Both Ends Count! Just How Good are LLM Agents at Text-to-"Big SQL"?

Germán T. Eizaguirre
Universitat Rovira i Virgili
Tarragona, Spain
germantelmo.eizaguirre@urv.cat

Lars Tissen
RWTH Aachen University
Aachen, Germany
lars.tissen@rwth-aachen.de

Marc Sánchez-Artigas
Universitat Rovira i Virgili
Tarragona, Spain
marc.sanchez@urv.cat

Abstract

Text-to-SQL and Big Data are both extensively benchmarked fields, yet there is limited research that evaluates them jointly. In the real world, Text-to-SQL systems are often embedded with Big Data workflows, such as large-scale data processing or interactive data analytics. We refer to this as **"Text-to-Big SQL"**. However, existing text-to-SQL benchmarks remain narrowly scoped and overlook the cost and performance implications that arise at scale. For instance, translation errors that are minor on small datasets lead to substantial cost and latency overheads as data scales, a relevant issue completely ignored by text-to-SQL metrics.

In this paper, we overcome this overlooked challenge by introducing novel and representative metrics for evaluating Text-to-Big SQL. Our study focuses on production-level LLM agents, a database-agnostic system adaptable to diverse user needs. Via an extensive evaluation of frontier models, we show that text-to-SQL metrics are insufficient for Big Data. In contrast, our proposed text-to-Big SQL metrics accurately reflect execution efficiency, cost, and the impact of data scale. These metrics enable new LLM-specific insights that are not revealed by traditional accuracy-based evaluations. For example, GPT-4o compensates for roughly 7% lower accuracy than the top-performing later-generation models with up to a 12.16× speedup, while GPT-5.2 is more than twice as cost-effective as Gemini 3 Pro at large input scales.

CCS Concepts: • **Computing methodologies** → **Natural language processing**; • **Information systems** → **Data analytics**.

Keywords: AI agents, Big Data, LLM, Text-to-SQL

ACM Reference Format:
Germán T. Eizaguirre, Lars Tissen, and Marc Sánchez-Artigas. 2026. Both Ends Count! Just How Good are LLM Agents at Text-to-"Big SQL"?. In *Sixth European Workshop on Machine Learning and Systems (EuroMLSys '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3805621.3807640



This work is licensed under a Creative Commons Attribution 4.0 International License.
EuroMLSys '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2605-7/2026/04
https://doi.org/10.1145/3805621.3807640

1 Introduction

Text-to-SQL is a longstanding problem in NLP that seeks to bridge natural language (NL) interfaces and structured query generation. Recent advances in production Large Language Models (LLMs) have substantially improved cross-domain performance, placing them as effective text-to-SQL engines [5, 11, 37] that can generalize across varying data schemas and terminology, improving state-of-the-art results [28, 55].

In practice, this generalization ability is further enhanced via the integration of AI agents, which act as task-specific LLM scaffolds to iteratively inspect database schemas, refine SQL generation, validate SQL syntax, enabling text-to-SQL execution to adapt to the unique traits of user-specific data sources [40]. Precisely, the existing ecosystem of open-source stacks for agent implementation [12, 24, 36], combined with the current state of production LLMs [5, 11, 37], has made text-to-SQL systems more accessible than ever.

However, when moving beyond traditional databases to Big Data systems, text-to-SQL faces additional complexities. For instance, systems such as Amazon Athena [3] enable interactive analytics on massive datasets without requiring ETL, supporting various formats (CSV, JSON, ORC, Avro) and serverless, on-demand execution. In Big Data systems such as Athena, focusing only on the text-to-SQL end is not enough. The Big Data end itself introduces critical constraints that directly affect overall performance and cost. First, incorrect SQL has amplified consequences: failed queries can consume substantial compute resources, scan massive volumes of data, and increase execution costs, making accuracy essential not only for correctness but also for efficiency. For instance, [23] report that running TPC-H benchmark at a moderate scale factor of 100 on Amazon Athena using Parquet data takes 132.3 seconds, illustrating how failed queries can sharply increase execution time and costs in Big Data environments.

Second, inefficiencies can arise not only from failed queries but also from the SQL generation process itself. In agentic text-to-SQL, the LLM instructs the agent to use structured tools to inspect schemas and extract data-specific traits [13, 49, 53]. While this enables context-aware query adaptation, the reasoning overhead and tool orchestration can increase latency, making efficient SQL generation critical in Big Data

systems. Simply put, if SQL generation becomes slower than physical query execution, interactive analysis may become impractical, undermining the performance gains of decades of optimizations in Query-as-a-Service (QaaS) engines like Athena and BigQuery [18].

Overall, these reasons lead to the following observation:

Considering both SQL generation and execution is crucial, making text-to-SQL in Big Data environments distinct, for we refer to it as "**Text-to-Big SQL.**"[1]

1.1 Why Current Text-to-SQL Approaches Fall Short for Big Data?

Traditional text-to-SQL methods have largely been evaluated on moderate-scale relational databases, focusing on query translation or isolated accuracy metrics [14, 27, 28, 52]. While these benchmarks provide insights into LLM capabilities, they often overlook the complexities of interactive execution, streaming data, and cost considerations [10].

One example of this is that many text-to-SQL benchmarks use binary correctness metrics, tagging each generated query with a simple 0/1 label, thereby diluting degrees of partial correctness [31, 52]. This is obvious, for example, when a generated query incorrectly projects an unnecessary column. In traditional text-to-SQL this counts as a wrong translation, but in text-to-Big SQL it should be partially acceptable, since re-running a query for a single extra column could be extremely costly. Consequently, new evaluation metrics are needed to jointly account for partial correctness and cost, reflecting practical text-to-SQL performance for Big Data.

While LLMs are effective text-to-SQL processors [28, 30], their performance depends on how agents scaffold tool use [33, 40, 50]. In a ReAct-style framework, the LLM controller guides reasoning, selects tools, and interprets feedback, while the executor runs the tools. Fast tools, such as fetching a table schema from a data catalog can be bottlenecked by extensive LLM reasoning for query validation, or vice versa. Efficient interaction between LLM, agent, and tools is thus critical for responsive interactive analytics. However, there are no evaluations that focus on this interplay and its effect on Big Data performance. In this paper, we start addressing this gap by jointly measuring agent and tool utilization, reasoning latency, and downstream query execution.

1.2 Our Contribution

In this work, we propose a benchmarking methodology for text-to-Big SQL agents that treats both ends, namely query generation and execution, as first-class citizens. We focus on **zero-shot** LLM agents to examine a worst-case scenario where no specific fine-tuning or additional optimization is applied, revealing the true impact of SQL generation, agent action, and tool interaction on performance and cost in Big Data settings. Our contributions are the following:

1. A novel evaluation framework for text-to-SQL agents designed to capture big query execution. We propose new metrics that jointly assess agent action, reasoning latency, and the cost-effectiveness of generated queries, reflecting both partial correctness and the practical implications of running queries on Big Data engines.
2. A systematic evaluation of state-of-the-art LLMs within a unified ReAct-style agent architecture. This analysis reveals insights beyond accuracy, identifying scenarios where newer models achieve high correctness but are less interactive due to reasoning or tool orchestration overhead, which happens for instance with Opus 4.6.
3. A discussion of the unique challenges, as well as open research questions in text-to-Big SQL, including the interplay between SQL generation, agent tool use, and execution performance, an area largely overlooked by existing benchmarks.

2 Text-to-Big SQL Demands New Metrics

2.1 Limitations of Current Metrics

Typically, a text-to-SQL benchmark suite comprises a set of triples containing a *natural language* (NL) query, a *golden query* in SQL, and a *ground truth* (V^n) result [31]. During evaluation, system performance is measured by comparing the generated SQL to the golden query, and the resulting output ($\hat{V}^n$) to V^n. Overall accuracy is then computed by aggregating these comparisons across the benchmark suite using standard evaluation metrics, such as Exact Matching (EM), Execution Accuracy (EA), and Valid Efficiency Score (VES) [20, 34, 55].

The major limitation of these metrics is their reliance on all-or-nothing correctness[2]. In practice, however, a generated query may deviate from the expected result without being entirely invalid, provided that end users may still be able to determine its correctness through simple validation, for example, by detecting an additional column in the projected output. We summarize the possible outcomes of an SQL execution as follows:

Ⓐ **Incorrect row count**: The result is invalid due to poor SQL translation, for example, mis-specified `WHERE` conditions or inappropriate join types (`INNER`, `LEFT`, etc.), leading to an incorrect number of rows. Such silent failures may remain unnoticed by the user, but always need a new translation and re-execution cycle to ensure correctness.

Ⓑ **Missing columns**: The result is invalid due to omission of mandatory attributes in the projection list, requiring query

[1] Here, "Big SQL" denotes a conceptual domain in SQL query processing; it is unrelated to IBM's Db2 Big SQL product [22]. "Big SQL" is a trademark of IBM Corporation, used descriptively and with no affiliation.

[2] The corresponding formulas are provided in the Appendix.

re-execution to retrieve the complete output by adding the missing attributes in the `SELECT` clause.

Ⓒ **Superfluous columns**: The result is valid, as we assume that experienced users can manually drop the extra columns without modifying the returned output, which is very cheap and fast. For instance, `df.drop("extra_col")` returns a new DataFrame without the extra column in Spark [45]. However, processing superfluous data affects execution performance and cost and should be penalized in the assessment.

2.2 Proposed Metrics

To encode our notion of query correctness in a measurable metric, we extend the standard text-to-SQL VES metric to account for superfluous columns. Specifically, to quantify the overhead introduced by including irrelevant columns, we compute the column-level precision: $P(S, \hat{S}) = \frac{|S \cap \hat{S}|}{|\hat{S}|}$, where S is the set of ground-truth columns and $\hat{S}$ is the set of columns in the execution result. This captures the fraction of retrieved columns that are actually relevant, penalizing additional, unnecessary columns without discarding partially correct results (Ⓒ).

Also, our new metric considers the total end-to-end (e2e) time (T_{e2e}), which includes all back-and-forth interactions between the LLM and the agent, the execution of interim tools, as well as the time required to run the generated SQL query on the underlying Big Data engine. Combining both concepts, we propose the novel text-to-Big SQL metric called VES*, which is defined as follows for N queries:

$$\text{VES}^* = \frac{1}{N} \sum_{i=1}^{N} \left(\mathbb{1}(V_i, \hat{V}_i) \cdot P(S_i, \hat{S}_i) \cdot \frac{T_{gold}}{T_{e2e}} \right), \tag{1}$$

where T_{gold} denotes the execution time of the golden query, and $\mathbb{1}(V_i, \hat{V}_i)$ is an indicator function that tells whether the output of the generated query $\hat{V}_i$ matches the expected result V_i (Ⓐ and Ⓑ):

$$\mathbb{1}(V, \hat{V}) = \begin{cases} 1, & \text{if } V \text{ is contained in expected output } \hat{V} \\ 0, & \text{otherwise} \end{cases}, \tag{2}$$

We also introduce the *Valid Cost-Efficiency Score* (VCES), a cost-oriented derivative of VES* to account for the overall execution cost C_{e2e}, including the iterative interactions between the LLM and agent, the execution of agent-invoked tools, and the runtime of the generated query on the Big Data engine. For text-to-Big SQL, benchmarking query execution cost is relevant, particularly in cloud deployments:

$$\text{VCES} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{\mathbf{1}(V_i, \hat{V}_i) \cdot P(S_i, \hat{S}_i) \cdot \frac{T_{gold}}{T_{e2e}}}{C_{e2e}} \right). \tag{3}$$

To complement the previous metrics, we introduce the *Expected Cost per Valid Query* (CVQ), which quantifies the anticipated cost to obtain a valid result under a retry-until-success strategy. Let p denote the single-shot validity rate, that is, the fraction of generated big queries that are valid. Consequently, the expected number of attempts until success follows a geometric distribution with mean $1/p$. Accordingly, we define $CVQ = \frac{C_{\text{e2e}}}{p}$.

3 AI Agent Design

3.1 Decision-Making Logic

To evaluate the novel text-to-Big SQL paradigm, we use a ReAct (Reasoning + Acting) agent [50], a well-established framework [32, 40, 43], which decomposes agentic operation into three intertwined components: *Thought*, *Action*, and *Observation*. Via *Thought*, the agent reasons about the task and decides its next step; through *Action*, it interacts with the environment or an external tool to gather or process information; and through *Observation*, it interprets feedback from these actions to refine subsequent reasoning steps.

We deliberately keep the agent simple to provide a broader perspective in our analysis. While more complex AI agents exist [29], we opt to leverage the long-context capabilities of production LLMs guided by iterative cycles of Thought, Action, and Observation, which has already demonstrated its effectiveness in text-to-SQL tasks [11, 33].

We define the *controller* as the connected LLM that guides ReAct-style reasoning, decides when and how to use tools, and produces the final answer. Conversely, the *executor* is the program that connects the LLM with external tools and handles the execution loop.

3.2 Agentic Architecture

The architecture of our agent is illustrated in Figure 1. For the downstream query engine, we choose Spark SQL [45] due to its widespread adoption for large-scale structured data analysis. Although this work focuses on Spark, the evaluated agentic workflow is compatible with most data analytics frameworks and backend deployments. The Spark session can be connected to either a local deployment or a distributed cluster.

The controller interacts with the Spark session through a set of four tools, whose specifications are fully inserted into the context in a zero-shot manner [21]. We define the tools as follows:

① `list_tables`: Look up the Spark Catalog [6] to retrieve available tables by executing a `SHOW TABLES` statement.
② `get_schema`: Retrieve the schema for one or more tables from the Spark Catalog using `SHOW CREATE TABLE t`. Optionally, the controller can fetch an adjustable number of sample rows from each table with the same tool call, which triggers an additional `SELECT * FROM t` statement.
③ `check_query`: Verify the syntax of a proposed query using predefined heuristics [11] with a connected LLM (the *checker*).

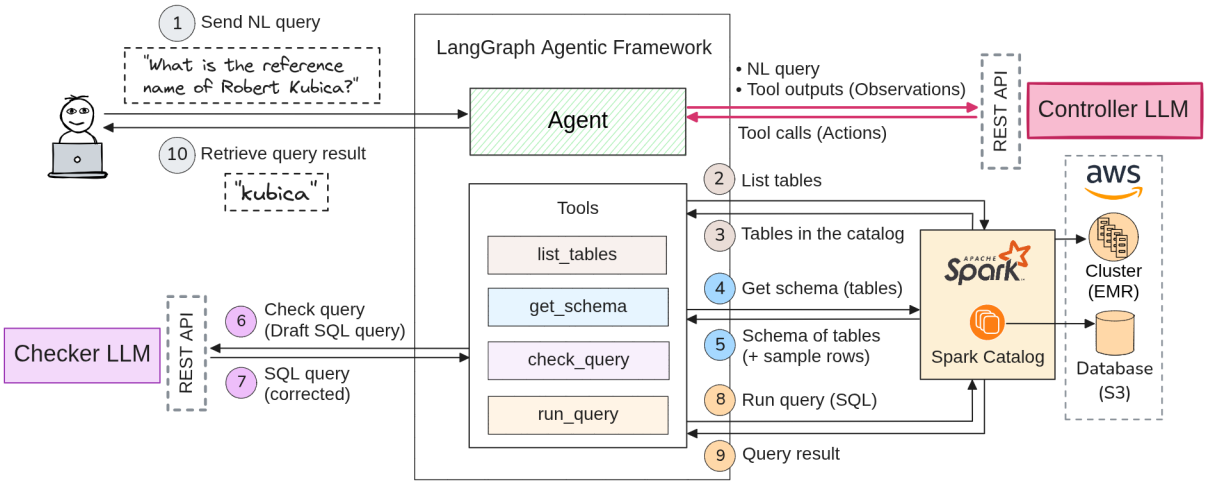


Figure 1. Architecture and typical execution flow of the evaluated Text-to-SQL agent.

④ `run_query`: Execute a SQL query in the connected Spark Session, regardless of whether the deployment is local or cluster-based.

We base our implementation on the original LangChain Spark SQL Toolkit [26] but port it to LangGraph [25]. We use the LangChain stack because it is actively maintained [24] and frequently serves as a research baseline [8, 15, 35].

The proposed agent could be adapted for more fine-grained component optimization or more flexible workflows. To wit, both our controller and checker operate on the same LLM, and the same controller is leveraged for all iterations of the ReAct framework. Instead, independent LLMs could be tuned for performance or cost-efficiency at the iteration level (see Section 5).

3.3 Text-to-Big SQL Workflow

Listing 1 provides the trace of a typical execution flow of the proposed agent[3]. The controller begins by listing all tables available within the connected database using the `list_tables` tool (steps ②-③). Although the agent currently supports a single connection, it can be extended to a multi-database catalog similar to other benchmarks [27], a feature Spark supports natively [4]. Next, the controller identifies the involved tables in the query and retrieves their attributes and data types from the catalog by calling `get_schema`. This tool returns the schema to the controller in a series of symbolic `CREATE TABLE`s. To further guide query construction the controller may optionally sample a custom number of rows as a practical hint (steps ④-⑤). Once the controller decides that it has gathered sufficient database metadata, it generates a SQL query and dispatches it to the checker for necessary syntax corrections (steps ⑥-⑦). The final SQL query is then executed within the Spark Session by calling the `run_query` tool (step ⑧), with either the resulting data or the execution error returned to the user (step ⑨).

Listing 1. Typical execution trace of the Text-to-SQL agent.

```
Input NL Query: "How many superheroes have blue eyes?"

Thought: I need to first check what tables are available...
Action: list_tables_sql_db
Action Input: {}
Observation: superhero, superpower, alignment...

Thought: I should check the schema of the tables...
Action: schema_sql_db
Action Input: {'table_names': 'superhero, colour'}
Observation: CREATE TABLE colour (id int, colour string);
CREATE TABLE superhero (id int, superhero_name string, ...);

Thought: Let me write the query and have it checked first.
Action: query_checker_sql_db
Action Input: {'query': "SELECT COUNT(s.id) FROM superhero s
JOIN colour c ON s.eye_colour_id = c.id WHERE c.colour =
    'Blue'"}
Observation: SELECT COUNT(`s`.`id`) FROM `superhero` `s`
    JOIN `colour` `c` ON `s`.`eye_colour_id` = `c`.`id`
    WHERE `c`.`colour` = 'Blue'

Thought: The query is correct, so I can execute it.
Action: run_query_sql_db
Action Input: {'query': "SELECT COUNT(`s`.`id`) FROM
    `superhero` `s` JOIN `colour` `c` ON
    `s`.`eye_colour_id` = `c`.`id` WHERE `c`.`colour` =
    'Blue'"}
Observation: [('234',)]

Thought: I now know the final answer.
Final Answer: 234
```

[3]Although the ReAct paradigm explicitly models reasoning as a 'Thought' step, in modern LLMs reasoning may occur internally and thus may not be observable in traces. Instead, LLMs may return tool calls directly [44].
[4]https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html

To prevent the controller from iteratively correcting and re-running queries, we terminate the agent immediately following the first `run_query` execution. We take this design decision because, in Big Data systems, unrestricted execution loops can lead to excessive resource consumption or high billing costs, and may even result in *stuck-in-the-loop* [10] scenarios without improving the accuracy of the inferred query.

While this trace reflects the most frequent tool-call sequence, the actual execution flow may vary depending on the controller LLM, which may repeat or omit specific tools.

4 Evaluation

Our evaluation proceeds as follows. §4.1 shows that text-to-SQL metrics lack informativeness, revealing opportunities from fine-grained agent benchmarking. §4.2 demonstrates the superior discriminability of text-to-Big SQL versus text-to-SQL metrics in differentiating agents across latency and cost objectives. Finally, §4.3 shows data scale matters and that its impact can be quantified via text-to-Big SQL metrics. We employ two benchmarks: a text-to-SQL-focused one and a Big Data-focused one.

- **BIRD [31]**, a text-to-SQL benchmark that assesses translation accuracy for realistic databases.
- **TPC-H [46]**, a classic data analytics benchmark that measures database performance on complex ad-hoc business queries over relational data. TPC-H is very useful for text-to-Big SQL because it allows for the deterministic scaling of data.

We conducted all experiments on an AWS `m5.xlarge` EC2 [4] instance in the `us-east-1` region. We selected a set of representative frontier models from various providers for the evaluation, according to the following criteria: (1) current frontier models from Google, Anthropic, and OpenAI; (2) previous-generation frontier models from these same providers; and (3) the three open-source models with the highest scores on SWE-rebench [5] (all at the time of the experiment, mid-February 2026) [6]. For all models, we used the official provider APIs and calculated costs based on their respective per-token pricing.

To ensure consistency for interactive use, all models are deployed with low-latency reasoning configurations. We standardize sampling hyperparameters, such as temperature, top-p, and maximum token limits, across all APIs, except where specific parameters are not exposed by the provider.

4.1 Accuracy is Not Enough in Text-to-Big SQL

When models achieve similar accuracy, standard text-to-SQL metrics fail to differentiate between setups. We prove this by inspecting the Execution-based Focused Evaluation (EX) [27], a SOTA text-to-SQL accuracy metric. Since text-to-SQL performance continues to improve independently [20], we design a practical testbed assuming near-perfect accuracy. To this end, we select eight queries from the BIRD dataset where all tested models (except the later-added GPT-5.2) attain an average EX of at least 0.85.

EX and e2e execution time alone lack the informativeness to discriminate models effectively. Figure 1 shows this: for faster models, the accuracy/speed tradeoff is unclear: e.g., is GPT-4o better than Gemini 3 Flash (27.79% faster but imperfect accuracy)? **If wrong-query costs are high, a slightly slower but more accurate model may be preferable in production**.

Later-generation models do not clearly outperform their predecessors in zero-shot agentic text-to-SQL. For example, Opus 4.6 achieves perfect accuracy but takes 92.37% longer execution time than GPT-4o. Similarly, Gemini 3 Pro and GLM-5 exhibit poor latency, further exacerbated by high variance from API instabilities. Notably, GPT-4o was released in 2024, nearly two years before the other two models.

Table 1. EX, total execution time and the breakdown of execution time per "Observation-Thought-Action" stage for each LLM, on selected BIRD queries. Results are displayed in ascending order of total execution time.

Model	EX	E2E (s)		% of E2E Time			
		Mean	σ	list	schema	check	run
GPT-4o	0.93	6.55	2.08	9.22	13.15	62.08	13.64
Gemini 3 Flash	1.00	8.37	2.61	9.74	10.15	66.76	11.71
GPT-5.2	0.69	8.44	2.74	13.18	21.36	48.88	14.88
Gemini 2.5 Flash	0.95	9.18	3.21	11.70	10.53	66.07	10.11
Claude Opus 4.5	1.00	11.40	2.82	16.90	18.57	42.32	20.93
Claude Opus 4.6	1.00	12.60	2.09	17.18	18.11	42.70	20.93
Kimi K2.5	0.98	13.61	5.72	9.99	14.35	62.89	11.74
GPT-5	0.88	15.45	8.63	7.91	11.09	72.04	7.98
Gemini 3 Pro	1.00	54.55	45.34	18.93	18.85	45.30	16.65
GLM-5	1.00	79.63	50.57	11.77	10.60	66.91	10.53

For better observability, we breakdown execution time, aggregating Observation-Thought-Action iterations that call the same tool into the *stage* abstraction. Common patterns emerge across models: the `check_query` stage dominates end-to-end time in all LLMs, as expected since it runs within the LLM rather than the local Spark session. Yet percentages vary widely (e.g., a 23% spread between GPT-5 and GPT-5.2).

These results suggest stage-specific optimization via model selection. For instance, the time split between `list_tables` and `run_query` differs by model. **Overall, smart per-stage model assignment (as in model ensembles [10]) offers clear optimization potential**.

4.2 Big SQL Metrics Zoom In on Performance

In the context of Big Data, text-to-SQL metrics fail to inform model selection. Instead, a Big SQL lens differentiates LLMs

[5] https://swe-rebench.com/

[6] Detailed results for each model are available in the Appendix.

more clearly, providing sharper selection criteria. Table 2 shows normalized VES and VES* with respect to the best scoring LLM setup: GPT-4o. As shown in the table, the **VES* metric better discriminates accurate models (809.09% dispersion range versus 54.93% in VES)**. The main reason is that VES considers only query execution time and result accuracy [20]. However, modern LLMs excel at text-to-SQL: their generated queries often functionally resemble the gold query (even if not identical) yielding similar execution times. At high accuracy levels, this hinders model discrimination. Our proposed metric addresses this limitation by breaking ties, favoring LLMs that combine efficient agentic interaction with minimal projection overhead (i.e., avoiding superfluous columns).

Table 2. VES and VES* for the selected BIRD queries. Both metrics are normalized to the best scoring LLM. LLMs are displayed in descending order of VES*.

Model	VES (norm)	VES* (norm)	Time Variation (x)			
			list	schema	check	run
GPT-4o	**1.00**	**1.00**	1.00x	1.00x	1.00x	1.00x
Gemini 3 Flash	1.06	0.81	1.35x	0.99x	1.37x	1.10x
Gemini 2.5 Flash	1.00	0.78	1.78x	1.12x	1.49x	1.04x
Claude Opus 4.5	1.09	0.57	3.19x	2.46x	1.19x	2.67x
Claude Opus 4.6	1.09	0.51	3.58x	2.65x	1.32x	2.95x
GPT-5	0.89	0.46	2.02x	1.99x	2.74x	1.38x
Kimi K2.5	1.05	0.45	2.25x	2.27x	2.10x	1.79x
GPT-5.2	0.71	0.39	1.84x	2.09x	1.01x	1.41x
Gemini 3 Pro	1.06	0.23	17.10x	11.95x	6.08x	10.17x
GLM-5	1.05	0.11	15.52x	9.81x	13.10x	9.39x

A high VES* effectively reflects both accurate and low latency models, ranking GPT-4o highest (Table 2). It also reveals nuances, such as the Opus models benefiting from perfect accuracy despite incurring higher execution times. When normalizing execution times relative to the fastest model (GPT-4o), GPT-4o consistently "wins" across all stages, indicating a clear separation between"fast" and "slow" model classes. VES is unable to capture this system behavior.

VCES matches the discriminative power of VES* while incorporating cost, as it factors in per-token billing and the expenses of suboptimal query execution. Table 3 presents VCES per LLM and identifies Gemini 3 Flash as the most cost-efficient option due to its low per-token pricing[7]. Ultimately, **VCES complements VES* by enabling the selection of cost-efficient LLM setups, thus addressing a significant gap in text-to-SQL metrics that directly ignore cost**. We attach CVQ to better support our finding: GPT-4o more than doubles the per-query cost of Gemini 3 Flash due to its lower accuracy, which leads to more failed queries and higher token consumption.

[7] 0.5/3.0 per input/output token for Gemini 3 Flash [17], vs. 2.5/10.0 for GPT-4o [38].

Table 3. VCES and CVQ for the selected BIRD queries. The VCES values are normalized to the best-performing LLM, and models are shown in descending order of VCES.

Model	VCES norm. ($\$^{-1}$)	CVQ ($)	Cost Variation (x)			
			list	schema	check	run
Gemini 3 Flash	**1.00**	0.0044	1.00x	1.00x	1.00x	1.00x
Gemini 2.5 Flash	0.85	0.0053	1.36x	1.18x	1.12x	0.98x
GPT-4o	0.55	0.0107	2.14x	2.93x	2.10x	2.64x
Kimi K2.5	0.53	0.0047	1.07x	1.48x	0.99x	1.05x
GPT-5.2	0.25	0.0124	2.62x	4.07x	1.42x	2.46x
GPT-5	0.24	0.0118	1.92x	2.58x	2.55x	1.61x
Claude Opus 4.5	0.08	0.0388	15.30x	16.13x	5.59x	15.76x
Claude Opus 4.6	0.08	0.0359	14.39x	14.56x	5.22x	14.58x
Gemini 3 Pro	0.06	0.0220	9.71x	9.29x	3.39x	7.11x
GLM-5	0.05	0.0129	3.54x	3.06x	2.94x	2.63x

As with execution time results, the leading cost performer in Table 3 dominates across nearly all stages. Interestingly, the results reveal a clear trade-off between **latency-optimal** and **cost-efficient** models: for example, GPT-4o ranks first in execution speed but achieves only half the cost-efficiency of Gemini 3 Flash, which is slightly slower but much cheaper. This distinction suggests that cost-efficient models may be preferable for stages with relaxed latency requirements, while VES*-optimal models are ideal when latency is critical.

4.3 The Aftermath of Data Scale

Text-to-SQL metrics do not capture data scale. However, data scale plays a key role in text-to-Big SQL, as shown in Figure 2. The key observation is that both ends can hinder interactive analytics, as seen across all three models. At small scale factors (SFs), agent interactions dominate the overall execution time, while at large SFs, fast responses are limited by the long query execution times; the execution time of TPC-H Query 21 increases by 13.3× when scaling from SF 10 to SF 1000 on the same cluster [8]. Even with highly optimized query engines, the end-to-end performance may remain constrained by the latency introduced through LLM–agent–tool interactions.

The remaining key question is the extent to which a new metric is needed to capture data scaling effects. To examine this, we chose four TPC-H queries where LLMs deliver poor accuracy (Figure 3): three complex ones (Q17, Q18, Q21) with nested subqueries, plus one simple single-table query (Q1). We used TPC-H business questions as NL inputs and official SQL as ground truth (see Appendix for details).

As illustrated in Figure 4, **VES (which incorporates SQL execution time) yields constant relationships between models across scale factors. In contrast, CVQ better**

[8] Tests ran on a Amazon EMR [2] cluster with `r5b.xlarge` master/core nodes, 32 core nodes (4 vCPUs each), and 128 GB gp3 EBS volumes.

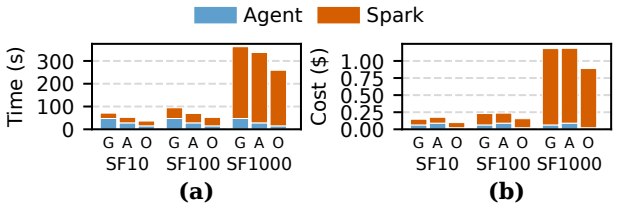


Figure 2. Breakdown of agent execution time (a) and cost (b) across different scale factors (SF) for TPC-H Query 21. Each bar series represents a specific model: Gemini 3 Pro (G), Claude Opus 4.5 (A), and GPT-5.2 (O).

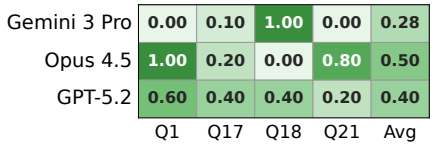


Figure 3. Execution accuracy across three models for TPC-H queries 1, 17, 18, and 21. Accuracy is computed as the average value of Equation 2 over 10 zero-shot translations of each corresponding natural language query.

captures each model's potential cost loss at varying scales: less accurate models (e.g., Gemini 3 Pro here) pose greater risk at large scale factors, where query errors incur substantially higher costs. Consequently, failing a query at SF 1000 is far more expensive than at SF 10. Even a modest accuracy gap (such as the 10% difference between Opus 4.5 and GPT-5.2) becomes critically amplified at higher scales.

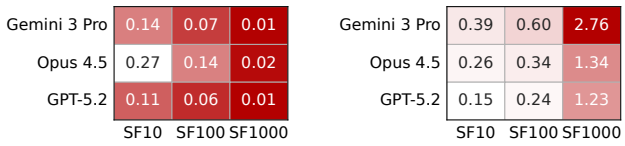


Figure 4. Text-to-SQL (a) and text-to-Big SQL (b) metrics across three models and scale factors, averaged for TPC-H queries 1, 17, 18, and 21.

Our evaluation clarifies the interpretation of text-to-Big SQL metrics. VES* and VCES provide practical assessments at fixed data scale factors, as they normalize generated SQL execution time and cost to the ground truth. In contrast, CVQ serves as an essential complement by quantifying the amplified impact of query inaccuracies as data scales.

5 Discussion

We discuss our results in two dimensions. First, we elaborate on how our contributions should be interpreted and extended to provide more representative benchmarking for production contexts. Second, we identify promising research directions emerging from these initial steps toward Text-to-Big SQL.

5.1 Benchmarking Real-World Deployments

Text-to-SQL benchmarks offer a practical and reproducible means of evaluating translators, yet they fail to faithfully capture business and production contexts. Constructing a realistic Text-to-Big SQL benchmark requires accounting for the specific characteristics of production Big Data environments.

In benchmarks, we assume the golden query is the sole correct translation of its corresponding natural language query. However, in Big Data, a correct query can take several semantically equivalent forms, affecting physical plan quality, shuffled data size, scan efficiency, and therefore latency and resource consumption [42].

The logical approach would designate the *optimal* or *worst-case* correct SQL as the golden query for each natural language query. Identifying these extremes is computationally infeasible due to the combinatorial explosion of possible query rewrite options, particularly for complex queries with multiple variables. Additionally, optimal queries are backend-dependent: a golden query for one system may be suboptimal for another based on pricing models, execution engines, or data partitioning strategies. Consequently, golden queries should serve not as universal baselines of optimality, but as reference points for relativizing translator efficiency.

Furthermore, text-to-SQL benchmarks execute each query an equal number of times to compute metrics [20, 31]. In practice, however, query frequencies follow skewed distributions where certain patterns dominate; for instance, certain operators may be significantly more frequent than others [47], which complicates absolute performance assessments [9]. Consequently, our method enables fair comparisons of text-to-SQL engines under identical conditions (same backend and golden query set) rather than providing absolute performance rankings.

Ideally, Text-to-Big SQL benchmarks should acknowledge this diversity of correct solutions by evaluating against multiple valid query variants rather than a single golden query. Meaningful comparison requires analyzing not merely syntactic structure, but the operator semantics of underlying physical execution plans, examined both pairwise and holistically across varying scale factors. We view this as a challenging challenge that merits broader future research.

[9]For instance, an abnormally frequent query structure accompanied by highly optimal golden queries would skew the overall benchmark assessment.

5.2 Future Opportunities in Text-to-Big SQL

Based on our result, we identify several promising avenues in the broader text-to-Big SQL domain. Figure 5 summarizes our key insight: even if text-to-SQL reached 99% accuracy since natural language ambiguity is unavoidable, text-to-Big SQL challenges would still remain.

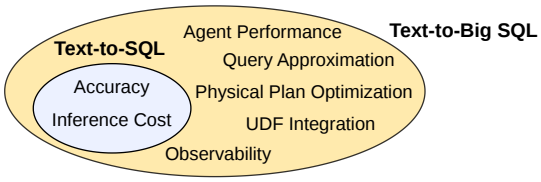


Figure 5. Challenges of Text-to-Big SQL include, but are not limited to, those of Text-to-SQL.

Agent performance tuning. Optimizing the internal stages of agents itself presents a significant research opportunity. Both our findings and prior work [10] demonstrate that both performance and cost could be improved by "strategically" assigning specialized models to different stages (e.g., navigating "fast" and "cheap" models). However, existing physical plan optimization approaches for semantic operators incur latencies of tens of seconds [39, 54], making them incompatible with interactive analytics. Adapting these models to meet the latency constraints of interactive Big Data analytics remains a major open challenge.

Text-to-SQL must be optimized for large-scale query execution. In Big Data, syntactically correct SQL may still be impractical if it triggers large shuffles, unnecessary joins, or full-table scans. Text-to-Big SQL must therefore optimize for both correctness and cost-efficient, large-scale execution, which remains an open challenge.

One promising direction is to leverage historical execution traces enriched with performance metrics and system-level quality indicators, such as VCES and CVQ. By semantically matching newly generated queries [16] to past executions [48], the system can estimate expected cost and runtime before execution and proactively rewrite inefficient queries [19, 42]. Incorporating physical plans and cost models [7] further enables extrapolation across data scales, supporting scale-aware optimization rather than static SQL translation. Alternatively, the agent can be few-shot with similar past queries and their Big SQL metrics to infer optimized SQL.

Beyond exact execution, Big Data systems often rely on approximate queries to trade precision for performance [9]. However, current text-to-SQL models rarely reason about such semantic alternatives. A text-to-Big SQL framework should instead consider approximate joins, sampling-based aggregations, or sketch-based summaries when they satisfy user intent while significantly reducing execution cost. User-provided QoS annotations (e.g., "run this fast") could further guide optimization.

User-defined functions (UDFs). Another key challenge in text-to-Big SQL arises from UDFs. Big Data engines like Spark, Athena, and BigQuery often leverage custom UDFs, which are not fully expressible in standard SQL. As a result, text-to-Big SQL solutions must produce UDF-compatible SQL or hybrid code, for example, combining SQL with Spark DataFrames, which goes beyond classical text-to-SQL.

6 Conclusion

In this work, we have started addressing the surprisingly underexplored domain of "text-to-Big SQL": the integration of text-to-SQL systems within Big Data engines. We have shown that standard text-to-SQL benchmarks fall short and, for the first time, introduced text-to-Big SQL measures that reflect the true interaction between agents, LLMs, Big Data systems, and data scale. We have evaluated state-of-the-art production LLMs to demonstrate that our metrics provide an effective assessment framework for text-to-Big SQL. Our work exposes new real-world challenges and guides future research in text-to-Big SQL.

Acknowledgments

This research work has been partly funded by the EU Horizon programme, grant no. 101092646 (CloudSkin), and by the Spanish MICIU/AEI, grant no. PID2023-148202OB-C21. Germán T. Eizaguirre is recipient of a pre-doctoral FPU grant from the Spanish Ministry of Universities (ref. FPU21/00630).

References

[1] Ahmed Ahmed, A. Feder Cooper, Sanmi Koyejo, and Percy Liang. 2026. Extracting books from production language models. arXiv:2601.02671 [cs.CL] https://arxiv.org/abs/2601.02671

[2] Amazon Web Services. 2026. Amazon EMR - Cloud Big Data Platform. https://aws.amazon.com/emr/. Accessed: February 22, 2026.

[3] Amazon Web Services, Inc. 2026. Amazon Athena - Serverless Interactive Query Service. https://aws.amazon.com/athena/ Accessed: 2026-02-12.

[4] Amazon Web Services, Inc. 2026. Amazon EC2. https://aws.amazon.com/ec2/ Accessed: 2026-02-22.

[5] Anthropic. 2024. Claude 3 Opus. https://www.anthropic.com/claude/opus Accessed: 2026-02-21.

[6] Apache Software Foundation. 2026. pyspark.sql.Catalog — PySpark 4.1.1 Documentation. https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.Catalog.html. Accessed: 2026-02-19.

[7] Lorenzo Baldacci and Matteo Golfarelli. 2019. A Cost Model for SPARK SQL . *IEEE Transactions on Knowledge & Data Engineering* 31, 05 (May 2019), 819–832. doi:10.1109/TKDE.2018.2850339

[8] Pranav Bhagat, K N Ajay Shastry, Pranoy Panda, and Chaitanya Devaguptapu. 2025. Evaluating Compound AI Systems through Behaviors, Not Benchmarks. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, Suzhou, China, 24193–24222. doi:10.18653/v1/2025.findings-emnlp.1314

[9] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *2017 ACM International Conference on Management of Data (SIGMOD'17)*. Association for Computing Machinery, New York, NY, USA, 511–519. doi:10.1145/3035918.3056097

[10] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. arXiv:2510.06189 [cs.AI] https://arxiv.org/abs/2510.06189

[11] Yeounoh Chung, Gaurav T. Kakkar, Yu Gan, Brenton Milne, and Fatma Özcan. 2025. Is Long Context All You Need? Leveraging LLM's Extended Context for NL2SQL. *Proc. VLDB Endow.* 18, 8 (April 2025), 2735–2747. doi:10.14778/3742728.3742761

[12] crewAI. 2026. crewAI: Multi AI Agents Systems. https://www.crewai.com/ Accessed: 2026-02-21.

[13] Minghang Deng, Ashwin Ramachandran, Canwen Xu, Lanxiang Hu, Zhewei Yao, Anupam Datta, and Hao Zhang. 2025. ReFoRCE: A Text-to-SQL Agent with Self-Refinement, Consensus Enforcement, and Column Exploration. arXiv:2502.00675 [cs.CL] https://arxiv.org/abs/2502.00675

[14] Saurabh Deochake and Debajyoti Mukhopadhyay. 2025. Cost-Aware Text-to-SQL: An Empirical Study of Cloud Compute Costs for LLM-Generated Queries. arXiv:2512.22364 [cs.DB] https://arxiv.org/abs/2512.22364

[15] Peng Ding and Rick Stevens. 2025. Unified Tool Integration for LLMs: A Protocol-Agnostic Approach to Function Calling. arXiv:2508.02979 [cs.AI] https://arxiv.org/abs/2508.02979

[16] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1534–1547. doi:10.14778/3583140.3583165

[17] Google. 2026. Gemini API Pricing. https://ai.google.dev/gemini-api/docs/pricing. Accessed: 2026-02-23.

[18] Google Cloud. 2026. BigQuery: Cloud Data Warehouse. https://cloud.google.com/bigquery?hl=en Accessed: 2026-02-12.

[19] Jiahao He, Yutao Cui, Cuiping Li, Jikang Jiang, Yuheng Hou, and Hong Chen. 2025. AQORA: A Fast Learned Adaptive Query Optimizer with Stage-Level Feedback for Spark SQL. arXiv:2510.10580 [cs.DB] https://arxiv.org/abs/2510.10580

[20] Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-Generation Database Interfaces: A Survey of LLM-Based Text-to-SQL . *IEEE Transactions on Knowledge & Data Engineering* 37, 12 (Dec. 2025), 7328–7345. doi:10.1109/TKDE.2025.3609486

[21] Cheng-Yu Hsieh, Si-An Chen, Chun-Liang Li, Yasuhisa Fujii, Alexander Ratner, Chen-Yu Lee, Ranjay Krishna, and Tomas Pfister. 2023. Tool Documentation Enables Zero-Shot Tool-Usage with Large Language Models. arXiv:2308.00675 [cs.CL] https://arxiv.org/abs/2308.00675

[22] IBM. [n. d.]. IBM Db2 Big SQL. https://www.ibm.com/es-es/products/db2-big-sql. Accessed: 2026-02-21.

[23] Dimitrios Koutsoukos, Renato Marroquín, Ingo Müller, and Ana Klimovic. 2025. Adaptive data transformations for QaaS. In *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org. https://vldb.org/cidrdb/2025/adaptive-data-transformations-for-qaas.html

[24] LangChain. 2026. LangChain GitHub Repository. https://github.com/langchain-ai/langchain. Accessed: 2026-02-14.

[25] LangChain Inc. 2024. LangGraph: Agent Orchestration Framework for Reliable AI Agents. https://www.langchain.com/langgraph Accessed: 2025-05-14.

[26] LangChain Inc. 2026. Spark SQL | LangChain. https://python.langchain.com/docs/integrations/tools/spark_sql/ Accessed: 2026-02-11.

[27] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. 2025. Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows. arXiv:2411.07763 [cs.CL] https://arxiv.org/abs/2411.07763

[28] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proc. VLDB Endow.* 17, 11 (July 2024), 3318–3331. doi:10.14778/3681954.3682003

[29] Boyan Li, Jiayi Zhang, Ju Fan, Yanwei Xu, Chong Chen, Nan Tang, and Yuyu Luo. 2025. Alpha-SQL: Zero-Shot Text-to-SQL using Monte Carlo Tree Search. arXiv:2502.17248 [cs.DB] https://arxiv.org/abs/2502.17248

[30] Haoyang Li, Shang Wu, Xiaokang Zhang, Xinmei Huang, Jing Zhang, Fuxin Jiang, Shuai Wang, Tieying Zhang, Jianjun Chen, Rui Shi, Hong Chen, and Cuiping Li. 2025. OmniSQL: Synthesizing High-Quality Text-to-SQL Data at Scale. *Proc. VLDB Endow.* 18, 11 (July 2025), 4695–4709. doi:10.14778/3749646.3749723

[31] 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* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1835, 28 pages.

[32] Chunwei Liu, Gerardo Vitagliano, Brandon Rose, Matthew Printz, David Andrew Samson, and Michael Cafarella. 2025. PalimpChat: Declarative and Interactive AI analytics. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 183–186. doi:10.1145/3722212.3725122

[33] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph E. Gonzalez, and Aditya G. Parameswaran. 2025. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. arXiv:2509.00997 [cs.AI] https://arxiv.org/abs/2509.00997

[34] Yuyu Luo, Guoliang Li, Ju Fan, Chengliang Chai, and Nan Tang. 2025. Natural Language to SQL: State of the Art and Open Problems. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 5466–5471. doi:10.14778/3750601.3750696

[35] Heng Ma, Alexander Brace, Carlo Siebenschuh, Ian Foster, and Arvind Ramanathan. 2025. LangChain-Parsl: Connect Large Language Model Agents to High Performance Computing Resource. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25)*. Association for Computing Machinery, New York, NY, USA, 78–85. doi:10.1145/3731599.3767349

[36] Microsoft. 2024. AutoGen: A Programming Framework for Agentic AI. https://microsoft.github.io/autogen/stable//index.html Accessed: 2026-02-21.

[37] OpenAI. 2025. GPT-5.2 Model Documentation. https://developers.openai.com/api/docs/models/gpt-5.2 Accessed: 2026-02-21.

[38] OpenAI. 2026. API Pricing. https://developers.openai.com/api/docs/pricing/. Accessed: 2026-02-23.

[39] Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael Cafarella, Tim Kraska, and Samuel Madden. 2026. Abacus: A cost-based optimizer for Semantic Operator Systems. https://arxiv.org/abs/2505.14661

[40] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. 2026. AI Agents vs. Agentic AI: A Conceptual taxonomy, applications and challenges. *Information Fusion* 126 (2026), 103599. doi:10.1016/j.inffus.2025.103599

[41] Jiawei Shen, Chengcheng Wan, Ruoyi Qiao, Jiazhen Zou, Hang Xu, Yuchen Shao, Yueling Zhang, Weikai Miao, and Geguang Pu. 2025. A Study of In-Context-Learning-Based Text-to-SQL Errors. arXiv:2501.09310 [cs.CL] https://arxiv.org/abs/2501.09310

[42] Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. 2026. QUITE: A Query Rewrite System Beyond Rules with LLM Agents. arXiv:2506.07675 [cs.DB] https://arxiv.org/abs/2506.07675

[43] Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. 2025. PaperBench: Evaluating AI's Ability to Replicate AI Research. arXiv:2504.01848 [cs.AI] https://arxiv.org/abs/2504.01848

[44] Ian Su, Gaurav Purushothaman, Jey Narayan, Ruhika Goel, Kevin Zhu, Sunishchal Dev, Yash More, and Maheep Chaudhary. 2026. Broken Chains: The Cost of Incomplete Reasoning in LLMs. arXiv:2602.14444 [cs.LG] https://arxiv.org/abs/2602.14444

[45] The Apache Software Foundation. 2026. Spark SQL & DataFrame. https://spark.apache.org/sql/ Accessed: 2026-02-15.

[46] Transaction Processing Performance Council. 2024. TPC Benchmark H (Decision Support) Standard Specification Revision 3.0.1. https://www.tpc.org/tpch/. Accessed: 2026-02-21.

[47] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1413–1425. doi:10.14778/3583140.3583156

[48] Pengyi Wang, Sibei Chen, Ju Fan, Bin Wu, Nan Tang, and Jian Tan. 2025. Andromeda: Debugging Database Performance Issues with Retrieval-Augmented Large Language Models. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 243–246. doi:10.1145/3722212.3725080

[49] Wenxuan Xie, Gaochen Wu, and Bowen Zhou. 2024. MAG-SQL: Multi-Agent Generative Approach with Soft Schema Linking and Iterative Sub-SQL Refinement for Text-to-SQL. arXiv:2408.07930 [cs.CL] https://arxiv.org/abs/2408.07930

[50] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[51] 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. 2019. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887 [cs.CL] https://arxiv.org/abs/1809.08887

[52] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024. FinSQL: Model-Agnostic LLMs-based Text-to-SQL Framework for Financial Analysis. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 93–105. doi:10.1145/3626246.3653375

[53] Tingkai Zhang, Chaoyu Chen, Cong Liao, Jun Wang, Xudong Zhao, Hang Yu, Jianchao Wang, Jianguo Li, and Wenhui Shi. 2024. SQLfuse: Enhancing Text-to-SQL Performance through Comprehensive LLM Synergy. arXiv:2407.14568 [cs.CL] https://arxiv.org/abs/2407.14568

[54] Junhao Zhu, Lu Chen, Xiangyu Ke, Ziquan Fang, Tianyi Li, Yunjun Gao, and Christian S. Jensen. 2025. Beyond Relational: Semantic-Aware Multi-Modal Analytics with LLM-Native Query Optimization. arXiv:2511.19830 [cs.DB] https://arxiv.org/abs/2511.19830

[55] Xiaohu Zhu, Qian Li, Lizhen Cui, and Yongkang Liu. 2024. Large Language Model Enhanced Text-to-SQL Generation: A Survey. arXiv:2410.06011 [cs.DB] https://arxiv.org/abs/2410.06011

Appendix

A Text-to-SQL formulas

A text-to-SQL benchmark suite includes a set of triples containing a *natural language* (NL) query, a *golden query* in SQL (Q^n), and a *ground truth* (V^n) result [31]. During evaluation, system performance is measured by comparing the generated SQL ($\hat{Q}^n$) to the golden query, and the resulting output ($\hat{V}^n$) to V^n [20].

Below we list standard formulas for text-to-SQL accuracy evaluation metrics, which serve as the foundation for the text-to-Big SQL metrics introduced in Section 2.

A.1 Exact Matching (EM)

$$EM = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}\left(Q_i, \hat{Q}_i\right), \tag{4}$$

where

$$\mathbb{I}(Q_i, \hat{Q}_i) = \begin{cases} 1, & Q_i = \hat{Q}_i \\ 0, & Q_i \neq \hat{Q}_i \end{cases}. \tag{5}$$

The $Q_i = \hat{Q}_i$ operation compares two queries to determine their equivalence, which depends on the specific context at hand. Therefore, multiple methods have been developed to address it. A common metric is the Spider exact matching accuracy [51], which verifies whether generated SQL queries match gold-standard references in both structure and specific components. To achieve this, this metric decomposes a SQL query into its constituent clauses, such as `SELECT`, `WHERE`, `HAVING`, `GROUP BY`, and `ORDER BY`, and performs set-based comparisons. This approach ignores ordering differences (e.g., varying sequences in conditions) to focus on **semantic equivalence**.

A.2 Execution Accuracy (EA)

$$EA = \frac{1}{N}\sum_{i=1}^{N} \mathbb{I}\left(V_i, \hat{V}_i\right), \tag{6}$$

where

$$\mathbb{I}(V_i, \hat{V}_i) = \begin{cases} 1, & V_i = \hat{V}_i \\ 0, & V_i \neq \hat{V}_i \end{cases}. \tag{7}$$

Usually, $V_i = \hat{V}_i$ uses a set-based equality: rows and columns must match exactly, ignoring order unless sorting is specified.

A.3 Valid Efficiency Score

$$\text{VES} = \frac{1}{N}\sum_{i=1}^{N} \left(\mathbb{I}(V_i, \hat{V}_i) \cdot \frac{T_{gold}}{T_{gen}}\right), \tag{8}$$

where T_{gold} and T_{gen} are the execution times of the golden query and the generated SQL, respectively.

B BIRD Evaluation

B.1 Metrics Breakdown for BIRD

Our BIRD evaluation averages each metric over 50 iterations. Table 4 provides a breakdown of the resulting Text-to-(Big) SQL metrics per query and model.

B.2 Translation Error Analysis

In the BIRD evaluation we ran 10 queries[10] on 11 models[11] with 50 zero-shot iterations of the translation for each, resulting in 5,500 total query executions. Out of these, 930 translations produced a result different from the expected output, yielding an Equation 7 result of 0. Table 5 presents the per-model aggregate distribution of the errors.

Listing 2. Correct query for Q886.

```
SELECT year
FROM races
GROUP BY year
ORDER BY COUNT(round) DESC
LIMIT 1
```

Listing 3. Incorrect query for Q886 (translated by Opus 4.5).

```
SELECT year, MAX(round) as max_races
FROM races
GROUP BY year
ORDER BY max_races DESC
LIMIT 1
```

Figure 6. Correct and incorrect SQL examples for query 886.

To give deeper insights, we classified the obtained errors based on the taxonomy from [41]. We used their proposed categories because they also focus on LLMs for text-to-SQL translation and provide a broad categorization of possible outcomes. Each SQL query with a result different from the expected output (i.e., '*incorrect*') could be affected by one or more errors. We manually gathered 1,730 errors across the 930 incorrect queries. We depict the error distribution in Figure 7 and a visual breakdown of the errors identified in Figure 8.

We extracted several interesting insights from the analysis. Within the **Output Format (F2)** errors, 627 (93.16%) were due to additional columns, meaning the result contained the expected data accompanied by unnecessary columns. Of those, 77.51% were due to just one additional column. Such results would be considered correct under our text-to-Big SQL metrics (Eq. 2). For instance, query 886 is considered incorrect by classical text-to-SQL metrics primarily because of an

[10] Query 886 was excluded from the main paper results since the Equation 7 results are low for most models. The query asks `"Which year has the most number of races? The most number of races refers to max(round)"`; most models returned the superfluous column `max(round)` instead of only the year.

[11] DeepSeek Chat is omitted in the main paper results due to its comparatively poor performance.

Table 4. Per-query BIRD metrics by model (mean ± std across runs). VES, VES*, and VCES follow the paper definitions; CVQ reports the expected cost per valid query.

QID	Model	Acc.	VES	VES*	VCES	CVQ
61	GPT-4o	1.00	1.08	0.011	1.11	0.010
	GPT-5.2	1.00	0.99	0.006	0.69	0.009
	Claude 4.6	1.00	1.06	0.006	0.20	0.030
	Gemini 3 Flash	1.00	1.06	0.006	0.91	0.006
	Claude 4.5	1.00	1.07	0.006	0.14	0.040
	Kimi K2.5	1.00	1.08	0.005	1.11	0.005
	DeepSeek Chat	0.86	0.90	0.003	1.73	0.002
	Gemini 2.5 Flash	0.60	0.62	0.003	0.40	0.013
	GPT-5	1.00	1.06	0.003	0.23	0.013
	Gemini 3 Pro	1.00	1.10	0.002	0.07	0.024
	GLM-5	1.00	1.02	0.001	0.05	0.015
606	Gemini 3 Flash	1.00	0.98	0.012	1.87	0.006
	Claude 4.5	1.00	1.01	0.011	0.30	0.038
	Gemini 2.5 Flash	1.00	0.97	0.011	1.62	0.007
	GPT-4o	0.52	0.51	0.011	1.10	0.019
	Claude 4.6	1.00	0.98	0.010	0.30	0.035
	Kimi K2.5	1.00	1.00	0.010	2.14	0.005
	DeepSeek Chat	0.92	0.83	0.006	2.87	0.002
	Gemini 3 Pro	1.00	0.98	0.003	0.14	0.022
	GLM-5	1.00	0.96	0.001	0.06	0.022
	GPT-5.2	0.04	0.02	0.001	0.08	0.198
	GPT-5	0.06	0.03	0.000	0.02	0.253
645	GPT-4o	1.00	0.98	0.012	1.31	0.009
	GPT-5.2	1.00	0.99	0.011	1.95	0.006
	Gemini 3 Flash	1.00	0.99	0.010	2.86	0.003
	Gemini 2.5 Flash	1.00	0.99	0.009	2.40	0.004
	Kimi K2.5	1.00	1.00	0.006	1.57	0.004
	Claude 4.5	1.00	0.99	0.006	0.15	0.037
	GPT-5	1.00	0.99	0.006	0.73	0.008
	Claude 4.6	1.00	1.02	0.006	0.16	0.034
	DeepSeek Chat	1.00	1.00	0.004	1.96	0.002
	Gemini 3 Pro	1.00	0.96	0.002	0.09	0.020
	GLM-5	1.00	0.98	0.001	0.15	0.009

QID	Model	Acc.	VES	VES*	VCES	CVQ
776	GPT-4o	0.98	0.96	0.009	0.86	0.011
	Gemini 3 Flash	1.00	0.99	0.009	2.19	0.004
	Gemini 2.5 Flash	1.00	1.00	0.008	1.94	0.004
	GPT-5.2	1.00	1.09	0.006	0.63	0.010
	Claude 4.5	1.00	1.08	0.006	0.15	0.040
	Claude 4.6	1.00	1.13	0.005	0.14	0.037
	Kimi K2.5	0.96	0.94	0.005	0.94	0.005
	GPT-5	1.00	1.04	0.004	0.49	0.009
	DeepSeek Chat	0.50	0.49	0.002	0.77	0.004
	Gemini 3 Pro	1.00	1.00	0.001	0.06	0.024
	GLM-5	1.00	1.12	0.001	0.11	0.011
607	GPT-4o	1.00	0.99	0.011	1.36	0.008
	Gemini 3 Flash	1.00	0.98	0.009	2.89	0.003
	GPT-5.2	1.00	0.96	0.009	1.58	0.006
	Gemini 2.5 Flash	1.00	0.96	0.009	2.47	0.003
	Claude 4.5	1.00	1.00	0.006	0.16	0.036
	GPT-5	1.00	1.01	0.005	0.72	0.007
	Kimi K2.5	1.00	0.99	0.005	1.27	0.004
	Claude 4.6	1.00	0.98	0.005	0.14	0.034
	DeepSeek Chat	1.00	0.98	0.004	2.08	0.002
	Gemini 3 Pro	1.00	0.98	0.002	0.10	0.017
	GLM-5	1.00	0.99	0.002	0.17	0.009
785	Gemini 3 Flash	0.98	0.98	0.007	1.97	0.004
	GPT-4o	0.90	0.88	0.007	0.66	0.012
	GPT-5.2	1.00	0.59	0.007	0.89	0.008
	Gemini 2.5 Flash	1.00	0.96	0.007	1.55	0.004
	Claude 4.5	1.00	0.99	0.005	0.12	0.039
	Kimi K2.5	1.00	0.97	0.004	0.84	0.005
	Claude 4.6	1.00	0.99	0.004	0.11	0.037
	GPT-5	0.96	0.53	0.004	0.41	0.010
	Gemini 3 Pro	1.00	0.97	0.001	0.06	0.022
	GLM-5	1.00	0.74	0.001	0.11	0.010
	DeepSeek Chat	0.26	0.16	0.001	0.39	0.006

QID	Model	Acc.	VES	VES*	VCES	CVQ
813	GPT-4o	0.98	1.46	0.016	1.57	0.011
	Gemini 3 Flash	1.00	1.47	0.015	3.21	0.005
	GPT-5.2	1.00	1.41	0.015	1.90	0.008
	Gemini 2.5 Flash	0.96	1.34	0.012	2.21	0.006
	Claude 4.5	1.00	1.51	0.011	0.28	0.040
	Claude 4.6	1.00	1.41	0.009	0.22	0.038
	Kimi K2.5	0.94	1.36	0.008	1.58	0.006
	GPT-5	1.00	1.43	0.008	0.79	0.010
	Gemini 3 Pro	1.00	1.45	0.004	0.15	0.024
	GLM-5	1.00	1.52	0.002	0.11	0.015
	DeepSeek Chat	0.18	0.26	0.001	0.89	0.007
895	GPT-4o	1.00	0.95	0.073	5.61	0.013
	Gemini 2.5 Flash	1.00	0.98	0.058	10.15	0.006
	Gemini 3 Flash	1.00	0.96	0.054	10.71	0.005
	GPT-5	0.94	0.90	0.037	2.70	0.015
	Claude 4.5	1.00	0.95	0.033	0.61	0.053
	Claude 4.6	1.00	0.96	0.031	0.65	0.048
	Kimi K2.5	1.00	0.97	0.024	3.82	0.006
	Gemini 3 Pro	1.00	0.97	0.020	0.77	0.026
	GLM-5	1.00	0.98	0.007	0.52	0.014
	DeepSeek Chat	0.28	0.26	0.006	3.24	0.007
	GPT-5.2	0.12	0.11	0.005	0.29	0.139
968	GPT-4o	1.00	0.92	0.003	0.39	0.009
	Claude 4.5	1.00	0.93	0.003	0.12	0.025
	Gemini 2.5 Flash	1.00	0.92	0.003	0.82	0.004
	Gemini 3 Flash	1.00	0.87	0.003	0.81	0.003
	Claude 4.6	1.00	0.95	0.002	0.07	0.031
	Kimi K2.5	0.96	0.85	0.002	0.54	0.004
	GPT-5	0.96	0.81	0.002	0.21	0.008
	DeepSeek Chat	0.74	0.65	0.001	0.47	0.002
	Gemini 3 Pro	1.00	0.87	0.001	0.04	0.018
	GLM-5	0.98	0.90	0.000	0.02	0.011
	GPT-5.2	0.04	0.04	0.000	0.02	0.151

Table 5. Total number of incorrect translations (Eq.2== 0) per model. Each model was evaluated on all 10 BIRD queries with 50 translations per query (500 iterations per model).

Model	Incorrect Translations
DeepSeek Chat	214
GPT-5.2	190
GPT-5	104
GPT-4o	78
Gemini 2.5 Flash	70
Kimi K2.5	57
Gemini 3 Flash	51
GLM-5	51
Claude Opus 4.5	50
Claude Opus 4.6	50
Gemini 3 Pro	15

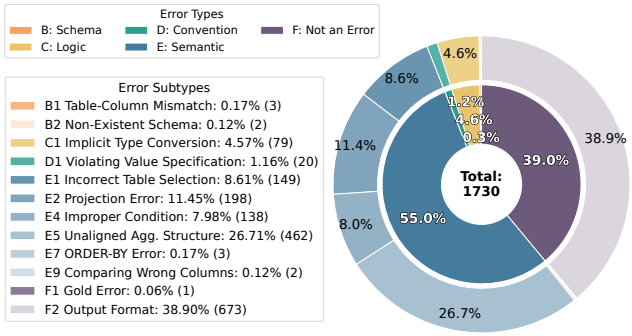


Figure 7. Distribution of identified text-to-SQL translation errors across all 930 incorrect BIRD query translations, categorized by the taxonomy proposed in [41]

additional column (see Listing 3). We consider our metric to better represent valid queries, as an untrained user could easily differentiate the required column in a practical case. We also identified an LLM difficulty in differentiating MAX and COUNT cases. Among **Unaligned Aggregation Structure (E5)** errors, 15.80% resulted from adding unnecessary `MAX` aggregations, while 5.41% were due to unnecessary `COUNT`s; notably, 44.16% involved mixing `MAX` and `COUNT` aggregations. Finally, in four queries, the agents ran an inspection operation (`DESCRIBE table` or `SHOW TABLES`) as the final query, a curious behavior despite the low number of cases.

We only tested the agent extensively on a subset of the BIRD benchmark; hence, these results should not be seen as absolute. However, we believe the above insights provide valuable information for future research.

C TPC-H evaluation

C.1 Model and Query Selection

We selected three last generation frontier models available at the time of our original experiments in January 2026: Gemini 3 Pro, Claude Opus 4.5, and GPT-5.2. We excluded additional models due to budget constraints, as a single TPC-H query run at the specified scale factor costs approximately $1 within our proposed EMR cluster. Given 50 replicas per model and the original set of 10 models from the BIRD evaluation, the total cost for four TPC-H queries would reach roughly $2,000, excluding lower scale factor tests. Because an exhaustive assessment of all available models and queries would not further clarify the primary contributions of this paper, we limited this evaluation to a representative subset.

Regarding query selection, we prioritized TPC-H queries that Claude Opus 4.5 could not translate correctly. We then

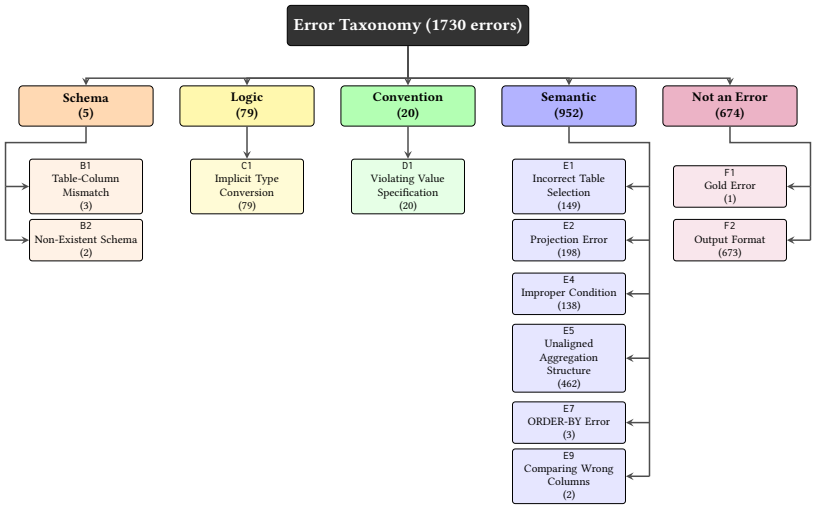


Figure 8. Taxonomy [41] of SQL translation errors from our LLM agent. Numbers in parentheses denote error counts. Categories: A=Syntax, B=Schema, C=Logic, D=Convention, E=Semantic, F=Not an Error. Total: 1730 errors across 5 categories.

verified that (1) accuracy remained imperfect in the other two models and (2) no model had memorized the TPC-H specification during training. While methods exist to prove memorization in production Large Language Models (LLMs), verifying the absolute absence of memorization remains an open challenge. As a workaround, we adapted the methodology from [1] [12], and attempted to detect TPC-H memorization in the models; however, these efforts were unsuccessful.

C.2 Metrics Breakdown for TPC-H

Our TPC-H evaluation averages each metric over 50 iterations at SF 1. Table 6 provides a breakdown of the resulting Text-to-(Big) SQL metrics for each query and model.

Table 6. Per-query metrics in TPC-H (SF1) by model (mean ± std across runs). VES, VES*, and VCES follow the paper definitions; CVQ reports the expected cost per valid query.

QID	Model	Acc.	VES	VES*	VCES	CVQ
1	GPT 5.2	0.60 ± 0.49	0.6507 ± 0.5318	0.2608 ± 0.2134	23.1794 ± 19.0188	0.0188 ± 0.0000
	Opus 4.5	1.00 ± 0.00	0.9567 ± 0.1057	0.2513 ± 0.0138	3.6030 ± 0.2671	0.0697 ± 0.0015
	Gemini 3 Pro (T)	0.00 ± 0.00	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
17	Opus 4.5	0.20 ± 0.40	0.2028 ± 0.4235	0.0323 ± 0.0647	0.3865 ± 0.6072	0.4182 ± 0.0025
	Gemini 3 Pro (T)	0.10 ± 0.30	0.1294 ± 0.3883	0.0140 ± 0.0420	0.3525 ± 0.7373	0.3973 ± 0.0000
	GPT-5.2	0.40 ± 0.49	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0168 ± 0.0004
18	Gemini 3 Pro (T)	1.00 ± 0.00	1.3223 ± 0.1182	0.2449 ± 0.0232	6.4131 ± 0.8326	0.0382 ± 0.0019
	GPT-5.2	0.40 ± 0.49	0.2206 ± 0.6619	0.0711 ± 0.2133	7.5480 ± 17.8560	0.0236 ± 0.0030
	Opus 4.5	0.00 ± 0.00	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
21	Opus 4.5	0.80 ± 0.40	0.7732 ± 0.3889	0.2639 ± 0.1321	2.8673 ± 1.7429	0.1150 ± 0.0003
	GPT-5.2	0.20 ± 0.40	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0915 ± 0.0000
	Gemini 3 Pro (T)	0.00 ± 0.00	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–

We then execute the generated SQL queries across different scale factors and average the proposed metrics for all queries. These results are shown in Table 7.

[12]We make the memorization test code available at https://github.com/GEizaguirre/memorization-LLM-prod

Table 7. TPC-H metrics by query, model, and scale factor (mean ± std across runs). VES, VES*, and VCES follow the paper definitions; CVQ reports the expected cost per valid query.

Query	SF	Model	VES	VES*	VCES	CVQ
Q1	10	Gemini 3 Pro	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		Opus 4.5	0.5668 ± 0.0000	0.2133 ± 0.0064	1.8351 ± 0.0783	0.1163 ± 0.0015
		GPT-5.2	0.3235 ± 0.2641	0.1857 ± 0.1517	3.0724 ± 2.5113	0.1007 ± 0.0000
	100	Gemini 3 Pro	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		Opus 4.5	0.2654 ± 0.0000	0.1494 ± 0.0031	0.8809 ± 0.0259	0.1697 ± 0.0015
		GPT-5.2	0.1599 ± 0.1306	0.1170 ± 0.0955	1.0526 ± 0.8598	0.1852 ± 0.0000
	1000	Gemini 3 Pro	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		Opus 4.5	0.0484 ± 0.0000	0.0424 ± 0.0002	0.0683 ± 0.0006	0.6199 ± 0.0015
		GPT-5.2	0.0289 ± 0.0236	0.0271 ± 0.0221	0.0478 ± 0.0390	0.9439 ± 0.0000
Q17	10	Gemini 3 Pro	0.0519 ± 0.1556	0.0121 ± 0.0362	0.1148 ± 0.3444	0.8777 ± 0.0000
		Opus 4.5	0.1040 ± 0.2079	0.0284 ± 0.0570	0.1814 ± 0.3637	0.6704 ± 0.0026
		GPT-5.2	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.1025 ± 0.0004
	100	Gemini 3 Pro	0.0332 ± 0.0997	0.0107 ± 0.0320	0.0807 ± 0.2422	1.1488 ± 0.0000
		Opus 4.5	0.0669 ± 0.1337	0.0247 ± 0.0494	0.1335 ± 0.2674	0.8111 ± 0.0026
		GPT-5.2	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.1357 ± 0.0004
	1000	Gemini 3 Pro	0.0051 ± 0.0154	0.0039 ± 0.0116	0.0071 ± 0.0213	5.2761 ± 0.0000
		Opus 4.5	0.0094 ± 0.0187	0.0075 ± 0.0151	0.0113 ± 0.0225	3.2374 ± 0.0026
		GPT-5.2	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.7089 ± 0.0004
Q18	10	Gemini 3 Pro	0.5006 ± 0.0000	0.1879 ± 0.0142	1.8557 ± 0.1625	0.1014 ± 0.0019
		Opus 4.5	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		GPT-5.2	0.1017 ± 0.3052	0.0517 ± 0.1551	1.0504 ± 3.1511	0.1172 ± 0.0029
	100	Gemini 3 Pro	0.2563 ± 0.0000	0.1382 ± 0.0080	0.8541 ± 0.0561	0.1619 ± 0.0019
		Opus 4.5	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		GPT-5.2	0.0616 ± 0.1849	0.0389 ± 0.1166	0.5274 ± 1.5823	0.1783 ± 0.0029
	1000	Gemini 3 Pro	0.0481 ± 0.0000	0.0414 ± 0.0008	0.0592 ± 0.0012	0.6997 ± 0.0019
		Opus 4.5	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		GPT-5.2	0.0187 ± 0.0562	0.0159 ± 0.0477	0.0737 ± 0.2211	0.5334 ± 0.0029
Q21	10	Gemini 3 Pro	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		Opus 4.5	0.3922 ± 0.1961	0.1984 ± 0.0992	1.2211 ± 0.6107	0.2233 ± 0.0003
		GPT-5.2	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.4713 ± 0.0000
	100	Gemini 3 Pro	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		Opus 4.5	0.2300 ± 0.1150	0.1462 ± 0.0731	0.6527 ± 0.3264	0.3002 ± 0.0003
		GPT-5.2	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	0.7582 ± 0.0000
	1000	Gemini 3 Pro	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	–
		Opus 4.5	0.0308 ± 0.0154	0.0286 ± 0.0143	0.0241 ± 0.0121	1.5039 ± 0.0003
		GPT-5.2	0.0000 ± 0.0000	0.0000 ± 0.0000	0.0000 ± 0.0000	4.4840 ± 0.0000