Are LLMs Overkill for Databases?: A Study on the Finiteness of SQL

Anonymous ACL submission

Abstract

Translating natural language to SQL for data retrieval has become more accessible thanks to code generation LLMs. But how hard is it to generate SQL code? While databases can become unbounded in complexity, the complexity of queries is bounded by real life utility and human needs. With a sample of 376 databases, we show that SQL queries, as translations of natural language questions are finite in *practical* complexity. There is no clear monotonic relationship between increases in database table count and increases in complexity of SQL queries. In their template forms, SQL queries follow a Power Law-like distribution of frequency where 70% of our tested queries can be covered with just 13% of all template types, indicating that the high majority of SQL queries are predictable. This suggests that while LLMs for code generation can be useful, in the domain of database access, they may be operating in a narrow, highly formulaic space where templates could be safer, cheaper, and auditable.

1 Introduction

The text-to-SQL task predates the rise of language models (Woods, 1973; Harris, 1977), but is currently dominated by LLM-based approaches. For instance, on the premier BIRD text-to-SQL benchmark, almost all of the top ranking methods on the public site are either agent-based (Pourreza and Rafiei, 2024) or LLM-based (LongShine AI Research, 2025). While these approaches are effective, they can get expensive, costing several dollars in token-count per query, and potentially unreliable, whereas enterprise database queries are highly sensitive to efficiency and security. To what extent is this code boilerplate, predictable, or templatizable? In this work, we probe whether full code generation is really necessary for SQL applications.

We explore the *practical* boundaries of SQL through empirical experiments. We curate 376 database schemas from benchmarks and open source database repository, *drawSQL*[1] and generate over 20,000 natural language questions (NLQ) and matching SQL queries based on schema designs. Using counts of proxies for complexity such as JOIN clause count, subclause count, and length of query, we show that there is no monotonic relationship between increase in table count of databases and increase in proxy count. We observe there is a "ceiling" of proxy-based complexity in SQL queries regardless of the complexity and size of the database schema design.

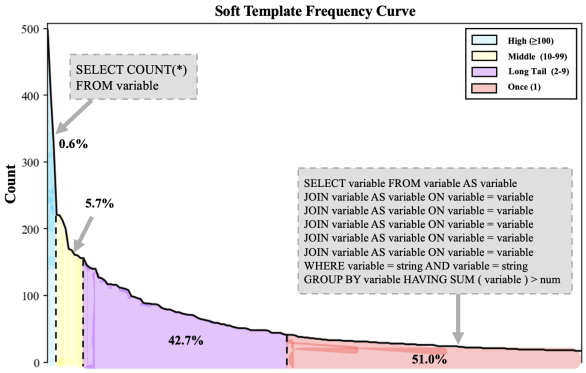


Figure 1: Frequency curve of soft templates.

When templated and analyzed by frequency, SQL queries exhibit a power-law-like distribution (Figure 1). We find that the top 13% of templates account for 90% of all queries: only 7 templates cover 10%, and about 40 cover 30%. High-frequency templates tend to be less complex (e.g., fewer JOINs), suggesting that relatively simple structures suffice to cover most user questions. Overall, a few hundred templates are sufficient to capture the vast majority of database queries.

2 Related Work

Text-to-SQL has long been studied as a semantic parsing task using deep learning models (Zhong et al., 2017; Xu et al., 2017). Recent advances

[1] drawsql.app/templates

Source	#DB	#T/DB	#Q
Bird23-train-filtered (Li et al., 2023)	69	7.57	6,601
Spider 1.0 (Yu et al., 2018)	196	5.15	11,245
Spider 2.0-lite (Lei et al., 2024)	15	14.87	287
KaggleDBQA (Lee et al., 2021)	8	2.12	244
drawSQL	88	14.34	2,112
Overall	**376**	**8.07**	**20,489**

Table 1: Statistics of database schemas collected from text-to-SQL benchmarks and drawSQL. #DB indicates the number of databases from each data source, #T/DB indicates the average number of tables per database, and #Q indicates the number of SQL queries per source. For drawSQL, we manually generate three difficulty levels of NLQs: easy, medium, and difficult. Appendix A provides concrete examples.

in LLMs have shifted this paradigm, with state-of-the-art systems adopting LLM-driven pipelines with iterative reasoning, self-correction, and agent-based workflows (Pourreza and Rafiei, 2023).

While LLM-based SQL generation has significantly improved accuracy, limited work has examined the empirical *science* of database queries. To our knowledge, this is the first work to explore the boundaries of SQL queries, enabled by the recent availability of large-scale, high-quality AI-generated SQL data that allows systematic analysis of query patterns and user intents.

3 Methodology

3.1 Database and Query Collection

We collect NLQ–SQL pairs and database schemas from online benchmarks and an open source repository, *drawSQL*. We use database schemas and gold label queries from classic text-to-SQL benchmarks such as Spider 1.0 (Yu et al., 2018). We take schema designs from *drawSQL* and construct NLQ–SQL pairs using Claude-Sonnet-4.6 (Anthropic, 2026) with varying levels of difficulty. Table 1 shows the statistics of database schemas collected from these sources. We manually verify all generated NLQ–SQL pairs to ensure semantic correctness and executable validity. To ensure consistency in template extraction and metric computation, we restrict all SQL queries to SQLite dialect.

3.2 SQL Templates Extraction

To analyze the frequency of repeat SQL queries by "type", we templatize or generalize the queries with uniform methods. We have two different categories of SQL templates: (1) **hard templates** keep more distinctive traits, and (2) **soft templates** uniformize more strictly to allow for looser categorization.

Type 1: Hard Templates. We construct hard SQL templates by abstracting dataset-specific identifiers and literals from SQL queries while preserving their structural and logical forms. This process normalizes queries into canonical templates that represent the underlying SQL pattern. Hard templates have stricter and more detailed extraction rules to transform SQL queries into SQL templates. We keep entity types such as table, column, and alias names as placeholders. Appendix C shows detailed hard templates extraction rules.

Type 2: Soft Templates. In addition to the masking presented by hard templates, we further generalize the variables and focus on preserving the underlying SQL structure and keywords. Function-specific keywords such as alias and table are all uniformly treated as variables. With this transformation, some distinct "hard" templates could be mapped to the same soft template. Appendix D shows detailed soft template extraction rules.

Example: Template Generation Procedure. The original SQL query is translated into templates in stages: *SQL* → *Hard Template* → *Soft Template*.

SQL Query

```
SELECT c.name
FROM customers AS c
JOIN orders AS o ON c.id = o.customer_id
WHERE o.amount > 100
ORDER BY o.amount DESC LIMIT 10
```

Hard Template

```
SELECT table_alias0.col_name
FROM table_name AS table_alias0
JOIN table_name AS table_alias1
ON table_alias0.col_name = t
able_alias1.col_name
WHERE table_alias1.col_name > num
ORDER BY table_alias1.col_name
DESC LIMIT num
```

Soft Template

```
SELECT variable
FROM variable AS variable
JOIN variable AS variable
ON variable = variable
WHERE variable > num
ORDER BY variable DESC LIMIT num
```

This example illustrates how hard templates preserve precise alias and schema structure, while soft templates collapse identifier roles into a single variable token. Appendix B presents more hard and soft template extraction examples.

3.3 Proxies for Complexity

One way to quantify "complexity" of SQL queries is to count the appearance of certain proxy traits. We identify 6 proxies used to characterize SQL query complexity. These include structural properties of queries (number of tables, joins, subqueries, and maximum nesting depth) as well as analytical operations such as aggregations with `GROUP BY` and advanced SQL features (e.g., window functions, percentile functions, `FILTER`, set operations, and CTEs). Table 2 provides the full list of all proxies.

4 Results

Non-monotonic SQL Query Complexity Trend There is no observable monotonic relationship between increase in database schema complexity and increase in SQL query complexity. SQL queries do not grow more and more complex as database schemas do. Figure 2 presents the moving average values of six proxies (window size = 15). All curves exhibit a similar pattern: they increase to some *peak value* at a *breaking point*, and then gradually stall or decline, suggesting that the SQL query complexity measured by these proxies does not continuously grow with the number of tables in the database. Instead, the complexity of SQL queries appears to follow a bounded trend rather than monotonically increasing as database size grows. Table 5 in Appendix F reports the Spearman Correlation results, showing the absence of a monotonic relationship between the proxies and the table count.

Table 3 summarizes the statistics of the six proxies across all SQL queries. Most of the proxies appear on average less than once per query. The count of table variables is higher: there are queries that include as many as 27 table variables. But the average table count seems bounded at around 2.

Power Law-like Distribution of Template Frequency. When templatized and grouped by frequency, the 20,489 queries follow a near Power Law distribution. The hard and soft templates both exhibit long-tail distributions: a small number of templates appear very frequently while most templates occur only a few times or once. Figure 4 in Appendix E shows the long-tail pattern. This

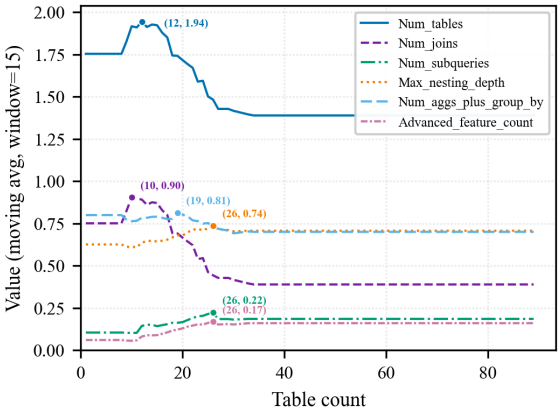


Figure 2: Moving average values of six proxies (window size = 15).

indicates a limited set of templates dominate while the majority of templates have more specialized SQL structures. The Power Law hypothesis is rejected by bootstrap goodness-of-fit test (Clauset et al., 2009) ($p \approx 0$), but the curves are Power Law-like in the tail.

We plot hard and soft template frequencies on log-log scale in figure 3 showing an approximately linear trend. We draw fitted lines following the linear representation of the Power Law distribution: $\log P(x) = -\alpha \log x + C$,where C is a constant. The fitted lines for hard and soft template frequency log–log scale distributions are $\log y = -0.7258 \log x + 6.1106$ (hard) and $\log y = -0.8858 \log x + 7.1968$ (soft), respectively. Not surprisingly, soft templates follow a steeper distribution, where the "top" frequency templates have higher counts. Table 7 in Appendix 7 shows soft templates have 28 templates with more than 100 counts whereas hard templates have just 19. Both distributions have long tails of single appearance queries making up 60% of the hard template distribution.

70% SQL queries can be covered using only 13.19% soft templates. Table 4 shows that a small number of templates can cover a disproportionate volume of queries with the appropriate schema linking of table and column names. The Power Law effect is more dramatic with soft templates, where up to 70% of all queries are derived from about 13% of templates. It takes only about 600 soft templates to cover 70% of 20,489 queries. The most frequent 140 templates cover about 50% of all queries. The two most frequent soft templates are `SELECT variable FROM variable WHERE`

Proxy	Definition
Num_tables	# of distinct tables referenced (i.e., number of data sources involved)
Num_joins	# of JOIN operations (capturing cross-table relational reasoning)
Num_subqueries	# of nested subqueries (capturing hierarchical reasoning)
Max_nesting_depth	Maximum depth of nested queries (i.e., levels of embedding)
Num_aggs_plus_group_by	# of aggregation operations (e.g., COUNT, SUM) and GROUP BY clauses
Advanced_feature_count	# of advanced constructs (e.g., window functions, FILTER, set operations, CTEs)

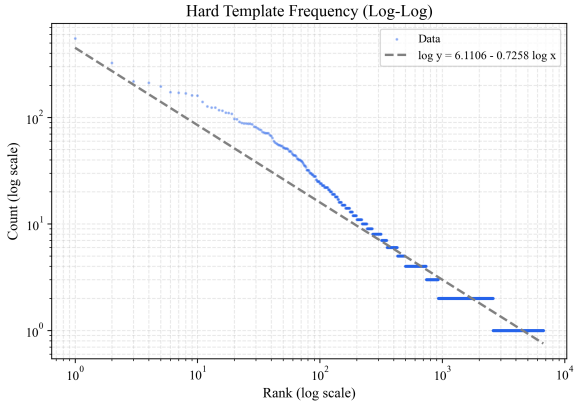
Table 2: Structural proxies for SQL query complexity, capturing aspects such as multi-table interactions, hierarchical structure, and advanced analytical operations.

Table 3: Summary statistics and peak characteristics of six SQL complexity proxies.

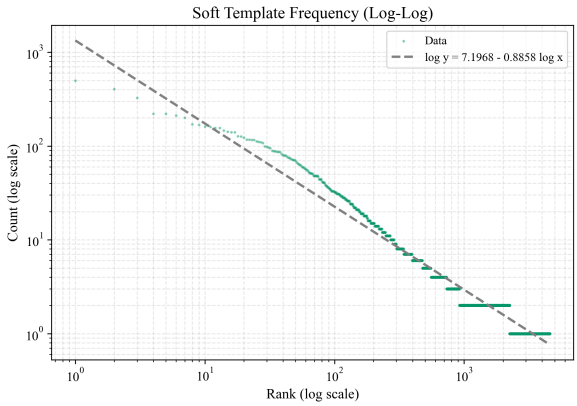
Proxy	Median	Average	Min	Max	Peak Value
Num_tables	2.00	1.73	1	27	1.94
Num_joins	1.00	0.73	0	26	0.90
Num_subqueries	0.00	0.11	0	7	0.22
Max_nesting_depth	1.00	0.62	0	5	0.74
Num_aggs_plus_group_by	0.00	0.74	0	9	0.81
Advanced_feature_count	0.00	0.06	0	6	0.17

Table 4: Number of templates required to cover different proportions of SQL queries. Numbers in parentheses denote the cumulative percentage of templates.

Template Type	Query Coverage (%)					
	10	30	50	**70**	90	100
Hard	9 (0.13%)	58 (0.87%)	306 (4.57%)	**1616 (24.12%)**	4677 (69.82%)	6699 (100%)
Soft	7 (0.15%)	42 (0.92%)	140 (3.05%)	**605 (13.19%)**	2565 (55.92%)	4587 (100%)



(a) Log–log plot of Hard Templates



(b) Log–log plot of Soft Templates

Figure 3: Log–log plots for hard and soft templates.

`variable = string` and `SELECT COUNT(*) FROM variable`. For example, the former corresponds to NLQ such as "List all users whose status is 'active'," while the latter corresponds to NLQ such as "How many users are there in total?"

Table 6 in Appendix H also shows the average proxy count of complexity negatively correlates with frequency as we expected. The simpler templates are more likely to be used in more contexts.

5 Conclusion

Relational and other databases are the middle ground of data storage and organization between human understanding and computational efficiency. SQL was proposed in the 1970s as a way to bridge the gap between human natural language and computer execution. LLMs have emerged as the additional layer over this with its seemingly magical code generation capabilities. But our experiments show LLMs code generation in the SQL domain may not be as impressive as they operate under a ceiling of complexity where just a few hundred templates cover the vast majority of query cases.

Limitations

One limitation of template-based analysis is that SQL queries do not have canonical forms and many different queries can be isomorphic. In other words, it is possible to generate code that looks completely different, and therefore have different templates, but retrieve the same "answer" or table result. We experimented with ways of homogenizing the overall style of generated queries to test different levels of maximizing template frequency, but ultimately decided that we wanted to test the queries (and resulting templates) in the most unadulterated, as-found forms.

Acknowledgments

We thank the anonymous reviewers and area chairs for their valuable feedback.

References

Anthropic. 2026. Introducing claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6.

Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. 2009. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703.

Larry R. Harris. 1977. Robot: A high performance natural language data base query system. *Proceedings of the National Computer Conference (NCC)*, pages 229–237.

Dongjoon Lee, Seokhwan Park, Jinhyuk Kim, and Sungjin Lee. 2021. Kaggledbqa: Realistic evaluation of text-to-sql parsers. In *ACL Findings*.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*.

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. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

LongShine AI Research. 2025. Longdata-sql: Scaling text-to-sql with long context modeling. Technical report.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. In *NeurIPS*.

Mohammadreza Pourreza and Davood Rafiei. 2024. Chase-sql: Multi-path reasoning and schema linking for text-to-sql. *arXiv preprint arXiv:2410.01943*.

William A. Woods. 1973. Progress in natural language understanding: An application to lunar geology. *AFIPS Conference Proceedings*, 42:441–450.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. In *ICLR*.

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. 2018. 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*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. In *EMNLP*.

A Examples of Generated NLQs by Difficulty Level

We use the *cachet* database from the drawSQL dataset as an illustrative example, and present NLQ–SQL pairs across three difficulty levels: easy, medium, and difficult.

A.1 Example 1 (Easy, cachet).

NLQ:
How many subscribers are there in total?
SQL Query:

```
SELECT COUNT(*)
FROM subscribers;
```

A.2 Example 2 (Medium, cachet).

NLQ:
How many activity log entries do not have a matching component?
SQL Query:

```
SELECT COUNT(*)
FROM actions a
LEFT JOIN components b
ON a.taggable_id = b.id
WHERE b.id IS NULL;
```

A.3 Example 3 (Hard, cachet).

NLQ:
For each class, count the number of activity log entries, and return only those classes whose count is above the average count across all classes. Sort the results in descending order of the count.
SQL Query:

```
WITH grouped AS (
    SELECT class_name, COUNT(*) AS cnt
    FROM actions
    GROUP BY class_name
)
SELECT *
FROM grouped
WHERE cnt > (
    SELECT AVG(cnt) FROM grouped
)
ORDER BY cnt DESC;
```

B Templates Extraction Examples

To illustrate the template abstraction process, we present several examples showing the transformation pipeline from SQL queries to hard templates and soft templates.

Example 1: Simple Selection

SQL

```
SELECT name FROM employees
WHERE salary > 50000
```

Hard Template

```
SELECT col_name FROM table_name
WHERE col_name > num
```

Soft Template

```
SELECT variable FROM variable
WHERE variable > num
```

Example 2: Join with Aliases

SQL

```
SELECT T1.name
FROM employees AS T1
JOIN departments AS T2
ON T1.dept_id = T2.id
WHERE T2.location = 'NY'
```

Hard Template

```
SELECT table_alias0.col_name
FROM table_name AS table_alias0
JOIN table_name AS table_alias1
ON table_alias0.col_name =
table_alias1.col_name
WHERE table_alias1.col_name = string
```

Soft Template

```
SELECT variable
FROM variable AS variable
JOIN variable AS variable
ON variable = variable
WHERE variable = string
```

Example 3: Aggregation

SQL

```
SELECT department, COUNT(*)
FROM employees
GROUP BY department
ORDER BY COUNT(*) DESC
LIMIT 5
```

Hard Template

```
SELECT col_name, COUNT(*)
FROM table_name
GROUP BY col_name
ORDER BY COUNT(*) DESC
LIMIT num
```

Soft Template

```
SELECT variable, COUNT(*)
FROM variable
GROUP BY variable
ORDER BY COUNT(*) DESC
LIMIT num
```

Example 4: Subquery

SQL

```
SELECT name
FROM employees
WHERE salary >
   (SELECT AVG(salary) FROM employees)
```

Hard Template

```
SELECT col_name
FROM table_name
WHERE col_name >
    (SELECT AVG(col_name)
    FROM table_name)
```

Soft Template

```
SELECT variable
FROM variable
WHERE variable >
   (SELECT AVG(variable) FROM variable)
```

Example 5: CTE with Join and Aggregation

SQL

```
WITH dept_avg AS (
 SELECT dept_id, AVG(salary) AS avg_salary
  FROM employees
  GROUP BY dept_id
)
SELECT d.name
FROM departments d
JOIN dept_avg a
ON d.id = a.dept_id
WHERE a.avg_salary > 70000
```

Hard Template

```
WITH CTE0 AS (
  SELECT col_name, AVG(col_name)
  AS column_alias0
  FROM table_name
  GROUP BY col_name
)
SELECT table_alias0.col_name
FROM table_name AS table_alias0
JOIN CTE0 AS table_alias1
ON table_alias0.col_name =
table_alias1.col_name
WHERE table_alias1.column_alias0 > num
```

Soft Template

```
WITH CTE0 AS (
  SELECT variable, AVG(variable)
  AS variable
  FROM variable
  GROUP BY variable
)
SELECT variable
FROM variable AS table_alias0
JOIN CTE0 AS table_alias1
ON variable = variable
WHERE variable > num
```

C Complete Hard Templates Extraction Rules

The generation procedure consists of the following steps:

1. **Preprocessing.** We remove comments (e.g., `--`, `/* */`) and blank lines from the SQL query and perform case-insensitive matching to standardize the input representation.

2. **Literal Abstraction.** All literal values are replaced with typed placeholders. Numeric constants are mapped to `num`, string literals to `string`, date literals to `date`, boolean values to `boolean`, and `NULL` values to `others`. This step removes value-level variability while preserving type signals.

3. **Schema-aware Identifier Replacement.** Using the database schema, table names are replaced with `table_name` and column references with `col_name`. Identifiers that do not appear in the schema are mapped to special placeholders such as `new_table`, `new_view`, or `new_column` depending on their SQL context.

4. **Alias Normalization.** Table aliases introduced in `FROM` or `JOIN` clauses are normalized as `table_alias0`, `table_alias1`, etc., according to their order of appearance. Column aliases defined via `AS` are similarly replaced with `column_alias0`, `column_alias1`, etc.

5. **Qualified Reference Resolution.** For qualified expressions (e.g., `alias.column`), the qualifier is replaced by its normalized form (e.g., `table_name` or `table_aliasN`), while the column component is mapped to `col_name` or `new_column`.

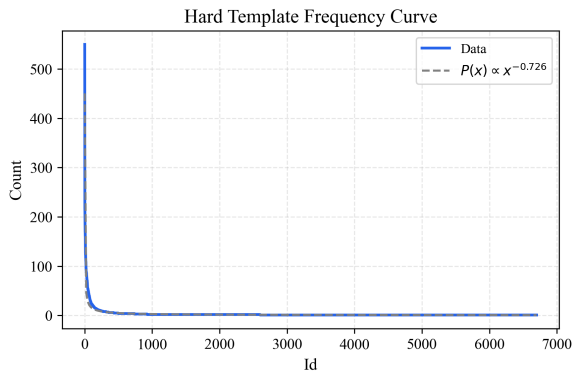
The resulting SQL statement forms a hard template that preserves the structural semantics of the original query while abstracting away dataset-specific details.

D Complete Soft Templates Extraction Rules

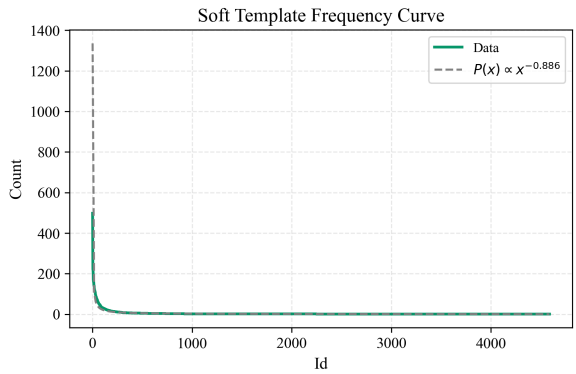
The generation procedure consists of the following steps:

1. **Identifier Generalization.** All identifier placeholders produced during hard template extraction (e.g., `table_name`, `col_name`, `new_table`, `new_column`, `cte`, and alias tokens such as `table_aliasN` and `column_aliasN`) are replaced with a single token `variable`. This token represents any table or column identifier, collapsing different schema roles into a unified symbol.

2. **Literal Type Preservation.** Unlike identifiers, typed literal placeholders introduced in the hard template (e.g., `num`, `string`, `date`,



(a) Hard templates



(b) Soft templates

Figure 4: Frequency curves for hard and soft templates.

boolean, jsonb, others) are preserved. This maintains the semantic role of constants in predicates and clauses such as `LIMIT num` or `variable = string`.

3. **Keyword and Structure Retention.** All SQL keywords, operators, and syntactic structure (e.g., `SELECT`, `JOIN`, `GROUP BY`, `ORDER BY`) remain unchanged so that the generalized template still reflects the underlying query logic.

The resulting SQL statement forms a soft template that preserves the structural semantics of the original query.

E Frequency Curves for Hard and Soft Templates

Figure 4 shows the frequency distributions of hard and soft templates, both of which exhibit a Power Law–like pattern. The fitted distributions follow $P(x) \propto x^{-0.726}$ and $P(x) \propto x^{-0.886}$, respectively.

F Spearman Correlation Test Results

Table 5 shows the Spearman Correlation test results.

Table 5: Spearman correlation test for the monotonic relationship between each proxy and the table count.

Proxy	ρ	**p-value**
Num_tables	-0.3673	2.33×10^{-2}
Num_joins	-0.3839	1.73×10^{-2}
Num_subqueries	0.4481	4.79×10^{-3}
Max_nesting_depth	0.3124	5.62×10^{-2}
Num_aggs_plus_group_by	-0.4263	7.61×10^{-3}
Advanced_feature_count	0.4921	1.70×10^{-3}

Table 6: Average proxy values across template frequency groups. The maximum value in each row is highlighted in bold.

Metric	**Type**	**High**	**Middle**	**Long Tail**	**Once**
Num_tables	Soft	1.41	1.74	1.83	**2.12**
	Hard	1.12	1.55	1.91	**2.21**
Num_joins	Soft	0.41	0.74	0.82	**1.10**
	Hard	0.12	0.55	0.91	**1.20**
Num_subqueries	Soft	0.03	0.11	0.14	**0.23**
	Hard	0.04	0.13	0.11	**0.15**
Max_nesting_depth	Soft	0.41	0.54	0.72	**1.06**
	Hard	0.43	0.54	0.64	**0.86**
Num_aggs_plus_group_by	Soft	0.49	0.64	0.94	**1.09**
	Hard	0.51	0.62	0.87	**0.90**
Advanced_feature_count	Soft	0.00	**0.10**	0.07	0.04
	Hard	0.00	**0.10**	0.07	0.02

G Individual Plots for Six Proxies

In figure 5, each point represents the average value of a proxy metric for a specific table count. The dashed line denotes the moving average of the proxy values with a window size of 15.

H Average Proxy Values Across template frequency groups.

Table 6 presents the average proxy values across hard and soft templates.

I Template Frequency Distribution.

Table 7 shows the template frequency distribution.

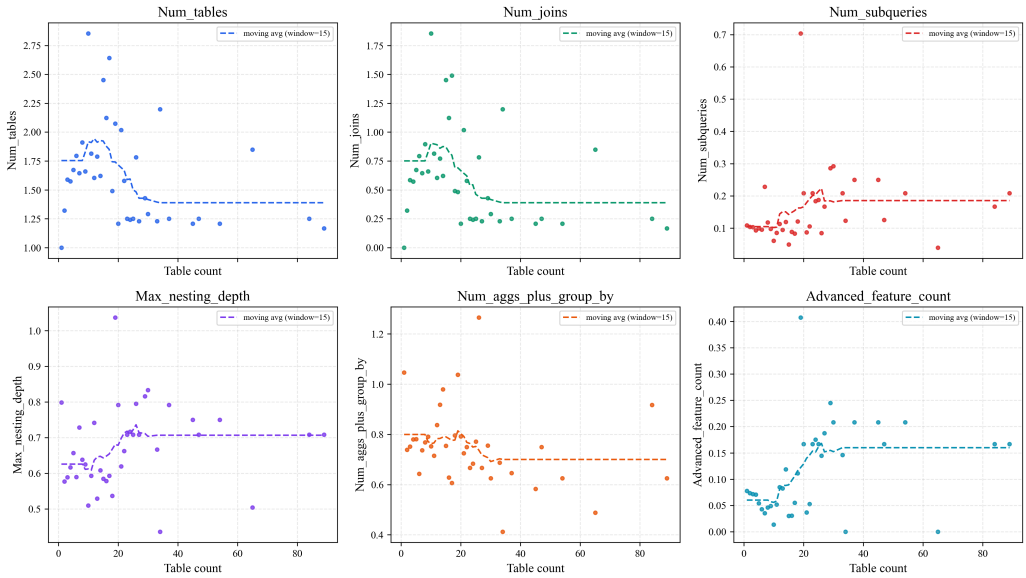


Figure 5: Individual Plots for Six Proxies. Each point represents the average value of a proxy metric for a specific table count. The dashed line denotes the moving average of the proxy values with a window size of 15.

Table 7: Template frequency distribution. Percentages indicate the proportion of templates in each frequency category.

Template Type	High (≥100)	Middle (10–99)	Long Tail (2–9)	Once (1)
Hard	19 (0.3%)	223 (3.3%)	2358 (35.2%)	4099 (61.2%)
Soft	28 (0.6%)	260 (5.7%)	1958 (42.7%)	2341 (51.0%)