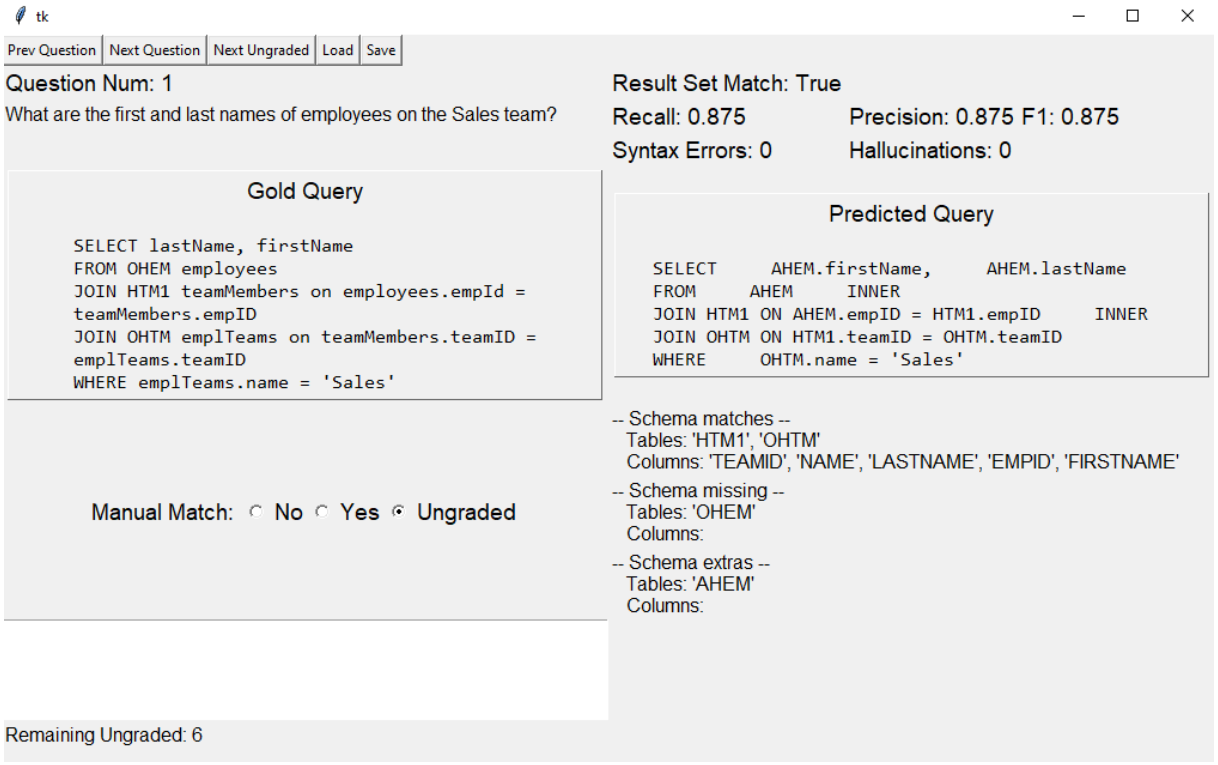


Figure 3.22. Screenshot of the query manual validation tool. This example depicts a query that passed set-subset matching, and is currently classified as ungraded. Helper information indicates that although the results matched, the incorrect table was selected during inference (AHEM instead of OHEM). This example was classified as incorrect.

between sets of identifiers within gold and predicted SQL queries. Recall is the primary metric. Precision and F1 are available, but less helpful, due to penalization for additional predicted columns that do not cause a query to be incorrect. The formulae for deriving linking metrics are printed in subsection 3.8.2 of the main report.

Linking Evaluation Example (ATBI Question 30, CodeLlama-34b)

This is an example of a predicted query that fails result set match comparison.

Natural Language Question

Which tree species were recorded as mature overstory but not as saplings? Include the species name and common name.

Gold Query

The gold query is quite complex, where using EXISTS clauses is a possible approach for solving.

```
SELECT species, CommonName
FROM tlu_PlantSpecies sp
  WHERE EXISTS(
    SELECT overstory_id
    FROM tbl_Overstory
    WHERE SpCode = sp.SpeciesCode
  ) AND NOT EXISTS (
    SELECT Seedlings_ID
    FROM tbl_Seedlings
    WHERE SpCode = sp.SpeciesCod
  )
```

Predicted Query (Code Llama 34b)

As we can see below, Code Llama 34b did not provide a correct solution. However, failure of result set matching does not preclude us from gleaning useful information from the result.

```
SELECT DISTINCT
 tlu_PlantSpecies.genus,
 tlu_PlantSpecies.subgenus,
 tlu_PlantSpecies.species,
 tlu_PlantSpecies.subspecies,
 tlu_PlantSpecies.SpeciesCode,
 tlu_PlantSpecies.CommonName
FROM
 tlu_PlantSpecies
 LEFT JOIN tbl_Overstory
    ON tbl_Overstory.SpCode
      = tlu_PlantSpecies.SpeciesCode
 LEFT JOIN tbl_Saplings
    ON tbl_Saplings.SpCode
      = tlu_PlantSpecies.SpeciesCode
WHERE
 tbl_Overstory.SpCode IS NOT NULL
 AND tbl_Saplings.SpCode IS NULL
ORDER BY
 tlu_PlantSpecies.genus,
 tlu_PlantSpecies.subgenus,
 tlu_PlantSpecies.species,
 tlu_PlantSpecies.subspecies,
 tlu_PlantSpecies.SpeciesCode,
 tlu_PlantSpecies.CommonName
```

Identifier Set Extraction

Using the purpose-built T-SQL parser, we extract a set of identifiers from the gold and predicted queries. Note that we do set comparison; so although columns are referenced in multiple clauses in the predicted query, we only measure the presence of a column or table once.

With a set QI_g of identifiers present in the gold query and a set of identifiers QI_p present in the generated (or predicted) query, we calculate recall, as well as f1 and precision.

Gold query identifiers QI_g :=

```
{
'TLU_PLANTSPECIES', 'TBL_OVERSTORY', 'TBL_SEEDLINGS',
'SPECIES', 'SPECIESCODE', 'COMMONNAME', 'SPCODE',
'OVERSTORY_ID', 'SEEDLINGS_ID'
}
```

Predicted query identifiers QI_p :=

```
{
'TLU_PLANTSPECIES', 'TBL_OVERSTORY', 'TBL_SAPLINGS'
'SPECIES', 'SPECIESCODE', 'COMMONNAME', 'SPCODE',
'GENUS', 'SUBSPECIES', 'SUBGENUS'
}
```

Identifier Set Comparisons

True positives are the intersection $QI_g \cap QI_p$ =

```
{
'TLU_PLANTSPECIES', 'TBL_OVERSTORY',
'SPECIES', 'SPECIESCODE', 'COMMONNAME', 'SPCODE'
}
```

$$QueryRecall = \frac{|QI_g \cap QI_p|}{|QI_g|} = \frac{6}{9} = 0.667$$

$$QueryPrecision = \frac{|QI_g \cap QI_p|}{|QI_p|} = \frac{6}{10} = 0.60$$

$$QueryF1 = \frac{2(Recall * Precision)}{Recall + Precision} = 0.632$$

So we see that although the predicted query failed in terms of execution result set comparison, we can still grade it in terms of linking performance. In other words, we can assign *partial credit* to predicted queries where correct schema identifiers are recalled.

3.8 NL-to-SQL Benchmarking Results

This section describes the process of evaluating the generated SQL query output from the prior section. We evaluate performance in terms of execution accuracy (result set comparison and manual evaluation) and schema linking (recall, precision, and F1).

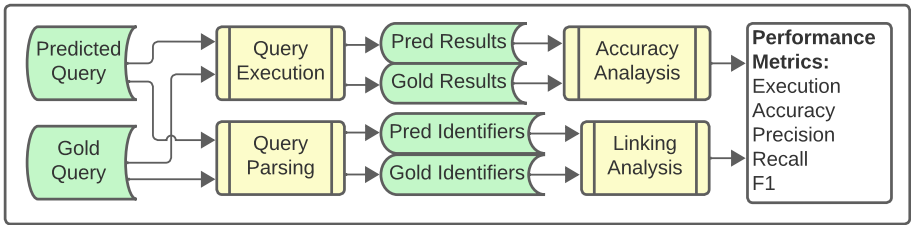


Figure 3.23. Benchmark results evaluation includes generated and gold query execution on target schemas, parser-based analysis, and identifier set comparisons. We evaluate performance in terms of execution accuracy and schema linking (precision, recall, and F1).

Key Takeaways

Overall, there is a model-dependent statistically significant correlation between identifier naturalness and execution accuracy, with smaller models exhibiting higher correlations between naturalness and performance. The presence of Least naturalness identifiers has the largest negative effect on schema linking. Additionally, while the performance difference between

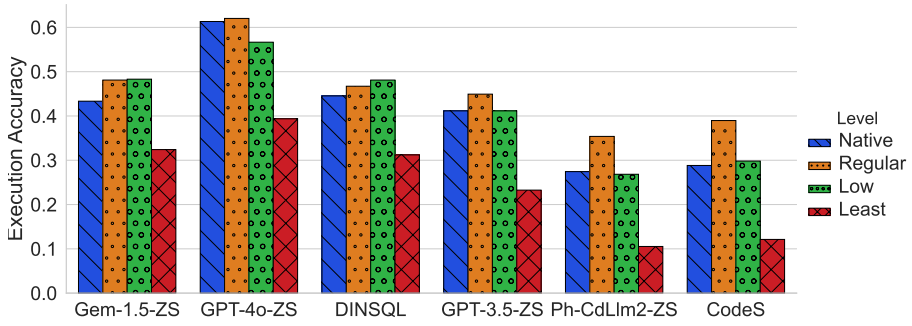


Figure 3.24. Execution accuracy (proportion of correct queries) by model. There is slight accuracy improvement from native schemas to schemas modified to regular naturalness. Accuracy drops significantly for schemas modified to low naturalness.

Regular and Low is visible, it is less impactful. So, modifying Least naturalness identifiers should be a higher priority than modifying Low naturalness identifiers.

3.8.1 Execution Accuracy

Execution Result Set Comparison

Execution accuracy is the standard measure of performance in most NL-to-SQL benchmarks [55, 119] where accuracy is determined using result set comparisons between gold and generated queries executed over one or more database instances. A drawback of existing methods is that strict set or bag comparisons risk increased false-negatives when a generated query includes additional fields that are not required, but do not render the result incorrect in terms of the natural language question [121, 25].

To reduce false negatives, the SNAILS approach to execution accuracy evaluation adopts 2 aspects of relaxed execution matching as described in [25]; it accounts for: (1) The possibility that a predicted query may contain additional columns beyond those retrieved by a gold query; and (2) That unless specified in the NL question, tuples may appear in any order. To achieve this, we perform result set-superset comparisons to ensure that the predicted result set column set is a superset of the gold result set column set. That is, a generated query is considered incorrect if it does not contain *all* gold query columns; but it is not considered incorrect (at this stage) if it includes columns not present in the gold query result. A more detailed description of this

approach is available in the appendix.

Manual Evaluation

Execution result set comparison cannot prove query correctness; so we rely on it only to rule out true negatives from further consideration. To validate correctness, the authors manually review generated queries that pass execution result set-superset comparison checks. We streamline this process by creating a Python-based manual validation user interface that makes the process of comparing gold and generated queries more user-friendly. Manual validation steps include ensuring the generated query answers the NL question, matches the gold query in terms of semantic structure, and does not contain semantically incorrect predicates, projections, or clauses.

Naturalness Effect on Execution Accuracy

Figure 3.24 shows execution accuracy for each LLM and naturalness level. There is a clear difference in overall performance between LLMs, most likely due to model size. We find that generally more natural database schemas yield more correct queries. Databases with more natural native schemas did not benefit from identifier renaming, though we observe that altering a schema to become less natural degrades accuracy in most cases. We find that for databases with Native schema combined naturalness scores less than 0.69, modifying the schema identifiers to increase naturalness improves execution accuracy.

Statistical Significance

The Kendall-Tau correlation between the naturalness of identifiers in a query and execution accuracy ranges from low ($\tau = 0.09, p < 0.0001$) for Gemini 1.5, to moderate ($\tau = 0.19, p < 0.0001$) for Phind-CodeLlama2 and CodeS. The most impactful relationship is between the presence of Least naturalness identifiers and performance, with Kendall-Tau correlations between the proportion of Least identifiers in a query and execution accuracy between $\tau = -.15$ and $\tau = -.22$ with $p < 0.0001$ for all models.

3.8.2 Schema Linking Evaluation

We make schema linking a "first class citizen" of our analysis, and study schema linking performance in queries irrespective of other aspects of correctness. Thus, we propose query-level and identifier-level schema linking measurements. We propose an approach similar to the Spider benchmark exact set matching system [119] in which we employ a schema linking-specific evaluation method using *recall* scoring of gold and generated query pairs. Other schema linking-focused research measure effects of schema linking improvements using ablation [113, 12, 115, 101]. In other cases, schema linking is described in post-hoc analysis of NL-to-SQL model performance, with schema linking accounting for roughly 30% of failures [21, 84].

Query-Level Linking Analysis

The set of all schema identifiers (table and column names) present in gold queries represents the minimum identifiers required to correctly answer an NL question. Our purpose-built ANTLR4-based [81] query parser extracts identifiers from gold and generated queries. With a set QI_g of identifiers present in the gold query and a set of identifiers QI_p present in the generated (or predicted) query, we calculate recall, as well as F1 and precision.

$$QueryRecall = \frac{|QI_g \cap QI_p|}{|QI_g|} \tag{3.3}$$

$$QueryPrecision = \frac{|QI_g \cap QI_p|}{|QI_p|} \tag{3.4}$$

$$QueryF1 = \frac{2(QueryRecall * QueryPrecision)}{QueryRecall + QueryPrecision} \tag{3.5}$$

We exclude 137 linking score calculations from analysis in situations where the predicted query contains invalid SQL that prevents query parsing and identifier extraction. We use recall as the primary measure for schema linking, as it does not penalize generated queries that contain extra

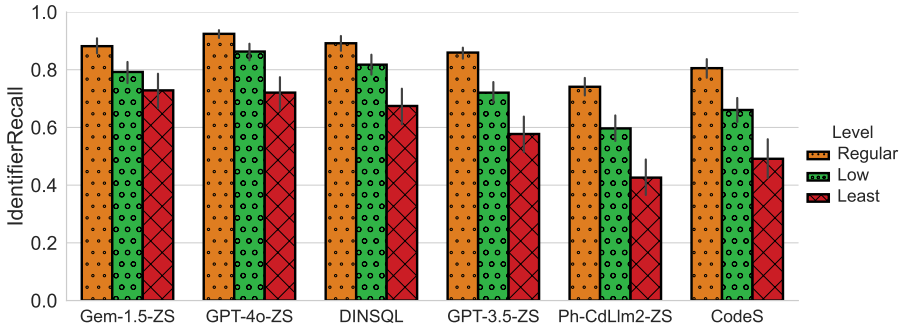


Figure 3.25. Native identifier recall scores by model and naturalness level. Error bars set with confidence interval of 0.95. For all models, identifiers in lower naturalness categories yield lower recall scores.

identifiers that do not render an answer incorrect in our setting, such as cases when an arbitrary column is referenced in a count function. Charts and tables depicting F1 and precision scores are available in the appendix.

Identifier-Level Linking Analysis

For an identifier-focused (rather than query-focused) metric, we perform identifier-level linking analysis. We derive recall linking scores for each Native schema identifier I as follows. I_{match} is the count of instances when I is correctly present in a predicted query. I_{gold} is the count of gold queries that contain I.

$$IdentifierRecall = \frac{I_{match}}{I_{gold}} \tag{3.6}$$

Figure 3.25 visualizes *IdentifierRecall* of Native identifiers in each naturalness level, and for each LLM. The chart indicates an observable difference in *IdentifierRecall* scores for each naturalness level, with *IdentifierRecall* increasing for higher naturalness levels. These results remain consistent relative to overall model performance across all 5 LLMs and various workflows.

Naturalness Effect on Schema Linking

Overall, we find that schema naturalness has a model-dependent and significant effect on schema linking performance with the highest correlations between *QueryRecall* and query

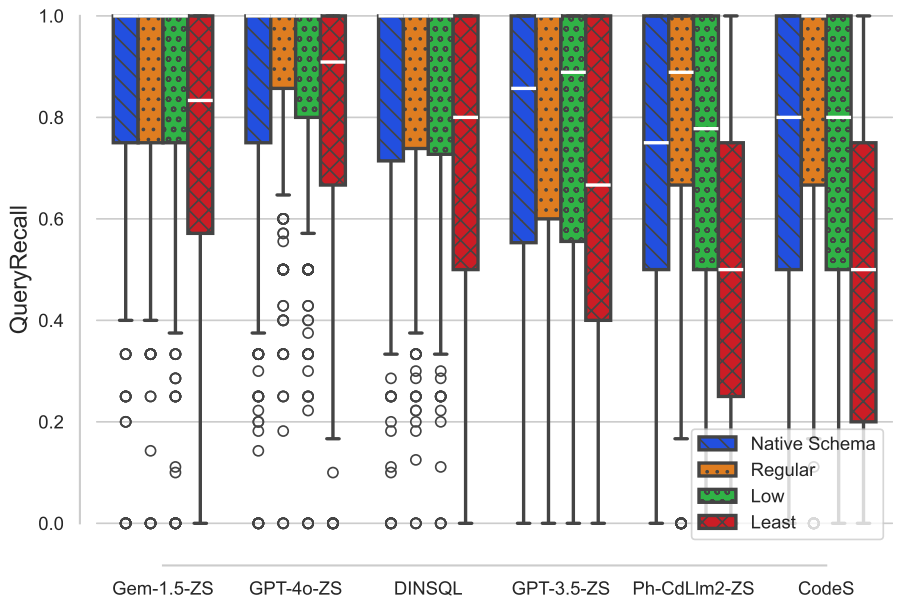


Figure 3.26. Schema linking performance across database schema naturalness levels generally yields equal or better performance for higher levels of naturalness, with open source models Phind-CodeLlama2 (Ph-CdLlm2-ZS) and CodeS as well as OpenAI's GPT-3.5 (GPT-3.5-ZS) exhibiting higher sensitivity to changes in naturalness. Zero-shot prompting NL-to-SQL methods are denoted as (ZS).

naturalness occurring with the open-source CodeLlama and CodeS models, and the lowest (though still significant) correlations occurring with Google's SoTA Gemini 1.5 Pro and OpenAI's GPT-4o models. The more complex DIN SQL and CodeS workflow *QueryRecall* results are also significantly affected by naturalness level differences.

Both DIN SQL and the CodeS complex NL-to-SQL workflows are sensitive to changes in naturalness, suggesting that these more complex workflows by themselves do not overcome schema naturalness effects. We also see that execution accuracy differences between the GPT-4o zero-shot prompting method and the DINSQL prompt chaining method suggest that applying more complex workflows to high-performing LLMs may be counterproductive for more recent SoTA LLMs.

Figure 3.26 illustrates *QueryRecall* across schema naturalness levels, and for each LLM. For GPT 3.5, Phind-CodeLlama2, and CodeS, we observe an improvement to *QueryRecall* when converting identifiers in a Native schema to Regular naturalness. This improvement did not manifest for Gemini and GPT-4o when observing the data in aggregate (i.e., between databases) due to their overall high performance relative to the other models, but improvements within databases of lower naturalness are still present (see Figure 3.27). The recall drop (approximately 20 percent decrease) associated with a modification from both Regular and Low to Least naturalness remains consistent across all LLMs.

Naturalness changes within specific SNAILS database schemas paints a clearer picture of the impact of naturalness. Figure 3.27 provides a drill-down view of the effect of schema modification on the PILB, SBOD, and NTSB schemas in terms of *QueryRecall*, and for each LLM and schema naturalness level. The center example (PILB) is a highly natural Native schema where schema naturalness modification would not be required. The leftmost example (NTSB) indicates linking performance improvement across all models for a native schema of lower naturalness converted to a higher naturalness schema, and presents a case where naturalness modification will improve NLI performance. The rightmost database (SBOD) represents a Least naturalness schema, and transformation from Native to Regular yields significant improvements

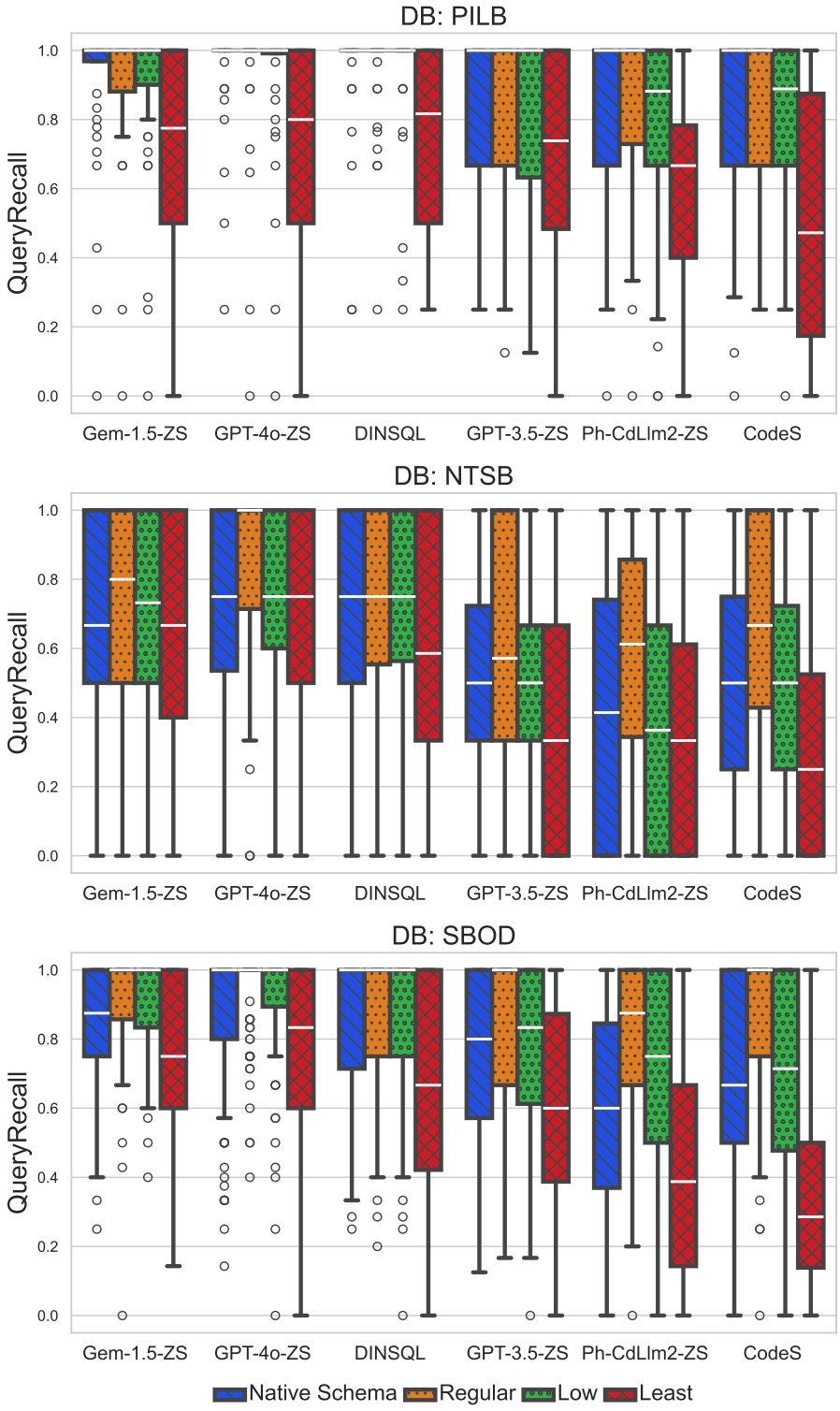


Figure 3.27. Schema linking performance (QueryRecall score) changes across 3 example databases' native and virtual schemas. We selected these 3 examples to showcase the diversity of the databases in our collection. PILB Native is a more natural schema with 65 percent Regular, 22 percent Low, and 13 percent Least; NTSB Native contains 42 percent Regular, 34 percent Low, and 24 percent Least; and SBOD Native is the lowest naturalness schema with 24 percent Regular, 49 percent Low, and 27 percent Least.

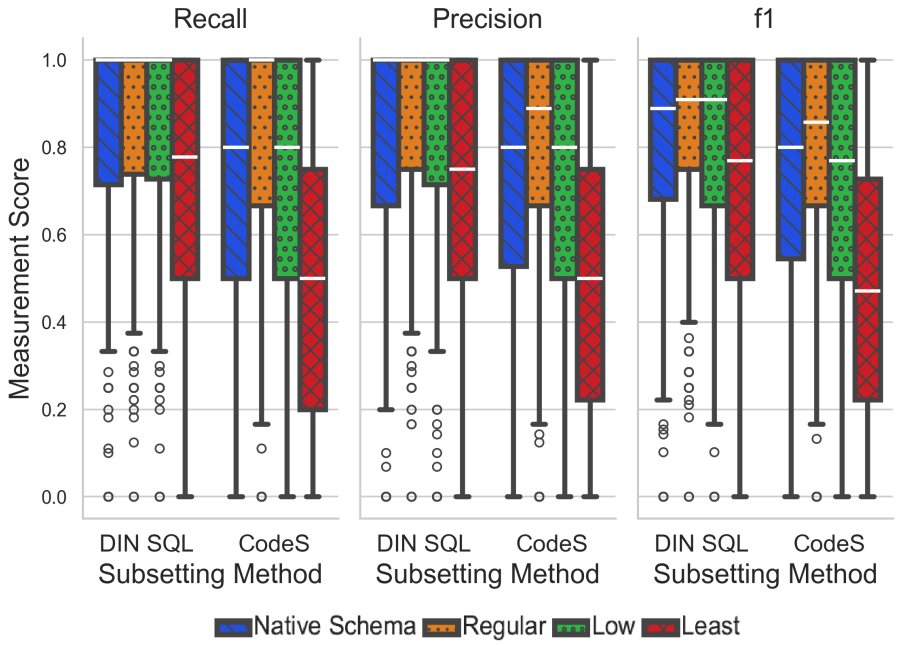


Figure 3.28. Schema subsetting performance, measured with recall, precision, and f1 score, varies by naturalness levels for both DIN SQL and CodeS. Measurement Score is Recall, Precision, or f1 respectively.

for all models. In all cases, we see that reducing naturalness to the Least level consistently degrades *QueryRecall*.

Statistical Significance

Kendall-Tau correlations between the proportion of Least identifiers and *QueryRecall* range from $\tau = -0.16$ (Gemini) to $\tau = -0.28$ (Phind-CodeLlama2), with $P < 0.001$ for all models. Both Regular and Low identifier proportions are significantly correlated with improved outcomes in terms of *QueryRecall*. Identifiers with Regular naturalness show the highest positive Kendall-Tau correlations ranging from $\tau = 0.07$ (Gemini) to $\tau = 0.20$ (Phind-CodeLlama2). Low naturalness identifier proportions correlate positively, but to a lesser degree, with Kendall-Tau values ranging from $\tau = 0.05$ (Phind-CodeLlama2) to $\tau = 0.07$ (Gemini).

Tables 3.46–3.80 indicate that proportions of a naturalness category within a set of query identifiers also have an effect on linking performance. Specifically, as the proportion of Regular naturalness identifiers increases, so does schema linking. We also observe that in most cases, as

the proportion of Low increases, schema linking generally improves, but not to the same degree as for Regular identifiers. The most striking effect comes from the proportion of Least identifiers, where as the proportion of Least naturalness increases, schema linking performance decreases.

Naturalness Effects on Schema Subsetting

We measure the schema subsetting (also known as schema filtering, or table retrieval) in terms of recall, precision, and f1 score, and present the results in Figure 3.28. We find that for the CodeS finetuned classifier approach, schema naturalness level differences result in observable differences in f1. For the DIN SQL LLM-based approach, naturalness effects are less pronounced, though still present, particularly for Least level schemas.

Performance Over Modified Spider Schemas

Figure 3.29 shows that with the SNAILS schema renaming artifacts applied to the Spider NL-to-SQL benchmark dev dataset [119], naturalness effects are the most significant between Low and Least levels of naturalness. Performance differences across naturalness levels for the highly natural Spider schemas resemble performance over similarly-natural schemas in the SNAILS collection.

Mean token to character ratio effect on schema linking

The Kendall-Tau correlation mean token-to-character ratio correlations (Tables 3.34 and 3.35) indicates that there is significant evidence that for GPT and Code Llama tokenizers over both native and modified schemas, that as the token-to-character ratio increases, schema linking performance decreases. This same observation does not hold for the Code Bison tokenizer ober the native schemas. Because of the inconsistent power of the token-character ratio measurement, we believe it may not serve as a viable proxy for naturalness in all cases.

3.9 Discussion and Limitations

The ability to assess the naturalness of existing schemas can inform the feasibility of "hooking up" an NL query interface to an existing database. We believe that practitioners who are

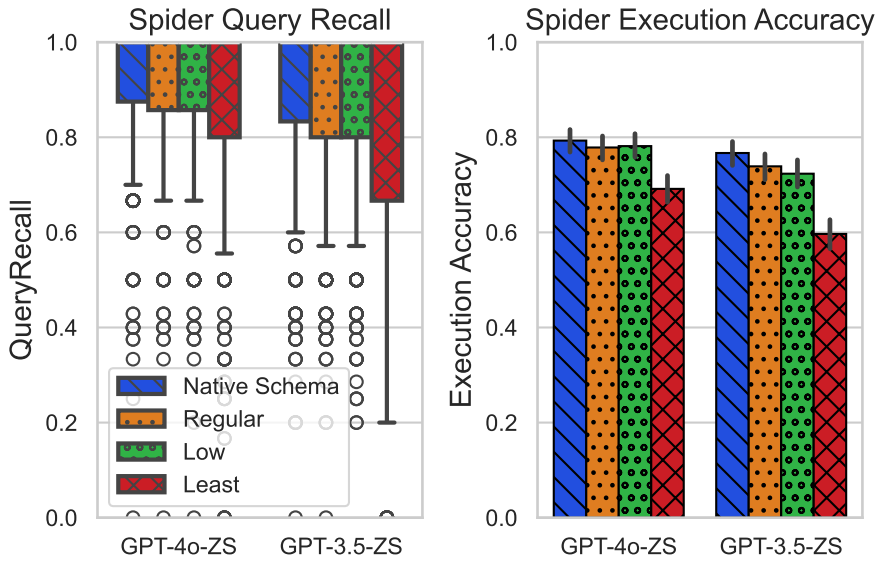


Figure 3.29. QueryRecall and Execution Accuracy differences over the Spider [119] dev set modified using SNAILS renaming artifacts.

considering the integration of an LLM into their database interaction workflows would benefit from making naturalness-focused schema analysis a key step in their integration process.

Other Naming Patterns in Real-World Schemas

To examine naming practices in the real-world, we classified the identifiers of SchemaPile dataset [20] with our CANINE-based classifier, and evaluated the identifiers for other LLM-unfriendly patterns. We observe that whitespace characters within schema identifiers contributes to identifier mutation during inference. That is, rather than encasing a whitespace-containing identifier with brackets or quotes, the LLM hallucinates the identifier into snake or camel case format. We find that in the SchemaPile collection, though whitespace is uncommon (less than 1 percent for both tables and columns), it appears in 808 columns and 63 tables, and is comparable to the proportions in the SNAILS dataset.

Another naming practice that yields disproportionate failures with some LLMs is the presence of the word *table* in the identifier name. In these instances, we find that the LLM tends to drop the word *table* from the name (e.g., table_employee becomes employee). There are over

700 identifiers (less than 1 percent of all identifiers) in the SchemaPile collection that employ this naming pattern.

These observations suggest that although these naming patterns are not necessarily a common occurrence in many real-world schema designs, they do appear in some cases. We suggest that practitioners would benefit from assessing the naming patterns of their database schemas.

Variations in LLM Sensitivity to Naturalness

There are many LLMs to select from for NLIDBs, and we can see even within the select 5 models in our work large variations in NL-to-SQL performance as well as the degree of sensitivity to schema naturalness. The Google Gemini and GPT-4o models demonstrate the highest overall performance, as well as the lowest sensitivity to naturalness differences between Regular and Low levels. Without access to the underlying model architectures and weights, it remains as a black box in our research, and we can merely speculate the reasons why it is not as affected by naturalness as the other 3 models in our study. Generally, we observe that the these models have an overall higher performance, and are less prone to linking errors such as selecting the incorrect identifier from the schema knowledge representation or committing a typo-like hallucination.

Though selecting the most performant model would seem to be an obvious course of action, competing factors such as an organization's policies, budget, or existing vendor contracts, may require the selection of a model that is more sensitive to schema naturalness differences. Thus, we believe that naturalness-aware NLI integration will remain important for at least the practitioners who use LLMs other than Gemini in the set that we have studied.

Modifying Existing Schemas

For already-existing schemas, renaming identifiers is generally a non-trivial effort, particularly for those databases for which documentation has been published and application interfaces have been integrated. Schema modifications may not be necessary (or helpful), if a schema is already classified as highly natural. DBAs should assess current naturalness levels

prior to committing to naming modifications. At a minimum, we recommend that any Least identifier be modified to a Regular naturalness level and, if feasible, Low identifiers as well. If renaming a less natural schema's identifiers is not feasible due to integration constraints, we suggest one of two approaches: 1) adopting a naturalness-as-a-view strategy by mapping Native identifiers to Regular naturalness identifiers using SQL views, or 2) a middleware approach that modifies schema knowledge in LLM prompts and generated SQL queries prior to execution on the database. We sketch a rough design of both options in the appendix.

We demonstrate a natural schema view proof of concept with our SNAILS database collection and their MS SQL Server instances. For each table and column in the collection's database schemas, we map the Native table or column to its Regular counterpart in the naturalness modified identifier dataset using SQL view creation DDL and a db_nl schema. This enables schema information retrieval for LLM-based NL-to-SQL prompting without prompt or generated query modification while still retaining the underlying Native schema naming patterns required for existing integrations.

In lieu of schema modificaftion, practitioners may elect to employ prompting techniques that augment schema representations with additional metadata or value samples. While these methods may improve schema linking performance in some contexts [67], they greatly increase schema representations on a per-identifier basis. Thus, the cost to do so is high in terms of token efficiency, latency, and implementation complexity, especially for very large schemas.

Designing New Schemas

For new schema development, our results show that making schema identifiers more natural from the start can make databases work better with LLMs. Specifically, database designers should try to avoid Least naturalness identifiers and would likely also benefit from limiting Low naturalness identifiers. Database practitioners can evaluate the naturalness of identifiers using the identifier naturalness classification techniques and model artifacts described in this paper and released publicly by us as part of the SNAILS collection.

Limitations

LLM research is advancing rapidly, and the LLMs represented in this paper may get superseded by newer versions or newer models (e.g., DBRX [108], Arctic [109]). But it does not negate our work's core value–the first in-depth characterization of how schema naturalness affects LLM-based NL-to-SQL–and our new labeled datasets, AI artifacts, and benchmarking framework can be used for future LLMs too. We leave it to future work to also include such very recent LLMs for further benchmark analyses.

We recognize that the correlation statistics indicate a moderate (in some cases only a weak) correlation between naturalness and *IdentifierRecall*. This suggests that other undiscovered factors also influence linking performance; and further research may reveal additional schema- and language-related correlations.

Our selection of 9 database schemas is of course not fully representative of *all* types of schemas available in the real-world. The SNAILS collection will benefit from continued growth in terms of both databases and NL-SQL pairs. We hope our open source datasets and artifacts can be built upon by the database and NLP communities to keep improving LLM-based NL-to-SQL.

Future Work

In addition to extending the SNAILS benchmark artifacts to include additional datasets and artifacts, we identify several NLP+DB directions for future work. First, we wish to ask why and how exactly do different naturalness levels alter schema linking performance so much? Is it due to the tokenization and embedding mechanics? If so, where in the latent space do these altered tokens end up, and how do the encoders make use of them? Second, why do the different foundational LLMs behave so differently? Is it related to their architectures, tokenization, (pre)training data, post-training finetuning process, or some other factors? We believe these open questions have the potential to lead to several interesting new lines of research at the DB and NLP intersection.

3.10 Practical Applications

It is clear that naturalness has an effect on multiple NL-to-SQL performance measurements, but what is less clear is what should be done about it. Adopting good schema naming practices, including the use of natural words, can be easily applied when designing *new* schemas, which makes the application of naturalness-based performance improvements relatively straightforward in these cases. For existing schemas, the challenge is much greater, as it is likely that external interfaces and documentation have coalesced around the database schema, making it difficult (or impossible) to change without overhauling external systems and artifacts.

3.10.1 For *New* Databases

We refer the reader to Section 3.3.1 for the descriptions of Regular, Low, and Least category criteria. Additionally, Table 3.1 provides some examples of database identifiers at each naturalness level.

When creating a new database schema, we recommend that designers apply the Regular definition criteria, where the identifier contains complete English words with no abbreviations or acronyms, or contains only acronyms in common usage. We also recommend avoiding the use of whitespace characters, as well as identifier type labels (e.g., *table* or *column*), as we observe that some LLMs tend to drop these words during NL-to-SQL inference.

3.10.2 For *Existing* Databases

Modifying existing database schemas directly is infeasible for a myriad of reasons ranging from external integrations to constraint management within the database. As such, we offer two viable approaches to making database interactions more natural: 1) schema and query modification middleware, and 2) a within-database natural view.

Schema Modification Middleware

This approach is the more complex of the two, but may be necessary in cases where practicioners do not have write access to the target database. This approach contains the following pre-processing steps:

1. Classify schema identifiers using *Artifact 3*.
2. For Low and Least identifiers classified in step 1: create Regular representations using *Artifact 5*.
3. Create a Native-to-Regular map (or crosswalk) for all identifiers using the output of step 2 (see *Artifact 4* for an example).

The output of the preprocessing steps is the Native-to-Regular map that maps every Native database identifier to a Regular representation. In the case where the Native identifier has a Regular classification, it should be mapped to itself.

The next step involves the development of a middleware that modifies schema knowledge during NL-to-SQL inference so that the LLM receives a Regular naturalness schema representation. It contains the following steps:

1. Modify prompt schema knowledge by replacing Low and Least identifiers with Regular representations drawn from the identifier map.
2. Incorporate modified schema knowledge into the NL-to-SQL workflows involving schema representations (e.g., schema filtering and SQL generation steps).
3. After SQL generation, modify the SQL query by replacing Regular naturalness representations of lower naturalness Native identifiers to enable compatibility with the Native schema.

The output of the inference-time steps is a SQL query that can be executed over the target Native database.

The SNAILS project repository contains a prototype of such a middleware system, which is incorporated into the NL-to-SQL workflow used in our experimental design.
The *nl_to_sql_inference_and_prompt_generation.py* file employs the *naturalize_prompt()* and *denaturalize_query()* functions to enable NL-to-SQL inference over natural schemas. While this is not an easily portable and standalone system, we encourage interested readers to trace the processes in these scripts for an example of a middleware solution.

Schema naturalization for LLM prompting is a fairly straightforward map lookup task. On the other hand, query "denaturalization" presents a more technical challenge due to the large variety of SQL queries that can be generated for a given NL question. To consistently replace identifiers in SQL queries, we create a Java-based SQL parser that supports both Sqlite and T-SQL syntax. This parser and query analyzer provides two important services: 1) query clause extraction, which we use for measuring query complexity; and 2) schema identifier tagging, which we use for query denaturalization. The latter feature (tagging) takes a SQL query as input, and returns the same query where all table and column names are encased within XML-like tags (e.g., <TABLE_NAME>*Customers*</TABLE_NAME>). We discuss this in more detail in Section 3.6.4.

Natural Views

The natural view concept is simple, but also very powerful. Rather than incorporating a relatively complex middleware strategy, for databases that support multiple schemas within an instance such as MS Sql Server we can create views that map a Regular naturalness representation of tables and their columns to their Native identifier counterparts within the base database schema. This approach is suitable when 2 main criteria can be met: 1) the user has schema and view creation privileges, and 2) the database supports multiple schemas for a database instance (e.g., a base *dbo* schema and a natural *db_nl* schema). Seperate schemas are required to avoid collisions between a natural view and a native schema where the native schema tables already have Regular naturalness levels.

The SNAILS project repository contains a prototype end-to-end natural view creation example that generates natural views for the SNAILS databases in a *db_nl* schema, which can be viewed and used by downloading the SNAILS real-world database collection (*artifact 1*). The *classify_rename_and_build_view.py* demonstrates the process of schema classification, identifier modification, and view creation over a target MS SQL Server database.

We first begin with the same pre-processing steps as the middleware approach where we:

1. Classify schema identifiers using *Artifact 3*.
2. For Low and Least identifiers classified in step 1: create Regular representations using *Artifact 5*.
3. Create a Native-to-Regular map (or crosswalk) for all identifiers using the output of step 2 (see *Artifact 4* for an example).

At this point, with the naturalness map (or crosswalk) as input, we generate a set of SQL view creation queries–one for each table in the schema. The resulting view query appears as follows:

```
CREATE VIEW db_nl.[table_deadwood] AS
  SELECT
    [Data_ID] AS [Data_ID],
    [Event_ID] AS [Event_ID],
    [OldPlot] AS [OldPlot],
    [Module] AS [Module],
    [Decay] AS [Decay],
    [MPD] AS [Midpoint_Diameter],
    [Length] AS [Length],
    [X_coord] AS [x_coordinate],
    [Y_coord] AS [y_coordinate]
```

```
FROM dbo.[tbl_Deadwood];
```

We make note of a few important aspects of the natural view: 1) Many identifiers map to themselves, as their Native naturalness is already Regular. 2) to avoid table name collisions, the views are mapped from the *dbo* schema to the *db_nl* schema. 3) This particular transformation contains an example of a poor naming habit (the word *table* in *table_deadwood*), and serves to remind us that we should typically review the output of the schema renamer and make necessary changes.

3.11 Additional Tables and Figures

The remaining pages contain several figures and tables of fine-grained analysis of dataset distributions and performance correlations.

Kendall-Tau Correlation Experiment Result Tables

Figures 3.34-3.44 provide Kendall-Tau experiment results for naturalness and token ratio correlations with linking performance (F1, Recall, and Precision).

- Figure 3.34: Token-Character ratio–Recall
- Figures 3.37–3.44: Combined naturalness–[Recall, F1, Precision]
- Figures 3.46–3.53: [Regular, Low, Least] naturalness–recall
- Figures 3.55–3.62: [Regular, Low, Least] naturalness–F1
- Figures 3.64–3.71: [Regular, Low, Least] naturalness–precision
- Figures 3.73–3.44: [Combined, Regular, Low, Least] naturalness–execution accuracy

Database-level Schema Linking Box and Whisker Plots

Figures 3.85-3.88 show additional database-level box and whisker plots depicting schema linking performance over individual database schemas and their naturalness levels for each LLM.

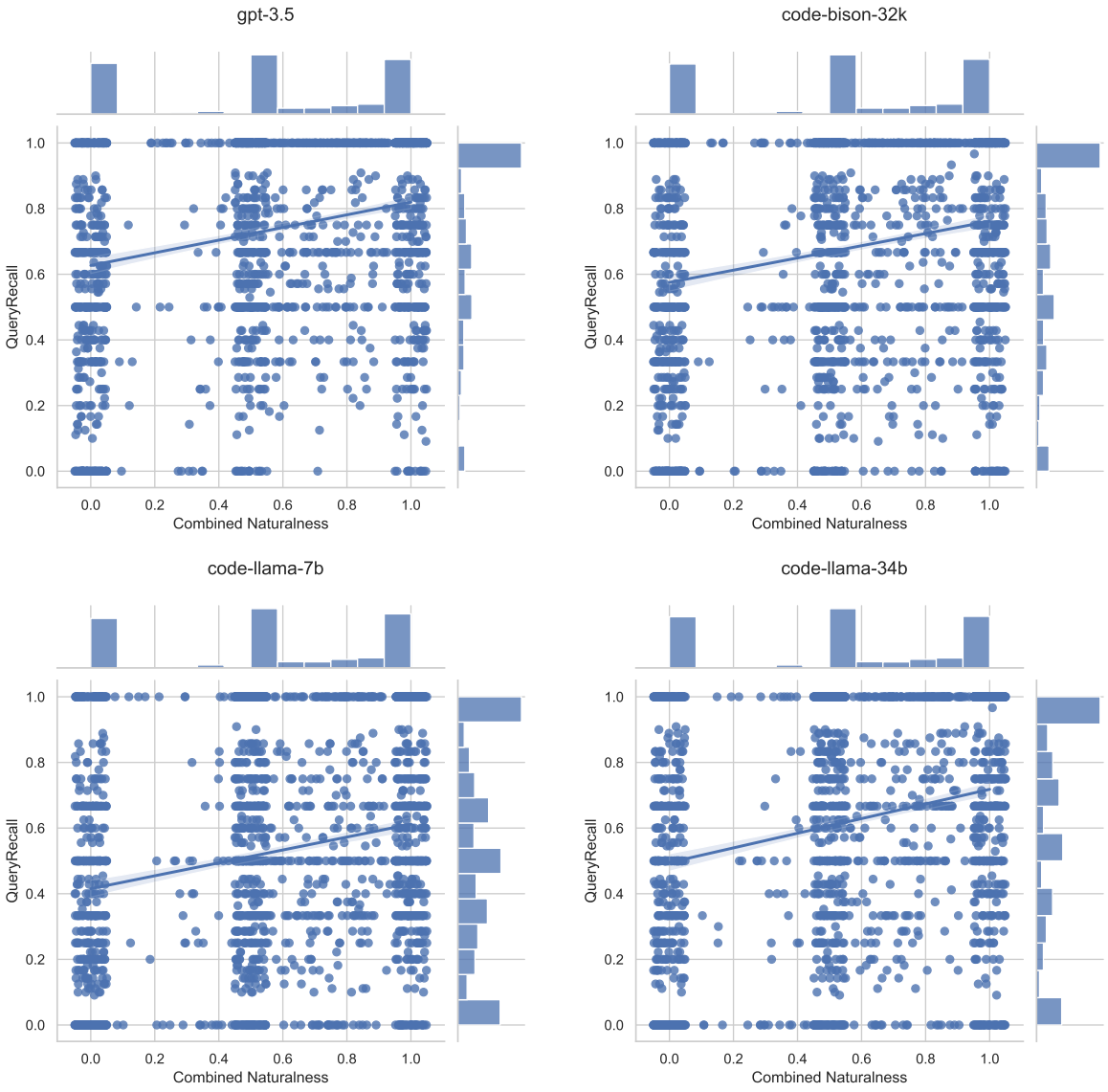


Figure 3.30. Schema linking performance (Recall Score) in terms of naturalness (Combined) by model on native schema identifiers. Histograms indicate non-parametric distributions of naturalness and Recall scores.

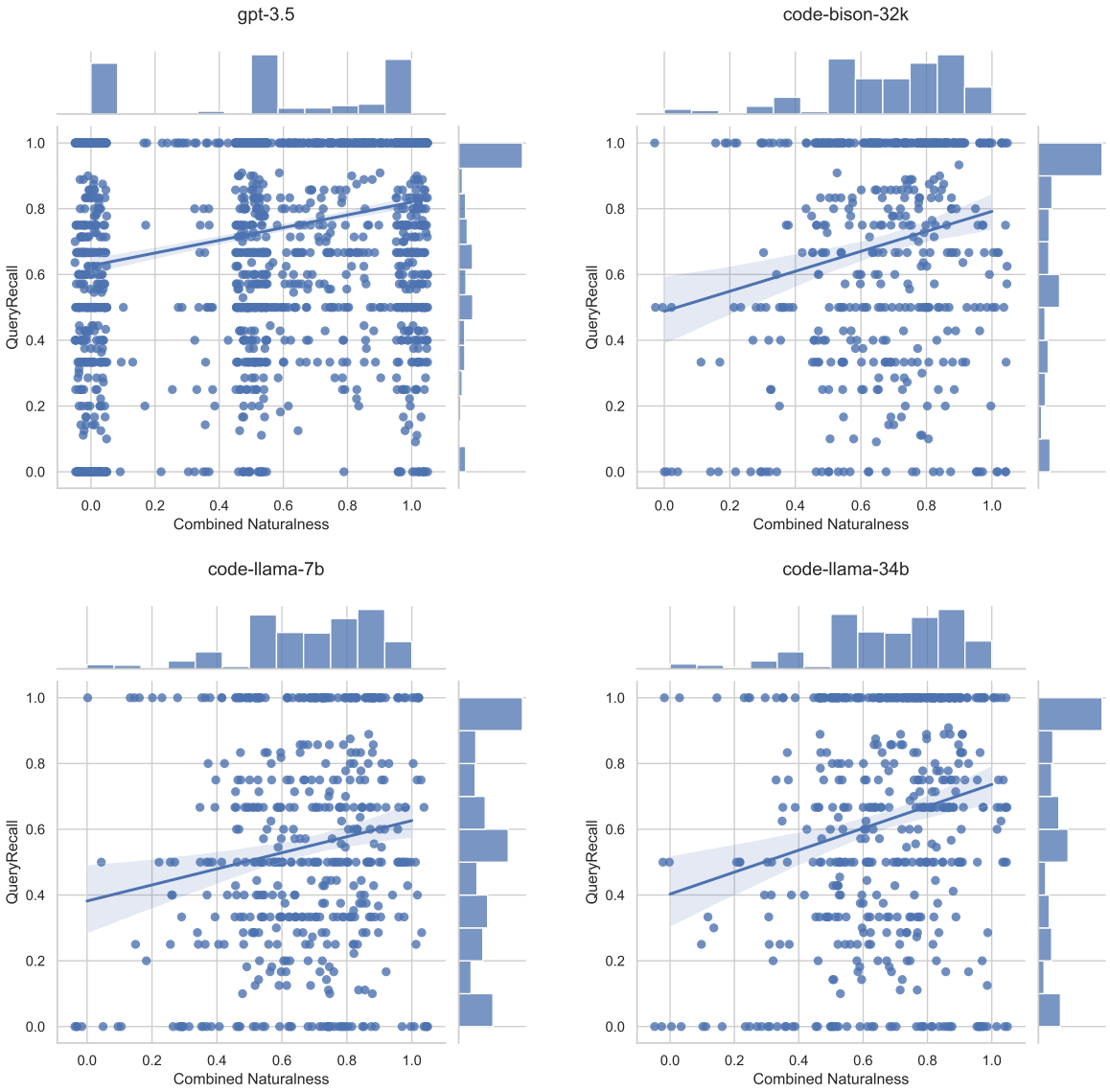


Figure 3.31. Schema linking performance (Recall Score) in terms of naturalness (Combined) by model on all schema identifiers (native and modified). Histograms indicate non-parametric distributions of naturalness and Recall scores with naturalness concentrations around modified schema identifier levels.

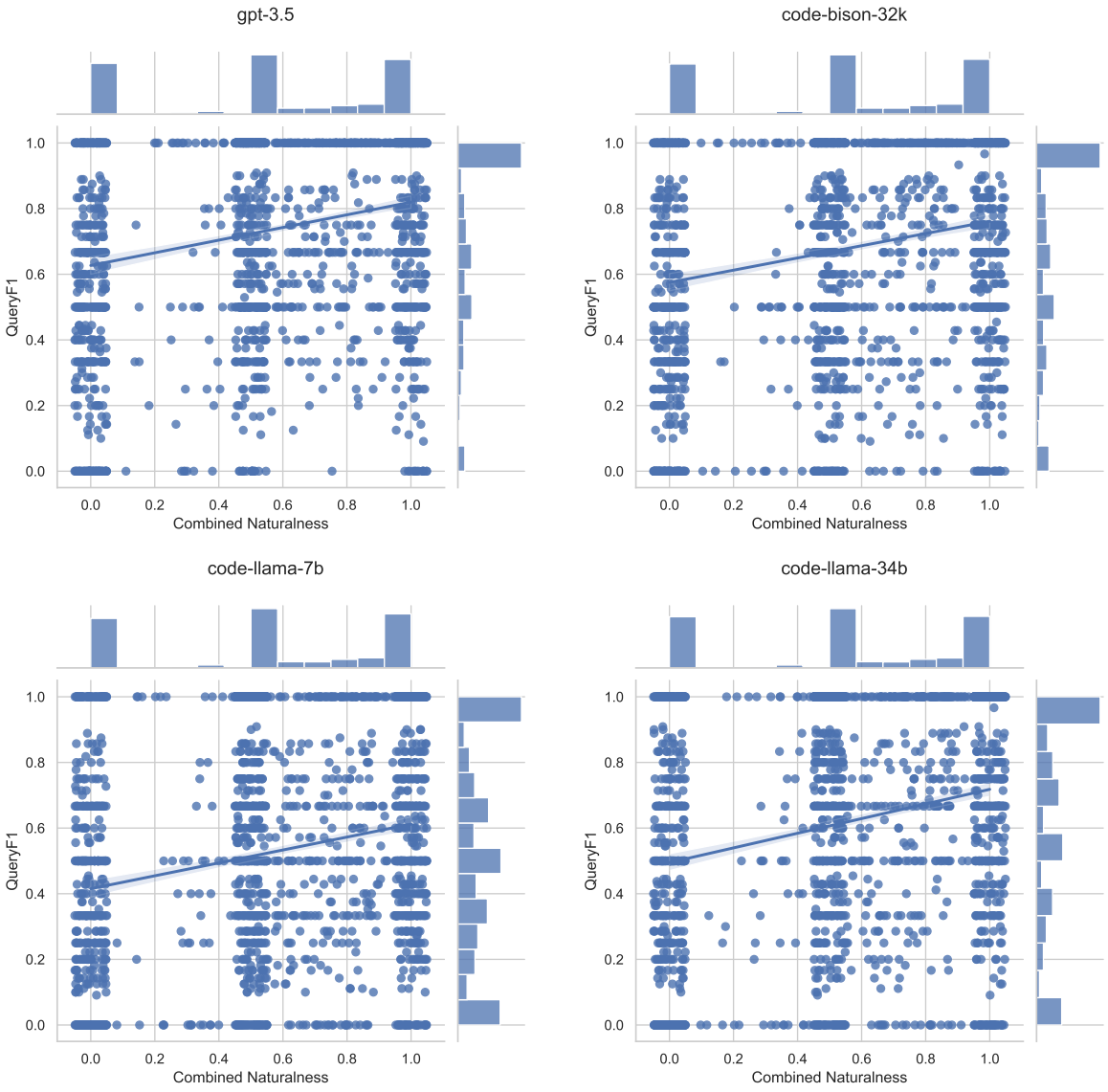


Figure 3.32. Schema linking performance (F1 Score) in terms of naturalness (Combined) by model on native schema identifiers. Histograms indicate non-parametric distributions of naturalness and F1 scores.

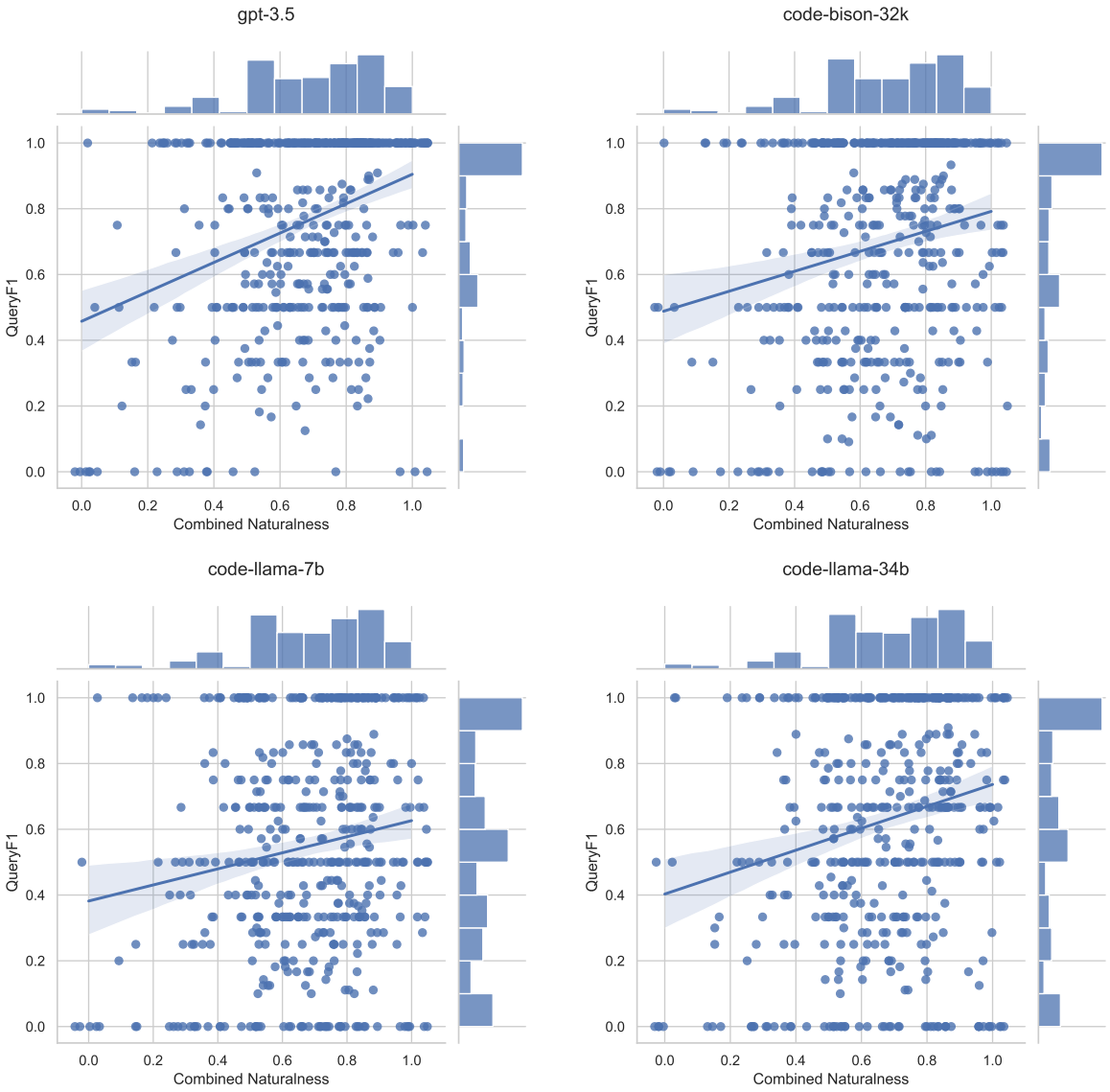


Figure 3.33. Schema linking performance (F1 Score) in terms of naturalness (Combined) by model on all schema identifiers (native and modified). Histograms indicate non-parametric distributions of naturalness and F1 scores with naturalness concentrations around modified schema identifier levels.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.142596	0.000023	492
gpt-4o	-0.125236	0.000182	512
DINSQL	-0.155634	0.000003	503
gpt-3.5	-0.259029	0.000000	500
Phnd-Llama2	-0.269220	0.000000	484
CodeS	-0.218149	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.130732	0.000000	1978
gpt-4o	-0.136292	0.000000	2009
DINSQL	-0.131631	0.000000	2007
gpt-3.5	-0.170698	0.000000	1998
Phnd-Llama2	-0.263657	0.000000	1936
CodeS	-0.270844	0.000000	2008

(b) All schemas (native + modified)

Figure 3.34. Kendall-Tau Correlations between the *Mean Token-to-Character Ratio* and *Query Recall*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.137770	0.000066	492
gpt-4o	0.146350	0.000020	512
DINSQL	0.182666	0.000000	503
gpt-3.5	0.209031	0.000000	500
Phnd-Llama2	0.254438	0.000000	484
CodeS	0.199335	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.113625	0.000000	1978
gpt-4o	0.154416	0.000000	2009
DINSQL	0.151862	0.000000	2007
gpt-3.5	0.171700	0.000000	1998
Phnd-Llama2	0.250113	0.000000	1936
CodeS	0.285891	0.000000	2008

(b) All schemas (native + modified)

Figure 3.35. Kendall-Tau Correlations between *Query Combined Naturalness* and *Query Recall*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.151056	0.000005	492
gpt-4o	0.130006	0.000069	512
DINSQL	0.137414	0.000027	503
gpt-3.5	0.214360	0.000000	500
Phnd-Llama2	0.253227	0.000000	484
CodeS	0.216096	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.110138	0.000000	1978
gpt-4o	0.141699	0.000000	2009
DINSQL	0.147045	0.000000	2007
gpt-3.5	0.185058	0.000000	1998
Phnd-Llama2	0.249083	0.000000	1936
CodeS	0.285834	0.000000	2008

(b) All schemas (native + modified)

Figure 3.36. Kendall-Tau Correlations between *Query Combined Naturalness* and *Query f1*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.165627	0.000001	492
gpt-4o	0.117021	0.000444	512
DINSQL	0.084476	0.012116	503
gpt-3.5	0.213515	0.000000	500
Phnd-Llama2	0.240645	0.000000	484
CodeS	0.222640	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.099675	0.000000	1978
gpt-4o	0.130333	0.000000	2009
DINSQL	0.149127	0.000000	2007
gpt-3.5	0.193870	0.000000	1998
Phnd-Llama2	0.240081	0.000000	1936
CodeS	0.279425	0.000000	2008

(b) All schemas (native + modified)

Figure 3.37. Kendall-Tau Correlations between *Query Combined Naturalness* and *Query Precision*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.117594	0.000720	492
gpt-4o	0.139545	0.000053	512
DINSQL	0.162703	0.000002	503
gpt-3.5	0.179904	0.000000	500
Phnd-Llama2	0.214458	0.000000	484
CodeS	0.155772	0.000003	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.069195	0.000265	1978
gpt-4o	0.113739	0.000000	2009
DINSQL	0.098608	0.000000	2007
gpt-3.5	0.122796	0.000000	1998
Phnd-Llama2	0.198193	0.000000	1936
CodeS	0.229038	0.000000	2008

(b) All schemas (native + modified)

Figure 3.38. Kendall-Tau Correlations between *Regular Identifier Proportion* and *Query Recall*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.020559	0.555850	492
gpt-4o	-0.033719	0.330627	512
DINSQL	-0.027539	0.424527	503
gpt-3.5	-0.012805	0.704874	500
Phnd-Llama2	-0.031016	0.359014	484
CodeS	0.019808	0.553404	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.073173	0.000115	1978
gpt-4o	0.044010	0.020955	2009
DINSQL	0.084950	0.000006	2007
gpt-3.5	0.073867	0.000058	1998
Phnd-Llama2	0.055309	0.002677	1936
CodeS	0.073957	0.000049	2008

(b) All schemas (native + modified)

Figure 3.39. Kendall-Tau Correlations between *Low Identifier Proportion* and *Query Recall*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.165916	0.000010	492
gpt-4o	-0.154217	0.000032	512
DINSQL	-0.207309	0.000000	503
gpt-3.5	-0.239186	0.000000	500
Phnd-Llama2	-0.285738	0.000000	484
CodeS	-0.255086	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.158525	0.000000	1978
gpt-4o	-0.174026	0.000000	2009
DINSQL	-0.198730	0.000000	2007
gpt-3.5	-0.212258	0.000000	1998
Phnd-Llama2	-0.279290	0.000000	1936
CodeS	-0.310967	0.000000	2008

(b) All schemas (native + modified)

Figure 3.40. Kendall-Tau Correlations between *Least Identifier Proportion* and *Query Recall*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.131560	0.000074	492
gpt-4o	0.139678	0.000022	512
DINSQL	0.123802	0.000172	503
gpt-3.5	0.182707	0.000000	500
Phnd-Llama2	0.226842	0.000000	484
CodeS	0.173666	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.070395	0.000111	1978
gpt-4o	0.109530	0.000000	2009
DINSQL	0.104732	0.000000	2007
gpt-3.5	0.137642	0.000000	1998
Phnd-Llama2	0.201929	0.000000	1936
CodeS	0.231121	0.000000	2008

(b) All schemas (native + modified)

Figure 3.41. Kendall-Tau Correlations between *Regular Identifier Proportion* and *Query f1*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.027036	0.417304	492
gpt-4o	-0.064241	0.051686	512
DINSQL	-0.021415	0.517452	503
gpt-3.5	-0.016076	0.623934	500
Phnd-Llama2	-0.061160	0.059740	484
CodeS	0.009374	0.771977	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.062756	0.000573	1978
gpt-4o	0.036763	0.043163	2009
DINSQL	0.068013	0.000170	2007
gpt-3.5	0.066247	0.000221	1998
Phnd-Llama2	0.044565	0.012124	1936
CodeS	0.075927	0.000017	2008

(b) All schemas (native + modified)

Figure 3.42. Kendall-Tau Correlations between *Low Identifier Proportion* and *Query f1*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.158945	0.000009	492
gpt-4o	-0.095678	0.006756	512
DINSQL	-0.136164	0.000124	503
gpt-3.5	-0.243548	0.000000	500
Phnd-Llama2	-0.240116	0.000000	484
CodeS	-0.244016	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.147469	0.000000	1978
gpt-4o	-0.149285	0.000000	2009
DINSQL	-0.175018	0.000000	2007
gpt-3.5	-0.220154	0.000000	1998
Phnd-Llama2	-0.263193	0.000000	1936
CodeS	-0.305808	0.000000	2008

(b) All schemas (native + modified)

Figure 3.43. Kendall-Tau Correlations between *Least Identifier Proportion* and *Query f1*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.140165	0.000039	492
gpt-4o	0.118622	0.000406	512
DINSQL	0.070004	0.038987	503
gpt-3.5	0.177250	0.000000	500
Phnd-Llama2	0.220023	0.000000	484
CodeS	0.174968	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.063466	0.000685	1978
gpt-4o	0.103105	0.000000	2009
DINSQL	0.110165	0.000000	2007
gpt-3.5	0.148501	0.000000	1998
Phnd-Llama2	0.196175	0.000000	1936
CodeS	0.225452	0.000000	2008

(b) All schemas (native + modified)

Figure 3.44. Kendall-Tau Correlations between *Regular Identifier Proportion* and *Query Precision*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.025641	0.453269	492
gpt-4o	-0.040198	0.232741	512
DINSQL	0.014076	0.679299	503
gpt-3.5	0.003413	0.919671	500
Phnd-Llama2	-0.060758	0.065561	484
CodeS	0.028242	0.397195	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.054467	0.003577	1978
gpt-4o	0.034060	0.066693	2009
DINSQL	0.067157	0.000305	2007
gpt-3.5	0.056122	0.002463	1998
Phnd-Llama2	0.041782	0.020555	1936
CodeS	0.084470	0.000003	2008

(b) All schemas (native + modified)

Figure 3.45. Kendall-Tau Correlations between *Low Identifier Proportion* and *Query Precision*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.170160	0.000004	492
gpt-4o	-0.097043	0.007088	512
DINSQL	-0.093633	0.010346	503
gpt-3.5	-0.248686	0.000000	500
Phnd-Llama2	-0.215408	0.000000	484
CodeS	-0.248516	0.000000	501

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.135749	0.000000	1978
gpt-4o	-0.136779	0.000000	2009
DINSQL	-0.170463	0.000000	2007
gpt-3.5	-0.222732	0.000000	1998
Phnd-Llama2	-0.247999	0.000000	1936
CodeS	-0.300108	0.000000	2008

(b) All schemas (native + modified)

Figure 3.46. Kendall-Tau Correlations between *Least Identifier Proportion* and *Query Precision*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.031553	0.402105	503
gpt-4o	0.117580	0.001612	513
DINSQL	0.050126	0.183182	503
gpt-3.5	0.130571	0.000526	503
Phnd-Llama2	0.133928	0.000376	503
CodeS	0.100636	0.007539	503

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.051044	0.013769	2012
gpt-4o	0.118405	0.000000	2022
DINSQL	0.055010	0.007940	2012
gpt-3.5	0.113802	0.000000	2012
Phnd-Llama2	0.156595	0.000000	2012
CodeS	0.154301	0.000000	2012

(b) All schemas (native + modified)

Figure 3.47. Kendall-Tau Correlations between *Regular Identifier Proportion* and *Execution Accuracy*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.067135	0.075780	503
gpt-4o	-0.050377	0.178385	513
DINSQL	0.005675	0.880684	503
gpt-3.5	-0.013996	0.711227	503
Phnd-Llama2	-0.051436	0.173713	503
CodeS	0.007070	0.851558	503

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.065896	0.001482	2012
gpt-4o	0.044061	0.032993	2022
DINSQL	0.069761	0.000767	2012
gpt-3.5	0.059034	0.004410	2012
Phnd-Llama2	0.025475	0.219203	2012
CodeS	0.036122	0.081613	2012

(b) All schemas (native + modified)

Figure 3.48. Kendall-Tau Correlations between *Low Identifier Proportion* and *Execution Accuracy*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.170360	0.000027	503
gpt-4o	-0.114226	0.004359	513
DINSQL	-0.092923	0.021956	503
gpt-3.5	-0.209644	0.000000	503
Phnd-Llama2	-0.167881	0.000035	503
CodeS	-0.185003	0.000005	503

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	-0.153130	0.000000	2012
gpt-4o	-0.171591	0.000000	2022
DINSQL	-0.141735	0.000000	2012
gpt-3.5	-0.193995	0.000000	2012
Phnd-Llama2	-0.210597	0.000000	2012
CodeS	-0.221692	0.000000	2012

(b) All schemas (native + modified)

Figure 3.49. Kendall-Tau Correlations between *Least Identifier Proportion* and *Execution Accuracy*.

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.077353	0.038579	503
gpt-4o	0.115466	0.001818	513
DINSQL	0.059435	0.111967	503
gpt-3.5	0.161218	0.000016	503
Phnd-Llama2	0.151421	0.000051	503
CodeS	0.136525	0.000259	503

(a) Native schemas

Model	Kendall-Tau	P Value	n
gemini-1.5-pro	0.095640	0.000002	2012
gpt-4o	0.154441	0.000000	2022
DINSQL	0.096971	0.000001	2012
gpt-3.5	0.156622	0.000000	2012
Phnd-Llama2	0.195768	0.000000	2012
CodeS	0.196615	0.000000	2012

(b) All schemas (native + modified)

Figure 3.50. Kendall-Tau Correlations between *Query Combined Naturalness* and *Execution Accuracy*.

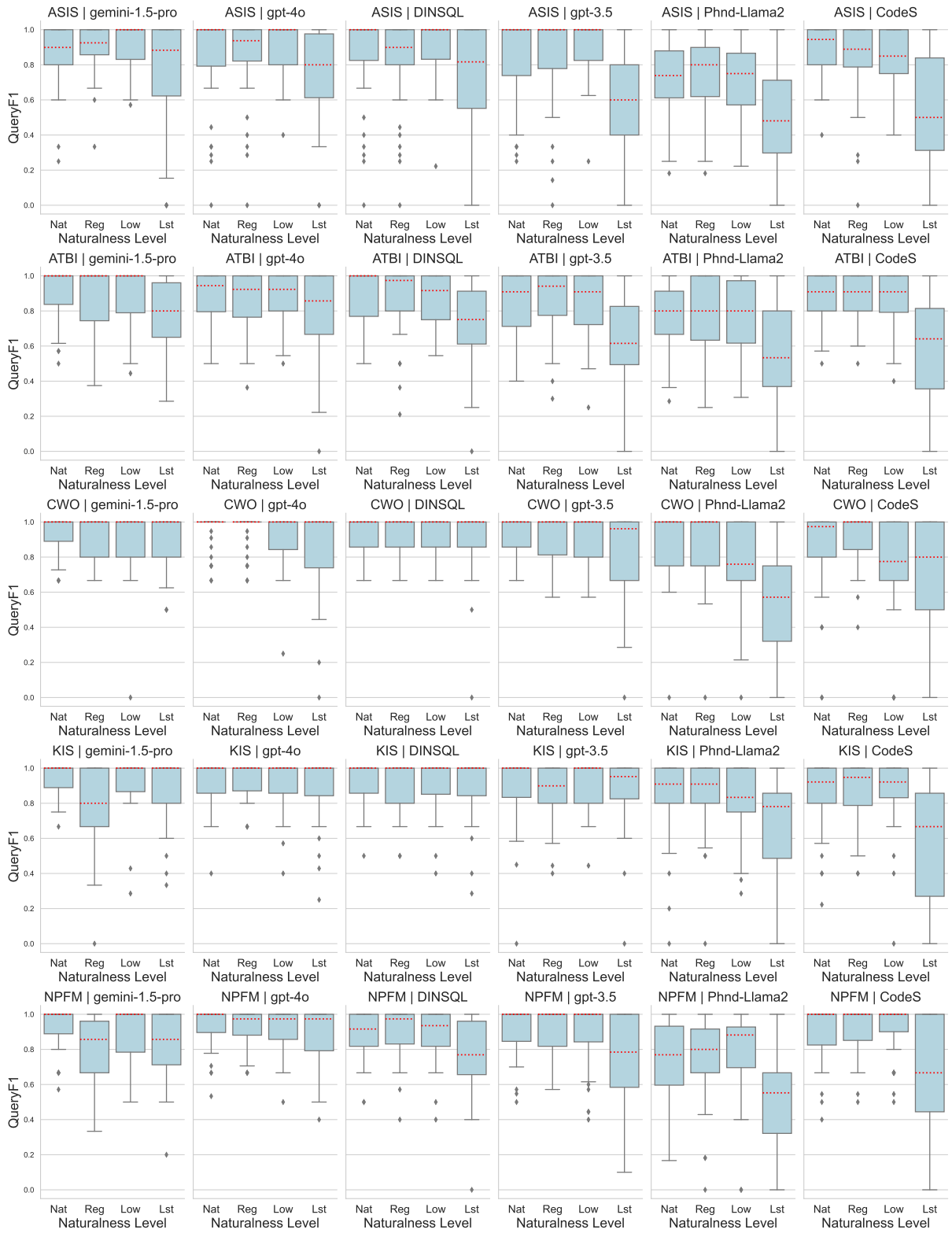


Figure 3.51. Schema linking performance (F1 score) changes across database naturalness levels.

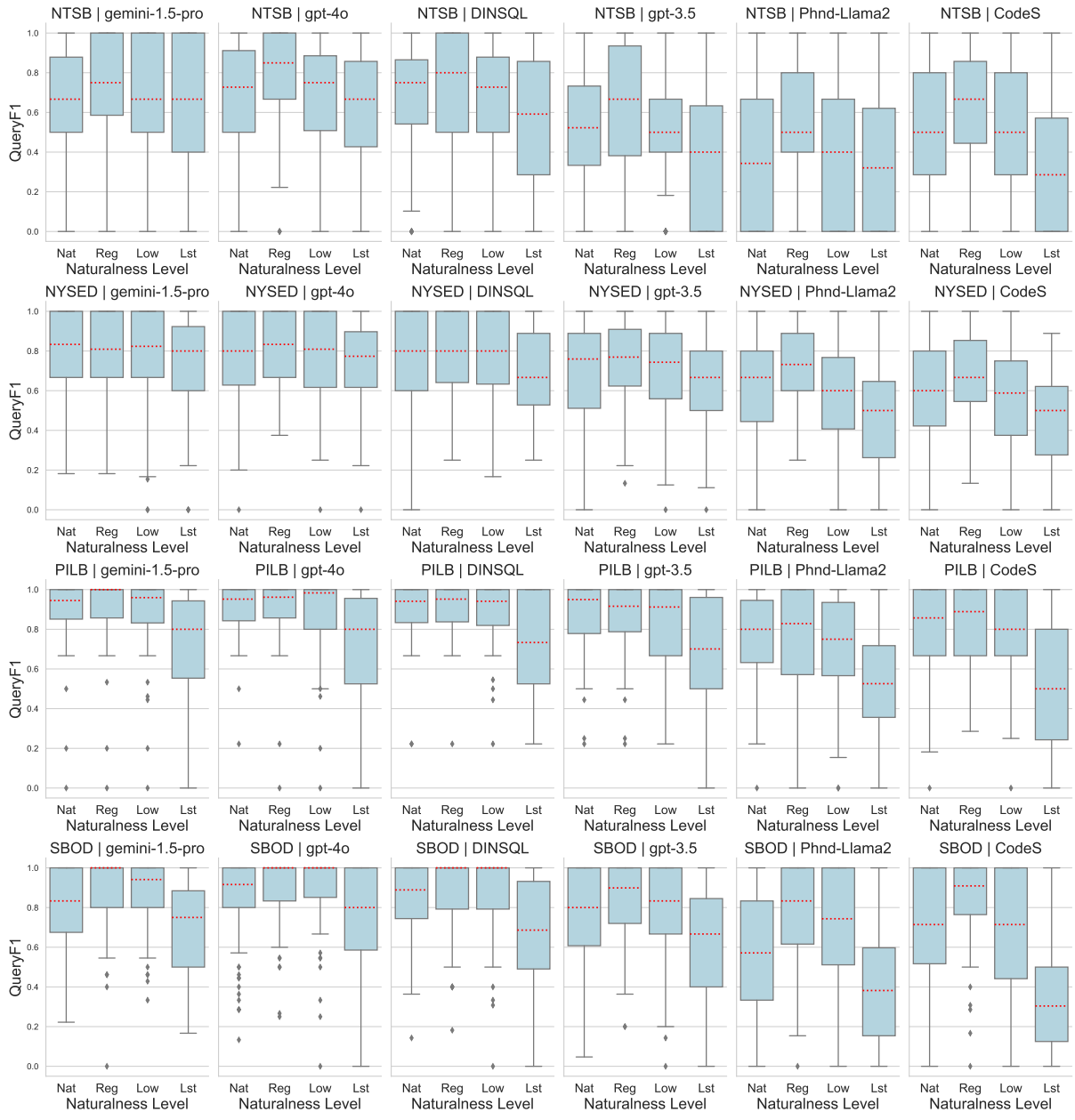


Figure 3.52. Schema linking performance (F1 score) changes across database naturalness levels.

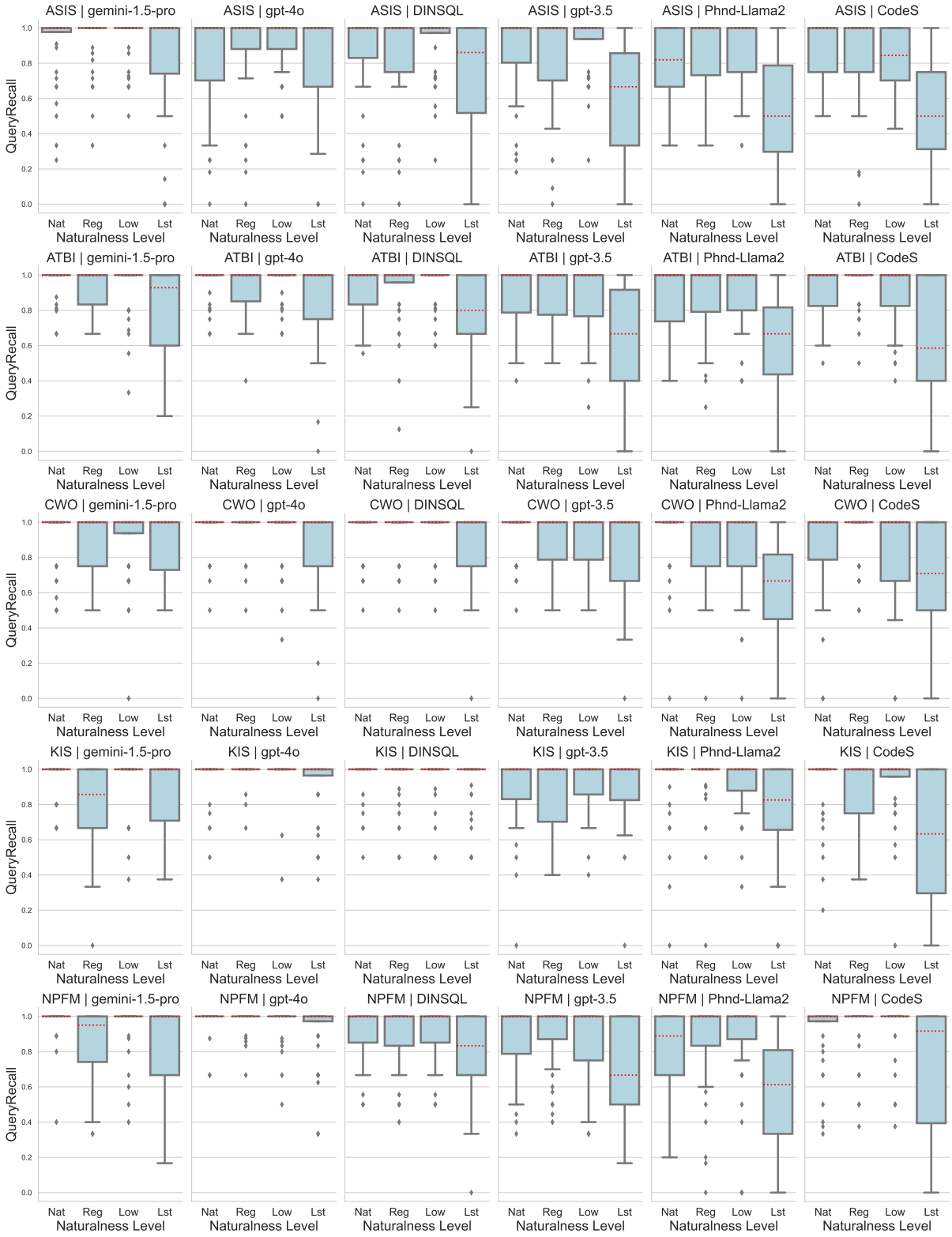


Figure 3.53. Schema linking performance (Recall score) changes across database naturalness levels.

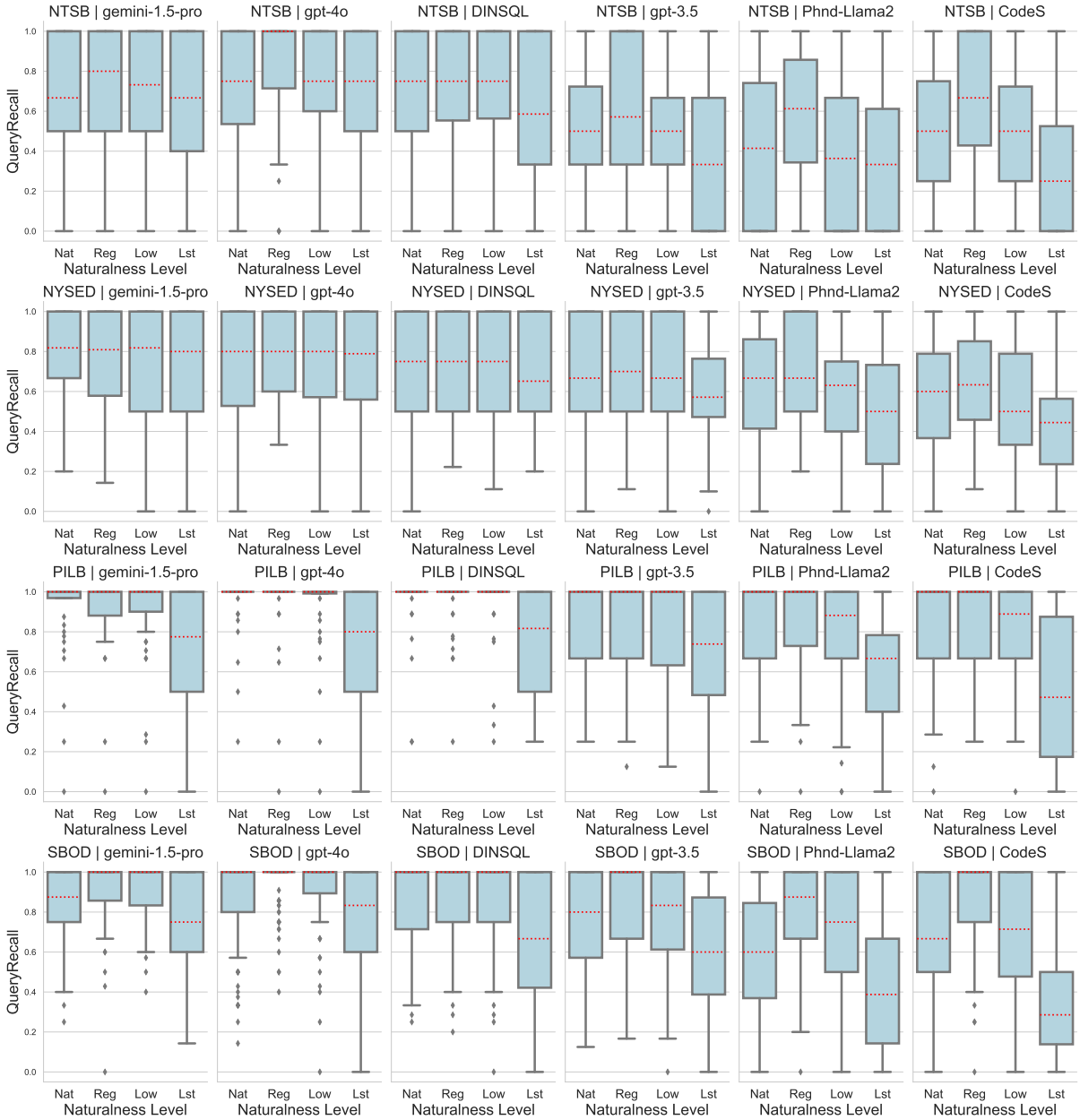


Figure 3.54. Schema linking performance (Recall score) changes across database naturalness levels.

Chapter 4

SKALPEL: Schema Knowledge Adjustments for LLM NL-to-SQL Performance Enhancements in Large Databases

4.1 Introduction

In the pursuit of better natural language interfaces with databases, Large Language Models (LLMs) have become the defacto standard for translating natural language questions into SQL queries. LLMs, especially those not finetuned on target database schemas, require LLM prompts to include text-based schema knowledge representations in order to avoid confabulation of non-existent schema elements and to ensure generated SQL queries contain valid schema identifiers. While including schema knowledge representations in prompts to modern LLMs is not a problem for moderately-sized schemas, some real-world databases including several enterprise resource planning systems, contain thousands of tables and tens of thousands of columns. In these cases, schema knowledge representations that include supplemental information such as column and table descriptions can exceed even SoTA LLM context window limitations, which can significantly degrade or eliminate the ability of an LLM to generate valid SQL queries.

In addition to the hard constraint of context window limitations, even the newest LLMs are still susceptible to degraded information recall due to large inputs. This tendency is commonly referred to as the *needle in a haystack* problem, that describes a model's ability (or failure) to retrieve information from input prompts [35]. Although some recent large context models such as GPT4.1 claim to have successfully mitigated needle in a haystack-related errors, the ability of a model to perform more complex tasks more akin to determining the usefulness of a table or column to a natural language question, such as *multi-round co-reference resolution* (MRCR) [112], are still negatively affected by high token counts [75].

The idea of reducing schema representation in LLM prompts to improve LLM-based SQL generation has taken form in several high-ranking NL-to-SQL methods on recent benchmark submissions, and some recent analysis of LLM-based schema linking methods provides a mixed view of the costs and benefits of such approaches. Our concurrent and complementary analysis of schema subsetting takes the body of schema research work even farther by asking two questions: 1) *of the multiple existing schema subsetting methods, which one is "the best" and has the most*

positive impact on NL-to-SQL? and 2) *even though schema subsetting seems to have only a marginal effect (at best) on NL-to-SQL performance over smaller benchmark database schemas, what effect does it have on very large database schemas?*

Our Focus

In this paper, we extend the field of recent schema linking research by examining 7 real-world schema subsetting modules reproduced from NL-to-SQL project source code, and testing them over 3 benchmark datasets including Bird, the emergent Spider 2, and schema linking-focused SNAILS–making this the first work to evaluate real-world schema linking modules, using schema linking-focused performance and efficiency metrics, over very large database schemas. Using the lessons learned from this analysis, we also present a prototype hybrid schema linking method which we call SKALPEL to motivate additional research toward more effective and efficient schema linking.

Schema Subsetting (AKA Schema Linking)

Schema subsetting (also known as schema filtering, schema linking, structured grounding, or entity retrieval) is an approach used by many database NLIs to reduce schema knowledge size as a method for avoiding context window limitations and mitigating the needle in the haystack problem that can occur with very large prompts. With LLM-based NL-to-SQL launching the usefulness of natural language interfaces (NLI) to databases into the realm of the practical, attempts to further improve the quality of SQL inference have begun to focus on sub-problems including schema linking. Recent analysis of schema linking usefulness has produced mixed results. On one hand, given an oracle (perfect) schema recall, decreasing the false positive rate (precision) generally improves NL-to-SQL generation for smaller open source LLMs [43]. On the other hand, schema linking proves to be less useful without oracle knowledge, and can even be detrimental to NL-to-SQL generation when the cost of omitting required tables and columns outweights the benefit of reduced schema representations in LLM prompts. This is primarily the case for the highest performing "flagship" LLMs [60]. Current schema linking methods

and analysis has relied on the Bird [54] and Spider [120] benchmarks which have schema sizes smaller than many real-world systems.

NL-to-SQL Benchmarks

The availability and difficulty of NL-to-SQL benchmarks have evolved at a pace relatively aligned with the advancement of NL-to-SQL systems. BIRD [120] is currently the most active benchmark, and offers a level of complexity that has posed a challenge for the current generation of LLMs and systems to achieve human-level performance. The BIRD benchmark consists of 95 databases, over 12,000 question-SQL pairs, and 37 domains. These data are split across a training, dev, and test set with the training and dev set available to the public.

Spider [120] was the most active NL-to-SQL benchmark at the beginning of the LLM-based NL-to-SQL era, and provides an interesting view of the associated advancement in NL-to-SQL capability over time. Spider has since reached benchmark saturation, and Spider 2.0 [49] supersedes it as the newest NL-based database interaction benchmark, focusing on multi-step data retrieval over large and complex schemas. Unlike its predecessors, Spider 2.0 does not provide dev or test set question-SQL pair data for model training. Instead, the authors make a subset of the test data question-SQL pairs available (256 total) to assist developers with prompt design. Spider 2.0 extends the set of target databases beyond Sqlite and also includes both Snowflake and BigQuery databases.

The SNAILS [57] project contains a benchmark dataset of 9 databases and 503 question-SQL pairs designed to represent real-world relational database schema design patterns including schema naming practices (defined as naturalness), schema size, and complex dependencies.

Contributions

This paper makes the following contributions:

- A schema subsetting-specific evaluation framework that isolates schema subsetting modules as independent processes.

Table 4.1. Schema size distribution for each benchmark dataset.

Size	Columns	Bird	Snails	Spider2	Total
S	<100	9	1	34	44
M	<1,000	2	6	43	51
L	<2,500	0	1	10	11
XL	<50,000	0	0	14	14
XXL	<100,000	0	1	4	5

- A reproduction and empirical analysis of 7 real-world schema subsetting modules from published code repositories.
- We present SKALPEL: a subsetter prototype for improving NL-to-SQL over very large schemas.
- Extension of recent research of schema subsetting to very large schemas and discovery of both the usefulness and degradation of schema subsetting performance over these schemas.

4.2 Methodology

To thoroughly evaluate multiple subsetting methods across multiple benchmarks, we create a modular object-oriented architecture to standardize interfaces between benchmarks, subsetting methods, evaluation processes, and NL-to-SQL generation. The core of this approach is the schema data object, represented in our project as a Python object that can be transformed to DDL and JSON representations and is exclusively used as the representation of both full and subset schemas throughout the project codebase.

4.2.1 Benchmarks

Benchmark Data

To evalute subsetting methods against a diverse set of schemas, we retrofit the Spider2 [49], SNAILS [57], and Bird [54] NL-to-SQL benchmarks. All benchmarks conform to the **Q** :=< $Q_{nl}, Q_{sqlGold}$ > format, where NL questions are paired with a ground truth $Q_{sqlGold}$ used for evaluating correctness of subsetting method-generated $Q_{sqlPred}$ queries.

Table 4.1 summarizes the benchmark schemas in terms of 5 size categories based on the total number of columns in a schema. The Bird dev set, the benchmark that appears most often in similar research, contains only small- and medium-sized schemas whereas the Snails and Spider 2 collections contain much larger schemas. Schema size categorization enables a cross-benchmark evaluation of subsetting that prioritizes overall schema size over the source benchmark, which is important in subsetting evaluation because some methods are constrained by model input context limits that may render them useless in a particular size category.

Benchmark Integration

The three benchmarks differ from each other in terms of how they perform schema representation, gold query and natural language question pairs formatting, and database storage and access. We define an abstract NL to SQL object with standard function calls and iterators that serves as the super class to the individual benchmark sub-classes that wrap the underlying benchmark behaviors in a standardized interface. This approach allows us to perform online benchmark instantiation using a factory class and greatly increases the interchangeability of benchmarks in the subsetting evaluation and NL-to-SQL experiments.

4.2.2 Schema Subsetting

We modularize and abstract the schema subsetters in a similar fashion as the benchmark class. Extracting the subsetting behavior from real-world NL-to-SQL systems varies in complexity by each system with some methods requiring more complex interfaces than others. Regardless of underlying method complexity, we are able to apply a singular schema subsetter interface to each one, thus also making the subsetter modular and relatively "easy" to modify existing methods and experiment with new methods. Schema subsetters take as input a benchmark question object that contains the associated schema object as well as a natural language question and return a subsetter result that contains a schema object (subset) and token usage data.

4.2.3 NL-to-SQL Generation

There are a variety of approaches to LLM-based NL-to-SQL generation, and the NL-to-SQL systems from which our evaluated subsetting modules employ an array of various post-subsetting techniques including zero-shot NL-to-SQL generation, error correction, few-shot example-based NL-to-SQL generation, and others. To maintain consistency in evaluation between subsetting methods, we forego more complex NL-to-SQL techniques and instead use a zero-shot prompting approach that includes SQL generation instructions, the schema subset (represented serially as table: column 1, column 2, ..., column n), and additional evidence or hints relating to the schema if provided in the benchmark dataset.

We evaluate 7 LLMs from the OpenAI GPT [76, 74], Meta Llama [105], and Google Gemini [104] model families. We categorize the models into 3 classes: *Flagship models* (GPT 4.1, Gemini 2.5 Pro), *economy models* (GPT 4.1-Nano, Gemini 2.5 Flash, Gemini 2.5 Flash-Lite), and *open source models* (Llama 3.3 70B Q8, GPT OSS 120B).

We use the OpenAI and Google API libraries for flagship and economy model integration. We host Llama 3.3 using Ollama on a dedicated server equipped with 2 AMD 9254 24-core processors, 541GB of system memory, and 2 NVIDIA A100 GPUs, with the model occupying 1 GPU. We host GPT OSS 120B using vLLM on a dedicated server equipped with 2 AMD 9654 96-core processors, 1,623 GB of system memory, and 8 NVIDIA H100 GPUs, with the model occupying 4 GPUs.

4.2.4 The SKALPEL Subsetter

In addition to a thorough evaluation of existing subsetting modules, we capitalize on our analysis to prototype a subsetting method targeted at improving NL-to-SQL for very large database schemas (Figure 4.1). The primary design goal of SKALPEL is to maximize table and column recall while minimizing subset size without compromising recall, minimizing token usage, and reducing latency.

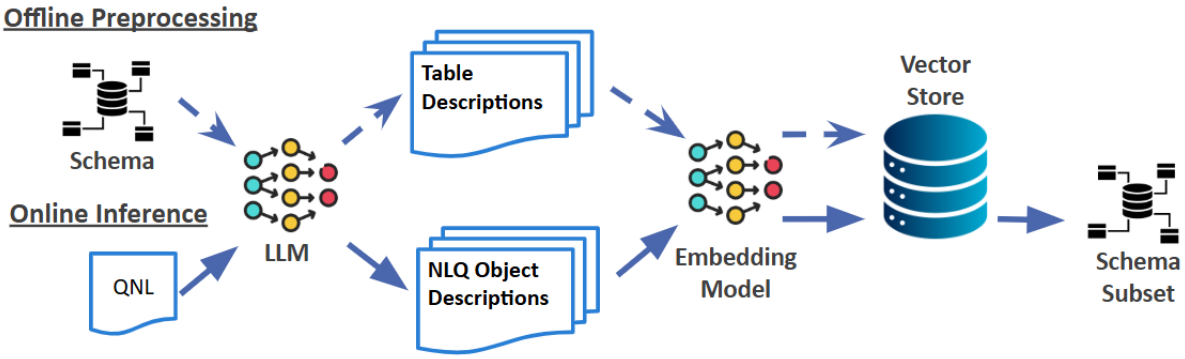


Figure 4.1. The prototype SKALPEL subsetter combines LLM-based question decomposition and embedding vector generation to perform table retrieval from a vector store. SKALPEL measures cosine distance between the embeddings of 2-3 sentence table descriptions and a set of object descriptions decomposed from an NLQ and returns tables and all of their columns whose distance are below a set threshold.

To accomplish this goal, we adopt a hybrid approach similar to Crush4SQL [47] except instead of schema hallucination, we task the LLM to decompose the question (similarly to the decomposition method described in [43]) with the instruction "describe the various objects, concepts, etc. in a natural language query." The output of the object description task is embedded using the NovaSearch *stella_en_1.5B_v5* embedding generation model [122].

During an offline preprocessing stage, SKALPEL uses an LLM to generate natural language descriptions of schema tables using the table name, column names, and example values. These descriptions are also embedded using NovaSearch and stored in a PGVector [107] database. During subset generation, SKALPEL retrieves relevant tables and all their columns via cosine distance search, retrieving all tables where the distance between question decomposition embeddings and table description embeddings are less than or equal to 0.525.

4.3 Schema Subsetting

Schema knowledge subsetting (also known as schema linking) is the process of extracting a subset of relations and attributes from a database schema with the objective of eliminating as many unneeded identifiers while retaining all identifiers required for a correct query generation.

In this section, we review recent subsetting modules in NL-to-SQL systems ranked highly on the Bird benchmark, we measure subsetting effectiveness by ablating schema knowledge representations from a full schema to only identifiers present in a correct query, we establish symbolic definitions of the subsetting problem, and articulate some challenges relating to mismatches between natural language questions and schema identifiers.

4.3.1 Survey of Subsetting Approaches

Many leading submissions to the Bird NL-to-SQL benchmark [54], including the top performer, use schema subsetting. As of June 2025, 5 out of the top 15 entries [97, 29, 117, 22, 102] on Bird's execution accuracy leaderboard use some form of schema subsetting in their SQL generation pipelines. While some analysis suggests that the risk of omitting required identifiers during the subsetting step negates any potential benefit [60], the overall performance of the top submissions that use schema subsetting suggest that, when done correctly, subsetting may be improving NL-to-SQL performance in some cases. These mixed results motivate additional scrutiny of existing schema subsetting methods to 1) determine their overall usefulness for improving NL-to-SQL, and 2) discover specific aspects of subsetting methods that either improve or degrade NL-to-SQL generation.

Selection Criteria for Evalution

Our research objective is to evaluate the best representations of a variety of subsetting methods, most of whose relatively high positions on the Bird execution accuracy leaderboard indicates good performance. Our ability to reproduce a given subsetting method is limited by code and model availability, and this constraint eliminates a large proportion of entries, as only 34 of the 74 entries provide links to their source code. Of the remaining entries, we select entries that indicate the use of a schema subsetting step and provide a fully reproducible codebase and, where applicable, model weights. Figure 4.2 shows the 7 subsetting methods we evaluate in our research and their classification in terms of subsetting model type and technique.

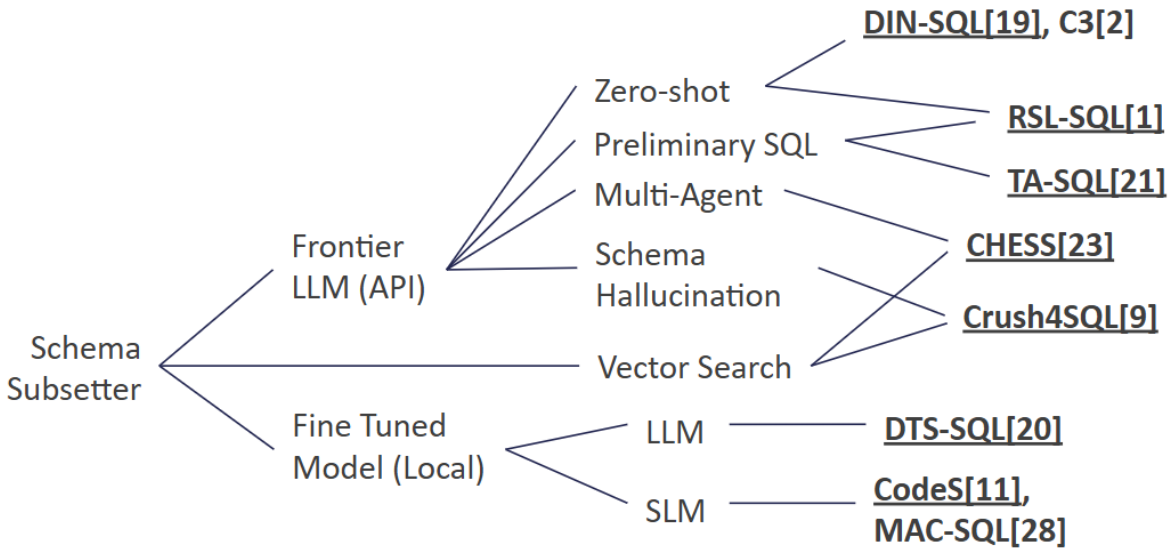


Figure 4.2. Schema subsetting methods use a variety of techniques and resources including frontier (API-based) LLMs, smaller locally-hosted and finetuned models (both LLM and SLM), and vector search. Some methods use more than one method such as CHESS which uses multiple LLM agents and vector search. Underlined methods indicate that a method is included in our experiments.

Subsetting with LLMs

Several approaches make use of the same LLM used for NL-to-SQL generation in a prior step that includes submitting a full schema, a natural language question, and instructions to identify the most-relevant schema entities. The output of this reduction step is typically used as input to an SQL generation prompt. LLM-based subsetters are subject to the same context limit constraints as an SQL generation prompt, and so a database schema representation that is too large for naive NL-to-SQL inference with full schema knowledge will also be too large for a schema subsetting prompt.

DIN-SQL [84], and C3 [21] use LLM-based schema subsetters as preliminary actions in a multi-step zero-shot NL-to-SQL generation pipelines. RSL-SQL [13] uses an LLM-based bi-directional subsetting method that combines the results of two actions: 1) a few shot prompt instructing the LLM to select relevant schema elements based on a natural language question, and 2) and extraction of table and column names from a *preliminary SQL* query generated via

zero-shot prompting and the full schema representation. TA-SQL [86] pre-processes a database with LLM-generated column descriptions which are used to supplement schema knowledge representations in a zero-shot SQL generation prompt. TA-SQL generates subsets through identifier extraction from the generated preliminary SQL queries.

Hybrid Subsetting Methods

Some methods employ multiple techniques including the use of both LLMs and semantic search. CHESS [102] is an agentic system that contains information retriever and schema selector agents that execute their tasks sequentially to generate schema subsets. The information retriever agent has access to information retrieval tools that use vector searches over identifier embeddings to retrieve the most relevant identifiers. The schema selector agent refines the information retriever's selections using LLM-based few shot prompting. Crush [47] is the only method that does not correspond to a Bird benchmark entry. It is an LLM-based subsetter that introduces the novel idea of schema hallucination as a pre-step to semantic search of vector representations of the target database's identifiers.

Subsetting with Finetuned Models

Another approach is to finetune a smaller language model (SLM) specifically to the task of schema subsetting. DTS-SQL [85] uses a finetuned 7B parameter Deepseek Coder [32] to subset schemas by selecting relevant tables. Code-S [53] employs a pre-trained T5-based model and adds a value lookup feature that seeks alignment between natural language symbols and text values in a target database. Code-S employs subsetting-specific analysis via area under curve (AUC) metrics for table and column recall. The authors of DTS-SQL introduce the use of precision and recall to evaluate schema linking performance separately from overall NL-to-SQL performance. MAC-SQL [114] uses a selector agent composed of an LLM prompt sequence run against a finetuned 7B parameter LLM to reduce a database to a subset database aligned with an input natural language question.

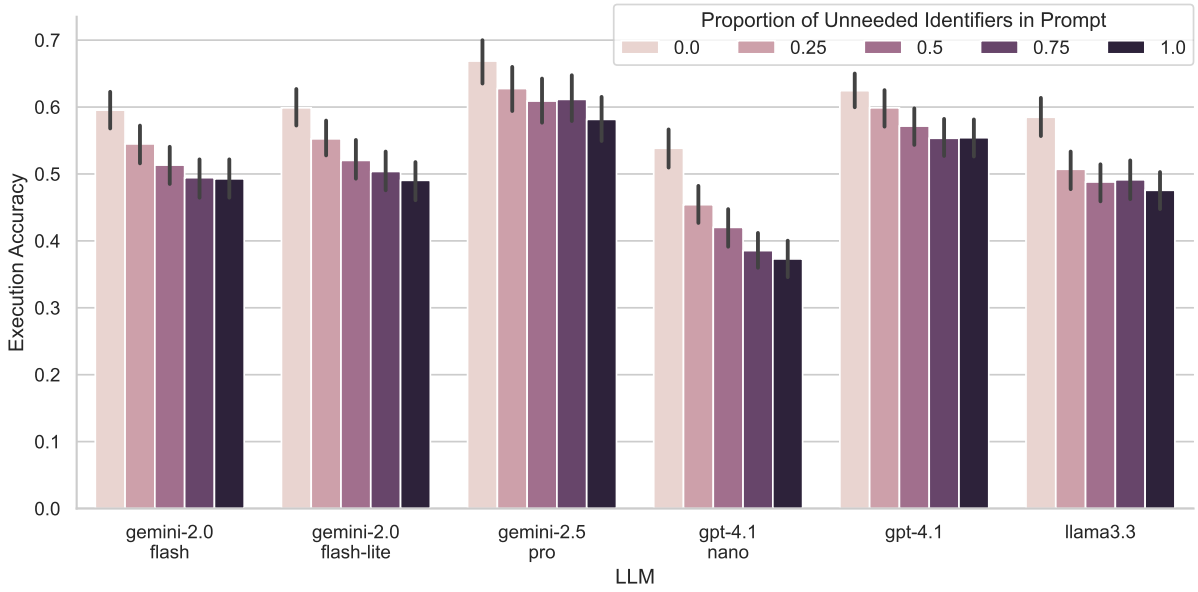


Figure 4.3. Execution accuracy (y-axis) over the medium- and large-sized schemas in the Snails benchmark generally exhibits a downward trend as the proportion of unneeded identifiers increases.

4.3.2 Subsetting Effectiveness

Schema Subset Proportion Effects on NL-to-SQL

To test the assumption that schema subsetting improves NL-to-SQL performance, we perform subset proportion ablation where we modify schema representations iteratively to randomly reduce the proportion of unneeded schema identifiers in a SQL creation prompt, where a proportion of 1.0 contains all schema identifiers, and a proportion of 0.0 contains only the identifiers required for a correct result. Figure 4.3 shows the result of a zero-shot prompt with varying proportions of schema knowledge executed over the SNAILS benchmark dataset at the Native naturalness level. Except for GPT-4.1 Nano, there is no improvement at the 0.75 proportion mark, and only the GPT 4 model family exhibits improvement at 0.50. All models show consistent improvement when the schema representation is further reduced to 0.25. When the schema representation sheds all unneeded identifiers (the case of a perfect subset), execution accuracy improves dramatically for all models. Logistic regression confirms a significant negative effect of more unneeded identifiers, with accuracy declining as unneeded identifier proportions

increase ($p < 0.0001$). The relatively low R^2 values between 0.01 and 0.025 (model-dependent) suggest that additional factors besides schema proportion also affect execution accuracy.

The takeaway from this evaluation is twofold: 1) the usefulness of schema subsetting at different proportions is model-dependent, and 2) the potential for performance improvement in the case of a subset with perfect recall motivates further evaluation and improvement of real-world subsetting methods.

4.3.3 Definitions

Schema and Database

We evaluate subsetting processes over a relational database schema $\mathbb{S}$ that contains a set of relations $\mathcal{R}$ and their attributes $\mathcal{A} \in \mathcal{R}$. Each relation $\mathcal{R}$ is a bag of tuples $t_{1...n}$ where $t[\mathcal{A}]$ references an atomic value v of the type of attribute $\mathcal{A}$. The database instance $\mathbb{D}$ is the realization of $\mathbb{S}$ such that $t[\mathcal{A}_{1...n}] \in \mathcal{R}_{1...n} \in \mathbb{S} \Rightarrow \mathbb{D}$, and an SQL query Q_{sql} over $\mathbb{D}$ returns a result r which is also a bag of tuples. We denote the set of all values v in $\mathbb{D}$ as $\mathbf{V}$, and specify the set of all values in a given relation or a relation and attribute as $\mathbf{V}_{\mathcal{R}}$ and $\mathbf{V}_{\mathcal{R}.\mathcal{A}}$ respectively.

SQL Queries and Identifiers

An SQL query Q_{sql} references relations $\mathcal{R}$ and attributes $\mathcal{A}$ in its projection and selection clauses. We differentiate between SQL queries that are members of benchmark question and query pairs as "gold" queries, with the notation $Q_{sqlGold}$ and queries that are predicted by an NL-to-SQL function with the notation $Q_{sqlPred}$. For the purposes of subsetting analysis, we consider Q_{sql} in a syntax-agnostic manner, simply as a set of its identifiers organized as a schema subset $\mathbb{S}'$ containing a subset of schema relations $\mathcal{R}'$ and attributes $\mathcal{A}'$.

We also define selection clause predicates $\mathcal{P}$ in simple terms of an attribute $\mathcal{A}$ and value v, and denote them in the form $\mathcal{P} :=< \mathcal{A}, v >$ where for a given predicate $\mathcal{P}$, $\mathcal{P}_{\mathcal{A}}$ and $\mathcal{P}_v$ are the attribute and value referenced in the clause. For example,
$\mathcal{P} :=< customer.name, jones >$ represents the selection expression $customer.name =$ "$jones$".

Benchmarks and Natural Language Question Symbols

A natural language question Q_{nl} is comprised of symbols $w \in Q_{nl}$ where symbols may be words, numbers, or special characters. Q_{nl} is an ordered concatenation of $w_{1...n}$ whereas the unordered set of $w \in Q_{nl}$ is denoted as $\mathbf{W}_{nl}$. The distinction between Q_{nl} as ordered, and $\mathbf{W}_{nl}$ as unordered is a necessary distinction in some contexts defined in subsequent sections.

Nl-to-SQL benchmarks $\mathbb{B}$ are collections of databases $\mathbb{D}$ and associated natural language question Q_{nl} and gold SQL query $Q_{sqlGold}$ pairs. We denote each pair of $Q_{nl}, Q_{sqlGold} \in \mathbb{D} \in \mathbb{B}$ as $\mathbf{Q}$ so that $\mathbf{Q} :=< Q_{nl}, Q_{sqlGold} >$.

Subsetting Function

We define an abstract schema subsetting function Ψ which generates a schema subset $\mathbb{S}'$ given the inputs Q_{nl} and $\mathbb{D}$. This function uses an abstract comparison operation denoted as $\sim$ (and its negation $\nsim$) which is satisfied if there is sufficient semantic similarity or other relevance-defining relationship between one or more symbols in Q_{nl} and one or more entities in a database $\mathbb{D}$.

$$\Psi(Q_{nl}, \mathbb{D}) \rightarrow \{\mathbb{S}' | \mathbb{S}' \subseteq \mathbb{S}\}$$

where relations and attributes are included in the subset by the criteria:

$$\Psi(Q_{nl}, \mathbb{D}) := \forall w \in Q_{nl} \exists \mathcal{R} | \mathcal{R} \in \mathbb{D} \wedge$$
$$(\mathcal{R} \sim w \vee (\exists \mathcal{A} \in \mathcal{R} | \mathcal{A} \sim w)) \quad (4.1)$$

That is, relations and attributes are members of the schema subset $\mathbb{S}'$ if they satisfy the semantic similarity comparison operation $\sim$ for at least one symbol w in Q_{nl}. Note that there is no restriction on the $\sim Q_{nl}$ operation so that we can overload Ψ and its $\sim$ operator in specific implementations in subsequent sections.

4.3.4 Subsetting Challenges

The schema subsetting process presents several challenges including relation and attribute name ambiguities, unmentioned attributes required for correct SQL statements, and unnatural schema elements with minimal association with common natural language terms. In this section, we formalize two such problems relating to recall errors caused by mismatches between natural language questions and their associated gold queries.

The Value Reference Problem

Q_{nl} references an instantiation of an object (e.g., proper nouns such as names and places) but not the type of object. For example, the question *how many Toyota Tacomas were involved in rollover accidents in 2022?* requires the retrieval of the Make and Model columns within a vehicle information table. However, the question does not contain keywords that will match *Toyota* or *Tacoma* to either the Make or Model attributes. We refer to this problem as the ***value reference*** problem. We can identify occurrences of the value reference problem by the criteria:

$$\forall \mathcal{P} \in Q_{sql} \exists w \in \mathbf{W}_{nl} | w \sim \mathcal{P}_v \wedge w \nsim \mathcal{P}_{\mathcal{A}} \tag{4.2}$$

That is, there is a satisfied similarity between a symbol in Q_{nl} and at least one value in any predicate $\mathcal{P}$ in Q_{sql}, where the symbol does not also satisfy a similarity comparison with at least 1 attribute or relation identifier in the same Q_{sql}.

Value lookup is an approach used by CodeS [53] where a $Q_{nl} \sim \mathbf{V}$ operation extracts attributes containing values relevant to Q_{nl}. The CRUSH subsetting approach relies on LLM prompting to hallucinate a schema in reponse to a Q_{nl}, which results in the generation of an intermediate representation of $\mathbf{W}_{nl}$ that better aligns with a value in $\mathbf{V}$ [47].

The Hidden Relation Problem

Q_{nl} contains keywords that bear semantic similarity to relations in $\mathbb{S}$, and at least one transitive dependency exists between relations that does not correspond to a Q_{nl} keyword. We

Table 4.2. Percent of each schema size category that each subsetting method is capable of processing. Only 3 of the 7 evaluated methods are capable of processing all schemas.

Ψ	S	M	L	XL	XXL
CodeS	100%	100%	100%	100%	100%
DINSQL	100%	100%	100%	100%	100%
chess	100%	100%	100%	-	-
crush4sql	100%	100%	100%	100%	100%
dtssql	66%	12%	-	-	-
rslsql	100%	100%	82%	79%	-
skalpel	100%	100%	100%	100%	100%
tasql	100%	100%	100%	86%	-

refer to this problem as the ***hidden relation*** problem. The hidden relation problem defined as:

$$\forall w \in \mathbf{W}_{nl} \exists \mathcal{R} \in Q_{sql} | \mathcal{R} \nsim w \wedge (\forall \mathcal{A} \in \mathcal{R} | \mathcal{A} \nsim w) \tag{4.3}$$

That is, there is at least 1 relation identifier $\mathcal{R}$ in a query Q_{sql} that neither itself, nor any of its attributes $\mathcal{A}$, satisfies a similarity comparison with any symbol w in $\mathbf{W}_{nl}$. This problem typically arises when a Q_{sql} must contain intermediate joins with relations that are not semantically similar to any keywords or phrases in a natural language question.

4.4 Experiments

4.4.1 Discovering Schema Size Limits

In order to discover the limits of subsetting methods executed over large schemas, we assess subsetting method capabilities in the context of schema size categories (described in Table 4.1). We do this by determining the percent of subsetting attempts that yield a non-empty set of schema identifiers, which indicates that the subsetting method did not encounter a condition that prevented it from generating a subset.

Size Constraints

Exposing subsetting methods built to solve smaller schemas in the Bird benchmark to larger schemas in the Snails and Spider 2 benchmarks reveals some limitations in most of the subsetting methods. Methods that load the entire schema representation into a single LLM prompt tend to exceed LLM context window limitations on very large schemas. Only 3 of the 7 methods that we evaluate successfully completing inference over all benchmark schemas. Table 4.2 reveals the inference completion percentages for each model and schema size. CodeS, a machine-learning based classifier approach, handles its small 512 token context via partitioning. Crush4SQL is not subject to context window constraints because it performs table and column retrieval using semantic search. DINSQL generates a minimal schema representation without additional value examples or identifier descriptions and thus does not exceed the GPT4.1 context window. Chess did not exceed any context limitations, however it relies on a column-by-column assessment to determine column relevance–a method that proves to be prohibitively expensive and time consuming for very large schemas. In the following experiments, we only compare results for subsetting attempts that do not exceed a subsetting method's capabilities.

4.4.2 Experiment 1: Subsetting Performance

We evaluate 7 subsetting methods used by high-performing NL-to-SQL submissions on the Bird [54] benchmark leaderboard. These methods represent a variety of approaches including frontier LLM prompting, SLM finetuning, and semantic search. We go beyond the standard measures of precision, recall, f1, and also include token usage (as applicable), inference time, preprocessing time (as applicable), and subset size as a proportion of the full schema. This allows us to assess the time and resource requirements for each subsetting method, and motivates the development of more economical methods.

Research Question

What are the best–measured in terms of recall, precision, f1, time-based performance, and resource usage–methods for correctly reducing the size of LLM prompt schema knowledge representations?

Evaluation Method

We evalute subsetting performance over the NL-SQL question-query pairs $\mathbf{Q} \in \mathbb{B}$ in the Bird, Snails, and Spider 2 benchmarks. The correct subsets used to derive recall, precision, and f1 scores include all identifiers (relations and attributes) present in each pair's gold SQL representation $Q_{sqlGold}$.

In order to derive the correct subset for each benchmark question, we define an SQL parsing function $P(Q_{sql}) \rightarrow \mathbb{S}'$ that arranges relations and attributes into a schema subset $\mathbb{S}'$ comprised of identifiers referenced in the input query Q_{sql}. For each $\mathbf{Q}$ in a benchmark $\mathbb{B}$, we derive a gold scema subset $\mathbb{S}'_{Gold}$ using $P(Q_{sqlGold}) \rightarrow \mathbb{S}'_{Gold}$. Because $Q_{sqlGold}$ is a syntactically correct query over its target $\mathbb{D} \in \mathbb{B}$, we assert that all $\mathbb{S}'_{Gold} \subseteq \mathbb{S}$ and that $\mathbb{S}'_{Gold}$ represents the execution of a perfect (or oracle) subsetter $\Psi(Q_{nl}, \mathbb{D}) \rightarrow \mathbb{S}'$. With this assertion, we evaluate the output $\mathbb{S}'_{Pred}$ of each subsetting method's implementation of Ψ against $\mathbb{S}'_{Gold}$ for each $\mathbf{Q}$.

$$\forall Q_{nl}, Q_{sqlGold} \in \mathbf{Q} \in \mathbf{B} :$$

$$\Psi(Q_{nl}, \mathbb{D}) \rightarrow \mathbb{S}'_{pred}; P(Q_{sqlGold}) \rightarrow \mathbb{S}'_{Gold} \quad (4.4)$$

The end state of the process is a collection of predicted and gold subsets for all question-query pairs used as the input to the analysis described in the following sections.

Evaluation Metrics

Schema linking-specific metrics have begun to appear in recent NL-to-SQL papers including precision and recall [85, 47, 57, 43, 60], and AUC [53]. To the best of our knowledge,

our work is the first application of schema linking metrics to a comparison of multiple reproductions of schema linking modules used in real-world NL-to-SQL systems. Additionally, we believe this to be the first work to evaluate schema subsetting in terms of efficiency by measuring token usage, offline pre-processing time, and inference time.

We evaluate performance of Ψ using the following metrics.

- SchRecall: $|\mathbb{S}'_{Gold} \cap \mathbb{S}'_{Pred}|/|\mathbb{S}'_{Gold}|$
- PerfRecall: 1 if *ScheRecall* = 1.0, 0 otherwise
- SchPrecision: $|\mathbb{S}'_{Gold} \cap \mathbb{S}'_{Pred}|/|\mathbb{S}'_{Pred}|$
- SchF1: $2 * (SchRecall * SchPrecision)/(SchRecall + SchPrecision)$
- Subset Proportion: $|\mathbb{S}'_{Pred}|/|\mathbb{S}|$; (lower is better)
- Inference Time: Time (in sec.ms) to execute Ψ
- Prompt Token Count (LLM-based): Sum of LLM tokens used by Ψ to generate $\mathbb{S}'_{Pred}$
- Preprocessing Rate: $SchemaProcessingTime/|\mathcal{A} \in \mathbb{S}|$ where *SchemaProcessing* is the time in seconds required to perform pre-processing tasks such as generating vector embeddings or data descriptions.

We expand the evaluation framework further by generating each of the listed metrics for each identifier type (relations and attributes).

- SchRelRecall: $|\mathcal{R}'_{Gold} \cap \mathcal{R}'_{Pred}|/|\mathcal{R}'_{Gold}|$
- SchRelPrecision: $|\mathcal{R}'_{Gold} \cap \mathcal{R}'_{Pred}|/|\mathcal{R}'_{Pred}|$
- SchRelF1: $2 * (SchRelRecall * SchRelPrecision)/$ $(SchRelRecall + SchRelPrecision)$
- Relation Proportion: $|\mathcal{R}'_{Pred}|/|\mathcal{R} \in \mathbb{S}|$; (lower is better)

- SchAtrRecall: $|\mathcal{A}'_{Gold} \cap \mathcal{A}'_{Pred}|/|\mathcal{A}'_{Gold}|$
- SchAtrPrecision: $|\mathcal{A}'_{Gold} \cap \mathcal{A}'_{Pred}|/|\mathcal{A}'_{Pred}|$
- SchAtrF1: $2 * (SchAtrRecall * SchAtrPrecision)/$
  $(SchAtrRecall + SchAtrPrecision)$
- Attribute Proportion: $|\mathcal{A}'_{Pred}|/|\mathcal{A} \in \mathbb{S}|$; (lower is better)

Performance Constraints

While variance in precision indicate method quality, other metrics indicate method feasibility where failure to meet specific constraints will result in NL-to-SQL translation failure. That is not to say that a method should be discounted outright in the event of a failure to meet the constraint in some cases, rather the frequency of failures should be compared between methods with the best methods being those with the fewest failure occurrences.

Perfect recall: Failure of the subsetting function Ψ to recall all $\mathcal{A}'_{Gold}$ and $\mathcal{R}'_{Gold}$ in $Q_{sqlGold}$ will almost always ensure failure in the downstream NL-to-SQL translation task. Thus, it is imperative that we evaluate recall in the strictest terms possible with the goal of identifying methods that yield recall values as close to 1 as possible. The proportion of subsetting attempts that achieve perfect recall over all attempts represents the likely upper bound of execution accuracy in a subsequent NL-to-SQL step. Subsetting methods that achieve perfect recall scores below downstream NL-to-SQL (sans subsetting) execution accuracy proportion scores are likely to reduce overall performance.

Subset size: schema subsets must be sufficiently small that generated prompts fit within the target LLM's context window token limitations. Subsetting attempts that fail due to context window constraints are omitted from analysis. Subsetting method schema coverage (the percentage of schemas each method is capable of handling) is measured in Table 4.2, and represented as a proportion in Figure 4.5.

Subsetting Challenges Analysis

We apply the subsetting challenge definitions in Section 4.3.4 to categorize failures and isolate failure causes. Identifying subsetting challenge-related failure states requires an implementation of the semantic similarity comparison operation denoted as $\sim$. We make use of the NovaSearch *stella_en_1.5B_v5* embedding generation model [122] with a sequence length of 1,024 and store the embeddings in a PGVector database. The similarity comparison operation $\sim$ is satisfied when the cosine distance between two sequences is below a similarity distance threshold. We derive the similarity threshold through iterative evaluation of threshold values, selecting the value that yields the most satisfactory results as determined by a human researcher.

4.4.3 Experiment 2: NL-to-SQL Performance

Because the objective of schema subsetting is improving NL-to-SQL performance, the most important measure of a subsetting method is to evaluate its effect on NL-to-SQL performance. To this end, we ask the following research questions:

Research Question 1

To what extent (if any) do schema subsetting methods improve NL-to-SQL execution accuracy?

Research Question 2

To what extent (if any) does the size of a schema affect the usefulness of schema subsetting?

Evaluation Method

The subsets $\mathbb{S}'_{Pred}$ generated in experiment 1 are the input to the NL-to-SQL generation in experiment 2. We evaluate the performance of $Q_{sqlPred}$ generated using prompts containing $\mathbb{S}'_{Pred}$ from all subsetting functions Ψ and measure them in terms of execution accuracy to determine the effect of schema subsetting outputs on the objective NL-to-SQL process.

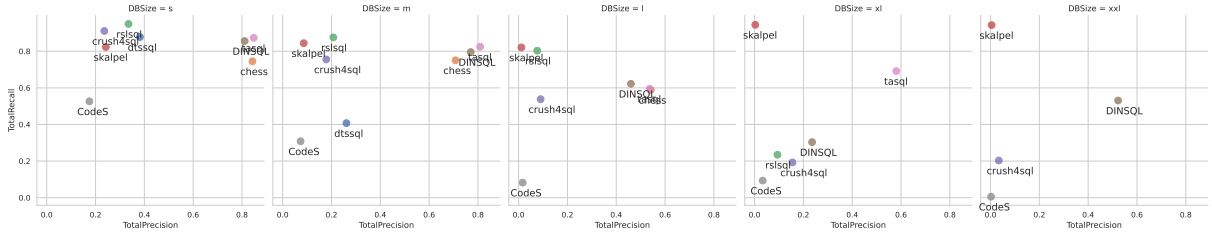


Figure 4.4. Mean total precision (x-axis) and mean total recall (y-axis) for each subsetting method for each database size. Besides the obvious drop in performance as database size increases, the data hint at some Pareto-like tradeoffs between recall and precision for most subsetting methods.

Evaluation Metrics

We evaluate NL-to-SQL in terms of execution result set matching, typically referred to as *execution accuracy*. Execution accuracy is the comparison of the results r_{Gold} returned from a $Q_{sqlGold}$ query over $\mathbb{D}$ and r_{Pred} returned from a $Q_{sqlPred}$ query over $\mathbb{D}$. We adopt a subset-set comparison approach that accounts for the possibility that predicted queries may contain additional attributes not required in the gold query, but to not render the predicted query incorrect.

$$\forall \mathcal{A}_{gold} \in r_{Gold} \exists \mathcal{A}_{pred} \in r_{Pred} : (\mathbf{V} \in \mathcal{A}_{gold}) = (\mathbf{V} \in \mathcal{A}_{pred}) \tag{4.5}$$

That is, every attribute in the gold query must be present in the predicted query, and the values in each attribute of the predicted query must equal their corresponding attribute values in the gold query.

4.5 Results

4.5.1 Experiment 1: Subsetting Performance

Overall Performance

Overall, there is no "best-performing method" across all metrics and schema sizes. Instead, we see varying performance over different schema sizes and subsetting methods that suggest the existence of Pareto-like trade spaces between various metrics (e.g., balancing precision and recall, or sacrificing latency for precision, etc.). Also, considering that the goal of schema subsetting is to

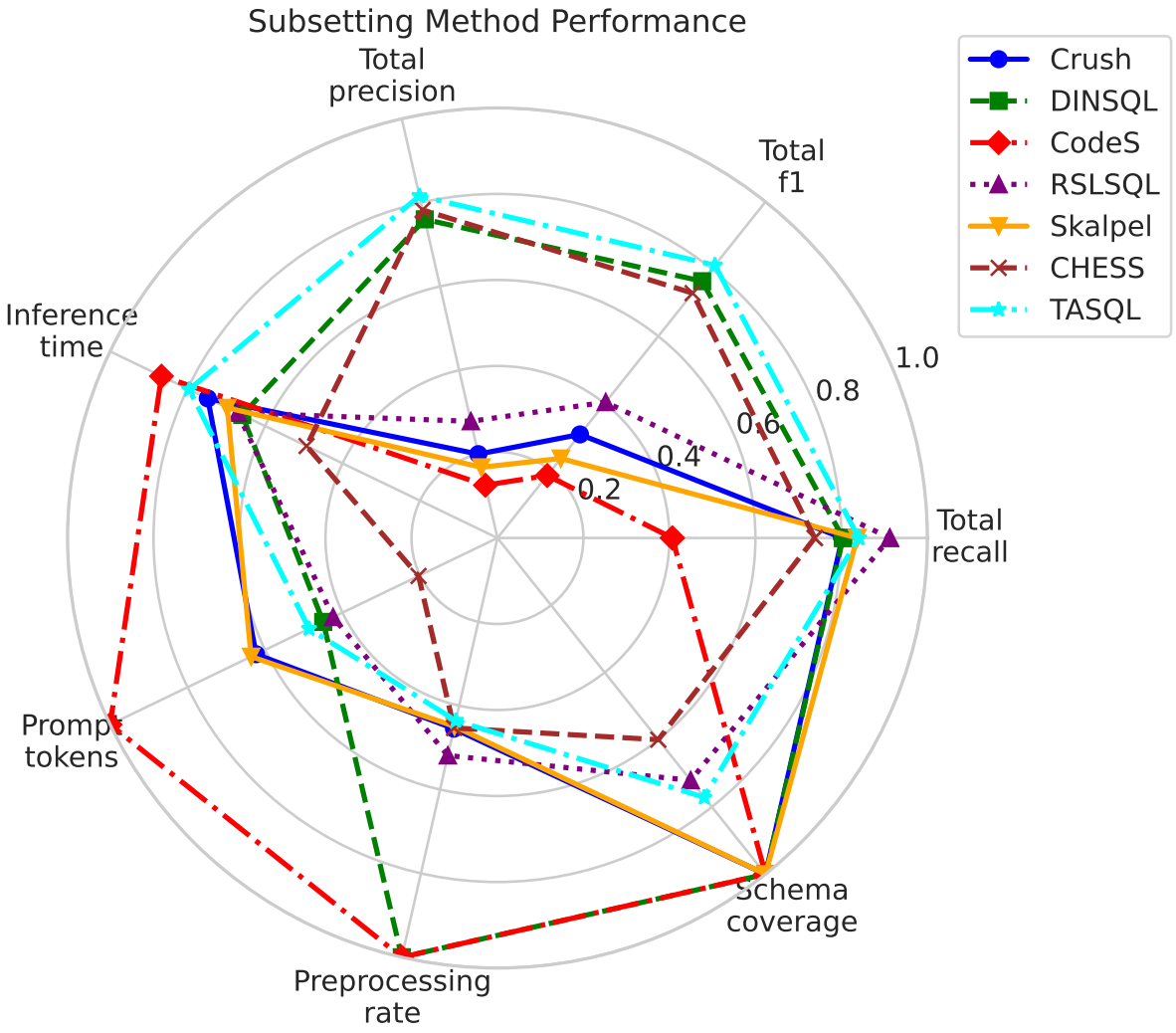


Figure 4.5. Inference time and Prompt tokens are log scale values with higher scores indicating better performance (e.g., lower token counts and inference time). Schema coverage indicates the proportion of the schemas in the combined benchmarks that the subsetting method can process.

improve NL-to-SQL performance, importance must be measured against NL-to-SQL generation outcomes which we do in experiment 2. Table 4.3 displays all subsetting performance metrics for each method at each schema size where only successful subsetting attempts for each method are included in the evaluation calculations. We also display the performance differences between all methods across performance and efficiency metrics with a radar visualization in Figure 4.5.

Frontier LLM-based methods including DINSQL, TASQL, CHESS, and RSLSQL generally outperform finetuned SLM- (CodeS, DTSSQL) and semantic search-based methods (Crush) by a significant margin in performance measures of f1, precision, and recall. However, the rapid growth of token usage across schema sizes exhibited by CHESS, RSLSQL, and TASQL portray the risk of over-reliance on knowledge augmentation and granular evaluation strategies. Knowledge enrichment, where schema representations are appended with sample values and verbose descriptions, cause both TASQL and RSLSQL to exceed the LLM's 1 million token context window for schemas in the XXL category. Granular column evaluation employed by CHESS, where each column is evaluated in isolation, invokes a very high token and time cost for relatively small schemas. The cost and time required to scale to XL and XXL schema sizes is prohibitive, and we consider it infeasible.

Recall is arguably the most important metric for evaluating the potential usefulness of a subsetting method. Failure to recall required identifiers all but guarantees that a subsequent NL-to-SQL translation will fail to produce a correct answer. No method consistently produces high recall scores across all schema sizes. Additionally, recall rates drop significantly for higher schema sizes with the "best" method in the XXL category only scoring an average recall of 0.53.

Setting aside the nearly terminal effect of low recall, we also want methods to reduce the schema proportion as much as possible through high precision. TASQL's preliminary SQL parsing and identifier extraction approach yields the best schema precision in the S, M, L, and XL categories. For the XXL category where TASQL is infeasible due to context window limitations, DINSQL achieves the highest schema precision.

Except for our own SKALPEL prototype subsetter, all methods reduce schema proportions

below the desired target derived from our analysis of subset proportion effects on NL-to-SQL.

Schema Size Effects on Performance

As schema column and table counts increase, all subsetting methods demonstrate a reduced performance across all performance measures which means that larger schemas lead to more expensive, more time consuming, and less useful results. The Schf1 (Schema f1) column in Table 4.3 reveals a *schema f1* performance reduction for each subsetting method as schema sizes increase. Additionally, Figure 4.6 radar charts portray consistent performance reduction for larger schemas across both performance and resource use metrics.

Recall vs. Precision Trade-Off

Intuitively, the should be a trade-off between precision and recall, where subsetting methods that prioritize precision run an increased risk of false negatives and methods that prioritize recall would likely reduce precision to maximize the chance of true positives. To test this intuition, we compare total precision and total recall for each method. Figure 4.4 shows mean *TotalPrecision* (x-axis) by mean *TotalRecall* (y-axis) for each subsetting method and database size. For the smallest databases, we can see a possible Pareto frontier with three subsetting methods (Crush, DTSSQL, and RSLSQL) clustered together with high recall and low precision, and three subsetting methods (DINSQL, TASQL, and CHESS) clustered together with high precision and slightly lower recall. We also note that the CodeS method is well inside the frontier for all database sizes.

As database size increases, some methods retreat from the apparent frontier or disappear as they are unable to process larger schemas. Additionally, methods that remain on the Pareto frontier generally exhibit decreased performance for larger schemas.

Subsetting Challenges

To evaluate the sensitivity of subsetting methods to each subsetting challenge described in Section 4.3.4, we first calculate the total percent of missing attributes and relations in $\mathbb{S}'_{Pred}$ from the total present in each $\mathbb{S}'_{Gold}$. Once we derive these percentages, we calculate sensitivity

Table 4.3. Experiment 1 Results: The mean of subsetting performance metrics for all $\mathbb{S}'$ by schema size (Sz) and subsetting and subsetting function Ψ. Metrics include Inference Time (T (s)), Prompt Token Count (Tokens), Perfect Recall (PRe), Schema Recall (SchRe), Schema Precision (SchPr), Schema f1 (Schf1), Relation Recall (RelRe), Relation Precision (RelPr), Relation f1 (Relf1), Attribute Recall (AtrRe), Attribute Precision (AtrPr), Attribute f1 (Atrf1), Relation Proportion (RelPn), and Attribute Proportion (AtrPn). Section 4.4.2 provides definitions for each listed metric.

Sz	Ψ	T (s)	Tokens	PRe	SchRe	SchPr	Schf1	RelRe	RelPr	Relf1	AtrRe	AtrP
S	CHESS	31.85	229825	0.26	0.75	0.84	0.78	0.88	0.89	0.87	0.69	0.82
	CodeS	**0.27**	0	0.20	0.53	0.17	0.25	0.66	0.33	0.42	0.47	0.14
	Crush	3.06	425	0.64	0.91	0.24	0.36	0.98	0.39	0.53	0.88	0.20
	DINSQL	5.89	4988	0.50	0.86	0.81	0.82	0.94	0.86	0.89	0.82	0.79
	DTSSQL	9.42	3582	0.76	0.88	0.38	0.49	0.87	**0.92**	0.88	0.88	0.33
	RSLSQL	4.52	8836	**0.86**	**0.95**	0.33	0.47	**0.99**	0.49	0.62	**0.93**	0.31
	Skalpel	4.09	**342**	0.68	0.82	0.24	0.33	0.82	0.52	0.57	0.82	0.21
	TASQL	1.51	2288	0.65	0.87	**0.85**	**0.85**	0.93	0.91	**0.91**	0.85	**0.83**
M	CHESS	96.69	891506	0.35	0.75	0.71	0.67	0.91	0.78	0.77	0.68	0.69
	CodeS	**1.15**	0	0.08	0.31	0.07	0.11	0.42	0.16	0.22	0.26	0.06
	Crush	2.60	431	0.41	0.76	0.18	0.28	0.91	0.39	0.51	0.69	0.15
	DINSQL	8.17	5989	0.50	0.80	0.77	0.76	0.90	0.86	0.86	0.74	0.73
	DTSSQL	8.82	7049	0.23	0.41	0.26	0.28	0.40	0.54	0.45	0.41	0.22
	RSLSQL	10.08	65715	**0.76**	**0.88**	0.21	0.32	**0.96**	0.37	0.49	0.83	0.18
	Skalpel	4.17	**363**	0.74	0.84	0.09	0.14	0.84	0.35	0.44	**0.84**	0.07
	TASQL	2.76	9096	0.57	0.82	**0.81**	**0.80**	0.90	**0.94**	**0.91**	0.79	**0.76**
L	CHESS	351.95	5254439	0.31	0.59	**0.54**	**0.54**	0.74	0.63	0.66	0.51	**0.49**
	CodeS	6.33	0	0.02	0.08	0.02	0.03	0.13	0.04	0.06	0.07	0.01
	Crush	**3.00**	441	0.24	0.54	0.09	0.14	0.82	0.19	0.30	0.41	0.07
	DINSQL	10.25	11566	0.43	0.62	0.46	0.50	0.71	0.54	0.59	0.56	0.42
	RSLSQL	27.74	191906	0.70	0.80	0.08	0.13	**0.93**	0.35	0.40	0.73	0.05
	Skalpel	4.21	**380**	**0.80**	**0.82**	0.01	0.02	0.82	0.13	0.18	**0.81**	0.01
	TASQL	4.97	47440	0.32	0.60	**0.54**	**0.54**	0.74	**0.69**	**0.70**	0.51	0.47
XL	CodeS	80.50	0	0.00	0.09	0.03	0.05	0.21	0.09	0.11	0.06	0.02
	Crush	**4.12**	**489**	0.00	0.19	0.16	0.13	0.47	0.22	0.23	0.12	0.11
	DINSQL	35.68	82289	0.19	0.30	0.24	0.25	0.34	0.28	0.30	0.29	0.22
	RSLSQL	39.02	498656	0.00	0.23	0.09	0.12	0.77	0.49	0.59	0.00	0.00
	Skalpel	5.70	533	**0.90**	**0.94**	0.00	0.01	**0.94**	0.10	0.14	**0.94**	0.00
	TASQL	14.62	294294	0.33	0.69	**0.58**	**0.59**	0.78	**0.81**	**0.78**	0.67	**0.53**
XXL	CodeS	381.43	0	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00	0.00
	Crush	**4.49**	431	0.03	0.20	0.03	0.06	0.36	0.04	0.07	0.15	0.03
	DINSQL	20.79	384137	0.34	0.53	**0.52**	**0.52**	0.57	**0.57**	**0.56**	0.51	**0.50**
	Skalpel	4.71	**371**	**0.89**	**0.94**	0.00	0.01	**0.94**	0.02	0.04	**0.95**	0.00

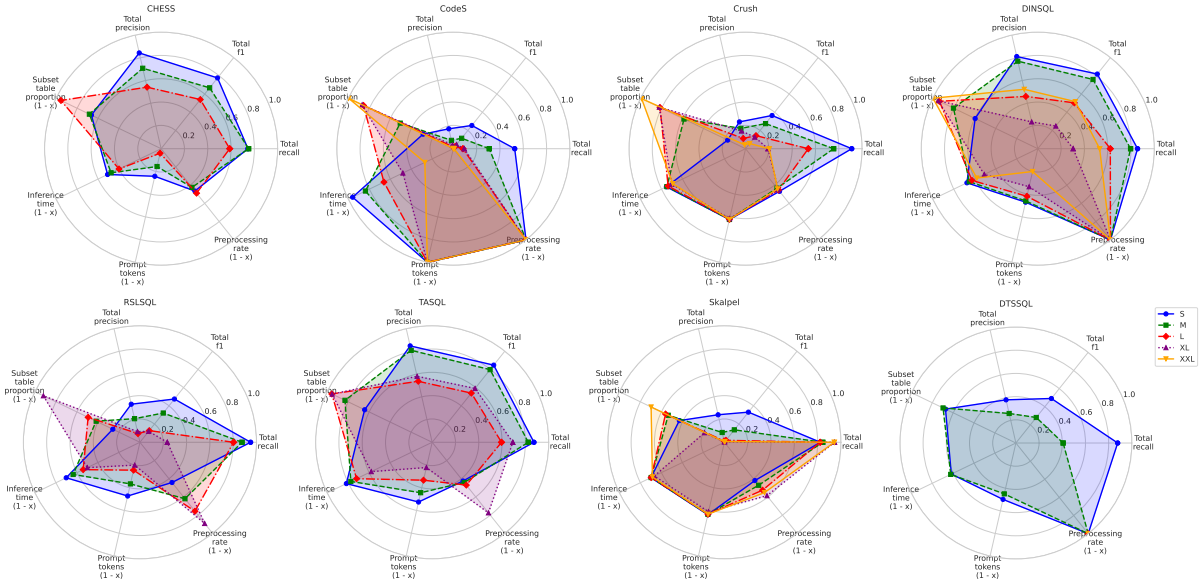


Figure 4.6. These radar charts display the tradeoffs between performance (measured by precision, f1, and recall) and time and resource usage (measured by inference time, prompt tokens, and pre-processing). Sensitivity to database size (S, M, L, XL, XXL) varies by both subsetting method and measure, and generally performance across all measures decreases as database size increases. Metrics where lower is better (schema proportion, inference time, token usage, preprocessing rate) are inverted (1 - x). Inference time, prompt tokens, and preprocessing rate are fit to the range [0, 1] via natural log functions.

Table 4.4. This table shows the Value Reference Problem (VRP) and Hidden Relation Problem (HRP) mean occurence percentages and sensitivities. Sensitivity (Sens.) is the proportion of a given problem occurence percentage (defined as realized problem over all instances where the problem might occur) to the percentage of all missing relations.

Ψ	% MA	% VRP	V-Sens.	% MR	% HRP	H-Sens.
RSLSQL	14%	7%	0.49	3%	11%	3.31
DINSQL	25%	17%	0.66	13%	22%	1.68
TASQL	20%	13%	0.66	11%	24%	2.12
DTSSQL	15%	11%	0.78	15%	29%	1.89
CodeS	65%	64%	0.97	47%	51%	1.09
Crush	32%	35%	1.11	11%	19%	1.71
CHESS	36%	43%	1.18	14%	31%	2.14
Skalpel	15%	21%	1.36	17%	58%	3.33

as the proportion of missing identifier percentages that occur when the potential for a problem manifest exists to the percentage of missing attributes in all question pairs. Sensitivity values greater than 1 indicate that the subsetting method is more likely to omit a relation or attribute when the problem potential exists in **Q**.

%MA (missing attributes) is the mean of the percent of the count of attributes missing in $\mathcal{A}'_{Pred}$ out of the total count of attributes in $\mathcal{A}'_{Gold}$ for each **Q**. **%VRP** (value reference problem) is the mean of the percent of the count of missing attributes in all $\mathcal{A}'_{Pred}$ where the conditions of $Q_{sql}, Q_{nl} \in \mathbf{Q}$ satisfy the conditions of the value reference problem definition. **%HRP** (hidden relation problem) and **%MR** (missing relations) are derived in the same manner as %MA, and %VRP for missing relations and the potential for the %HRP for each **Q**.

Except for the hybrid methods that rely on vector search (SKALPEL, Crush and CHESS), the value reference problem manifests at a rate lower than the overall missing attribute percentage, suggesting that subsetting methods that use LLM-based relation and attribute selection are resistant to failure in situations where there is a reference to a value in the natural language question that doesn't align semantically with any attribute or relation name.

On the other hand, the hidden reference problem occurs at a higher rate than the rate of all missing relations, suggesting that cases where required relations that are not semantically aligned to phrases or keywords in the natural language question cause subsetting to fail at a higher rate for all subsetting methods.

Token Usage and Inference Time

Token usage is a cost driver for LLM-based systems either in terms of costs incurred through API transactions with LLM providers or as a surrogate for power consumption in self-hosted LLMs. Thus, the economnic use of LLMs necessitates the monitoring of token usage and motivates the discovery of methods that achieve performance objectives while minimizing token requirements.

In Figure 4.7 we see that token usage varies considerably by method, and all LLM-based

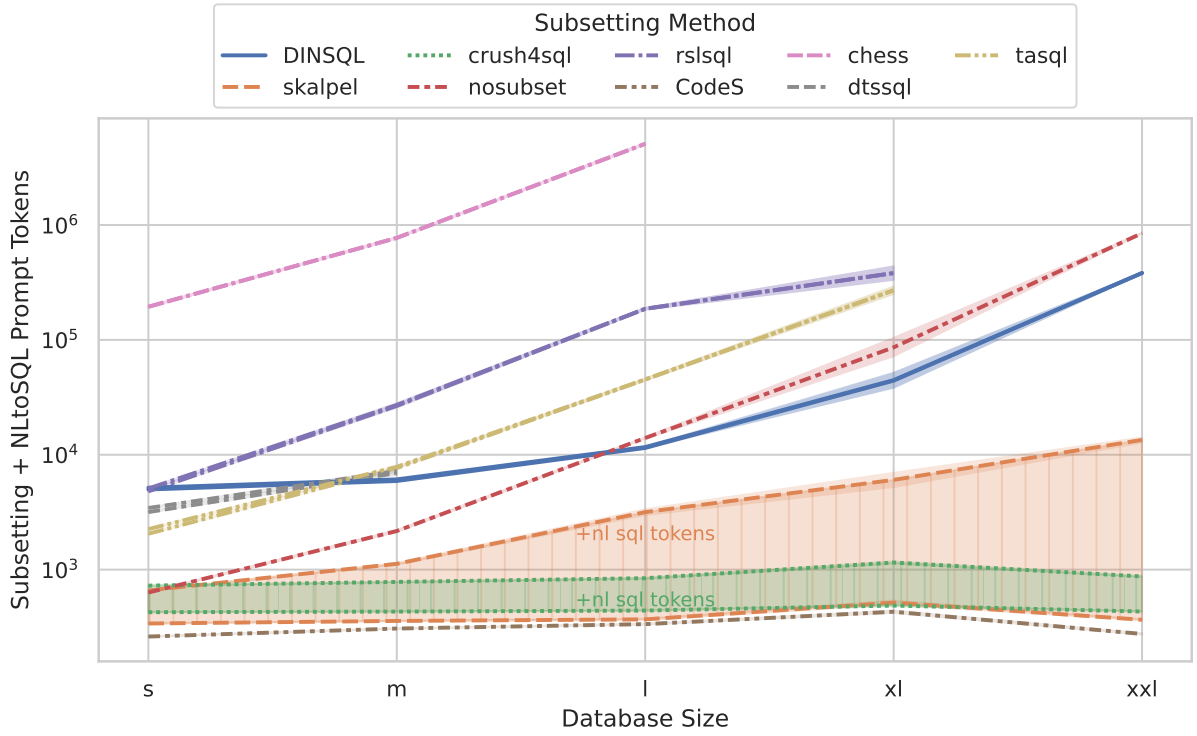


Figure 4.7. This chart displays total token (subsetting prompt tokens + NL-to-SQL prompt tokens) usage along the Y-axis (log_{10} of prompt tokens) and database size as defined in Table 4.1. Except for Crush and SKALPEL, the subsetting task dominates token usage to the point where NL-to-SQL prompt tokens are not visible. The filled area between the Crush and SKALPEL lines represent the additional tokens required for NL-to-SQL prompts, where the lower line represents subsetting tokens and the space between the lower and upper line represents the NL-to-SQL prompt tokens.

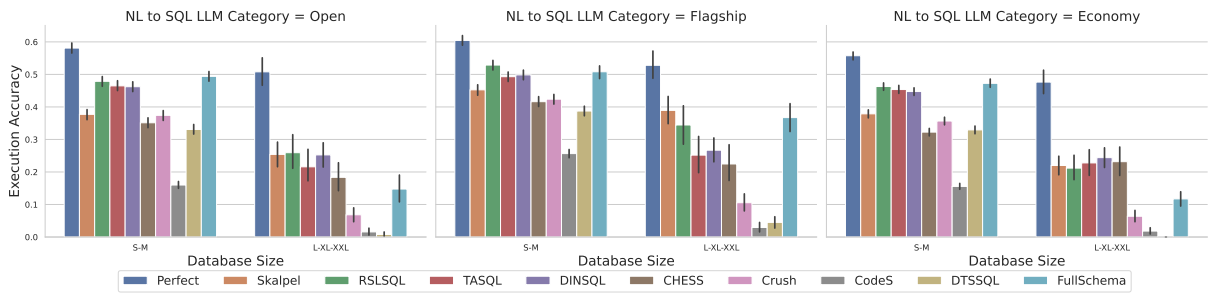


Figure 4.8. Without considering schema size, in aggregate all subsetting methods underperform vs. full schemas on NL-to-SQL execution accuracy. We discover that subsetting effects on execution accuracy (y-axis) are schema size-dependent (x-axis). Here, we bin schema sizes into two categories (s-m, and l+ which includes l, xl, and XXL categories), and we categorize flagship models (GPT-4.1, Gemini-2.5-Pro), economy models (GPT-4.1-nano, Gemini-2.5-Flash/Flash-Lite), and open source models (Llama 3.3, GPT-OSS).

methods consume more tokens than simply using the full schema for NL-to-SQL over small and medium databases. DIN SQL is the sole LLM-based method that consumes fewer tokens for large, and larger, databases, whereas other LLM-based methods continue increasing in excess of full schema NL-to-SQL prompting.

The most token-efficient LLM-based methods are the Crush subsetting method and our own SKALPEL prototype which use an LLM only to either hallucinate a schema or decompose an NL question without referencing the target database schema. While both of these hybrid methods do not exhibit a subsetting prompt token growth correlation with database size, token usage from resulting higher schema proportions and subsequent larger NL-to-SQL prompts does cause some token usage increase during NL-to-SQL inference. Nevertheless, both Crush and SKALPEL consume fewer tokens than all LLM-based methods as well as full schema NL-to-SQL prompting. CHESS expends the highest number of tokens primarily during its column selection phase which issues an LLM prompt for every column in a schema to determine the column's relevance to the NL question, a method that yields high precision scores on small, medium, and large schemas.

4.5.2 Experiment 2: NL-to-SQL Performance

Subsetting Metric Effects on Execution Accuracy

In the results of experiment 1, we are presented with a broad array of performance measures with an equally broad variety of results across subsetting methods and schema sizes. To determine the importance of each metric, we test correlations between the dependent variable NL-to-SQL execution accuracy and various performance measures from experiment 1. Many performance measure independent variables exhibit a high degree of covariance (e.g., f1 is derived from precision and recall, table and column recall are highly correlated, etc.), so we use a subset of all outputs from experiment 1 as coefficients in a logistic regression model over NL-to-SQL generation attempts over subsets generated by the evaluated subsetting methods (excluding perfect subsets and full schema attempts).

Table 4.5 indicates that there are significant correlations between the independent measures of *TableRecall*, *ColumnRecall*, and *SubsetTableProportion*. Precision- and proportion-based independent variables indicate a statistically significant, albeit weaker, correlation to execution accuracy. As expected, identifier recall is the prime measure of subset method performance evaluation. Precision is a secondary measure that should be used to differentiate subsetting methods that achieve relatively equal levels of table and column recall.

Execution Accuracy Differences Between Schema Sizes

Figure 4.8 shows *Execution Accuracy* for each subsetting method, a perfect subsetter, and a full schema (no subset) for two size categories of databases: small-to-medium (S-M), and large+ (L+, including L, XL, and XXL) whose labelling is consistent with the definitions in Table 4.1. For small-to-medium sized schemas, only one method (RSLSQL + Flagship LLM) improves performance over full schema use; this holds for all categories of LLM (Open, Flagship, and Economy). Given the token usage comparisons (see Figure 4.7), it is safe to conclude that, with one exception (RSLSQL with a flagship LLM), none of the schema subsetters improve execution accuracy *or* efficiency. Instead, for small ane medium schemas (fewer than 1,000 columns), it is

Table 4.5. Logistic regression coefficients, standard error, and significance (p value) for non-colinear parameters with execution accuracy as the dependent variable. n = 10,241, Pseudo R-squared = 0.099. Execution accuracy is binary (1, 0) where 1 indicates a correct NL-to-SQL generation given a schema subset.

Metric	Coef	StdErr	pValue
Const	-4.679	0.151	0.000
TableRecall	1.107	0.213	0.000
TablePrecision	0.599	0.181	0.001
ColumnRecall	0.933	0.175	0.000
ColumnPrecision	0.597	0.190	0.002
SubsetTableProportion	-0.055	0.163	0.736
SubsetColumnProportion	0.189	0.217	0.383

better to simply perform NL-to-SQL inference with a full schema representation.

Because we have extended subsetting analysis beyond the Bird benchmark onto the much larger Spider2 and Snails benchmarks, we can now see that for much larger schemas, both Open and Economy models benefit from RSLSQL, TASQL, DINSQL, SKALPEL, and CHESS subsetting methods during NL-to-SQL inference. Of these methods, SKALPEL performs at the same level as (Open LLMs), slightly better than (Flagship LLMs), or slightly under (Economy LLMs) the LLM-based methods that require a much higher number of tokens. In the case of larger schemas (more than 1,000 columns), we find that schema subsetting *can be* beneficial in terms of both execution *and potentially* token efficiency given the right subsetting method and NL-to-SQL LLM category.

4.6 Discussion and Limitations

4.6.1 Discussion

Subsetting Usefulness

Contemporary research expresses mixed results around the usefulness of schema subsetting, and in our work we expose some of the underlying reasons for these inconstencies: namely that subsetting performance–as well as downstream NL-to-SQL inference–is database dependent,

where schema size plays a large role in subsetting effectiveness and usefulness. From our analysis, we determine that subsetting small and medium schemas (less than 1,000 columns) actually tends to worsen NL-to-SQL performance for all types of LLMs, and so we recommend full schema representations in these cases, at least until real-world schema subsetting modules can improve recall.

A case can be made for subsetting when dealing with schemas that have more than 1,000 columns. This is because we see that for both open source and economy LLMs NL-to-SQL execution accuracy improves when using the LLM-based subsetters (e.g., RSLSQL, TASQL, DINSQL). In these cases, the decision to subset, and which subsetter to use, is more nuanced and depends on the tradeoff between performance and token usage. Alternatively, our prototype SKALPEL hybrid subsetter offers a solution that has the potential to reduce token usage while maintaining execution accuracy at the same levels as LLM-based subsetters.

Future Research

In this paper we refine the idea of a hybrid subsetter using LLM-aided question decomposition, LLM-generated table descriptions, and semantic search (cosine distance) to retrieve schema tables that align to the decomposed natural language questions. Although we performed some preliminary distance search tuning based on the competing goals of high recall and reduced schema proportion, additional work can be done to identify additional ways to improve schema recall without negating the benefit of reduce schema sizes. Specifically, would the join path detection method in [43] be an effective way to mitigate the hidden relation problem that SKALPEL is sensitive to?

4.6.2 Limitations

Schema Size NL-to-SQL Question Distributions

Although, to our knowledge, this is the first work to evaluate subsetting over very large database schemas, the majority of L-, XL-, and XXL-sized schemas come from the Spider 2 benchmark, which only comprises 221 of the 2,258 total NL-to-SQL questions across all 3

benchmarks. SNAILS provides 100 additional NL-to-SQL questions over its XXL-sized database. Future research into subsetting over very large schemas would benefit from an increased question pool of NL-SQL question query pairs over very large schemas.

Chapter 5

Related Work

5.1 Related Work for Speakql

Natural Language Interfaces

Section 2.2 discussed NLIs in detail and their relationship with SpeakQL. In particular, note that while we envision future integration of SpeakQL with NLIs for databases, the primary focus of this paper is the new dialect for spoken querying, not building a new NLI. We describe how our features can help improve ease of use, while preserving a context free grammar and correct-by-construction guarantees of regular SQL. In contrast, NL-to-SQL and chat-based interfaces today are primarily aimed at NL typing-based interactions, which lack such correctness guarantees.

NLIs such as FLAME [41] that provide code repair, autocompletion, and syntax reconstruction (i.e. inserting missing delimiter symbols) for targeted domains (spreadsheets in this case) are compelling candidates for integration with SpeakQL in a speech recognition context, where errors may result from either user or speech recognition errors.

Natural Language Interfaces With Structural Elements

Some data NLIs straddle the line between NL and controlled NL. SODA (Search over DAta Warehouse) [10] employs a simple, high level, query language as input to a pattern and meta-data enabled NL-to-SQL system. While this system employs a formal language for input queries, it differs from SpeakQL in that 1) it does not represent the full expressive power of SQL

and 2) there is not a direct mapping between the input sequence and an equivalent SQL query, and ambiguity resolution remains a challenge.

ATHENA [93] is an NL-to-SQL system that employs a novel intermediate ontology query language (OQL) for disambiguation which is then translated into SQL. While the OQL is similar to SpeakQL in the sense that they are both grammar-based languages, OQL serves as an intermediate representation of NL inputs and is transparent to database users; whereas SpeakQL is the input language designed for user interaction. Additionally, OQL does not resolve NL ambiguities, and the ATHENA system generates multiple possible SQL queries for a single NL input.

NaLIR [50] is an NL-to-SQL system that provides an interactive communicator as a means to disambiguate NL queries that yield more than one valid SQL statement. It provides feedback to users in the form a natural language explanation of query tree-based choices and a disambiguation widget that allows users to select the correct interpretation from a list of possibilities. Similar to ATHENA, NaLIR uses a structured intermediate interpretation of an NL expression to resolve ambiguities whereas SpeakQL gives the task of disambiguation to users through its syntax structure.

Speech-Based Database Interaction

EchoQuery [58] is a stateful query-by-voice system that allows users to interact with a relational database in a conversational manner. It only presents a tool demonstration though, without a rigorous grammar or thorough user study evaluation. It does use a speech-friendly dialect, but its syntax represents a subset of SQL expressiveness, e.g., joins are not explicitly defined at all and disjunctive predicates are not possible at all. In contrast, our work formalizes SpeakQL as a new dialect of SQL for spoken querying and defines its grammar. We extend a non-trivial subset of SQL and retain its full expressive power, including on joins and predicates, because SpeakQL subsumes that SQL subset. We also evaluate the ease of use of the SpeakQL dialect using a thorough A/B user study.

Prior work on SpeakQL 1.0 [8] is a speech+touch multimodal query interface that allows users to modify query intent using a novel SQL touch keyboard. SpeakQL 1.0 also targets a non-trivial subset of SQL. In contrast, SpeakQL 2.0 (this paper) proposes and evaluates a new dialect of SQL designed for spoken querying, not a new multimodal interface with touchscreen focus.

Data@Hand [46] is another speech+touch query interface that allows interaction with personal health data on a smartphone device. Users issue natural language instructions to initiate interaction, and then disambiguate or modify their requests with on-screen widgets. Data@Hand differs from other systems mentioned here as it is targeted at lay users and limited to personal health data in scope, whereas SpeakQL is targeted at SQL-trained data analysts and database experts and is intended to function on any arbitrary relational database.

CiceroDB [110] uses Google Assistant to infer user query intent from NL statements and vocalizes query result sets using computer-generated voice. It is orthogonal to both the general goals of the SpeakQL line of work and this paper's specific goals of a new speech-first dialect of SQL. One can use both SpeakQL and CiceroDB together to enable fully speech-based interactions with databases to both query data and to consume results.

Setlur and Tory [95] design and evaluate a speech-only interaction system as well as a speech+visual multimodal interaction system for querying data and viewing results on a tablet device. Rather than using a structured, or semi-structured language, these systems employ a chatbot-style natural language interface. Among other results, the authors found that augmenting a voice-based system with visual feedback resulted in richer dialogue and data exploration sessions.

Spoken Programming as Assistive Technology

Spoken programming systems have been considered and evaluated as assistive technologies for programmers with motor impairments. Much prior art in this space ended up as ultimately unconvincing due to weak ASR capabilities [9]. But the recent wave of highly accurate deep

learning-based ASR has rekindled interest in this space. Usability interviews with motor impaired programmers suggested a promising future for an NL-based programming system that can avoid dictation of symbols and variables [69]. The SpeakQL dialect is inspired by a similar vision and our user study survey responses support a similar conclusion, viz., a desire for more natural-feeling constructs to avoid dictating symbols.

5.2 Related Work for Snails

Ontology Mapping

Schema modifications and intermediate representations to enhance performance in a specific context extend beyond NL-to-SQL applications. Mapping relational database schemas to ontologies is an approach used to improve schema-to-schema integration and web application application-database interfaces [118]. This improves the semantic description of underlying data, which is often a desirable feature in web applications that interact within the semantic web [34]. While ontological mapping of a relational database can improve performance in this context; we see less evidence that such an approach is useful or necessary in NL-to-SQL applications, though this may serve as a compelling opportunity for future research.

NL-to-SQL Benchmarks

Spider [119], soon to be superseded by a more challenging benchmark for the LLM era, was a popular NL-to-SQL benchmark that still offers a publically-available dataset consisting of 166 multi-table databases and 1,034 NL questions and gold queries over the databases in a development dataset. *Spider-Syn* [26] and Spider-Realistic [26] are extensions of the Spider benchmark that perform NL question synonym replacement to reduce the occurrences of lexical matching between NL question keywords and schema identifiers. *BIRD* [55] is an emergent benchmark containing 95 large databases over 37 domains that seeks to better replicate real-world databases in order to better challenge highly capable LLM-based NL-to-SQL systems. While Spider and its variants as well as BIRD intend to better-replicate real-world database designs,

our naturalness-focused analysis indicates that their schema identifiers are more natural than those we encountered in our real-world database selection process (see the statistics in Figure 3.3). Additionally, Spider and BIRD both evaluate performance using either exact set matching or execution result set comparison while we use the more pragmatic set-superset matching as proposed in [25] and schema linking-specific recall metrics.

Archerfish [25] is a benchmarking framework that relaxes execution matching and accounts for semantic ambiguity in NL questions by allowing for multiple correct answers derived from candidate key analysis. This framework relies on the binary "correct, or not" evaluation approach common to other benchmarks, whereas in addition to relaxed execution matching, SNAILS evaluates target schema linking performance via query identifier recall. Overall, we find that our benchmark and findings complement this existing and ongoing research by enhancing our ability to target specific schema-related aspects of NL-to-SQL performance in future NLI development.

Impacts of Schema on NL-to-SQL Performance

Spider-Syn [26] demonstrates degraded NL-to-SQL performance of language models trained for NL-to-SQL tasks when the occurrence of lexical matching between NL questions and schema identifiers is reduced. This approach differs from our experiments in that it evaluates a LM specifically trained on NL-to-SQL tasks using the Spider training set as opposed to the more general-purpose foundational LLMs evaluated in this work. They also make no apparent attempt to reduce the naturalness of database schema identifiers.

Semantics-preserving schema transformation is a design feature of MT-teql [59], an NL-to-SQL evaluation framework that modifies natural language utterances and schema properties to stress LM robustness. MT-teql provides a holistic view of the effect of NL utterance variances and schema design on LM performance. However, it does not address the question of schema identifier naturalness, nor does it make modifications to schema elements that are necessary for answer generation.

Some recent work has examined the effects of schema ambiguity, where semantically different tables or columns have identical or synonymous names. Schema ambiguity, where a schema contains one or more semantically similar pairs of elements, degrades semantic parsing (i.e., NL-to-SQL) performance by recalling undesired tables or columns in response to a NL question that contains patterns or keywords that align with more than one schema element in the latent space [79]. Documentation, combined with agent-based column selection, can improve Text-to-SQL performance in the presence of data and schema ambiguity [36]. Though we did not focus on ambiguity in our work, identifier naturalness and ambiguity are complementary efforts that provide a potential future direction for the expansion of the SNAILS benchmark artifacts.

5.3 Related Work for Skalpel

Evaluating Schema Subsetting Methods

CRUSH [47] is a recent work that performs subsetting evaluation of a novel halucination-based subsetting method and compares it to variations of dense passage retrieval. The authors provide benchmark datasets including SpiderUnion, and BirdUnion–unions of the disjoint schemas in the Spider [120] and Bird [54] benchmarks respectively. They also introduce SocialDB, a collection of database schemas and schema descriptions without associated database instances and NL-SQL question-query pairs. In our work, we extend this line of research further by adopting the SNAILS collection to better-represent real-world schema subsetting challenges. Additionally, we adopt a more robust benchmark evaluation strategy by introducing precision and f1 metrics in addition to the recall metric used by the CRUSH authors. Finally, our subsetting method comparison is more representative of the current SOTA in the NL-to-SQL domain by evaluating subsetting methods described by systems with top placement on current benchmarks.

A recent ArXiv preprint that critically evaluates the usefulness of schema subsetting using SoTA LLMs suggests that as LLM capability increases, the need for schema subsetting diminishes [60]. In this work, the authors evaluate four LLM-based schema subsetting methods using the Bird SQL benchmark and measure performance using execution accuracy, false positive

rates during subsetting, and schema linking recall. While this concurrent work bears some resemblance to our SKALPEL project, we provide additional measures of recall, precision, and F1 for various combinations of table and column identifiers. We also extend the scope of subsetting evaluation to non-LLM-based subsetting methods such as semantic similarity search- and finetuned classifier-based approaches. In addition to the Bird benchmark, we also evaluate SoTA linking methods using the Spider2 and SNAILS benchmarks.

The authors of the *In-depth Analysis of LLM-based Schema Linking* [43] concurrently and independently developed a schema subsetting (or schema linking) evaluation methodology focused on LLM-based schema linking approaches in which they replicate several LLM-prompting methods for schema linking and evaluate them in terms of precision, recall, and accuracy using the Spider and Bird benchmarks [120, 54]. Our work complements this approach and makes use of the Spider 2 and SNAILS [49, 57] benchmarks which contain very large schemas. We also opt to reproduce existing subsetting methods using original code, and additionally evaluate subsetting performance in terms of token usage as well as both online inference and offline pre-processing time requirements.

Chapter 6

Conclusion and Future Work

Bibliography

[1] NPS IRMA portal. https://irma.nps.gov/Portal/. Accessed: April 2023.

[2] SAP TABLES. https://sap.erpref.com/. Accessed: June 2023.

[3] Survey of occupational injuries and illnesses data, 2020.

[4] grammars-v4. https://github.com/antlr/grammars-v4, 2022.

[5] Report Card Database 2021-22. https://data.nysed.gov/files/essa/21-22/SRC2022.zip, 2022. Accessed: May 2023.

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

[7] Alireza Ahadi, Julia Prior, Vahid Behbood, and Raymond Lister. A quantitative study of the relative difficulty for novices of writing seven different types of sql queries. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, page 201–206, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Removed author list for anonymity. Speakql: Towards speech-driven multimodal querying of structured data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2363–2374, 2020. Anonymized version included as supplementary material in PCS.

[9] A. Begel and S.L. Graham. An assessment of a speech-based programming environment. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*, pages 116–120, Sep. 2006.

[10] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. Soda: Generating sql for business users. *Proc. VLDB Endow.*, 5(10):932–943, jun 2012.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner,

Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[12] Ruisheng Cao, Lu Chen, Zhi Chen, Yanbin Zhao, Su Zhu, and Kai Yu. LGESQL: Line graph enhanced text-to-SQL model with mixed local and non-local relations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2541–2555, Online, August 2021. Association for Computational Linguistics.

[13] Zhenbiao Cao, Yuanlei Zheng, Zhihao Fan, Xiaojin Zhang, Wei Chen, and Xiang Bai. Rsl-sql: Robust schema linking in text-to-sql generation, 2024.

[14] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade. Sequel 2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev.*, 20(6):560–575, November 1976.

[15] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, page 249–264, New York, NY, USA, 1974. Association for Computing Machinery.

[16] Pankaj Kumar Choudhary. Naming conventions in sql. https://www.c-sharpcorner.com/UploadFile/f0b2ed/what-is-naming-convention/, December 2022. Last accessed on 2024-01-01.

[17] Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. Canine: Pre-training an efficient tokenization-free encoder for language representation. *Transactions of the Association for Computational Linguistics*, 10:73–91, 2022.

[18] Robert Cook. Field Data for Assateague Island National Seashore Amphibian and Reptile Inventory. https://irma.nps.gov/DataStore/Reference/Profile/2236826, 2016. Accessed: April 2023.

[19] LTC Jason Crist. From the server to the battlefield, how data scientists are key to winning future conflicts. https://www.army.mil/article/249036/from_the_server_to_the_battlefield_how_data_scientists_are_key_to_winning_future_conflicts, 2021. Accessed: 2023-01-20.

[20] Till Doehmen, Radu Geacu, Madelon Hulsebos, and Sebastian Schelter. Schemapile: A large collection of relational database schemas. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2024.

[21] Xuemei Dong, Chao Zhang, Yuhang Ge, Yuren Mao, Yunjun Gao, lu Chen, Jinshu Lin, and Dongfang Lou. C3: Zero-shot text-to-sql with chatgpt, 2023.

[22] Yusuf Denizay Dönder, Derek Hommel, Andrea W Wen-Yi, David Mimno, and Unso Eun Seo Jo. Cheaper, better, faster, stronger: Robust text-to-sql without chain-of-thought or fine-tuning, 2025.

[23] Thomas Evans. Great Smoky Mountains All Taxa Biodiversity Inventory (ATBI) Plot Vegetation Monitoring Database. https://irma.nps.gov/DataStore/Reference/Profile/2221324, 2015. Accessed: April 2023.

[24] Catherine Finegan-Dollak, Jonathan K. Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. Improving text-to-SQL evaluation methodology. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 351–360, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[25] Avrilia Floratou, Fotis Psallidas, Fuheng Zhao, Shaleen Deep, Gunther Hagleither, Wangda Tan, Joyce Cahoon, Rana Alotaibi, Jordan Henkel, Abhik Singla, Alex Van Grootel, Brandon Chow, Kai Deng, Katherine Lin, Marcos Campos, Venkatesh Emani, Vivek Pandit, Victor Shnayder, Wenjing Wang, and Carlo Curino. Nl2sql is a solved problem... not! In *Proceedings of the CIDRDB 2024 Conference*, January 2024.

[26] Yujian Gan, Xinyun Chen, Qiuping Huang, Matthew Purver, John R. Woodward, Jinxia Xie, and Pengsheng Huang. Towards robustness of text-to-SQL models against synonym substitution. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2505–2515, Online, August 2021. Association for Computational Linguistics.

[27] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R. Woodward, John Drake, and Qiaofu Zhang. Natural SQL: Making SQL easier to infer from natural language specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2030–2042, Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[28] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. Text-to-sql empowered by large language models: A benchmark evaluation, May 2024.

[29] Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, Jinyang Gao, Liyu Mou, and Yu Li. A preview of xiyan-sql: A multi-generator ensemble framework for text-to-sql, 2025.

[30] General Services Administration. Security policy for generative artificial intelligence (ai) large language models (llms). https://www.gsa.gov/directives-library/security-policy-for-generative-artificial-intelligence-ai-large-language-models-llms, 2023. Last accessed on 2024-05-28.

[31] Frederic Piesschaert Gert Van Spaendonk, Jo Loos. Database naming conventions. https://inbo.github.io/tutorials/tutorials/database_conventions/. Last accessed on 2024-01-01.

[32] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xieand Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[33] Sree Hari Krishnan Parthasarathi, Lu Zeng, and Dilek Hakkani-Tür. Conversational text-to-sql: An odyssey into state-of-the-art and challenges ahead. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5, 2023.

[34] Mohamed A. G. Hazber, Ruixuan Li, Yuxi Zhang, and Guandong Xu. An approach for mapping relational database into ontology. In *2015 12th Web Information System and Application Conference (WISA)*, pages 120–125, 2015.

[35] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024.

[36] Zezhou Huang, Pavan Kalyan Damalapati, and Eugene Wu. Data ambiguity strikes back: How documentation improves GPT's text-to-SQL. In *NeurIPS 2023 Second Table Representation Learning Workshop*, 2023.

[37] Klamath Inventory and Monitoring Network. Exotic and Invasive Plants Monitoring Database. https://irma.nps.gov/DataStore/Reference/Profile/2288667, 2021. Accessed: April 2023.

[38] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Comput. Surv.*, nov 2022. Just Accepted.

[39] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Lilong Jiang, Michael Mandel, and Arnab Nandi. Gesturequery: A multitouch database query interface. *Proc. VLDB Endow.*, 6(12):1342–1345, aug 2013.

[41] Harshit Joshi, Abishai Ebenezer, José Cambronero, Sumit Gulwani, Aditya Kanade, Vu Le, Ivan Radiček, and Gust Verbruggen. Flame: A small language model for spreadsheet formulas, 2023.

[42] Seth Judge and Kevin Kozar. Pacific Island Network Landbird Monitoring Dataset. https://irma.nps.gov/DataStore/Reference/Profile/2300107, 2023. Accessed: April 2023.

[43] George Katsogiannis-Meimarakis, Katsiaryna Mirylenka, Paolo Scotton, Francesco Fusco, and Abdel Labbi. In-depth analysis of llm-based schema linking. In *Proceedings of the 29th International Conference on Extending Database Technology (EDBT 2026)*, volume 29 of *Advances in Database Technology*, pages 117–130, Tampere, Finland, March 2026.

[44] Brad Kelechava. The sql standard - iso/iec 9075:2016 (ansi x3.135), Jul 2020.

[45] Hyeonji Kim, Byeong Hoon So, Wook Shin Han, and Hongrae Lee. Natural language to sql: Where are we today? *Proceedings of the VLDB Endowment*, 13:1737–1750, 6 2020.

[46] Young-Ho Kim, Bongshin Lee, Arjun Srinivasan, and Eun Kyoung Choe. Data@hand: Fostering visual exploration of personal data on smartphones leveraging speech and touch interaction. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Mayank Kothyari, Dhruva Dhingra, Sunita Sarawagi, and Soumen Chakrabarti. CRUSH4SQL: Collective retrieval using schema hallucination for Text2SQL. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 14054–14066, Singapore, December 2023. Association for Computational Linguistics.

[48] Tobias Kuhn. A Survey and Classification of Controlled Natural Languages. *Computational Linguistics*, 40(1):121–170, 03 2014.

[49] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*, 2024.

[50] Fei Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, 8(1):73–84, sep 2014.

[51] Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. Resdsql: decoupling schema linking and skeleton parsing for text-to-sql, 2023.

[52] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql. *Proc. ACM Manag. Data*, 2(3), May 2024.

[53] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. Codes: Towards building open-source language models for text-to-sql, may 2024.

[54] Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Chenhao Ma, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls, 2023.

[55] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls, 2024.

[56] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! 2023.

[57] Kyle Luoma and Arun Kumar. Snails: Schema naming assessments for improved llm-based sql inference. *Proc. ACM Manag. Data*, 3(1), February 2025.

[58] Gabriel Lyons, Vinh Tran, Carsten Binnig, Ugur Cetintemel, and Tim Kraska. Making the case for query-by-voice with echoquery. volume 26-June-2016, pages 2129–2132. Association for Computing Machinery, 6 2016.

[59] Pingchuan Ma and Shuai Wang. Mt-teql: Evaluating and augmenting neural nlidb on real-world linguistic and schema variations. *Proc. VLDB Endow.*, 15(3):569–582, nov 2021.

[60] Karime Maamari, Fadhil Abubaker, Daniel Jaroslawicz, and Amine Mhedhbi. The death of schema linking? text-to-sql in the age of well-reasoned language models, 2024.

[61] I. Scott MacKenzie. *Human-Computer Interaction: An Empirical Research Perspective*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[62] Daphne Miedema, Efthimia Aivaloglou, and George Fletcher. Identifying sql misconceptions of novices: Findings from a think-aloud study. *ACM Inroads*, 13(1):52–65, feb 2022.

[63] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Comput. Surv.*, 56(2), sep 2023.

[64] Mehdi Mohammadpoor and Farshid Torabi. Big data analytics in oil and gas industry: An emerging trend. *Petroleum*, 6(4):321–328, 2020. SI: Artificial Intelligence (AI), Knowledge-based Systems (KBS), and Machine Learning (ML).

[65] Jesse Mu and Advait Sarkar. Do we need natural language? exploring restricted language interfaces for complex domains. In *37th Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems (CHI '19 Extended Abstracts)*. ACM, May 2019.

[66] Ian Muirhead. Northern Great Plains Fire Management: FFI Database. https://irma.nps.gov/DataStore/Reference/Profile/2297267, 2021. Accessed: April 2023.

[67] Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies, 2023.

[68] National Center for Statistics and Analysis. Overview of the 2021 crash investigation sampling system, December 2022. Traffic Safety Facts Research Note. Report No. DOT HS 813 397.

[69] Sadia Nowrin, Patricia OrdóñEz, and Keith Vertanen. Exploring motor-impaired programmers' use of speech recognition. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*, ASSETS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[70] William C. Ogden and Susan R. Brooks. Query languages for the casual user: Exploring the middle ground between formal and natural languages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, page 161–165, New York, NY, USA, 1983. Association for Computing Machinery.

[71] OpenAI. Chatgpt: Optimizing language models for dialogue. https://openai.com/blog/chatgpt/, [2022].

[72] OpenAI. Openai api documentation. https://platform.openai.com/docs/guides/gpt, 2023. Last accessed on 2023-10-30.

[73] OpenAI. Openai tokenizer. https://github.com/openai/tiktoken, 2023. Last accessed on 2023-10-30.

[74] OpenAI. gpt-oss-120b and gpt-oss-20b model card, 2025.

[75] OpenAI. Introducing gpt-4.1 in the api, 2025. Accessed: 2025-05-02.

[76] OpenAI. Introducing GPT-4.1 in the API, April 2025. Accessed: 2025-09-23.

[77] Oracle. Database object names and qualifiers. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/Database-Object-Names-and-Qualifiers.html, November 2024. Last accessed on 2025-01-04.

[78] Oracle. Table naming standards and conventions. https://docs.oracle.com/cd/E92917_01/PDF/8.1.x.x/common/HTML/DM_Naming/2_Table_and_Column_Naming_Standards.htm, November 2024. Last accessed on 2025-01-04.

[79] Simone Papicchio, Paolo Papotti, and Luca Cagliero. Evaluating ambiguous questions in semantic parsing. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*, pages 338–342, 2024.

[80] Simone Papicchio, Paolo Papotti, and Luca Cagliero. Qatch: benchmarking sql-centric tasks with table representation learning models on your data. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

[81] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 579–598, New York, NY, USA, 2014. Association for Computing Machinery.

[82] Phind. Phind-codellama-34b-v2. https://huggingface.co/Phind/Phind-CodeLlama-34B-v2, 2023.

[83] Marie-Laurence Poujois. Localized demo databases now available for sap business one 10.0 fp 2011. https://blogs.sap.com/2021/01/29/localized-demo-databases-now-available-for-sap-business-one-10.0-fp-2011/, January 2021. Accessed: April 2023.

[84] Mohammadreza Pourreza and Davood Rafiei. 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*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

[85] Mohammadreza Pourreza and Davood Rafiei. Dts-sql: Decomposed text-to-sql with small large language models, 2024.

[86] Ge Qu, Jinyang Li, Bowen Li, Bowen Qin, Nan Huo, Chenhao Ma, and Reynold Cheng. Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation, 2024.

[87] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022.

[88] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[89] G. A. Radja, E.-Y. Noh, and F. Zhang. Crash investigation sampling system 2021 analytical user's manual, December 2022. Accessed: April 2023.

[90] P. Reisner. Use of psychological experimentation as an aid to development of a query language. *IEEE Transactions on Software Engineering*, SE-3(3):218–229, May 1977.

[91] Phyllis Reisner, Raymond F. Boyce, and Donald D. Chamberlin. Human factors evaluation of two data base query languages: Square and sequel. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, page 447–452, New York, NY, USA, 1975. Association for Computing Machinery.

[92] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

[93] Diptikalyan Saha, Avrilia Floratou, Karthik Sankaranarayanan, Umar Farooq Minhas, Ashish R. Mittal, and Fatma Özcan. Athena: An ontology-driven system for natural language querying over relational data stores. *Proc. VLDB Endow.*, 9(12):1209–1220, aug 2016.

[94] Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. PICARD: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901. Association for Computational Linguistics, November 2021.

[95] Vidya Setlur and Melanie Tory. How do you converse with an analytical chatbot? revisiting gricean maxims for designing analytical conversational behavior. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[96] Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. Smelly relations: Measuring and understanding database schema quality. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 55–64, New York, NY, USA, 2018. Association for Computing Machinery.

[97] Vladislav Shkapenyuk, Divesh Srivastava, Theodore Johnson, and Parisa Ghane. Automatic metadata extraction for text-to-sql, 2025.

[98] StackOverflow. Database, table and column naming conventions? https://stackoverflow.com/questions/7662/database-table-and-column-naming-conventions. Last accessed on 2024-01-01.

[99] Charles Stefanic. Wildlife Observations Database: Craters of the Moon National Monument and Preserve 1921-2021. https://irma.nps.gov/DataStore/Reference/Profile/2192964, 2021. Accessed: April 2023.

[100] Kil Soo Suh and A. Milton Jenkins. A comparison of linear keyword and restricted natural language data base interfaces for novice users. *Information Systems Research*, 3(3):252–272, 1992.

[101] Alane Laughlin Suhr, Kenton Lee, Ming-Wei Chang, and Pete Shaw. Exploring unexplored generalization challenges for cross-database semantic parsing. In *ACL 2020*, 2020.

[102] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. Chess: Contextual harnessing for efficient sql synthesis, 2024.

[103] Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

[104] Gemini Team. Gemini: A family of highly capable multimodal models, 2024.

[105] Llama Team. The llama 3 herd of models, 2024.

[106] PaLM 2 Team. Palm 2 technical report. Technical report, 2023.

[107] PGVector Team. Pgvector: Open-source vector similarity search for postgres. https://github.com/pgvector/pgvector/, 2021. Accessed: 2025-09-25.

[108] The Mosaic Research Team. Introducing dbrx: A new state-of-the-art open llm. https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm, 2024.

[109] The Snowflake Research Team. Snowflake arctic: The best llm for enterprise ai - efficiently intelligent, truly open. https://www.snowflake.com/blog/arctic-open-efficient-foundation-language-models-snowflake/, 2024.

[110] Immanuel Trummer. Demonstrating the voice-based exploration of large data sets with cicerodb-zero. *Proc. VLDB Endow.*, 13(12):2869–2872, sep 2020.

[111] Fatma undefinedzcan, Abdul Quamar, Jaydeep Sen, Chuan Lei, and Vasilis Efthymiou. State of the art and open challenges in natural language interfaces to data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2629–2636, New York, NY, USA, 2020. Association for Computing Machinery.

[112] Kiran Vodrahalli, Santiago Ontanon, Nilesh Tripuraneni, Kelvin Xu, Sanil Jain, Rakesh Shivanna, Jeffrey Hui, Nishanth Dikkala, Mehran Kazemi, Bahare Fatemi, Rohan Anil, Ethan Dyer, Siamak Shakeri, Roopali Vij, Harsh Mehta, Vinay Ramasesh, Quoc Le, Ed Chi, Yifeng Lu, Orhan Firat, Angeliki Lazaridou, Jean-Baptiste Lespiau, Nithya Attaluri, and Kate Olszewska. Michelangelo: Long context evaluations beyond haystacks via latent structure queries, 2024.

[113] Bailin Wang, Richard Shin, Xiaodong Liu, Alex Polozov, and Matthew Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. In *ACL 2020*, June 2020.

[114] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. Mac-sql: A multi-agent collaborative framework for text-to-sql, 2024.

[115] Lihan Wang, Bowen Qin, Binyuan Hui, Bowen Li, Min Yang, Bailin Wang, Binhua Li, Jian Sun, Fei Huang, Luo Si, and Yongbin Li. Proton: Probing schema linking information from pre-trained language models for text-to-sql parsing. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 1889–1898, New York, NY, USA, 2022. Association for Computing Machinery.

[116] Tianbao Xie, Chen Henry Wu, Peng Shi, Ruiqi Zhong, Torsten Scholak, Michihiro Yasunaga, Chien-Sheng Wu, Ming Zhong, Pengcheng Yin, Sida I. Wang, Victor Zhong, Bailin Wang, Chengzu Li, Connor Boyle, Ansong Ni, Ziyu Yao, Dragomir Radev, Caiming Xiong, Lingpeng Kong, Rui Zhang, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models, 2022.

[117] Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. Opensearch-sql: Enhancing text-to-sql with dynamic few-shot and consistency alignment, 2025.

[118] Zhuoming Xu, Shichao Zhang, and Yisheng Dong. Mapping between relational database schema and owl ontology for deep annotation. In *2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2006 Main Conference Proceedings)(WI'06)*, pages 548–552, 2006.

[119] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, 2018. Association for Computational Linguistics.

[120] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, 2018. Association for Computational Linguistics.

[121] Lu Zeng, Sree Hari Krishnan Parthasarathi, and Dilek Hakkani-Tur. N-best hypotheses reranking for text-to-sql systems. In *2022 IEEE Spoken Language Technology Workshop (SLT)*, pages 663–670, 2023.

[122] Dun Zhang, Jiacheng Li, Ziyang Zeng, and Fulong Wang. Jasper and stella: distillation of sota embedding models, 2025.

[123] Jiani Zhang, Zhengyuan Shen, Balasubramaniam Srinivasan, Shen Wang, Huzefa Rangwala, and George Karypis. NameGuess: Column name expansion for tabular data. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13276–13290, Singapore, December 2023. Association for Computational Linguistics.