Replacing Training with Memory: Listwise Selection for Text-to-SQL

Yeonseok Jeong[1], Soyoung Yoon[1], Seongjun Lee[2], Seung-won Hwang[1]*
IPAI, Seoul National University[1], KAIST[2]
{jys3136, soyoung.yoon, seungwonh}@snu.ac.kr
epaksa@kaist.ac.kr

Abstract

Modern Text-to-SQL systems often follow *generate-execute-select* pipelines, generating multiple candidate queries then selecting the best one. Listwise selection, by jointly comparing multiple candidates, has been widely adopted, but fine-tuning listwise selectors is costly. We thus propose a fine-tuning-free listwise selector. We replace two major fine-tuning objectives with inference-time strategies: (1) learning selection criteria as ordering and (2) mitigating positional bias. First, we build reusable structured memories instead of learning selection behavior as model parameters. Given a question, **MAP-SQL** retrieves memories distilled from training data that encode how natural language maps to schema elements, SQL operations, and expected outputs. These memories serve as explicit decision criteria for evaluating candidates in a listwise manner. Second, to mitigate ordering bias of listwise selectors, we aggregate rankings across multiple input permutations, with inference cost optimized by execution results and pointwise scoring. Our approach improves selection accuracy while maintaining efficiency and compatibility with existing large language models. Across Text-to-SQL benchmarks, it produces more stable selection without fine-tuning and fewer unnecessary comparisons than existing methods. On BIRD-dev, it outperforms the previous state-of-the-art selector-based method R^3-SQL by 2.02 execution accuracy points on average using the same candidate sets, with 2.92× fewer tokens.[1]

1 Introduction

Modern Text-to-SQL systems (Sheng and Shuai, 2025; Dönder et al., 2025; Liu et al., 2026) increasingly follow a *generate–execute–select* pipeline, where multiple SQL candidates are produced and a selector chooses the best one (Li et al., 2025; Yao et al., 2026; Wang et al., 2025). Figure 1 illustrates such a pipeline: four candidate queries are generated and then evaluated by a selector. Depending on how many candidates are evaluated at a time, selectors can be categorized as pointwise (Agrawal and Nguyen, 2025; Tritto et al., 2026), pairwise (Pourreza et al., 2025; Bai et al., 2025), or listwise. Pointwise selectors score candidates independently and miss cross-candidate differences, while pairwise selectors compare candidates more directly but require $O(n^2)$ comparisons. Listwise selection avoids exhaustive quadratic pairwise comparisons by evaluating multiple candidates jointly, allowing it to capture subtle differences that pointwise or

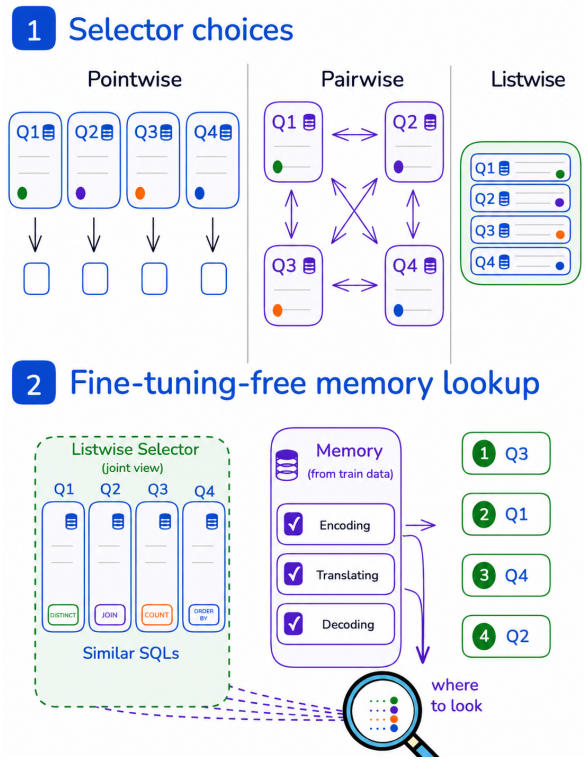

Figure 1: Overview of our proposed SQL selection. Our method uses (1) a listwise selector to jointly compare SQL candidates and (2) fine-tuning-free memory lookup to better capture subtle SQL differences.

*Corresponding Author
[1]https://github.com/ldilab/MAP-SQL

Findings of the Association for Computational Linguistics: EMNLP 2026, pages 53049–53067
October 24-29, 2026 ©2026 Association for Computational Linguistics

pairwise methods may miss (Sun et al., 2023).

However, this benefit comes at a cost, as fine-tuning listwise selectors is costly. Each training example involves multiple SQL queries and their execution results, resulting in long input contexts and high computational overhead. As this makes learned listwise selection difficult to scale, our approach replaces the two main roles of fine-tuning with inference-time strategies.

First, fine-tuning provides selection criteria that guide ranking decisions. We replace this by constructing structured memories, as illustrated in Figure 1(2), distilled from training data. Each memory encodes how natural language maps to schema elements, SQL operations, and expected outputs. Given a test question, we retrieve relevant memories and use them as explicit decision criteria for listwise candidate comparison.

Second, fine-tuning aims to mitigate a well-known positional bias in listwise selection, known as "lost-in-the-middle" bias (Liu et al., 2024; Tang et al., 2024). We instead address this at inference time by aggregating rankings across multiple input permutations, with a cost-aware design that leverages execution signals and selective pointwise scoring.

By combining these two components, we present **M**emory **a**nd **P**ermutation for Listwise **SQL** selection (**MAP-SQL**). Our approach eliminates the need for additional fine-tuning while retaining the benefits of joint candidate comparison. In this paper, *fine-tuning-free* means that MAP-SQL does not update selector parameters. It still uses pre-trained models and labeled question–SQL pairs to construct retrieval memories.

Our approach is simple, efficient, and compatible with off-the-shelf language models, making it practical for real-world Text-to-SQL systems. On BIRD-dev (Li et al., 2024), Spider-test (Yu et al., 2018), and EHRSQL (Lee et al., 2022), our method improves over the previous state-of-the-art selector R^3-SQL by 2.02, 0.53, and 0.68 execution accuracy points on average when both methods use the same candidate pools. Our method also requires 6.54×, 6.85×, and 7.29× fewer selector calls and 2.92×, 2.12×, and 4.16× fewer tokens on the three benchmarks.

2 Related Work

We review existing selection paradigms and their training objectives. Lastly, we present our distinctions of fine-tuning-free approaches.

2.1 Selector Paradigms in Text-to-SQL

Recent Text-to-SQL systems include a selection step after generating multiple SQL candidates to improve performance at test time. Majority voting (Sheng and Shuai, 2025) selects the most frequent execution outcome, but a larger incorrect group can dominate a smaller correct one. Pointwise selectors (Agrawal and Nguyen, 2025; Tritto et al., 2026) score each candidate independently. This lacks comparative insights across candidates, leading to inconsistent scoring and suboptimal performance (Long et al., 2025). Pairwise selectors (Pourreza et al., 2025; Wang et al., 2025) compare all pairs of candidates within the candidate set. However, the number of comparisons grows quadratically with the candidate set, making it computationally inefficient. XiYan-SQL (Liu et al., 2026) adopts a listwise selector that compares all candidates within a window simultaneously. MCS-SQL (Lee et al., 2025a) performs multiple-choice selection without selector fine-tuning. It sorts candidates before presenting them to the selector. In this work, however, the challenges of listwise selection and potential improvements to address them have not been explored.

2.2 Training for Listwise Reranking and Positional Bias

Existing listwise selectors require training to encode selection behavior in model parameters. However, jointly considering a long list of generations requires expensive training on candidate queries and execution results. In addition, long inputs expose the model to positional bias, commonly known as the "lost-in-the-middle" problem (Liu et al., 2024). R^3-SQL (Han et al., 2026) addresses positional bias in training by using a pointwise selector as a tie-breaker when the pairwise selector cannot confidently distinguish between candidates. Delaying bias mitigation to inference time through self-consistency can incur up to $O(n!)$ calls in principle (Tang et al., 2024; Zeng et al., 2026). Although optimization strategies for relevance ranking have been studied (Lee et al., 2025b), no such work has addressed our target problem.

2.3 Our Distinction: Inference-time Strategies

We identify two key roles of fine-tuning: fine-tuning selection criteria and mitigating positional bias, and show that both can be replaced by

inference-time memory retrieval and permutation aggregation. First, for memory retrieval, we draw on the framework of Deng et al. (2022), which decomposes the disconnect between natural language and SQL structure into *encoding* (understanding natural language semantics), *translating* (mapping those semantics to SQL), and *decoding* (generating executable SQL). To address this challenge, PAS-SQL (Kong et al., 2026) extracts question structures and maps each question phrase to database schemas, then uses both to generate SQL. Second, for optimizing permutation aggregation, we leverage execution feedbacks to group candidates with same results to drastically reduce permutation space from $O(n!)$ to $O(g!)$, where g denotes the number of groups such that $g \ll n$.

Unlike prior work, MAP-SQL uses retrieved selection criteria and permutes candidates only within execution-result groups, with confidence-based comparison. This design improves both accuracy and efficiency over the corresponding baselines.

3 Problem Setup

Input. We start with a natural language question x, database schema S, and retrieved memories M. A candidate generator produces a set of SQL candidates $C = \{q_i\}_{i=1}^{n}$.[2] Each SQL query q_i is executed to produce an execution result e_i (result table, empty result, or error), which is also provided as input to the selector with q_i. Each execution result is computed once and cached for reuse across all sliding windows and permutations.

Output. Our selector chooses a single SQL query $\hat{q} \in C$ that maximizes correctness (measured by execution match).

We evaluate execution accuracy (exact match) across benchmarks, with BIRD as the primary benchmark (Li et al., 2024). We additionally measure efficiency, by the number of selector LLM calls per question (or input tokens per question).

4 Proposed Method

In this section, we present MAP-SQL, a fine-tuning-free listwise selection framework that replaces training with inference-time strategies. Specifically, we use memory retrieval (Section 4.1) to provide selection criteria and permutation-based aggregation to mitigate positional bias (Section 4.2).

[2]We experiment with n=8 and n=32 candidates.

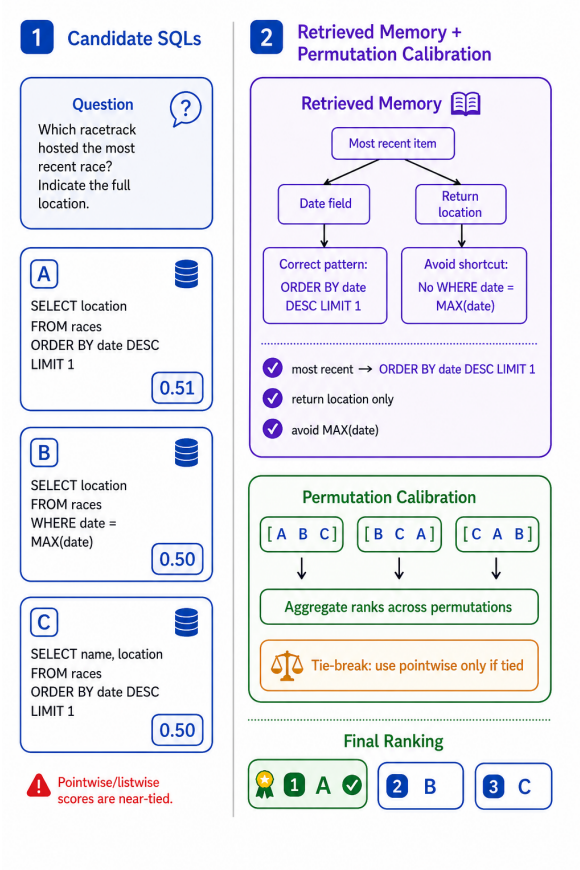

Figure 2: Overview of our fine-tuning-free listwise selection framework. Given a question, multiple SQL candidates are generated. To select, the selector retrieves structured memories that provide explicit criteria for validating candidate queries (e.g., schema grounding, ordering patterns, and constraints). To address positional bias, candidates are evaluated across multiple permutations and their rankings are aggregated, with a pointwise tie-break when needed.

4.1 Fine-tuning-free Selection with Memory

As an alternative to training a selector, we may store the full selection history (Packer et al., 2023; Lee et al., 2026), but using histories is difficult under limited context. Instead, we propose generating a compact memory, avoiding the need for selector fine-tuning.

Step 1: Memory Generation. To select the correct candidate, the selector must verify whether the *semantic gap* (Guo et al., 2022; Kong et al., 2026) between x and q_i is resolved. We therefore store how natural language expressions map to SQL operations and schema elements as reusable memories:

$$M = \{m_j\}_{j=1}^{|\mathcal{D}|}, \quad m_j = f_{\text{mem}}(x_j, S_j, q_j^*), \quad (1)$$

where f_{mem} generates m_j from (x_j, S_j, q_j^*) using the prompt in Figure 5.[3] Borrowing terms from Deng et al. (2022), each memory m_j is organized into three groups:

- **Encoding** captures how natural language phrases are grounded to the database schema and conditions.
- **Translating** captures how the grounded meaning is converted into SQL operations.
- **Decoding** captures how the final SQL output should be formed and validated.

Figure 2 illustrates an example in which the generated candidates are superficially similar, making them difficult for a listwise selector to distinguish. The retrieved memory supplies complementary criteria across the three groups: an encoding criterion for the relevant date field, a translating criterion such as `ORDER BY ... DESC LIMIT 1` for recency, and a decoding criterion specifying that the location should be returned. Together, these criteria guide the selector toward the correct SQL candidate. Definitions of the memory keys in each group are provided in Table 1.

Step 2: Memory Retrieval. Each memory is generated from a question in the training set, so we retrieve memories based on the similarity of the question. Given a test question x, we retrieve the top-k most relevant memories using a dense retriever:

$$\mathcal{J}^* = \underset{|\mathcal{J}|=k}{\arg\,\text{top-}k}\, \text{sim}(\mathbf{h}_x, \mathbf{h}_{x_j}), \tag{2}$$

where $\mathbf{h}_x$ and $\mathbf{h}_{x_j}$ are dense embeddings of x and x_j. Rather than fixing k, we include as many memories as fit within the context limit of the selector to provide diverse perspectives for selection. The retrieved memories $\mathcal{M}(x) = \{m_j\}_{j \in \mathcal{J}^*}$ are prepended to the prompt.

Step 3: Listwise selection. After retrieval, we initialize a candidate order and apply listwise reranking using the retrieved memories. We set the initial order by placing candidates with more frequent execution results first, as majority voting (Sheng and Shuai, 2025) is a reliable prior for correctness. We apply a sliding window of size w and stride s from back to front.[4] For each window W_t, the selector produces a local ranking:

$$\pi_t = f_{\text{list}}(x, S, \mathcal{M}(x), \{(q_i, e_i)\}_{q_i \in W_t}), \tag{3}$$

and updates the order by $R[W_t] \leftarrow \pi_t$. The retrieved memories $\mathcal{M}(x)$ are included once per prompt. This allows the selector to verify all candidates in the window against the same specifications simultaneously.

4.2 Fine-tuning-free Bias Mitigation with Permutation

To reduce positional bias without selector fine-tuning, we aggregate selection results across multiple candidate orderings. Prior permutation-based methods (Zeng et al., 2026; Lee et al., 2025b) mitigate positional bias by evaluating candidates across multiple orderings. Our goal is to mitigate positional bias. However, they require a large number of such evaluations to be effective, e.g., up to $O(n!)$ permutations. Thus, we propose a method that reduces positional bias efficiently with far fewer runs. Table 6 confirms that preserving the between-group ordering leads to better accuracy and less positional bias than prior methods.

We first partition candidates into groups based on execution outcomes, treating each group as a unit during permutation. This reduces the permutation space from individual candidates to groups, e.g., $O(g!)$, typically $g \ll n$.[5] We then consider individual elements only when distinguishing between groups is necessary.

This two-level strategy significantly reduces the number of required permutations while preserving the ability to resolve fine-grained differences between competing candidates.

Group-Based Permutation. We propose a group-based permutation strategy to reduce the number of required selection runs. As in Step 3 of Section 4.1, we group candidates by their execution results and place larger groups first. Prior method (Tang et al., 2024) shuffles all candidates globally because they lack a reliable prior over candidate importance, requiring more runs for stable results. In Text-to-SQL, majority voting serves as a strong prior for correctness. Preserving this signal

[3] We use the same LLM as the selector for f_{mem}, to avoid using a separate model that provides no additional gains.

[4] In our experiments, we use $w = 8$ and $s = 4$. Thus, each selector call contains at most 8 candidates even when $n = 32$.

[5] In our experiments with $n = 8$ candidates, the average g was approximately 2.

Group	Specification Key	Description	Covered SQL Keywords / Constructs
Encoding	`schema_grounding`	Maps question phrases to tables/columns, including confusable ones.	(All table and column references.)
	`join_path`	Specifies required join paths and join types between tables.	`JOIN`, `INNER JOIN`, `LEFT JOIN`, `CROSS JOIN`
	`filter_semantics`	Identifies WHERE conditions and NULL-related predicates.	`WHERE`, `IN`, `LIKE`, `BETWEEN`, `IS NULL`, `IS NOT NULL`
Translating	`aggregation`	Defines grouping columns, aggregate functions, and HAVING conditions.	`GROUP BY`, `HAVING`, `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, `DISTINCT`
	`ordering_and_scope`	Specifies ordering direction and row-count constraints.	`ORDER BY`, `ASC`, `DESC`, `LIMIT`, `OFFSET`
	`conditional_and_null`	Describes conditional branching and null-safe value expressions.	`CASE`, `IIF`, `COALESCE`, `IFNULL`, `NULLIF`
Decoding	`output_form`	Specifies result shape, return columns, and value format.	`SELECT`, `DISTINCT`
	`query_constraints`	Meta-level constraints to avoid spurious joins or wrong aggregations.	(Meta-level; no specific SQL syntax.)
	`extra_keywords`	Covers advanced SQLite constructs beyond the core keys. Also included as `[]` when empty.	All other SQL keywords and constructs not covered by the core keys above.

Table 1: Specification keys and their roles in the Encoding–Translating–Decoding framework for Text-to-SQL selection.

avoids noisy rankings from uninformative orderings. We therefore fix the ordering between groups and shuffle only within each group.

We rerank candidates by their average rank and apply a confidence estimation procedure to determine whether the top-1 candidate is better than the top-2 candidate. For each candidate c_i, we compute the average rank μ_i across K runs:

$$\mu_i = \frac{1}{K} \sum_{k=1}^{K} r_i^{(k)} \tag{4}$$

where $r_i^{(k)}$ is the rank of c_i in the k-th run, and candidates are sorted by μ_i in ascending order. Let a and b denote the top-1 and top-2 candidates after sorting. We estimate confidence using pairwise rank differences $d_{a,b}^{(k)} = r_b^{(k)} - r_a^{(k)}$ and compute the confidence score $P(a > b)$ as:[6]

$$P(a > b) = T_{K-1}\left(\frac{\bar{d}_{a,b}}{s_{a,b}/\sqrt{K}} \right), \tag{5}$$

where $\bar{d}_{a,b}$ and $s_{a,b}$ are the mean and standard deviation of $\{d_{a,b}^{(k)}\}$, and $T_{K-1}(\cdot)$ is the CDF of Student's t-distribution with $K-1$ degrees of freedom. A tie is declared when $P(a > b) < \tau$.[7]

Tie-Breaking with Pointwise Selector. When a tie is declared, we can resolve it using a pointwise selector as an optional secondary step. We score only the tied candidates independently using a pointwise reward model and select the one with the higher score. Since the pointwise selector is invoked only when a tie occurs, the dominant computational path remains listwise operations.

5 Experiments

5.1 Experimental Setup

Benchmarks. We evaluate on three Text-to-SQL benchmarks: BIRD-dev (Li et al., 2024), Spider-test (Yu et al., 2018), and EHRSQL (Lee et al., 2022). BIRD is our primary benchmark, consisting of 1,534 development queries over large-scale, realistic databases. Spider-test is a widely used cross-domain Text-to-SQL benchmark consisting of 2,147 queries, while EHRSQL contains 1,008 questions in the electronic health record domain.

Metrics. We report execution accuracy (Acc.), which measures whether the predicted SQL produces the same result as the gold SQL. We also

[6] We use paired differences rather than individual rank variances, since ranks within the same run are not independent. We use this score as a lightweight ranking-confidence heuristic rather than as a formal hypothesis test.

[7] We set $\tau = 0.95$ in all experiments.

Generator	Agentar-Scale-SQL-Generation-32B						Arctic-Text2SQL-R1-7B					
	n=8			n=32			n=8			n=32		
	Acc. ↑	Calls ↓	Tokens ↓	Acc. ↑	Calls ↓	Tokens ↓	Acc. ↑	Calls ↓	Tokens	Acc. ↑	Calls ↓	Tokens ↓
Baselines												
greedy	69.95	–	–	69.95	–	–	68.19	–	–	68.19	–	–
majority voting (Sheng and Shuai, 2025)	71.38	–	–	70.93	–	–	70.08	–	–	69.95	–	–
Single Selector												
pointwise (Agrawal and Nguyen, 2025)	71.45	8.00	16,793	71.71	32.00	67,299	70.27	8.00	16,798	70.47	32.00	67,168
pairwise (Pourreza et al., 2025)	71.25	6.11	14,251	71.06	129.93	309,564	69.30	9.71	23,498	68.12	184.59	443,713
listwise w/ Memory (ours)	72.23	**0.71**	**3,347**	72.75	**4.72**	**22,190**	71.90	**1.12**	**5,147**	72.10	**5.91**	**27,440**
Multiple Selectors												
R^3-SQL (not trained) (Han et al., 2026)	71.51	14.11	31,044	71.97	161.93	376,863	69.36	17.71	40,295	69.56	216.59	510,881
MAP-SQL (ours)	**72.62**	2.97	15,765	**73.08**	19.04	98,234	**72.16**	4.65	23,567	**72.62**	23.85	122,985

Table 2: Full results across generation and selection strategies on BIRD-dev. Acc. = execution accuracy (%). Calls = # calls. Tokens = # tokens. '–' = not reported or not applicable.

Dataset	Spider-test						EHRSQL					
	n=8			n=32			n=8			n=32		
	Acc. ↑	Calls ↓	Tokens ↓	Acc. ↑	Calls ↓	Tokens ↓	Acc. ↑	Calls ↓	Tokens ↓	Acc. ↑	Calls ↓	Tokens ↓
Baselines												
greedy	86.40	–	–	86.40	–	–	36.52	–	–	36.52	–	–
majority voting (Sheng and Shuai, 2025)	87.07	–	–	87.02	–	–	38.10	–	–	39.59	–	–
Single Selector												
pointwise (Agrawal and Nguyen, 2025)	86.88	8.00	4,848	86.72	32.00	19,399	38.79	8.00	24,611	43.52	32.00	98,579
pairwise (Pourreza et al., 2025)	86.83	4.54	4,191	86.63	74.40	69,682	38.23	22.61	74,708	43.67	408.11	1,341,930
listwise w/ Memory (ours)	87.36	**0.57**	**1,377**	87.40	**3.13**	**7,612**	39.08	**2.11**	**10,340**	44.20	**9.84**	**48,060**
Multiple Selectors												
R^3-SQL (not trained) (Han et al., 2026)	87.21	12.54	9,040	86.82	106.40	89,081	39.25	30.61	99,319	44.03	440.11	1,440,510
MAP-SQL (ours)	**87.49**	2.38	6,013	**87.59**	12.63	32,659	**39.93**	8.77	49,516	**44.71**	39.70	228,070

Table 3: Results on Spider-test and EHRSQL using Arctic-Text2SQL-R1-7B as the generator. Acc. = execution accuracy (%). Calls = # calls. Tokens = # tokens. '–' = not reported or not applicable.

report the average number of LLM calls (Calls) and input tokens per query (Tokens) to assess computational efficiency.

Baselines. We compare against three selection baselines: pointwise, pairwise, and a multiple-selector approach using both (e.g. R^3-SQL and MAP-SQL). For the **pointwise** selector, we use Contextual-RM-32B (Agrawal and Nguyen, 2025)[8], a strong reward model trained from Qwen2.5-Coder-32B-Instruct and released by Contextual-SQL. For the **pairwise** selector, we follow the approach of CHASE-SQL (Pourreza et al., 2025). In this method, candidates with identical execution results are grouped and not compared against each other, and only cross-group pairs are compared, by using the prompt in Figure 8. The final ranking of pairwise is obtained by aggregating these pairwise outcomes. For **R^3-SQL** (Han et al., 2026), we reproduce its selection algorithm by using the same generator and selector as ours, since neither CHASE-SQL nor R^3-SQL has released their original models. This follows the same evaluation protocol as R^3-SQL, which also uses a shared selector to isolate the contribution of the selection algorithm for fair comparison. For all listwise and pairwise components, we use Qwen3-Coder-30B-A3B-Instruct[9] as the selector. For MAP-SQL, Contextual-RM-32B is used only for optional tie-breaking. Appendix A reports results without tie-breaking and with Qwen3-Coder-30B-A3B-Instruct as the tie-breaker.

Generators. We use two SQL generators to assess robustness across different candidate pools: Agentar-Scale-SQL-Generation-32B (Wang et al., 2025) and Arctic-Text2SQL-R1-7B (Yao et al., 2026). For each question, we generate either n=8 or n=32 candidates and evaluate selection performance on top of these fixed candidate pools.

[8] Contextual-RM-32B

[9] Qwen3-Coder-30B-A3B-Instruct

5.2 Implementation Details

Memory Retrieval. We generate memories from training questions following the procedure described in Section 4.1. At inference time, we retrieve the top-k most relevant memories using bge-m3 (Chen et al., 2024) as the dense retriever, based on question similarity. Rather than fixing k, we include as many memories as fit within the selector's context limit. For BIRD and Spider, we construct the memory bank from each benchmark's training split.Because EHRSQL does not provide a training split, we use a memory bank constructed from the combined BIRD and Spider training splits.

Selection Prompts. The listwise selector uses the prompt template shown in Figure 7.

Experimental Infrastructure. All experiments are conducted on a single node equipped with 1 NVIDIA RTX PRO 6000 GPU.

5.3 Experimental Results

Tables 2 and 3 present the results across all benchmarks. Our method consistently outperforms all baselines in accuracy while requiring substantially fewer LLM calls and input tokens.

Accuracy. Our full method achieves the highest execution accuracy across all settings. On BIRD-dev with Agentar-32B and n=32, it reaches 73.08%, outperforming R^3-SQL by 1.11 points and pairwise by 2.02 points. The gains are consistent across both generators and both candidate pool sizes. Notably, our single listwise selector already surpasses R^3-SQL in most settings, despite R^3-SQL combining multiple selectors.

Efficiency. Our listwise selector reduces computational cost significantly compared to pairwise and R^3-SQL. On BIRD-dev with n=32, pairwise requires on average 184.59 calls and 443,713 tokens per query with Arctic-R1-7B, whereas our listwise selector uses only 5.91 calls and 27,440 tokens. Compared to R^3-SQL, our full method uses 9.07× fewer calls and 4.16× fewer tokens in the same setting. On EHRSQL with n=32, the token reduction is even more pronounced: pairwise consumes over 1.3M tokens per query, while our listwise selector requires only 48,060.

Generalization. On Spider-test and EHRSQL, our method also outperforms all baselines. With n=32 on Spider-test, our full method achieves 87.59% accuracy, surpassing R^3-SQL by 0.77 points using 8.43× fewer calls. On EHRSQL with n=32, our method reaches 44.71%, improving over R^3-SQL by 0.68 points while requiring 11.09× fewer calls. These results confirm that the advantage of listwise selection holds beyond the primary benchmark.

Method	Agentar	Arctic
MAP-SQL	**72.62**	**72.16**
w/o Permutation	72.23	71.90
w/o Memory	71.94	71.74
w/o Memory & Permutation	71.64	71.32

Table 4: Ablation study on BIRD-dev with n=8 candidates generated by Agentar-Scale-SQL-Generation-32B (Agentar) and Arctic-Text2SQL-R1-7B (Arctic). We report execution accuracy (%).

6 Analysis

6.1 Ablation Study

To verify the contribution of each component in our method, we conduct an ablation study on BIRD-dev with $n = 8$ candidates. Our full system consists of two components: memory retrieval, which provides structured selection criteria, and permutation-based aggregation, which mitigates positional bias. As shown in Table 4, the full MAP-SQL system achieves the best performance on both generators, with 72.62 on Agentar-Scale-SQL-Generation-32B (Wang et al., 2025) (Agentar) and 72.16 on Arctic-Text2SQL-R1-7B (Arctic) (Yao et al., 2026). Removing either component consistently lowers accuracy, which shows that both components contribute to the final selection performance.

The two components provide complementary benefits. Without Permutation, the average accuracy decreases from 72.39 to 72.07, resulting in a drop of 0.32 points. Without Memory, the average accuracy decreases to 71.84, resulting in a larger drop of 0.55 points.

6.2 Memories with a Strong Code Selector

One concern is that a strong agentic code model may already form memory-like criteria during its own reasoning. Table 5 shows that explicit retrieved memories still help. With a `gpt-5.1-codex-mini` selector, accuracy rises from 71.90% to 72.23% for Agentar-32B and from 70.01% to 70.40% for Arctic-R1-7B.

The gain is modest, but it is consistent across

Generator	Listwise	+ Memory
Agentar-32B	71.90	**72.23** (+0.33)
Arctic-R1-7B	70.01	**70.40** (+0.39)

Table 5: Memory gains on BIRD-dev with a `gpt-5.1-codex-mini` selector and n=8 candidates per query.

both generators. This suggests that the memory complements the selector's internal reasoning rather than duplicating it. It is closer to a retrieved evaluation template than to open-ended memory. The selector does not revisit past trajectories or maintain long interaction histories. Instead, it receives a short structured criterion set once and applies it to the current candidate list.

6.3 Mitigating Positional Bias

We examine whether our method selects the same candidate regardless of the order in which candidates are presented. To this end, we provide the model with two candidate sets that contain the same candidates but in different orderings and measure how often the model selects the same candidate from both. We refer to this agreement rate as *consistency* and separately measure whether the agreed-upon selection is actually correct, which we call *consistency & correct*. We set the candidate set size to $n = 8$. As a reference, random selection yields a consistency of 12.58%. Each compared method applies permutations according to its own strategy before aggregating the final selection.

Table 6 demonstrates that each component of our aggregation method contributes to mitigating positional bias. The baseline listwise method achieves a consistency of 18.98%, only modestly above the random baseline. Adding global permutation improves this to 23.04%, but the gain remains limited. Group-based permutation, by contrast, raises consistency substantially to 29.04%, a gain of 6.0 percentage points over global permutation. The tie-break strategy further pushes consistency to 33.17% with a corresponding consistency & correct of 22.63%. These results show that both components contribute meaningfully and that group-based permutation accounts for the larger share of the improvement.

Figure 3 further reveals how the two permutation strategies behave as the number of permutations increases. Global permutation yields lower consistency throughout and improves only slowly with more permutations. Group-based permutation achieves higher consistency from the start and saturates more quickly. This faster saturation is because group-based permutation avoids noisy orderings that can mislead the model. The tie-break strategy makes this even more efficient. At four permutations, applying the tie-break reaches 33.17%, which is comparable to running group-based permutation eight times without it. The tie-break thus achieves the benefit of doubling the number of permutations without requiring additional inference.

Method	Consistency (%) ↑	Consistency & Correct (%) ↑
Random	12.58	8.67
Listwise	18.98	11.46
+ Global Permutation	23.04	14.18
+ Group-based Permutation (ours)	29.04	19.18
+ Tie-break (ours)	**33.17**	**22.63**

Table 6: Positional-bias analysis with shuffled initial candidate orders ($n = 8$). Group-based permutation and tie-break strategies improve selection consistency while mitigating positional bias.

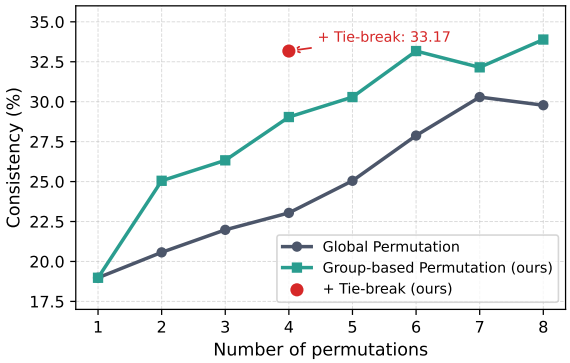

Figure 3: Consistency under different numbers of permutations. Group-wise permutation improves selection consistency over global permutation, and the tie-break strategy further improves consistency at four permutations.

6.4 Comparison with a Fine-Tuned Selector

To assess whether fine-tuning-free selection can match fine-tuned selectors, we compare MAP-SQL against the reported result of R^3-SQL (Han et al., 2026) using its fine-tuned selector. Since the fine-tuned selector of R^3-SQL is not publicly released, we rely on its reported result: 71.84% execution accuracy on BIRD-dev with OmniSQL-7B (Li et al., 2025) as the generator and n=32 candidates.

As shown in Table 7, MAP-SQL achieves 71.90% with OmniSQL-7B, slightly surpassing this figure without selector fine-tuning. This result suggests that our inference-time strategies are

Method	Acc.
majority voting	68.45
R^3-SQL (fine-tuned)	71.84
MAP-SQL	**71.90**

Table 7: Comparison with a fine-tuned selector on BIRD-dev using n=32 candidates generated by OmniSQL-7B. We report execution accuracy (%).

sufficient to reach a comparable level of selection quality to a trained selector.

7 Conclusion

We present MAP-SQL, and (1) show that listwise selection outperforms pairwise and pointwise baselines for SQL candidate selection. The selector contrasts SQL structures, execution outcomes, and schema usage across multiple candidates in a single call. We also (2) show that retrieved memories further improve listwise selection. With a strong code-focused selector, the gain remains consistent across multiple generators. (3) Listwise selection reduces call complexity from $O(N^2)$ to $O(N)$–$O(N \log N)$, using up to 27.92× fewer input tokens per query than pairwise selection in our experiments.

Limitations

We believe there is still room for improvement on applying advanced coordination techniques for listwise selection, other than just using sliding windows, such as TourRank (Chen et al., 2025), or Set-based ranking (Zhuang et al., 2024). We leave the exploration of more advanced coordination strategies as future work.

The overall execution accuracy (70–72%) lags behind SOTA systems using proprietary models (75–76%), partly due to generator quality rather than selection strategy. Our results characterize selector performance on the same candidate pools rather than end-to-end SOTA performance on BIRD.

For enterprise-scale databases with hundreds of tables, we assume upstream schema linking to prune the schema before generation and selection (Talaei et al., 2024), while schema-scale context management itself is beyond the scope of our selector.

We focus on Text-to-SQL because it is a challenging listwise selection setting involving long schemas, execution results, and multiple candidate SQL queries. Exploring whether our memory-guided selection and group-based permutation generalize beyond Text-to-SQL is left for future work.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00414981), the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2025-2020-0-01789) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) [NO.RS-2021-II211343, Artificial Intelligence Graduate School Program (Seoul National University)].

References

Sheshansh Agrawal and Thien Nguyen. 2025. Open-sourcing the best local text-to-sql system.

Jiayuan Bai, Xuan-guang Pan, Chongyang Tao, and Shuai Ma. 2025. Judgesql: Reasoning over sql candidates with weighted consensus tournament. *arXiv preprint arXiv:2510.15560*.

Shuaichen Chang, Jun Wang, Mingwen Dong, Lin Pan, Henghui Zhu, Alexander Hanbo Li, Wuwei Lan, Sheng Zhang, Jiarong Jiang, Joseph Lilien, Steve Ash, William Yang Wang, Zhiguo Wang, Vittorio Castelli, Patrick Ng, and Bing Xiang. 2023. Dr.Spider: A diagnostic evaluation benchmark towards text-to-SQL robustness. In *The Eleventh International Conference on Learning Representations*.

Jianlyu Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2318–2335, Bangkok, Thailand. Association for Computational Linguistics.

Yiqun Chen, Qi Liu, Yi Zhang, Weiwei Sun, Xinyu Ma, Wei Yang, Daiting Shi, Jiaxin Mao, and Dawei Yin. 2025. Tourrank: Utilizing large language models for documents ranking with a tournament-inspired strategy. In *Proceedings of the ACM Web Conference 2025*.

Naihao Deng, Yulong Chen, and Yue Zhang. 2022. Recent advances in text-to-sql: A survey of what we have and what we expect. In *Proceedings of the 29th International conference on computational linguistics*, pages 2166–2187.

Yusuf Denizay Dönder, Derek Hommel, Andrea W Wen-Yi, David Mimno, and Unso Eun Seo Jo. 2025. Cheaper, better, faster, stronger: Robust text-to-sql without chain-of-thought or fine-tuning. *Preprint*, arXiv:2505.14174.

Mengfei Guo, Yufeng Chen, Jinan Xu, and Yujie Zhang. 2022. Kg-sql: Hybrid knowledge-guided semantic understanding for text-to-sql. In *2022 4th International Conference on Natural Language Processing (ICNLP)*, pages 409–413. IEEE.

Hojae Han, Yeonseok Jeong, Seung-won Hwang, Zhewei Yao, and Yuxiong He. 2026. R3-sql: Ranking reward and resampling for text-to-sql. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 43257–43275.

Yonghui Kong, Hongbing Hu, Dan Zhang, Zhaohui Xu, and Wei Wang. 2026. Bridging the gap: transforming natural language questions into sql queries via abstract query pattern and contextual schema markup. In *ICASSP 2026-2026 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 16902–16906. IEEE.

Dohyeon Lee, Yeonseok Jeong, and Seung-Won Hwang. 2026. Beyond markovian forgetfulness: Episodic memory for reasoning-intensive retrieval. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 37266–37280.

Dongjun Lee, Choongwon Park, Jaehyuk Kim, and Heesoo Park. 2025a. MCS-SQL: Leveraging multiple prompts and multiple-choice selection for text-to-SQL generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 337–353, Abu Dhabi, UAE. Association for Computational Linguistics.

Gyubok Lee, Hyeonji Hwang, Seongsu Bae, Yeonsu Kwon, Woncheol Shin, Seongjun Yang, Minjoon Seo, Jong-Yeup Kim, and Edward Choi. 2022. Ehrsql: A practical text-to-sql benchmark for electronic health records. *Advances in Neural Information Processing Systems*, 35:15589–15601.

Youngwon Lee, Seung-won Hwang, Daniel F Campos, Filip Gralinski, Zhewei Yao, and Yuxiong He. 2025b. Inference scaling for bridging retrieval and augmented generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 7324–7339.

Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, and 1 others. 2025. Omnisql: Synthesizing high-quality text-to-sql data at scale. *arXiv preprint arXiv:2503.02240*.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024. Can LLM already serve as a database interface? A BIg bench for large-scale database grounded text-to-SQLs. *Advances in Neural Information Processing Systems*, 36.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics*, 12:157–173.

Yifu Liu, Yin Zhu, Yingqi Gao, Zhiling Luo, Xiaoxia Li, Xiaorong Shi, Yuntao Hong, Jinyang Gao, Yu Li, Bolin Ding, and Jingren Zhou. 2026. Xiyan-sql: A novel multi-generator framework for text-to-sql. *IEEE Transactions on Knowledge and Data Engineering*, 38(4):2474–2487.

Kehan Long, Shasha Li, Chen Xu, Jintao Tang, and Ting Wang. 2025. Precise zero-shot pointwise ranking with llms through post-aggregated global context information. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2384–2394.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G Patil, Ion Stoica, and Joseph E Gonzalez. 2023. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*.

Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2025. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *The Thirteenth International Conference on Learning Representations*.

Lei Sheng and Xu Shuai Shuai. 2025. Csc-sql: Corrective self-consistency in text-to-sql via reinforcement learning. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 1473–1496.

Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023. Is chatgpt good at search? investigating large language models as re-ranking agents. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 14918–14937.

Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual harnessing for efficient SQL synthesis. *arXiv preprint arXiv:2405.16755*.

Raphael Tang, Crystina Zhang, Xueguang Ma, Jimmy Lin, and Ferhan Türe. 2024. Found in the middle: Permutation self-consistency improves listwise ranking in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 2327–2340.

Mattia Tritto, Giuseppe Farano, Dario Di Palma, Gaetano Rossiello, Dharmashankar Subramanian, Fedelucio Narducci, and Tommaso Di Noia. 2026.

Gradesql: Outcome reward models for intelligent text-to-sql generation from llms. *Journal of Intelligent Information Systems*, pages 1–24.

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, Mengdie Chu, Yingqi Gao, Xiang Qi, Peng Zhang, and Ying Yan. 2025. Agentar-scale-sql: Advancing text-to-sql through orchestrated test-time scaling. *arXiv preprint arXiv:2509.24403*.

Zhewei Yao, Guoheng Sun, Łukasz Borchmann, Zheyu Shen, Minghang Deng, Bohan Zhai, Hao Zhang, Ang Li, and Yuxiong He. 2026. Arctic-text2sql-r1: Simple rewards, strong reasoning in text-to-sql. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 26966–26995.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 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.

Yifan Zeng, Ojas Tendolkar, Raymond Baartmans, Qingyun Wu, Lizhong Chen, and Huazheng Wang. 2026. LLM-rankfusion: Mitigating intrinsic inconsistency in LLM-based ranking. *Transactions on Machine Learning Research*.

Shengyao Zhuang, Honglei Zhuang, Bevan Koopman, and Guido Zuccon. 2024. A setwise approach for effective and highly efficient zero-shot ranking with large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 38–47.

Appendices

A Additional Experiments

We provide additional experiments to validate MAP-SQL from several perspectives. We first examine whether MAP-SQL depends on a separate pointwise reward model for tie-breaking (Section A.1). We then evaluate whether the gains over R^3-SQL are statistically reliable and consistent across different candidate pools (Sections A.2 and A.3). We also compare MAP-SQL with the sorted-list selection strategy of MCS-SQL (Section A.4). Finally, we evaluate the robustness of question-based memory retrieval when the wording of a question changes (Section A.5).

A.1 Effect of Pointwise Tie-Breaking

We evaluate whether MAP-SQL depends on a separate pointwise reward model for tie-breaking. We use BIRD-dev (Li et al., 2024) with n=8 candidates generated by Agentar-Scale-SQL-Generation-32B (Wang et al., 2025). We compare the results with R^3-SQL (Han et al., 2026) under the same setting. We evaluate three configurations of MAP-SQL. The first configuration does not use tie-breaking. The second configuration reuses Qwen3-Coder-30B-A3B-Instruct[10] with a pointwise prompt. The third configuration uses Contextual-RM-32B (Agrawal and Nguyen, 2025) as the pointwise tie-breaker.

Table 8 shows that MAP-SQL remains competitive without a separate pointwise model. Without tie-breaking, MAP-SQL achieves 72.23% execution accuracy. This result is higher than the 71.51% of R^3-SQL in the same setting. Using Qwen3-Coder for tie-breaking improves the accuracy to 72.42%. This configuration reuses the same selector and does not require an additional reward model. Using Contextual-RM-32B further improves the accuracy to 72.62%. These results show that the pointwise reward model is an optional accuracy enhancement. It is not required by the core method.

Method	Acc.
R^3-SQL	71.51
MAP-SQL w/o tie-breaking	72.23
MAP-SQL + Qwen3-Coder tie-break	72.42
MAP-SQL + Contextual-RM tie-break	**72.62**

Table 8: Tie-breaking ablation on BIRD-dev with n=8 candidates generated by Agentar-Scale-SQL-Generation-32B. Qwen3-Coder denotes Qwen3-Coder-30B-A3B-Instruct. Contextual-RM denotes Contextual-RM-32B. We report execution accuracy (%).

A.2 Statistical Reliability of MAP-SQL improvements

We evaluate whether the accuracy gains over R^3-SQL are reliable at the question level. We use the n=32 settings from our main experiments. The evaluation covers BIRD-dev, Spider-test (Yu et al., 2018), and EHRSQL (Lee et al., 2022). The candidate pools are generated by Agentar and Arctic-Text2SQL-R1-7B (Yao et al., 2026). Both methods are evaluated on the same candidate pool. We use a paired bootstrap to estimate a 95% confidence interval for the difference in execution accuracy. We also use the exact McNemar test to compare questions for which the two methods have different correctness outcomes.

Table 9 shows significant improvements in both BIRD settings and in Spider. Their confidence intervals exclude zero. Their exact McNemar test p-values are also below 0.05. The EHRSQL confidence interval includes zero. Its p-value is 0.125. We therefore treat the EHRSQL improvement from this candidate pool as inconclusive. The other three settings provide statistical support for the improvement of MAP-SQL over R^3-SQL.

A.3 Robustness Across Randomly Generated Candidate Pools

The preceding analysis evaluates both methods on a fixed candidate pool. We further evaluate whether the comparison with R^3-SQL depends on a particular candidate pool. We generate three candidate pools with random seeds 42, 43, and 44. Each pool contains n=32 candidates. Both selectors are evaluated on every pool. We report the mean execution accuracy and the sample standard deviation across the three pools.

Table 10 shows that MAP-SQL has higher mean execution accuracy in every dataset and generator setting. This pattern also holds for EHRSQL despite the inconclusive result from the single candidate pool in Table 9. These results show that the improvement is consistent across the three generated candidate pools.

[10] Qwen3-Coder-30B-A3B-Instruct

Dataset	Generator	R^3-SQL Acc.	MAP-SQL Acc.	95% CI for Δ Acc.	Exact McNemar p
BIRD-dev	Agentar-Scale-SQL-Generation-32B	71.97	**73.08**	[0.20, 2.02]	0.021
BIRD-dev	Arctic-Text2SQL-R1-7B	69.56	**72.62**	[1.76, 4.37]	< 0.001
Spider-test	Arctic-Text2SQL-R1-7B	86.82	**87.59**	[0.14, 1.39]	0.023
EHRSQL	Arctic-Text2SQL-R1-7B	44.03	**44.71**	[-0.06, 1.43]	0.125

Table 9: Statistical significance of MAP-SQL over R^3-SQL with n=32 candidates. We report paired bootstrap 95% confidence intervals for the execution accuracy difference and exact McNemar test p-values.

Dataset	Generator	R^3-SQL	MAP-SQL
BIRD-dev	Agentar-32B	71.82 $\pm$ 0.16	**73.01 $\pm$ 0.36**
BIRD-dev	Arctic-R1-7B	69.73 $\pm$ 0.20	**72.34 $\pm$ 0.25**
Spider-test	Arctic-R1-7B	86.92 $\pm$ 0.10	**87.89 $\pm$ 0.26**
EHRSQL	Arctic-R1-7B	43.86 $\pm$ 0.45	**45.28 $\pm$ 0.52**

Table 10: Robustness across candidate pools generated with seeds 42, 43, and 44 for n=32. We report mean execution accuracy and sample standard deviation.

Method	BIRD-dev Agentar	BIRD-dev Arctic	Spider-test Arctic	EHRSQL Arctic
MCS-SQL	71.51	70.47	86.97	39.25
MAP-SQL	**72.62**	**72.16**	**87.49**	**39.93**

Table 11: Comparison with the MCS-SQL (Lee et al., 2025a) sorted-list strategy using n=8 candidates. We report execution accuracy (%).

A.4 Comparison with MCS-SQL

We compare our group-based permutation strategy with the sorted-list strategy of MCS-SQL (Lee et al., 2025a). The official implementation of MCS-SQL was not publicly available when we conducted this experiment. We therefore reproduce its multiple-choice selection strategy. We use Qwen3-Coder-30B-A3B-Instruct as the selector for both methods. Both methods also use the same n=8 candidate pools. This setup isolates the difference between the two selection strategies.

Table 11 shows that MAP-SQL achieves higher execution accuracy in all four settings. The improvement appears with both generators on BIRD-dev. The same pattern also appears on Spider-test and EHRSQL. These results show that group-based permutation performs better than the reproduced sorted-list strategy under the same selector and candidate pools.

A.5 Robustness on Dr.Spider

We evaluate whether question-based memory retrieval remains effective when the wording of a question changes. Our retriever uses bge-m3 (Chen et al., 2024) to retrieve memories based on question similarity. We use the nine natural-language question perturbations from Dr.Spider (Chang et al., 2023). These perturbations change how the original questions are expressed while preserving their intended meaning. All methods use n=8 candidates generated by Arctic-Text2SQL-R1-7B. We report accuracy before perturbation as Pre. We report accuracy after perturbation as Post. Drop is calculated as Pre minus Post.

Method	Pre ↑	Post ↑	Drop ↓
greedy	89.54	76.22	13.32
majority voting	89.23	76.43	12.80
pointwise	89.93	76.66	13.27
pairwise	89.56	76.13	13.43
listwise w/o Memory	90.13	77.06	13.13
listwise w/ Memory (ours)	**90.52**	**77.80**	**12.72**

Table 12: Average execution accuracy (%) across the nine natural-language question perturbations of Dr.Spider with n=8 candidates generated by Arctic-Text2SQL-R1-7B. Drop is calculated as Pre minus Post.

Table 12 shows that listwise selection with memory achieves the highest Post accuracy of 77.80%. It also has the smallest Drop at 12.72 points. Listwise selection without memory reaches 77.06% after perturbation. Its Drop is 13.13 points. These results show that the retrieved memories remain useful when questions are expressed with different wording. They also show that question-based retrieval remains robust when the intended meaning is preserved but the wording changes.

Taken together, these experiments support the reliability and robustness of the reported improvements. The gains over R^3-SQL receive statistical support in three of the four fixed-pool settings and remain consistent across the three generated candidate pools. The results also show that MAP-SQL remains competitive without a separate pointwise reward model and outperforms the reproduced MCS-SQL strategy in all four evaluated settings.

Finally, the Dr.Spider results show that retrieved memories remain useful when questions are expressed with different wording.

B Additional Examples and Prompts

B.1 Memory Examples and Generation Prompt

Our memory design follows the Encoding, Translating, and Decoding decomposition of Deng et al. (2022). Figure 4 shows an example memory generated from a training question and its gold SQL. Figures 5 and 6 show the generation prompt and an example of how the retrieved memory guides listwise selection.

B.2 Selection Prompts

We provide the full prompts used for listwise and pairwise selection. Figure 7 shows the prompt used by our listwise selector. Figure 8 shows the pairwise prompt used for the CHASE-SQL-style baseline (Pourreza et al., 2025).

C Usage of AI Assistants

We utilized ChatGPT to improve the clarity and grammatical accuracy of the writing. It provided suggestions for rephrasing sentences and correcting grammatical errors to make the text flow more naturally.

Example of Retrieved Memory Used for Listwise Selection

Question:

```
How many athletes from region 151 have won a medal?
```

Database Schema (relevant subset):

```
CREATE TABLE competitor_event (
event_id INTEGER,
competitor_id INTEGER,
medal_id INTEGER,
FOREIGN KEY (competitor_id) REFERENCES games_competitor(id)
);

CREATE TABLE games_competitor (
id INTEGER,
person_id INTEGER,
PRIMARY KEY (id),
FOREIGN KEY (person_id) REFERENCES person(id)
);

CREATE TABLE person_region (
person_id INTEGER,
region_id INTEGER
);
```

Specification (JSON):

```
{
  "schema_grounding": {
    "mappings": [
      "athletes -> person_region.person_id",
      "region 151 -> person_region.region_id = 151",
      "won a medal -> competitor_event.medal_id != 4"
    ]
  },
  "join_path": {
    "paths": [
      "competitor_event --(competitor_id)--> games_competitor",
      "games_competitor --(person_id)--> person_region"
    ],
    "join_types": ["INNER", "INNER"]
  },
  "filter_semantics": {
    "conditions": ["person_region.region_id = 151", "competitor_event.medal_id != 4"]
  },
  "aggregation": {
    "group_by_columns": [],
    "aggregate_functions": "COUNT(person_region.person_id)"],
    "distinct_within_aggregate": "false"
  },
  "output_form": {
    "result_shape": "single_value",
    "return_columns": ["COUNT(person_region.person_id)"],
    "distinct": "false",
    "format": "number"
  },
  "query_constraints": {
    "avoid_unnecessary_distinct": "true", "avoid_extra_join": "true",
    "avoid_wrong_aggregation_column": "true", "avoid_extra_filter": "true",
    "prefer_schema_defined_columns": "true"
  },
  "extra_keywords": []
}
```

Figure 4: Example of an LLM-generated memory derived from a training question–SQL pair.

Prompt Template for Memory Generation

```
You are an expert in Text-to-SQL reasoning.
Analyze the given natural language question, database schema, and correct SQL query.
Generate a structured specification memory that helps an LLM select the correct SQL query
among multiple candidates.

--------------------------------------------------
INPUT

[Question]
{question}

[Database Schema]
{schema}

[Correct SQL Query]
{gold_sql}

[Detected Extra SQLite Constructs in Gold SQL]
The following keywords / functions were found in the correct SQL.
You MUST populate "extra_keywords" with one entry per detected keyword.
Detected: {detected_extra_keywords}

--------------------------------------------------
OUTPUT FORMAT

Return a single flat JSON object.

IMPORTANT RULES:
- Include a key ONLY when its corresponding SQL construct actually appears in the correct SQL.
  Omit keys entirely when they are not relevant.
- "extra_keywords" is ALWAYS included. Use [] when no extra keywords are detected.

{
  "schema_grounding": {
    "mappings": ["question phrase -> table.column", ...]
  },
  "join_path": {
    "paths": ["tableA --(fk)--> tableB"],
    "join_types": ["INNER | LEFT | CROSS | ..."]
  },
  "filter_semantics": {
    "conditions": ["column op value, e.g. release_year = 1945, price BETWEEN 10 AND 50"],
  },
  "aggregation": {
    "group_by_columns": ["col1"],
    "aggregate_functions": ["COUNT(*)", "SUM(amount)"],
    "having_condition": "e.g. COUNT(*) > 3",
    "distinct_within_aggregate": "true | false"
  },
  "ordering_and_scope": {
    "order_by": ["col ASC | DESC"], "limit": "N", "offset": "M"
  },
  "conditional_and_null": {
    "construct": "CASE | IIF | COALESCE | IFNULL | NULLIF", ...
  },
  "output_form": {
    "result_shape": "single_value | single_row | multiple_rows",
    "format": "number | year | text | id | boolean | percentage | custom", ...
  },
  "query_constraints": {
    "avoid_unnecessary_distinct": "true | false", ...
  },
  "extra_keywords": [{"keyword": "X", "usage": "one sentence on how it is used in this query"}, ...
  ]
}
```

Figure 5: Prompt template used for memory generation.

Qualitative Example — memory resolves three near-tied candidates for one question

Question (CODEBASE_COMMUNITY): `Among posts by Harvey Motulsky and Noah Snyder, which one has higher popularity?`

Retrieved memory (most relevant of the k retrieved; green = decisive):

```
{
  "schema_grounding": { "mappings": ["reviewer -> ProductReview.ReviewerName"] },
  "output_form": { "result_shape": "single_row",
      "return_columns": [ "ProductReview.ReviewerName" ], "format": "text" },
  "query_constraints": { "avoid_extra_filter": "true", "prefer_schema_defined_columns": "true" }
}
```

Candidates (all execute; they disagree on *what to return*)

(A) `SELECT u.DisplayName ... GROUP BY u.DisplayName ORDER BY SUM(p.ViewCount) DESC;` ⇒ `Harvey Motulsky`

(B) `SELECT p.Title ... ORDER BY p.ViewCount DESC LIMIT 1;` ⇒ `"Power of Holm's ..."`

(C) `SELECT u.DisplayName, SUM(p.ViewCount) ... GROUP BY u.DisplayName ...;` ⇒ `Harvey Motulsky | 23065`

Listwise output: `(A)` > `(C)` > `(B)` > ··· (A selected)

`memory.output_form`: return only the queried entity's name → (A) `DisplayName` ✓ a person (B) `Title` × a post (C) `+SUM` × extra column

first-stage rank **18** ⟹ final rank **1**

Figure 6: **Memory resolves three near-tied candidates.** For a single question, three candidates all execute and return a plausible value, so execution alone cannot rank them: (A) returns a display name (`Harvey Motulsky`), (B) returns a post title (`"Power of Holm's ..."`), and (C) returns a name together with its view count (`Harvey Motulsky, 23065`). The question asks which of the two *people* is more popular, so the answer should be a single person's name. The retrieved memory's `output_form` field prescribes returning exactly the queried entity's name column—mirroring its twin *"Which reviewer... → ReviewerName"*—which selects (A) over the post-valued (B) and the over-specified (C); conditioning the listwise selector on the memory promotes (A) from rank $18/32$ to the top.

Full Prompt Template for Listwise SQL Selection

Prompt template:

```
Given the db info and question, there are {num_candidates} candidate queries.
Compare all candidates, analyze the differences in their queries and execution results.
{score_instruction}
Based on the original question and the provided database info, rank them based on their correctness.

**************************
Database Schema
{schema}
**************************
Question: {question}
Evidence: {hint}
**************************
Candidate A
```sql
{candidate_A_sql}
```
Execution result:
{candidate_A_execution_result}
{optional_consensus_or_judge_score_A}
****
Candidate B
```sql
{candidate_B_sql}
```
Execution result:
{candidate_B_execution_result}
{optional_consensus_or_judge_score_B}
****
...
Candidate {last_label}
```sql
{candidate_last_sql}
```
Execution result:
{candidate_last_execution_result}
{optional_consensus_or_judge_score_last}
****
****

Rank the passages in descending order of correctness to the query.
The most correct query identifier should be placed first.
Output ONLY letter identifiers in the format of, e.g., [A] > [B] > ... .
Do NOT say any word or explain.

Ranking:
```

Figure 7: Full prompt template used by the listwise SQL reranker. Candidate SQLs, execution results, optional scores, and retrieved memory sections are filled at runtime before the model is asked to output a ranking such as `[A] > [B] > ....`

Full Prompt Template for Pairwise SQL Selection

Prompt template:

```
Given the DB info and question, there are two candidate queries.
There is correct one and incorrect one, compare the two candidate answers,
analyze the differences of the query and the result.
{score_instruction}
Based on the original question and the provided database info, choose the correct one.

**************************
Database Schema
{schema}
**************************
Question: {question}
Evidence: {hint}
**************************
Candidate A
```sql
{candidate_a}
```
Execution result:
{result_a}{score_text_a}{judge_text_a}
****
Candidate B
```sql
{candidate_b}
```
Execution result:
{result_b}{score_text_b}{judge_text_b}
****

Output ONLY "A" or "B" to indicate the correct candidates.
Do NOT say anything or explain.

Correct Answer:
```

Figure 8: Full prompt template used by the pairwise SQL reranker. For each non-identical pair of candidate execution results, the model compares two SQL queries and returns exactly `A` or `B`.