Large Databases Need Small, Open-Weight Language Models

Parker Glenn
Capital One
parker.glenn@capitalone.com

Alfy Samuel
Capital One
alfy.samuel@capitalone.com

ABSTRACT

Language model systems built around proprietary APIs often operate on a token-based cost model. This becomes prohibitively expensive in the context of large databases, where LM-enhanced relational operators can incur costs exceeding $10,000 for a single set of experiments, hindering thorough research and practical deployment. In this paper, we demonstrate that quantized, open-weight models running locally on just 16GB of VRAM can match or exceed the accuracy of closed-source counterparts at lower latency and a fraction of the price, challenging the prevailing assumption that closed-source LM APIs are necessary for effective LM-database integration. We present and analyze the key system optimizations required to efficiently deploy these open-weight models within an LM-DB system. By integrating these local models into the BlendSQL v0.1.0 framework, we demonstrate a 390x reduction in overall costs and 3.8x reduction in latency compared to a proprietary LM API. We make our code available at https://github.com/CapitalOne-Research/play-by-the-type-rules.

VLDB Workshop Reference Format:
Parker Glenn and Alfy Samuel. Large Databases Need Small, Open-Weight Language Models. VLDB 2026 Workshop: Applied AI for Database Systems and Applications (AIDB 2026).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CapitalOne-Research/play-by-the-type-rules.

1 INTRODUCTION

Relational databases remain the ubiquitous choice for storing structured information. Combined with the increasing popularity of language models, a research question has emerged: what is the optimal method for combining the flexible reasoning capabilities of language models with the deterministic and reliable processing of traditional structured query languages? This notion of an *optimal* approach can be decomposed into three key dimensions: cost, latency, and quality. To illustrate, consider an auto repair shop with a database of customer complaints containing columns `summary` and `date`. A user might seek all complaints describing an engine-related issue submitted before 2024. Determining whether a complaint is engine-related requires interpreting free-text summaries, which is out-of-scope for native SQL operators. A recent trend is to accomplish this via a call to a language model (LM): `SELECT summary FROM complaints WHERE LM('Is this engine-related?', summary) = TRUE AND date < '2024-01-01'`. Queries containing LM functions may be made more complex by referencing multiple tables via `JOIN` clauses, injecting additional native-SQL conditions, or adding additional LM functions within a single expression context, all of which may change the terms of what makes a query plan *optimal*. Importantly, the introduction of non-deterministic LM functions inherently changes the optimization landscape for database systems.

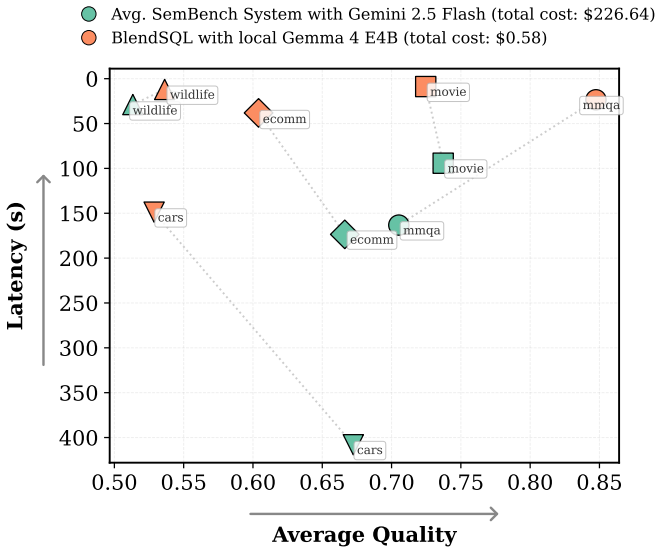

Figure 1: Plotting the quality, latency, and cost over the five scenarios in SemBench. Latency and quality are averaged across five total runs, and cost is computed as the cumulative total of all runs.

A growing body of research shows that the quality of the software infrastructure wrapping language model calls can be as predictive of downstream task performance as the choice of base LM. In agentic coding flows, harness-level changes such as self-verification loops, context management, and loop detection, have produced large benchmark improvements without any change to the underlying model [30, 31, 60]. In tool calling, structured generation systems have been shown to elevate open-source models to performance meeting or exceeding closed-source counterparts [14, 59]. These gains are driven not by innovations in pre/post-training or parameter scaling, but by embedding task-specific inductive bias into the systems that orchestrate inference.

We argue that, in spite of recent progress made in hybrid LM-DB systems, there are still large cost, quality, and latency gains to

Proceedings of the VLDB Endowment. ISSN 2150-8097.

be made with small, open-weight models by designing effective software infrastructure for hybrid LM-DB systems. To demonstrate this, we sample a recent body of work applying language models to relational databases via a UDF-style pattern. Of 21 total evaluated language models across 8 published works, only 6 are open-weight models [8, 25, 29, 32, 36, 50, 51, 64]. Of the 6 evaluated open-weight models across published works, none are below 20B parameters. While the cost of these proprietary, API-based models is often undisclosed, Lao et al. [29] report total costs of $10,167.80 in their published experiments using Gemini 2.5 Flash. Importantly, this cost is a subset of total experimental cost, as it does not include preliminary experiments leading up to final results. We show that not only are these costly proprietary models slow and unnecessary, but, within certain domains, they are strictly worse than a properly harnessed local open-weight model.

Our contributions are the following:

- We present BlendSQL v0.1.0, a LM-DB system that combines query-level optimizations with constrained decoding to enable small, quantized open-weight models running on a single 16GB GPU to achieve a win-or-tie rate of 57% against closed-source alternatives at 390× lower cost and 3.8× lower latency on the SemBench benchmark.
- We demonstrate that BlendSQL circumvents the linear memory scaling deficiencies of existing baseline systems at large database scales, maintaining a near-constant memory footprint.
- We identify a modality gap in current small open-weight models: while competitive with closed counterparts on text, they fall behind on image and audio, suggesting that multimodal understanding remains a bottleneck for cost-effective local deployment.
- We argue for a future of LM-DB research built on open-weight models, ensuring accessibility and full reproducibility of experimental results.

2 SYSTEM OVERVIEW

2.1 Terminology

Several terms have recently been introduced to describe systems that combine traditional relational operators with AI-powered operators. Lao et al. [29] introduce *semantic query processing engines (SQPEs)* to describe systems that extend relational algebra with natural language *semantic operators*. Similarly, Su et al. [51] use *LLM-Enhanced Relational Operators (LROs)* to denote components that enhance relational processing by invoking a language model. More broadly, Yan et al. [63] categorize this intersection of language models and databases as *DBMS–LLM systems*, a space further refined by Zhou et al. [65], who formally distinguish the *UDF (user-defined function)-centric* integration pattern presently relevant to our work.

We adopt the umbrella term *LM-DB systems* to refer to systems that integrate language model functions with existing structured data manipulation interfaces such as Pandas and SQL. We use the broader term "language model" (LM) to avoid restricting our scope to models of any particular scale. Ultimately, each system we study is a UDF-centric system, and can be decomposed into two components: (1) a *query optimizer*, which routes subsets of data to LM functions, and (2) the *LM function implementation* itself, encompassing the prompts, generation parameters, and post-processing logic used at inference time. We describe each component of our proposed system below.

2.2 BlendSQL v0.1.0 Implementation

BlendSQL is a query language that compiles to SQL [16]. It allows for combining traditional SQL operators with generalizable language model (LM) functions capable of unstructured reasoning. Each BlendSQL LM function is denoted by double curly braces, "`{{`" and "`}}`". Using a pre-determined prompt template, a response is generated from a local or remote language model with optional type-constraints to yield a function output. This function output is then integrated into the wider SQL query using the `sqlglot` parser [37]. Once all LM functions have run, a final compiled SQL query is executed against the native DBMS to yield the final result. Currently, SQLite, DuckDB, and PostgreSQL backends are supported. In this work, we introduce BlendSQL v0.1.0. Since the original v0.0.0 release in Glenn et al. [16], the following changes have been made:

- An improved type inference system that constrains LM generations to valid datatypes.
- Support for image and audio datatypes as arguments to LM functions.
- Query optimizations including cascade filtering and early exiting.
- Faster aggregation of in-memory LM function outputs via `polars` [45].

2.3 LM Functions

Unlike other LM-DB systems that rely on numerous operators coupled with specific prompt templates (e.g., `sem_map`, `sem_filter`, `sem_extract`, `sem_topk`), BlendSQL implements a small number of general-purpose polymorphic functions defined by their input/output cardinality. In this work, we focus on two core polymorphic functions and demonstrate how they enable complex reasoning patterns such as ranking, RAG, and entity linking. For comprehensive descriptions and function signatures for all available functions, refer to the actively maintained online documentation[1].

Constrained Decoding All BlendSQL v0.1.0 LM functions accept a `return_type` argument, which is either specified explicitly or inferred implicitly from query syntax [17]. This `return_type` is then used to perform constrained decoding, guaranteeing adherence to datatype structures defined via a context-free grammar. When provided, the `options` argument behaves similarly - constrained decoding will be used to restrict LM generations to a value in the option set. If the `return_type` is a collection, the `quantifier` argument can be used to apply a regular expression-style modifier to restrict the number of generated items (e.g. `{3}` for "exactly three", `{1,5}` for "between one and five").

One-Shot Prompting All BlendSQL v0.1.0 LM functions employ type-aligned one-shot prompting. From a pre-defined pool, the system selects a single example that matches the current function's expected return type.

[1] https://parkervg.github.io/blendsql/reference/functions/

2.3.1 LLMQA

The LLMQA function is an aggregate function which transforms a subset of data into a single-cell output. This output can be a single scalar type (`return_type='str'`), or a collection (`return_type='List[str]'`).

Example 1: LLMQA for top-k ranking

```
SELECT * FROM (
    VALUES {{
        LLMQA(
            'Select the 3 most
            negative reviews.',
            options=(
                SELECT reviewText
                FROM Reviews
                WHERE reviewYear = 2025
            ),
            return_type='List[str]',
            quantifier='{3}'
        )
    }}
) AS rankedReviews(review1, review2, review3)
```

Example 2: LLMQA for unstructured-structured linking

```
SELECT name FROM state_flowers
WHERE state = {{
    LLMQA(
        'Which state is known as
        ''The Golden State''?',
        context=(SELECT * FROM documents)
    )
}}
```

2.3.2 LLMMAP

The LLMMAP function is a row-wise function that takes a natural language question and one or more column names. For each row, the corresponding values $v_1, \ldots, v_k$ are inserted into the prompt alongside the question Q and passed to the underlying LM, which returns the result of $f(Q, v_1, \ldots, v_k)$. As with LLMQA, f returns either a single scalar type (`return_type='str'`) or a collection (`return_type='List[str]'`).

Refined LLMMAP Prompting Pattern Previously in BlendSQL <v0.1.0, the LLMMAP function would run with a default `batch_size` argument of 5, which determined the number of input values to pass in a single prompt. The language model was then expected to generate the mapped outputs for each value, separated by a semicolon. For example, given the input:

```
What is the capital of the country?
France
Croatia
Canada
Australia
Ukraine
```

the LM would be expected to generate the string `Paris;Zagreb;Ottawa;Canberra;Kyiv`. While this approach greatly decreased token usage, it also led to sub-optimal accuracy in many studies [32, 51]. We adopt the more traditional single-in, single-out prompting format (i.e. `batch_size=1`) as the default for BlendSQL v0.1.0.

Example 3: LLMMAP for filtering

```
SELECT * FROM Reviews
WHERE {{
LLMMap(
    'Does this review have a positive
    sentiment?',
    reviewText
)
}} = TRUE
```

Example 4: LLMMAP for classification and grouping

```
SELECT GROUP_CONCAT(Name, ', ') AS 'Names',
{{
    LLMMap(
        'In which century was
        this person born?',
        p.Name,
        options=('1800s', '1900s', '2000s')
    )
}} AS Born
FROM People
GROUP BY Born
```

2.4 Query Optimization

Optimizing user-defined functions (UDFs) is a notoriously difficult task. A large body of research has explored this in the general setting [3, 12, 13, 61]: given some user-provided function of unknown computational cost, what is the optimal method of execution within a wider SQL query? Notably, when embedding language models into SQL, the problem space is different in an important way: Given prior knowledge that a function f_{LM} calls an LM while other functions f do not, the cost calculation $C(f(D)) \ll C(f_{LM}(D))$ typically holds for any data subset D. Then, given LLMQA calls translate to exactly one LM generation call and LLMMAP potentially > 1, we derive the following cost model by abstracting away D:

$$C(f) \ll C(\text{LLMQA}) < C(\text{LLMMAP}) \tag{1}$$

We implement a rule-based optimizer built on this heuristic cost model. When executing a program, the query is parsed into an abstract syntax tree (AST) using `sqlglot` [37]. Each `SELECT` expression is iteratively processed via depth-first search, and the following strategy is applied:

(1) **Pre-filtering**: Blocking operators are removed (`ORDER BY`, `GROUP BY`, etc.) and LM function nodes are replaced with the constant `TRUE`, allowing the underlying DBMS to execute all vanilla SQL predicates first. The `SELECT` argument is modified to only select those columns used by LM functions, minimizing I/O overhead.
(2) **Materialization**: The filtered query is executed against the DBMS and written to a temporary table, effectively pushing down all non-LM operations.
(3) **LM Function Execution**: Following the standard SQL order of operations (`FROM`/`JOIN`→`WHERE`→`GROUP BY`, etc.), the LM functions are executed, reading from the filtered temporary tables to fetch inputs.

The `TRUE` substitution in Step 1 ensures the optimizer honors disjunctive conditions between LM functions and vanilla SQL functions (`WHERE` $f_{LM}()$ `OR f()` becomes `WHERE TRUE OR f()`, preventing an overly-eager pre-filter via `f()`).

Cascade Filtering BlendSQL implements cascade filtering for conjunctive predicates involving LM functions. The LM predicates are sequenced into a pipeline of operators, and subsequent functions only process rows of data which satisfied all preceding predicates. This avoids redundant, expensive calls for rows already disqualified by a prior LM-based filter. This logic applies when two or more LM functions appear within a `WHERE` clause, provided that no disjunctions (`OR` relationships) exist between them. Currently, our optimizer prioritizes LLMQA before LLMMAP based on the cost model in Eq. 1. Future work may refine the ordering of multiple LLMMAP calls by estimating and prioritizing the most selective predicates first.

Early Exiting In expressions where a `LIMIT` keyword is used, LLMMAP execution may terminate as soon as the required number of rows is satisfied. This optimization is only applied when the expression contains no blocking operators (e.g. `ORDER BY`, `GROUP BY`, etc.) that would require a full scan of the LM function's output. Building on the `AsyncOpenAI` API, we utilize streaming to ensure no unnecessary tokens are generated: as soon as the exit condition is met, a cancel event is issued and all outstanding asynchronous generation requests are terminated.

Early Deduplication of Database Values Before passing a sequence of values to an LLMMAP function, the input set is deduplicated via a `SELECT DISTINCT` clause, and the mapped outputs are then aligned to the original table via a `LEFT JOIN` operation. Consider the query `SELECT name FROM People WHERE {{LLMMap('Are they an NBA player?', name)}} = TRUE` over a table with $N = 10^6$ rows but only $|\text{dom}(name)| \ll 10^6$ distinct name values. Without deduplication, each row would result in a separate LM invocation; with early deduplication, the number of generation calls is reduced to $|\text{dom}(name)|$, and results are broadcast back to all matching rows via the join. By paying a relatively small upfront cost of deduplicating and re-joining to the source table, this optimization yields a reduction in LM calls proportional to the duplication factor $\rho = N/|\text{dom}(name)|$. Given the aforementioned cost heuristic $C(f(D)) \ll C(f_{LM}(D))$, this translates to substantially lower latency and cost. This optimization applies for both single-column and multi-column inputs to the LLMMAP function.

3 EXPERIMENTS

3.1 SemBench

SemBench is a benchmark designed to evaluate the performance of hybrid LM-DB systems. The benchmark can be viewed as an extension of TPC-H [56] and TPC-DS [55], with a specific emphasis on the cost, quality, and latency of queries containing LM functions. It consists of five scenarios, requiring LM functions to operate over a mix of text, image, and audio data. In the example below from the ECOMM scenario, a system must orchestrate calls to a multi-modal language model to determine the color of products, given local image files.

```
SELECT
i.id as id,
{{
  LLMMap(
    'What is the primary color of the product
    in this image?',
    i.local_image_path
  )
}} AS category
FROM styles_details s
JOIN image_mapping i on s.id = i.id
WHERE s.baseColour IN
('Black', 'Blue', 'Red', 'White',
'Orange', 'Green')
```

3.2 Setup

Model Inference All experiments with open-weight models are run locally with an NVIDIA RTX 5080 GPU with 16GB VRAM, 64GB of RAM, and an AMD Ryzen 5 7600X processor. We experiment with the Gemma 3 [53] and Gemma 4 [52] families of models for their strong performance at sizes that fit within 16GB VRAM. For brevity, we use the shorthands in Table 1 to refer to the quantized HuggingFace checkpoints used in our experiments. We use

Table 1: Model shorthands and HuggingFace identifiers.

Shorthand	HuggingFace Model
Gemma 4 E2B	prithivMLmods/gemma-4-E2B-it-FP8
Gemma 4 E4B	prithivMLmods/gemma-4-E4B-it-FP8
Gemma 3 4B	RedHatAI/gemma-3-4b-it-quantized.w4a16
Gemma 3 12B	RedHatAI/gemma-3-12b-it-quantized.w4a16

`vLLM==0.21.0` for model inference [27]. The `llguidance` engine is used for all constrained decoding capabilities [20]. All experiments are run five times and their averages reported.

System Versions We use `blendsql==0.1.26`, `lotus-ai==1.1.4`, `palimpzest==0.8.2`, and `thalamusdb==0.1.15` in our experiments.

Cost Calculations While on-demand GPU pricing is highly volatile, hourly rental costs for an NVIDIA RTX 5080 range from $0.14 to $0.18 as of April 2026 [49, 58]. We use the upper-end price of $0.18 per hour for cost reporting of local models. For Gemini 2.5 Flash, we follow the cost reporting of Lao et al. [29]: $0.30 per million input text, image, and video tokens, $1.00 per million input audio tokens, and $2.50 per million output tokens.

Dataset We evaluate systems using the five scenarios of SemBench. Tables for each scenario are stored in a local DuckDB database [47], which serves as the backend for both ground-truth definitions and system predictions. We use scale factors identical to those of Lao et al. [29] to generate data for each scenario.

Scenario	Scale Factor Used	Modalities
Movie	2,000	Text
Wildlife	200	Text, Image, Audio
E-Commerce	500	Text, Image
Cars	19,672	Text, Image, Audio
MMQA	200	Text, Image

Table 2: Overview of SemBench scenarios.

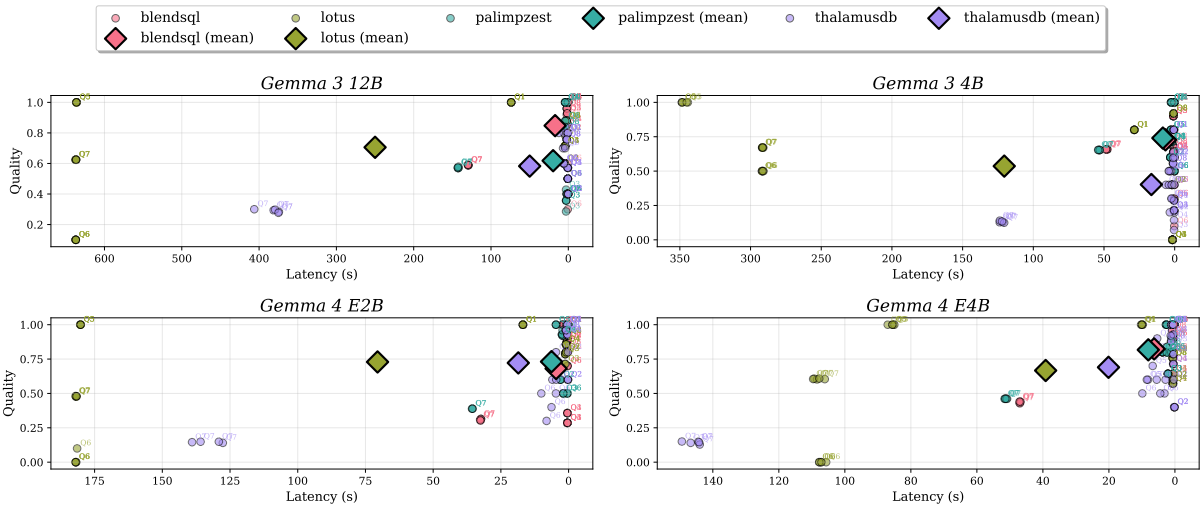

Figure 2: Benchmarking the quality and latency of various LM-DB systems with open-weight LMs on the text-only MOVIE scenario from SemBench. All systems are run with a max concurrency of 32.

Metrics We adopt the evaluation methodology of Lao et al. [29] and report F1 score for retrieval queries, Spearman's rank correlation for ranking queries, and absolute error for aggregation queries. All metrics are normalized to the range [0, 1] and reported as "quality".

3.3 Baseline Systems

We describe the LM-DB systems evaluated in Lao et al. [29] below.

LOTUS LOTUS [44] extends the pandas API [41] with additional LM-based functions. As a result, it adopts the eager execution of pandas, and therefore executes operators in the order specified by the user, without reordering. Some functions, such as the `sem_filter` function, can be equipped with a smaller proxy model to apply a semantic cascade filter, where a decision threshold is learned at inference time and optionally used to bypass calls to the larger "main" LM. Other local optimizations are implemented within the specific LM functions, described in Patel et al. [44]. All global logical plan optimizations (such as predicate push-down, join re-ordering, early exiting, etc.) are up to the user to implement in Python code. Given our memory-constrained local setup, we evaluate LOTUS identically to other systems, restricting each system to a single "main" LM.

Palimpzest PALIMPZEST [34] is a declarative system for optimizing AI workloads over hybrid structured/unstructured data. It exposes a Python API based on lazy evaluation: users define a sequence of mixed LM/non-LM operators, which is optimized and executed upon a final `.run()` invocation. Though we do not evaluate it in this study, recent work such as ABACUS [48], builds on this declarative system, exploring optimization objectives when given access to LM APIs with different capability/cost trade-offs.

ThalamusDB ThalamusDB [24] is a deterministic approximate query processing system. It implements traditional relational optimizations such as single-table predicate pushdown to select table subsets in order to minimize approximation error. Additionally, it is capable of early-exiting LM functions when a `LIMIT` condition is satisfied. Its optimizer is unique in its multi-objective optimization over approximation error, resource usage (execution time, LLM calls, etc.), and number of labeled examples[2].

BigQuery Released in November 2025, Google BigQuery AI functions [18] enable integration of LM calls into standard SQL query flows. The implementation and hyperparameters of LM inference (max concurrency, temperature, etc.) are hidden from users. Because the implementation is proprietary, we treat it as a black-box baseline.

4 RESULTS

4.1 Comparison to Other LM-DB Systems

System	EE	CF	CD	ED	SemBench Coverage
THALAMUSDB	✓	×	×	×	65.45%
LOTUS	×	×	×	×	78.18%
BIGQUERY	?	?	?	?	94.55%
PALIMPZEST	✓	✓	×	×	98.18%
BLENDSQL	✓	✓	✓	✓	100%

Table 3: System comparison on SemBench coverage and supported features. Features evaluated include early exiting (EE), cascade filtering (CF), constrained decoding (CD), and early deduplication (ED) across 55 questions.

Query Coverage As shown in Table 3, the design of LM-DB systems may restrict their expressivity, inhibiting full coverage across the diverse SemBench dataset. For example, THALAMUSDB does not support LM aggregation operations, PALIMPZEST does not

[2]We do not explore the labeled examples component of ThalamusDB in the present study.

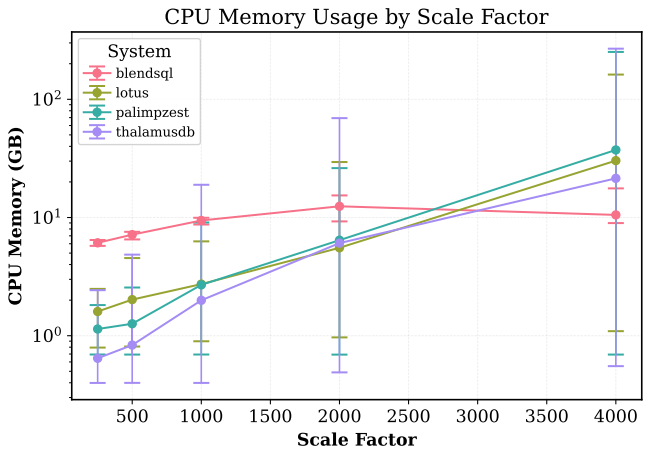

Figure 3: Plotting the average, minimum, and maximum CPU memory usage across different scale factors in the ECOMM scenario. BLENDSQL memory accounts for a local vLLM server running Gemma 4 E4B in addition to normal LM-DB system memory; all other systems use a remote Gemini 2.5 Flash endpoint and only account for memory usage induced by LM-DB system processing.

support top-k reranking, and LOTUS does not support audio inputs to LM functions. By using the two expressive polymorphic LM functions described in Section 2.3, BLENDSQL is the only system to display 100% coverage over all 55 SemBench queries.

Benchmarking on the MOVIE Scenario We benchmark the performance of BLENDSQL against other open-source LM-DB systems in a 16GB VRAM setting. For all systems, we provide access to exactly one open-weight language model. While PALIMPZEST and LOTUS offer additional optimizations for multi-model settings, this would exceed our memory budget and is thus outside the scope of the current study. Both systems support the widest coverage of SemBench scenarios; however, we encountered unresolved compatibility issues when running either system on non-text inputs with a local vLLM backend, limiting our cross-system comparison to the text-only MOVIE scenario.

Figure 2 shows results on the text-only MOVIE scenario of SemBench. Since many queries in this scenario involve `LIMIT` clauses (e.g., "Find five clearly positive reviews"), the ability to early exit LM functions once a condition is satisfied is critical for achieving reasonable latency. The performance of LOTUS illustrates the cost of eager execution in a hybrid space of relational operators and LM functions: since the stateless LM functions have no knowledge of a future `LIMIT` operation, the entire subset of data must be passed to the LM function prior to executing `.head(n)`, leading to significantly higher latency. Despite this high latency, LOTUS yields slightly higher average quality than ThalamusDB across the four models.

Across all models, BLENDSQL exhibits the lowest latency. It matches the quality of PALIMPZEST across Gemma 3 4B and Gemma 4 E4B, and achieves the highest quality of all systems with Gemma 3 12B. Importantly, small variations in reported quality can largely be attributed to minor prompting variations and the nondeterminism of asynchronous generation requests: as all systems except for LOTUS implement early exiting, final results are a product of which generation requests finish first to satisfy the exit condition.

Scalability to Large Databases Following Lao et al. [29], we perform a study on the scalability of BLENDSQL across various database scales. We plot the CPU memory usage across ECOMM scale factors of 250, 500, 1,000, 2,000, and 4,000 for the three baseline open-source LM-DB systems and BLENDSQL in Figure 3. While the three baseline systems' memory usage scales linearly with database size, BLENDSQL's usage is relatively constant, impacted only by variations in vLLM server usage. This is due to the design of the baseline systems' semantic join function: in queries such as Q8 and Q9, image pairs are constructed and evaluated against a language model without releasing the memory used to store the image pairs. As a result, each system can potentially load 25,000 .jpg images into memory at once under a scale factor of 4,000. The three open-source systems THALAMUSDB, PALIMPZEST, and LOTUS each reach a peak usage of 268GB, 252GB, and 161GB respectively under this scale factor of 4,000. Comparatively, BLENDSQL uses a maximum of only 18GB at this scale factor to both serve the local language model with vLLM and execute the underlying multi-modal LM-DB system. This memory usage is primarily attributable to components such as the vLLM KV cache and scheduler management under high concurrency workloads: for reference, starting an idle Gemma 4 E4B vLLM server alone consumes 5GB of CPU memory.

This relatively constant memory usage is made possible by the mechanism BLENDSQL uses to achieve joins via the asynchronous LLMMAP function. Given the abbreviated ECOMM Q8 program below, there are at most `max_concurrency` × 2 images loaded into memory at a given time.

```
WITH image_pairs AS (
    SELECT
    s1.id AS id1,
    img1.local_image_path AS image1,
    s2.id AS id2,
    img2.local_image_path AS image2
    FROM styles_details s1
    JOIN image_mapping img1 ON s1.id = img1.id
    JOIN styles_details s2 ON s1.id < s2.id
    JOIN image_mapping img2 ON s2.id = img2.id
)
SELECT id1 || '-' || id2 AS id
FROM image_pairs
WHERE {{
    LLMMap(
        'Do both images display objects of
        the same category?',
        image1,
        image2
    )
}}
```

Running all 14 ECOMM queries five times under the five scale factors with Gemma 4 E4B and BLENDSQL took 38.6 hours and cost $6.96. A direct comparison to other LM-DB systems using Gemini 2.5 Flash is difficult, since as reported in Lao et al. [29], no open-source system successfully executed all 70 queries. The closest system, PALIMPZEST, completed a single run of 65 queries in

Table 4: Latency, quality, and cost across various SemBench scenarios. All queries are clickable hyperlinks directing to the corresponding BlendSQL query. Green highlights indicate a better score compared to the other system. Yellow highlights indicate a tie. Quality differences within the range ± 0.02 are considered insignificant and labeled as ties with a yellow highlight. The BlendSQL setting is run with a max concurrency of 64.

(a) MMQA results.

	BlendSQL with gemma-4-E4B-it-FP8			Avg. SemBench System with gemini-2.5-flash		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**0.07**	**1.00**	**$2e-5**	8.67	**1.00**	$0.06
Q2a	**55.15**	**1.00**	**$0.01**	119.77	0.61	$3.35
Q2b	**64.22**	**0.97**	**$0.02**	113.00	0.61	$3.42
Q3a	**1.35**	0.78	**$3e-4**	9.82	**0.82**	$1.34
Q3f	**1.41**	**0.92**	**$4e-4**	10.38	**0.92**	$0.06
Q4	**0.45**	**0.56**	**$1e-4**	1.20	**0.54**	$0.03
Q5	**0.24**	**1.00**	**$6e-5**	0.49	**1.00**	$5e-3
Q6a	**0.73**	**1.00**	**$2e-4**	8.22	0.59	$0.05
Q6b	**0.71**	0.67	**$2e-4**	7.67	**0.76**	$0.05
Q6c	**0.83**	**0.91**	**$2e-4**	10.02	0.67	$0.05
Q7	**133.26**	**0.51**	**$0.03**	1501.57	0.21	$50.73
Avg.	**23.49**	**0.85**	**$6e-3**	163.19	0.71	$5.37
Avg. Std.	0.11	0.03	$3e-5	-	-	-
Wins	11	5	11	0	2	0
Total Cost			**$0.06**			$59.12

(b) ECOMM results.

	BlendSQL with gemma-4-E4B-it-FP8			Avg. SemBench System with gemini-2.5-flash		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**8.18**	**1.00**	**$2e-3**	20.10	0.90	$0.26
Q2	**23.15**	0.40	**$6e-3**	95.03	**0.65**	$6.04
Q3	**8.53**	0.78	**$2e-3**	18.27	**0.97**	$0.52
Q4	**13.65**	0.40	**$3e-3**	174.23	**0.56**	$1.25
Q5	**4.48**	0.96	**$1e-3**	13.23	**0.98**	$0.47
Q6	**13.34**	0.67	**$3e-3**	97.80	**0.89**	$1.50
Q7	222.74	**0.77**	**$0.06**	**157.53**	0.75	$5.08
Q8	**51.63**	0.50	**$0.01**	211.80	**0.57**	$23.18
Q9	**24.43**	**0.43**	**$6e-3**	302.27	0.41	$0.85
Q10	**10.95**	0.00	**$3e-3**	856.00	**0.03**	$2.70
Q11	**57.46**	0.25	**$0.01**	145.55	**0.76**	$2.20
Q12	**13.58**	**0.91**	**$3e-3**	33.13	0.52	$0.57
Q13	**32.99**	0.51	**$8e-3**	178.50	**0.73**	$1.77
Q14	**48.73**	**0.88**	**$0.01**	125.90	0.62	$11.23
Avg.	**38.13**	0.60	**$1e-2**	173.52	**0.67**	$4.11
Avg. Std.	0.15	0.02	$4e-5	-	-	-
Wins	13	5	14	1	9	0
Total Cost			**$0.13**			$57.60

(c) MOVIE results.

	BlendSQL with gemma-4-E4B-it-FP8			Avg. SemBench System with gemini-2.5-flash		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**0.39**	**1.00**	**$1e-4**	16.85	**0.99**	$0.18
Q2	**0.45**	0.96	**$1e-4**	10.80	**0.98**	$0.03
Q3	**0.48**	**0.69**	**$1e-4**	5.20	**0.67**	$0.03
Q4	**0.32**	0.64	**$8e-5**	5.60	**0.69**	$0.04
Q5	**0.63**	0.46	**$2e-4**	148.80	**0.72**	$4.25
Q6	**0.57**	0.46	**$1e-4**	122.72	**0.76**	$3.53
Q7	**149.03**	**0.54**	**$0.04**	321.00	**0.54**	$16.24
Q8	**0.68**	**0.92**	**$2e-4**	6.08	0.84	$0.04
Q9	**0.81**	**0.87**	**$2e-4**	7.97	0.77	$0.15
Q10	**8.88**	**0.71**	**$2e-3**	34.07	0.42	$1.07
Avg.	**16.22**	0.72	**$4e-3**	67.91	**0.74**	$2.55
Avg. Std.	0.22	0.03	$6e-5	-	-	
Wins	10	3	10	0	4	0
Total Cost			**$0.04**			$25.54

(d) CARS results.

	BlendSQL with gemma-4-E4B-it-FP8			Avg. SemBench System with gemini-2.5-flash		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**191.33**	0.64	**$0.05**	476.75	**0.78**	$8.74
Q2	**1.81**	0.00	**$5e-4**	10.77	**0.06**	$0.04
Q3	**1.53**	**1.00**	**$4e-4**	166.23	0.94	$3.78
Q4	**182.48**	**0.99**	**$0.05**	525.77	**0.99**	$8.59
Q5	**2.16**	0.00	**$5e-4**	182.97	**1.00**	$5.15
Q6	**215.09**	0.88	**$0.05**	415.67	**0.96**	$10.78
Q7	**364.85**	**0.54**	**$0.09**	1038.30	**0.53**	$17.83
Q8	**27.79**	0.21	**$7e-3**	431.18	**0.29**	$7.12
Q10	**351.99**	**0.50**	**$0.09**	425.00	**0.50**	$17.47
Avg.	**148.78**	0.53	**$0.04**	408.07	**0.67**	$8.83
Avg. Std.	0.54	0.00	$1e-4	-	-	
Wins	9	1	9	0	5	0
Total Cost			**$0.33**			$79.51

(e) WILDLIFE results.

	BlendSQL with gemma-4-E4B-it-FP8			Avg. SemBench System with gemini-2.5-flash		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**9.41**	**0.78**	**$2e-3**	44.20	**0.79**	$0.57
Q2	**4.16**	0.00	**$1e-3**	5.57	**0.17**	$0.05
Q3	**9.29**	0.20	**$2e-3**	38.02	**0.25**	$0.47
Q4	**4.01**	**0.40**	**$1e-3**	4.47	0.33	$0.04
Q5	13.39	**0.93**	**$3e-3**	**11.67**	0.75	$0.43
Q6	**13.34**	0.20	**$3e-3**	18.93	**0.23**	$0.52
Q7	**18.58**	**1.00**	**$5e-3**	71.40	**1.00**	$0.98
Q8	**26.68**	**0.83**	**$7e-3**	31.40	0.75	$0.80
Q9	**11.92**	**0.61**	**$3e-3**	24.63	**0.61**	$0.55
Q10	**9.27**	**0.40**	**$2e-3**	38.02	0.25	$0.47
Avg.	**12.01**	**0.54**	**$3e-3**	28.83	0.51	$0.49
Avg. Std.	0.08	0.22	$2e-5	-	-	-
Wins	9	4	10	1	3	0
Total Cost			**$0.03**			$4.88

18.6 hours and $597.67, translating to a five-run reproduction cost of 92.8 hours and $2,988.32.

4.2 Cost, Latency, and Quality of Open-Weights Models

Table 4 contains the cost, latency, and quality results for the evaluated SemBench scenarios across two settings: BlendSQL with Gemma 4 E4B, and the average SemBench system with Gemini 2.5 Flash. We define the "average" system as the query-level mean of the four baseline systems described in Section 3.3; full system-level results can be found in Appendix 7. Visualized in Figure 1, the properly harnessed small, open-weight model is competitive with the closed Gemini 2.5 Flash at a fraction of the cost: the BlendSQL setting demonstrates a query-level win-or-tie rate of 57% across all scenarios and decreases experiment costs from $226.64 to just $0.58. Notably, the small-model BlendSQL setting outperforms the average SemBench system in average quality on the MMQA scenario

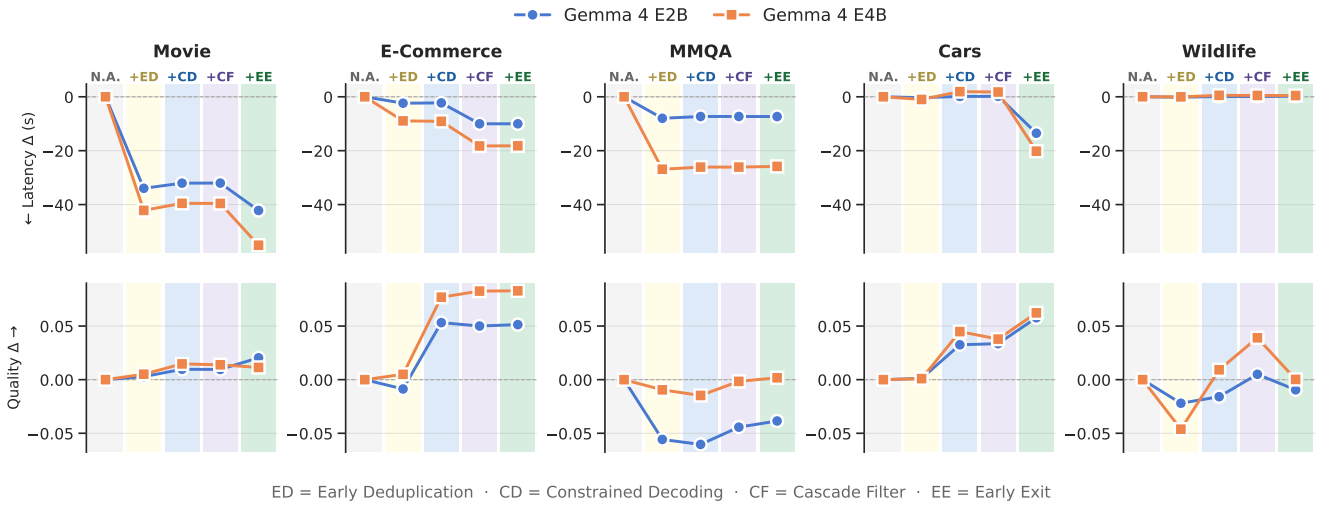

Figure 4: Measuring the relative quality and latency change when incrementally adding query processing features to BlendSQL. Results are averaged across 5 runs and use a max concurrency of 64. "N.A." refers to v0.0.0 presented in Glenn et al. [16]; all other features are new additions in v0.1.0.

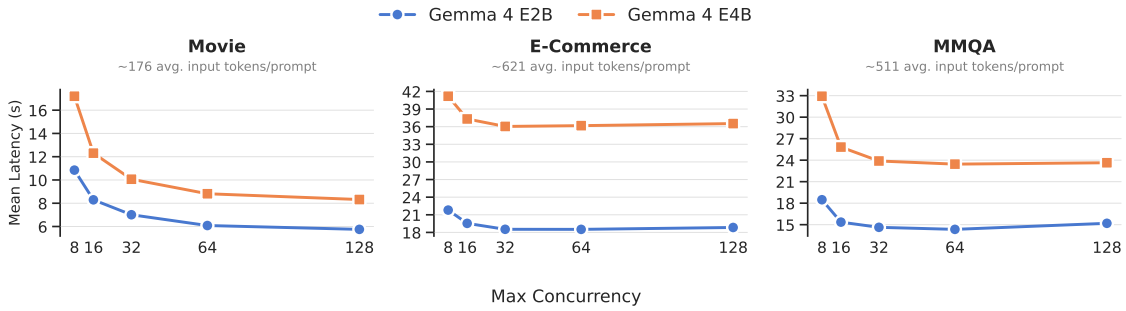

Figure 5: Evaluating the impact of concurrency limits with LM inference requests on a single 16GB RTX 5080. This controls the maximum number of concurrent async requests sent to vLLM at once, which in turn make use of continuous batching to keep GPU usage high.

(0.85 vs. 0.70) and the WILDLIFE scenario (0.54 vs. 0.51). The MOVIE scenario shows the BlendSQL system nearly matching the average SemBench system (0.72 vs. 0.74) while decreasing cost from $25.54 to $0.04.

We see a quality gap emerge in the audio-heavy CARS scenario, where average quality of the BlendSQL setting is 0.53, falling behind the average SemBench system's 0.67. As shown in Figure 6, this gap is attributable to the modality of the data being passed to the LM functions: while the small, properly harnessed Gemma E4B model outperforms the closed Gemini 2.5 Flash across text data, it falls behind on audio and image data. In Q5 of CARS, for instance, BlendSQL with Gemma 4 E4B achieves a quality of 0.0: the LLMMAP function tasked with identifying recordings of damaged cars returns `FALSE` for every row, disqualifying all candidates and producing an empty result. This pattern recurs in Q2, where the model fails to identify any audio file as resembling a car with a dead battery.

Impact of Concurrency Limits Local deployment eliminates provider-enforced rate limits, allowing request parallelism bounded only by available compute. Figure 5 displays the relationship between concurrency limits and system latency across the scenarios of SemBench.[3] The text-only MOVIE scenario sees the greatest speedup with increased concurrency limits, where the Gemma 4 E4B model displays a latency drop from 17.2s to 8.3s when scaling concurrency from 8 to 128. As each LLMMAP prompt is relatively small (176 tokens on average), even just 16GB VRAM allows for the headroom required to speed up inference at large concurrency settings. By contrast, we see that the ECOMM scenario's latency flattens at a concurrency of 32 due to the long-context prompts formed via the lengthy product descriptions (621 tokens on average).

[3] As `vLLM` 0.21.0 does not currently support continuous batching for Gemma 4 models with audio data (https://github.com/vllm-project/vllm/pull/39459), we refrain from evaluating the two scenarios with audio data.

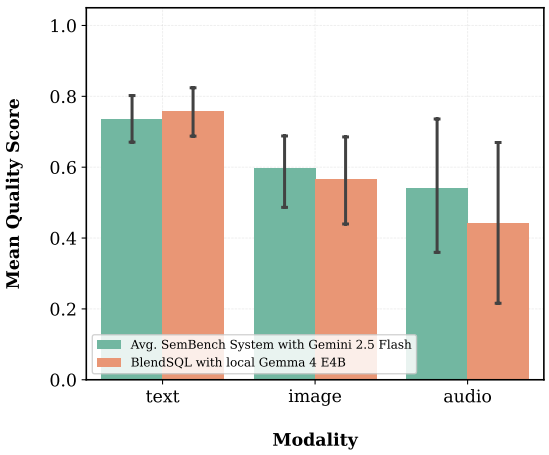

Figure 6: While a properly harnessed quantized Gemma 4 E4B slightly outperforms the closed Gemini 2.5 Flash on queries involving text, the quality gap widens on image and audio inputs, where the open-weight model exhibits a gap of 0.03 for images and 0.10 for audio.

Feature Ablations We plot the relative impact of the query optimizations described in Section 2.4 in Figure 4[4]. The `LIMIT`-heavy MOVIE scenario sees a large improvement in latency when enabling early exiting, where a Gemma E4B model drops from 24.3s to 8.8s average latency. The largest change in latency is seen in the CARS scenario, driven by two questions: In Q3, instead of performing inference over all 1,270 rows of data where `transmission = 'Manual'`, early exiting allows the process to complete once 10 outputs of the LM function satisfy the `= FALSE` predicate, resulting in latency savings of 55 seconds. In Q8, rather than calling the LLMMAP function over all 19,672 rows, processing can terminate once 100 images of cars with both punctures and paint scratches are found, saving 141 seconds.

Early deduplication is responsible for the single largest drop in latency across the dataset, saving 40 seconds on the MOVIE scenario. The reason for this is the relatively large scale factor (2,000) and level of duplication within the columns passed to LM functions: of the 2,000 values in the `reviewText` column, only 1,864 are distinct. This saves 136 calls to an LM on a simple LLMMAP across the column, but the effect is compounded when performing joins: the self-join logic in Q5, Q6, and Q7 decreases the size of the resulting data subset from 122,004 rows to 72,675. We see latency gains in the MMQA scenario for the same reasons, where Q2a and Q2b select the 6 distinct `Track` values rather than the 13 total values (with duplicates), resulting in the cross-join passing 1,200 values to the LM instead of 2,600.

Constrained decoding is a major factor in the quality of the language model outputs. In the CARS scenario, the average quality scores are boosted by 0.03 and 0.04 for Gemma 4 E2B and Gemma 4 E4B, respectively. The largest increase in quality for the CARS scenario occurs in Q10, in which the LM function must classify a customer complaint into one of 24 possible categories (electrical system, power train, engine, etc.). Applying constrained decoding results in a +0.35 improvement in quality for Gemma 4 E4B, showing that despite being prompted to return a response within a valid set of options, small language models often fail to generate a valid selection without formal runtime constraints. The largest change in quality is seen in the ECOMM scenario, where the addition of constrained decoding boosts average quality by 0.06 for Gemma 4 E2B and 0.07 for Gemma 4 E4B. This large boost in performance is primarily due to Q12 (+0.90 increase after applying constrained decoding), which tasks the language model with generating a JSON object with a specified schema, and Q5 / Q11 (+0.05, +0.04), in which the language model plays the role of a 5-way multi-class classifier. Critically, this quality improvement is achieved without sacrificing latency: enabling constrained decoding via LLGUIDANCE adds little-to-no overhead, adding 0.064 seconds of latency to the Gemma 4 E2B model and saving 0.20 seconds with the Gemma 4 E4B model.

As very few queries in the SemBench benchmark contain multiple LM functions within the same subquery, cascade filtering shows little impact within this context. However, we see Q14 in the ECOMM scenario save 142 seconds of latency after enabling cascade filtering. In this query, a program is written to (1) Find all images containing white socks, and (2) Identify which image and description pairs are aligned. By filtering on the "white socks" criteria first, the second function only receives the 426 filtered inputs rather than the full set of 3,000 image / description pairs.

5 ENVIRONMENTAL IMPACT

The total runtime for all reported experimental results amounted to 81 hours on the workstation described in Section 3.2. To estimate the carbon footprint of our evaluation, we followed the widely adopted methodology proposed by Lannelongue et al. [28][5]. This amounts to an estimated energy consumption of 27.21 kWh, which translates to approximately 11.4 kgCO_2. For context, this carbon footprint is roughly equivalent to the emissions of driving 46.67 km (29 miles) in an average gasoline-powered passenger car [1].

6 LIMITATIONS

Described in Section 4.1, a full comparison of BLENDSQL against the baseline LM-DB systems with Gemma 4 E4B is not possible, as the baseline systems raise errors when using a local vLLM endpoint on multi-modal inputs. We hope that in the future, a complete comparison on the SemBench dataset with local open-weight models may be made possible.

The notion of "financial cost" is highly volatile and subject to the volatility of hardware rental and proprietary API pricing. The rates used in this study ($0.18/hour for an NVIDIA RTX 5080 and $0.30 per million input tokens for Gemini 2.5 Flash) reflect market conditions at the time of writing, and are influenced by economic factors outside the scope of this study. While monetary costs will inevitably fluctuate, our broader conclusion relies not on specific price points, but on the structural shift from a volume-based (per-token) to a time-based (hourly compute) pricing paradigm. Because BLENDSQL v0.1.0 reduces execution time while matching or improving accuracy, this structural advantage remains robust regardless of future economic shifts. To emphasize the financial cost savings

[4]Complete query-level results can be found in Appendix 9

[5]https://calculator.green-algorithms.org/

independent of on-demand rental availability in third-party GPU marketplaces, we present an alternative setting. The RTX 5080 has an MSRP of $999.99; with standard markup pricing for high-demand graphics cards, it is available for $1,300 as of May 2026. The AMD Ryzen 7600X is available for $167. Once these hardware costs are accounted for, only energy costs must be considered: Electric Choice report average U.S. residential energy costs of 0.18 / kWh [7]. Factoring in these variables, the cost to buy the workstation described in Section 3.2 and run the ECOMM scalability experiment in Section 4.1 is approximately $1,469 (derived via $1300+167+(13.13*0.18)$). This is still substantially less than the $2,988 cost of using PALIMPZEST with Gemini 2.5 Flash for the same experiment.

7 RELATED WORK

Combining Language Models with Database Systems Combining language models with structured data operators is a widely studied topic. The origin of such an approach can be traced back to early SQL operators enabling statistical machine learning and data mining [21, 23, 38]. Later, database management systems with native predictive functions utilizing models such as logistic regression and support vector machines became more common, such as SQL Server's `PREDICT` function in 2017 and BigQuery ML in 2018 [19, 22, 39]. To the best of our knowledge, Bae et al. [4] were the first to propose the idea of putting calls to a multi-modal neural model into a SQL query. Others have since continued exploration into domain-specific languages for combining the generalized computations of language models with the structured reasoning of traditional database query languages [6, 11, 16, 43, 54].

Modern approaches integrate language models with database management systems at varying levels. While Patel et al. [43] intervenes via a Pandas API, Dorbani et al. [11] build out a set of custom UDFs for the online analytical processing DBMS DuckDB [47]. Tjangnaka et al. [54] build out UDFs for the PostgreSQL DBMS [46], with additional calls to language models to determine the semantic equivalency LM-generated values against native database values. Palimpzest [34] is unique in its formulation of the cost optimization problem, allowing the system to search over a space of various-priced closed LM APIs. A subset of work specifically explores efficient methods for optimizing LM functions in relational systems [26, 35].

Constrained Decoding Constrained decoding refers to the process of controlling the output of language models by applying masks at the decoding level, such that generations adhere to a specific predetermined constraint [9]. These constraints are typically encoded via regular expressions or context-free grammars, and optimized decoding engines have emerged for deriving and applying masks [10, 15, 20, 42, 62]. Recent work has explored alternatives to the predominant locally-constrained decoding paradigm [33]. Most relevant to our work is Mündler et al. [40], who present an algorithm to enforce the well-typedness of LLM-generated TypeScript code. Whereas they tackle the problem of determining whether a partial program can be completed into a well-typed program, we explore constraints for integrating LM outputs into a declarative query language like SQL.

8 CONCLUSION

Under token-based API pricing, cost scales with the number of tokens processed, independent of inference speed. Open-weight models on local, hourly-billed hardware invert this relationship: by tuning parameters such as max concurrency (Figure 5) and query optimizations (Figure 4), faster inference directly translates to lower cost. This shift in incentive structure allows researchers to focus on core inference innovations with known architectures and hardware, rather than engineering around black-box models, arbitrary rate limits, and opaque infrastructure. The cost difference becomes stark at scale and massively improves the accessibility of LM-DB research: Lao et al. [29] report spending an average of $2,988.32 evaluating PALIMPZEST on the ECOMM scenario five times across five scale factors. We show that the equivalent workload with BLENDSQL and Gemma 4 E4B costs $6.96. Perhaps most importantly, numerous studies have shown API quality fluctuates over time, making full reproducibility impossible [2, 5, 57]. Open-weight models guarantee full reproducibility by decoupling experimental results from the volatility of proprietary LM endpoints.

We presented BLENDSQL v0.1.0, a LM-DB system that elevates small, open-weight language models to performance competitive with closed API solutions. We demonstrated a cost reduction of 390x and latency reduction of 3.8x, with performance matching or exceeding systems with closed-weight models on text and image inputs. We highlighted the modality gap between small open-weight and closed-weight models, demonstrating a stark quality difference on audio data. Finally, we examined the components aiding the success of LM-DB systems through a series of ablations.

REFERENCES

[1] United States Environmental Protection Agency. 2026. Greenhouse Gas Equivalencies Calculator. https://www.epa.gov/energy/greenhouse-gas-equivalencies-calculator. Accessed: 2026-06-06.

[2] Florian Angermeir, Maximilian Amougou, Mark Kreitz, Andreas Bauer, Matthias Linhuber, Davide Fucci, Daniel Mendez, Tony Gorschek, et al. 2025. Reflections on the Reproducibility of Commercial LLM Performance in Empirical Software Engineering Studies. *arXiv preprint arXiv:2510.25506* (2025).

[3] Samuel Arch, Yuchen Liu, Todd C Mowry, Jignesh M Patel, and Andrew Pavlo. 2024. The key to effective udf optimization: Before inlining, first perform outlining. *Proceedings of the VLDB Endowment* 18, 1 (2024), 1–13.

[4] Seongsu Bae, Daeun Kyung, Jaehee Ryu, Eunbyeol Cho, Gyubok Lee, Sunjun Kweon, Jungwoo Oh, Lei Ji, Eric Chang, Tackeun Kim, et al. 2023. Ehrxqa: A multi-modal question answering dataset for electronic health records with chest x-ray images. *Advances in Neural Information Processing Systems* 36 (2023), 3867–3880.

[5] Lingjiao Chen, Matei Zaharia, and James Zou. 2024. How is ChatGPT's behavior changing over time? *Harvard Data Science Review* 6, 2 (2024).

[6] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. 2023. Binding Language Models in Symbolic Languages. In *International Conference on Learning Representations (ICLR 2023)(01/05/2023-05/05/2023, Kigali, Rwanda)*.

[7] Electric Choice. 2026. Electricity Rates by State. https://www.electricchoice.com/electricity-prices-by-state/. Accessed: 2026-05-28.

[8] Suparno Roy Chowdhury, Manan Roy Choudhury, Tejas Anvekar, Muhammad Ali Khan, Kaneez Zahra Rubab Khakwani, Mohamad Bassam Sonbol, Irbaz Bin Riaz, and Vivek Gupta. 2026. Diagnosis, Bad Planning & Reasoning. Treatment, SCOPE–Planning for Hybrid Querying over Clinical Trial Data. *arXiv preprint arXiv:2604.25120* (2026).

[9] Daniel Deutsch, Shyam Upadhyay, and Dan Roth. 2019. A general-purpose algorithm for constrained sequential inference. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*. 482–492.

[10] Yixin Dong, Charlie F Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. 2024. Xgrammar: Flexible and efficient structured generation engine for large language models. *arXiv preprint arXiv:2411.15100* (2024).

[11] Anas Dorbani, Sunny Yasser, Jimmy Lin, and Amine Mhedhbi. 2025. Beyond Quacking: Deep Integration of Language Models and RAG into DuckDB. (2025).

[12] Yannis Foufoulas, Theoni Palaiologou, and Alkis Simitsis. 2025. The UDFBench Benchmark for General-purpose UDF Queries. *Proceedings of the VLDB Endowment* 18, 9 (2025), 2804–2817.

[13] Yannis Foufoulas and Alkis Simitsis. 2023. Efficient execution of user-defined functions in SQL queries. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3874–3877.

[14] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. 2025. Jsonschemabench: A rigorous benchmark of structured outputs for language models. *arXiv preprint arXiv:2501.10868* (2025).

[15] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. *arXiv preprint arXiv:2305.13971* (2023).

[16] Parker Glenn, Parag Dakle, Liang Wang, and Preethi Raghavan. 2024. BlendSQL: A Scalable Dialect for Unifying Hybrid Question Answering in Relational Algebra. In *Findings of the Association for Computational Linguistics ACL 2024*. 453–466.

[17] Parker Glenn, Alfy Samuel, and Daben Liu. [n.d.]. Play by the Type Rules: Inferring Constraints for Small Language Models in Declarative Programs. In *EurIPS 2025 Workshop: AI for Tabular Data*.

[18] Google. [n.d.]. Announcing BigQuery-managed AI functions for better SQL. https://cloud.google.com/blog/products/data-analytics/sql-reimagined-for-the-ai-era-with-bigquery-ai-functions. Accessed: 2026-05-07.

[19] Google. 2018. What a week! 105 announcements from Google Cloud Next '18. https://blog.google/innovation-and-ai/infrastructure-and-cloud/google-cloud/100-plus-announcements-from-google-cloud-next-18/. Accessed: 2026-05-13.

[20] Guidance. 2023. Guidance: A language model programming framework. https://github.com/guidance-ai/guidance. Accessed: 2025-08-11.

[21] Jiawei Han, Yongjian Fu, Wei Wang, Krzysztof Koperski, Osmar Zaiane, et al. 1996. DMQL: A data mining query language for relational databases. In *Proc. 1996 SiGMOD*, Vol. 96. 27–34.

[22] Joe Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. 2012. The MADlib analytics library or MAD skills, the SQL. *arXiv preprint arXiv:1208.4165* (2012).

[23] Tomasz Imieliński and Aashu Virmani. 1999. MSQL: A query language for database mining. *Data Mining and Knowledge Discovery* 3, 4 (1999), 373–408.

[24] Saehan Jo and Immanuel Trummer. 2024. Thalamusdb: Approximate query processing on multi-modal data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[25] Rohit Khoja, Devanshu Gupta, Yanjie Fu, Dan Roth, and Vivek Gupta. 2025. Weaver: Interweaving SQL and LLM for Table Reasoning. *arXiv preprint arXiv:2505.18961* (2025).

[26] Kyoungmin Kim, Kijae Hong, Caglar Gulcehre, and Anastasia Ailamaki. 2024. Optimizing LLM Inference for Database Systems: Cost-Aware Scheduling for Concurrent Requests. *arXiv preprint arXiv:2411.07447* (2024).

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[28] Loïc Lannelongue, Jason Grealey, and Michael Inouye. 2021. Green algorithms: quantifying the carbon footprint of computation. *Advanced science* 8, 12 (2021), 2100707.

[29] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, et al. 2025. SemBench: A Benchmark for Semantic Query Processing Engines. *arXiv preprint arXiv:2511.01716* (2025).

[30] Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. 2026. Meta-Harness: End-to-End Optimization of Model Harnesses. *arXiv preprint arXiv:2603.28052* (2026).

[31] Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, et al. 2025. Deepagent: A general reasoning agent with scalable toolsets. *arXiv preprint arXiv:2510.21618* (2025).

[32] Yin Lin, Tianjing Zeng, Zhongjun Ding, Rong Zhu, Bolin Ding, HV Jagadish, and Jingren Zhou. 2026. SEMA-SQL: Beyond Traditional Relational Querying with Large Language Models. *arXiv preprint arXiv:2604.23477* (2026).

[33] Benjamin Lipkin, Benjamin LeBrun, Jacob Hoover Vigly, João Loula, David R MacIver, Li Du, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Timothy J O'Donnell, et al. 2025. Fast Controlled Generation from Language Models with Adaptive Weighted Rejection Sampling. *arXiv preprint arXiv:2504.05410* (2025).

[34] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2.

[35] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. 2024. Optimizing llm queries in relational workloads. *CoRR* (2024).

[36] Qiuyang Mang, Yufan Xiang, Hangrui Zhou, Runyuan He, Jiaxiang Yu, Hanchen Li, Aditya Parameswaran, and Alvin Cheung. 2026. Horrila: Cost-Based Placement of Semantic Operators in Hybrid Query Plans. *arXiv preprint arXiv:2604.09944* (2026).

[37] Toby Mao. [n.d.]. SQLGlot: Python SQL Parser and Transpiler. https://github.com/tobymao/sqlglot.

[38] Rosa Meo, Giuseppe Psaila, Stefano Ceri, et al. 1996. A new SQL-like operator for mining association rules. In *VLDB*, Vol. 96. 122–133.

[39] Microsoft. 2017. What's new in SQL Server 2017. https://learn.microsoft.com/en-us/sql/sql-server/what-s-new-in-sql-server-2017?view=sql-server-ver17. Accessed: 2026-05-13.

[40] Niels Mündler, Jingxuan He, Hao Wang, Koushik Sen, Dawn Song, and Martin Vechev. 2025. Type-Constrained Code Generation with Language Models. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 601–626.

[41] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

[42] Kanghee Park, Timothy Zhou, and Loris D'Antoni. 2025. Flexible and efficient grammar-constrained decoding. *arXiv preprint arXiv:2502.05111* (2025).

[43] Liana Patel, Siddharth Jha, Parth Asawa, Melissa Pan, Carlos Guestrin, and Matei Zaharia. 2024. Semantic Operators: A Declarative Model for Rich, AI-based Analytics Over Text Data. arXiv:2407.11418 [cs.DB] https://arxiv.org/abs/2407.11418

[44] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2024. Semantic Operators: A Declarative Model for Rich, AI-based Data Processing. *arXiv preprint arXiv:2407.11418* (2024).

[45] Polars. 2022. Polars: Extremely fast Query Engine for DataFrames, written in Rust. https://github.com/pola-rs/polars. Accessed: 2026-04-23.

[46] PostgreSQL. 2025. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. https://www.postgresql.org/

[47] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[48] Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael Cafarella, Tim Kraska, and Samuel Madden. 2025. Abacus: A Cost-Based Optimizer for Semantic Operator Systems. *arXiv preprint arXiv:2505.14661* (2025).

[49] salad.com. [n.d.]. Salad.com 5080 Pricing. https://salad.com/pricing. Accessed: 2026-04-25.

[50] Nima Shahbazi, Seiji Maekawa, Nikita Bhutani, and Estevam Hruschka. 2026. OmniTQA: A Cost-Aware System for Hybrid Query Processing over Semi-Structured Data. *arXiv preprint arXiv:2604.02444* (2026).

[51] Yunxiang Su, Tianjing Zeng, Zhongjun Ding, Yin Lin, Rong Zhu, Zhewei Wei, Bolin Ding, and Jingren Zhou. 2026. Large Language Model-Enhanced Relational Operators: Taxonomy, Benchmark, and Analysis. *arXiv preprint arXiv:2603.02537* (2026).

[52] Gemma Team. [n.d.]. Gemma 4. https://deepmind.google/models/gemma/gemma-4/.

[53] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram

Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] https://arxiv.org/abs/2503.19786

[54] Shicheng Liu Jialiang Xu Wesley Tjangnaka, Sina J Semnani Chen Jie Yu, and Monica S Lam. 2024. SUQL: Conversational Search over Structured and Unstructured Data with Large Language Models. (2024).

[55] Transaction Processing Performance Council (TPC). [n.d.]. TPC-DS Benchmark. https://www.tpc.org/tpcds/.

[56] Transaction Processing Performance Council (TPC). [n.d.]. TPC-H Benchmark. https://www.tpc.org/tpch/.

[57] Shangqing Tu, Chunyang Li, Jifan Yu, Xiaozhi Wang, Lei Hou, and Juanzi Li. 2023. Chatlog: Carefully evaluating the evolution of chatgpt across time. *arXiv preprint arXiv:2304.14106* (2023).

[58] Vast.ai. [n.d.]. Vast.ai 5080 Pricing. https://cloud.vast.ai. Accessed: 2026-04-25.

[59] Darren Yow-Bang Wang, Zhengyuan Shen, Soumya Smruti Mishra, Zhichao Xu, Yifei Teng, and Haibo Ding. 2025. Slot: Structuring the output of large language models. *arXiv preprint arXiv:2505.04016* 1, 2 (2025), 3.

[60] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[61] Johannes Wehrstein, Tiemo Bang, Roman Heinrich, and Carsten Binnig. 2025. GRACEFUL: A Learned Cost Estimator For UDFs. *arXiv preprint arXiv:2503.23863* (2025).

[62] Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702* (2023).

[63] Zhengtong Yan, Gongsheng Yuan, Qingsong Guo, and Jiaheng Lu. 2025. DBMS-LLM Integration Strategies in Industrial and Business Applications: Current Status and Future Challenges. *arXiv preprint arXiv:2507.19254* (2025).

[64] Fuheng Zhao, Divyakant Agrawal, and Amr El Abbadi. 2024. Hybrid querying over relational databases and large language models. *arXiv preprint arXiv:2408.00884* (2024).

[65] Lixi Zhou, Qi Lin, Kanchan Chowdhury, Saif Masood, Alexandre Eichenberger, Hong Min, Alexander Sim, Jie Wang, Yida Wang, Kesheng Wu, et al. 2023. Serving Deep Learning Model in Relational Databases. *arXiv preprint arXiv:2310.04696* (2023).

9 APPENDIX

9.1 vLLM Config

All local vLLM servers were launched using the following command with `vLLM==0.21.0`

```
vllm serve "${model_path}" \
--host 0.0.0.0 \
--port 8000 \
--tensor-parallel-size 1 \
--enable-prefix-caching \
--max-model-len 32000 \
--structured-outputs-config.backend guidance \
--gpu_memory_utilization 0.9 \
--mm-processor-kwargs '{"max_length": 480000}' \
--enable-prompt-tokens-details
```

9.2 Prompts

LLMMAP Prompt

```
You are a helpful assistant. You will be
presented with some context and a question.
{{return_type_instructions}}

{% if quantifier is not none %}
{{quantifier_disclaimer}}
{% endif % }

An example is shown below.

{{one_shot_example}}

---

QUESTION:
{{question}}

CONTEXT:
{{context_as_dict}}

{% if options is not none %}
OPTIONS:
{{options}}
{% endif % }

ANSWER:
```

LLMQA Prompt

```
Answer the question given the context, if provided.
Keep the answers as short as possible,
without leading context. For example, do not
say 'The answer is 2', simply say '2'.

Your response format should match the specified
'Return type', if provided.

Question: {{question}}

{% if context is not none %}
Context: {{context_as_dict}}
{% endif % }

{% if options is not none %}
Options: {{options}}
{% endif % }

{% if return_type is not none %}
Return type: {{return_type}}
{% endif % }

Answer:
```

Figure 7: Full SemBench results. BlendSQL uses `gemma-4-E4B-FP8`, other systems use `gemini-2.5-flash`.

(a) Full MMQA results.

	BlendSQL			BigQuery			LOTUS			Palimpzest			ThalamusDB		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**0.07**	**1.00**	**$2e-5**	14.10	**1.00**	$0.05	7.40	**1.00**	$0.10	4.50	**1.00**	$0.01	–	–	–
Q2a	55.15	**1.00**	**$0.01**	**53.80**	0.00	$0.40	169.60	0.83	$4.45	135.90	**1.00**	$5.20	–	–	–
Q2b	64.22	0.97	**$0.02**	**38.40**	0.00	$0.60	166.80	0.83	$4.45	133.80	**1.00**	$5.20	–	–	–
Q3a	**1.35**	0.78	**$3e-4**	19.60	0.72	$0.05	4.20	0.83	$0.05	4.50	**1.00**	$5.20	11.00	0.75	$0.05
Q3f	**1.41**	0.92	**$4e-4**	22.30	0.67	$0.05	4.20	**1.00**	$0.05	8.00	**1.00**	$0.10	7.00	**1.00**	$0.05
Q4	**0.45**	0.56	**$1e-4**	9.70	**0.60**	$0.01	–	–	–	1.20	0.54	$0.03	–	–	–
Q5	**0.24**	**1.00**	**$6e-5**	–	–	–	0.48	**1.00**	**$5e-3**	0.50	**1.00**	**$5e-3**	–	–	–
Q6a	**0.73**	**1.00**	**$2e-4**	18.90	0.03	$0.02	4.40	**1.00**	$0.05	4.00	**1.00**	$0.10	5.60	0.33	$0.01
Q6b	**0.71**	0.67	**$2e-4**	17.30	0.04	$0.02	3.80	**1.00**	$0.05	4.10	**1.00**	$0.10	5.50	**1.00**	$0.01
Q6c	**0.83**	0.91	**$2e-4**	17.40	0.13	$0.02	4.60	**1.00**	$0.05	4.70	**1.00**	$0.10	13.40	0.53	$0.01
Q7	133.26	**0.51**	**$0.03**	**91.70**	0.00	$5.90	2311.40	0.32	$68.05	2101.60	0.31	$78.25	–	–	–
Avg.	23.49	0.85	**$6e-3**	32.61	0.29	$0.79	267.27	0.88	$7.73	218.44	**0.90**	$8.57	**8.50**	0.72	$0.03
Avg. Std.	0.11	0.03	$3e-5	–	–	–	–	–	–	–	–	–	–	–	–
Wins	8	1	10	3	1	0	0	0	0	0	2	0	0	0	0
Total Cost			**$0.06**			$7.11			$77.30			$94.30			$0.13

(b) Full ECOMM results.

	BlendSQL			BigQuery			LOTUS			Palimpzest			ThalamusDB		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**8.18**	**1.00**	**$2e-3**	21.20	0.59	$0.20	12.20	**1.00**	$0.30	12.20	**1.00**	$0.40	34.80	**1.00**	$0.15
Q2	**23.15**	0.40	**$6e-3**	55.70	0.21	$19.80	166.10	**0.87**	$0.90	57.50	0.83	$1.90	100.80	0.67	$1.55
Q3	**8.53**	0.78	**$2e-3**	21.20	**0.97**	$0.60	17.10	**0.97**	$0.35	16.50	**0.98**	$0.60	–	–	–
Q4	**13.65**	0.40	**$3e-3**	31.00	**0.69**	$1.85	156.70	0.45	$0.70	335.00	0.53	$1.20	–	–	–
Q5	**4.48**	0.96	**$1e-3**	25.70	**0.98**	$0.85	7.40	**0.99**	$0.20	6.60	**0.98**	$0.35	–	–	–
Q6	**13.34**	0.67	**$3e-3**	34.90	**0.88**	$1.75	114.80	**0.89**	$0.60	143.70	**0.89**	$2.15	–	–	–
Q7	222.74	0.77	**$0.06**	**45.40**	0.83	$4.30	199.40	0.75	$6.65	287.60	**0.92**	$8.95	97.70	0.51	$0.40
Q8	51.63	0.50	**$0.01**	126.20	0.29	$91.15	4.10	**1.00**	$0.02	**3.90**	**1.00**	$0.05	713.00	0.00	$1.50
Q9	**24.43**	0.43	**$6e-3**	48.60	**0.58**	$1.05	243.40	0.55	$0.30	44.90	0.49	$0.50	872.20	0.00	$1.55
Q10	**10.95**	0.00	**$3e-3**	–	–	–	519.50	0.00	$0.30	1192.50	**0.06**	$5.10	–	–	–
Q11	**57.46**	0.25	**$0.01**	–	–	–	158.70	**0.78**	$1.35	132.40	0.73	$3.05	–	–	–
Q12	**13.58**	0.91	**$3e-3**	31.10	**0.97**	$0.50	36.40	0.60	$0.50	31.90	0.00	$0.70	–	–	–
Q13	32.99	0.51	**$8e-3**	**22.40**	0.70	$1.90	274.80	**0.74**	$1.20	238.30	**0.74**	$2.20	–	–	–
Q14	**48.73**	**0.88**	**$0.01**	73.60	0.37	$21.30	178.20	**0.87**	$1.15	–	–	–	–	–	–
Avg.	**38.13**	0.60	**$1e-2**	44.75	0.67	$12.10	149.20	**0.75**	$1.04	192.54	0.70	$2.09	363.70	0.44	$1.03
Avg. Std.	0.15	0.02	$4e-5	–	–	–	–	–	–	–	–	–	–	–	–
Wins	11	0	14	2	3	0	0	2	0	1	2	0	0	0	0
Total Cost			**$0.13**			$145.25			$14.52			$27.15			$5.15

(c) Full MOVIE results.

	BlendSQL			BigQuery			LOTUS			Palimpzest			ThalamusDB		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**0.39**	**1.00**	**$1e-4**	26.30	**1.00**	$0.25	33.10	**1.00**	$0.45	3.80	**1.00**	**$5e-3**	4.20	0.95	**$2e-3**
Q2	**0.45**	0.96	**$1e-4**	9.50	**1.00**	$0.01	2.10	**1.00**	$0.05	29.70	**1.00**	$0.05	1.90	0.92	$0.01
Q3	**0.48**	0.69	**$1e-4**	11.00	0.64	$0.01	2.10	0.64	$0.03	4.60	0.64	$0.05	3.10	**0.74**	$0.01
Q4	**0.32**	0.64	**$8e-5**	11.40	0.64	$0.01	2.80	0.64	$0.03	4.40	**0.74**	$0.10	3.80	**0.74**	$0.01
Q5	**0.63**	0.46	**$2e-4**	54.50	0.89	$5.05	536.50	0.59	$11.90	1.90	0.39	$0.05	2.30	**1.00**	**$5e-3**
Q6	**0.57**	0.46	**$1e-4**	54.50	0.69	$5.00	432.40	0.67	$9.05	2.30	**0.83**	$0.05	1.70	**0.84**	**$4e-3**
Q7	**149.03**	0.54	**$0.04**	198.00	**0.70**	$16.55	431.80	0.21	$9.05	1056.10	**0.68**	$38.60	649.90	0.57	$0.75
Q8	**0.68**	**0.92**	**$2e-4**	10.90	0.76	$0.01	2.30	**0.93**	$0.02	4.30	0.86	$0.10	6.80	0.83	$0.03
Q9	**0.81**	**0.87**	**$2e-4**	13.30	0.78	$0.10	4.90	0.75	$0.10	5.70	0.78	$0.25	–	–	–
Q10	**8.88**	**0.71**	**$2e-3**	32.10	0.44	$0.65	30.90	0.40	$0.65	39.20	0.42	$1.90	–	–	–
Avg.	**16.22**	0.72	**$4e-3**	42.15	0.75	$2.77	147.89	0.68	$3.13	115.20	0.73	$4.12	84.21	**0.82**	$0.10
Avg. Std.	0.22	0.03	$6e-5	–	–	–	–	–	–	–	–	–	–	–	–
Wins	10	2	7	0	0	0	0	0	0	0	0	0	0	2	0
Total Cost			**$0.04**			$27.66			$31.32			$41.16			$0.82

(a) Full CARS results.

	BlendSQL			BigQuery			LOTUS			Palimpzest			ThalamusDB		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	201.92	0.63	**$0.05**	**61.70**	0.71	$7.20	550.00	**0.90**	$8.70	465.60	0.69	$12.20	829.70	0.81	$6.85
Q2	**1.97**	0.00	**$5e-4**	14.10	**0.08**	$0.05	–	–	–	4.10	0.00	$0.02	14.10	**0.09**	$0.05
Q3	**1.69**	0.94	**$4e-4**	36.40	**1.00**	$8.30	456.20	0.90	$3.00	6.10	0.92	$0.05	–	–	–
Q4	193.15	**0.99**	**$0.05**	**68.70**	**0.99**	$7.05	822.00	**0.99**	$8.55	443.60	**0.99**	$12.05	768.80	**1.00**	$6.70
Q5	**2.25**	0.00	**$6e-4**	58.90	**1.00**	$7.35	–	–	–	6.30	**1.00**	$0.05	483.70	**1.00**	$8.05
Q6	219.79	0.75	**$0.05**	**44.30**	**0.96**	$10.00	–	–	–	427.30	**0.96**	$12.55	775.40	**0.97**	$9.80
Q7	378.20	0.55	**$0.09**	**86.00**	0.45	$15.85	–	–	–	882.70	**0.56**	$22.35	2146.20	**0.58**	$15.30
Q8	173.18	0.07	**$0.04**	**38.20**	0.24	$8.45	1349.80	**0.45**	$8.90	268.70	0.29	$10.10	68.00	0.20	$1.05
Q10	350.62	0.50	**$0.09**	**62.00**	**0.57**	$13.50	618.10	0.41	$15.45	594.90	0.51	$23.45	–	–	–
Avg.	169.20	0.49	**$0.04**	**52.26**	0.67	$8.64	759.22	**0.73**	$8.92	344.37	0.66	$10.31	726.56	0.66	$6.83
Avg. Std.	0.10	0.01	$3e-5	–	–	–	–	–	–	–	–	–	–	–	–
Wins	3	0	9	6	2	0	0	2	0	0	0	0	0	0	0
Total Cost			**$0.38**			$77.75			$44.60			$92.82			$47.80

(b) Full WILDLIFE results.

	BlendSQL			BigQuery			LOTUS			Palimpzest			ThalamusDB		
Query	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**	**Lat. (s)**	**Quality**	**Cost (5 runs)**
Q1	**9.41**	**0.78**	**$2e-3**	32.00	**0.79**	$0.55	92.40	**0.79**	$0.55	32.80	**0.79**	$0.65	19.60	**0.79**	$0.55
Q2	4.16	0.00	**$1e-3**	9.40	**0.19**	$0.05	–	–	–	**2.80**	**0.17**	$0.05	4.50	0.14	$0.05
Q3	9.29	0.20	**$2e-3**	25.70	0.00	$0.55	99.40	**1.00**	$0.55	22.70	0.00	$0.65	**4.30**	0.00	$0.15
Q4	4.01	0.40	**$1e-3**	9.40	**1.00**	$0.05	–	–	–	2.70	0.00	$0.05	**1.30**	0.00	**$5e-3**
Q5	13.39	**0.93**	**$3e-3**	19.20	0.75	$0.60	–	–	–	13.50	0.75	$0.65	**2.30**	0.75	$0.05
Q6	13.34	0.20	**$3e-3**	24.30	0.20	$0.60	–	–	–	19.30	0.00	$0.65	**13.20**	**0.50**	$0.30
Q7	**18.58**	**1.00**	**$5e-3**	24.60	**1.00**	$1.10	188.30	**1.00**	$1.15	43.90	**1.00**	$0.65	28.80	**1.00**	$1.00
Q8	26.68	**0.83**	**$7e-3**	35.50	0.75	$1.15	–	–	–	34.90	0.75	$0.65	**23.80**	0.75	$0.60
Q9	**11.92**	0.61	**$3e-3**	37.60	0.59	$0.60	–	–	–	19.10	0.57	$0.65	17.20	**0.67**	$0.40
Q10	9.27	0.40	**$2e-3**	40.30	0.00	$0.55	87.80	**1.00**	$0.55	19.60	0.00	$0.65	**4.40**	0.00	$0.15
Avg.	12.01	0.54	**$3e-3**	25.80	0.53	$0.58	116.98	**0.95**	$0.70	21.13	0.40	$0.53	**11.94**	0.46	$0.33
Avg. Std.	0.08	0.22	$2e-5	–	–	–	–	–	–	–	–	–	–	–	–
Wins	3	2	9	0	1	0	0	2	0	1	0	0	6	2	0
Total Cost			**$0.03**			$5.80			$2.80			$5.30			$3.25

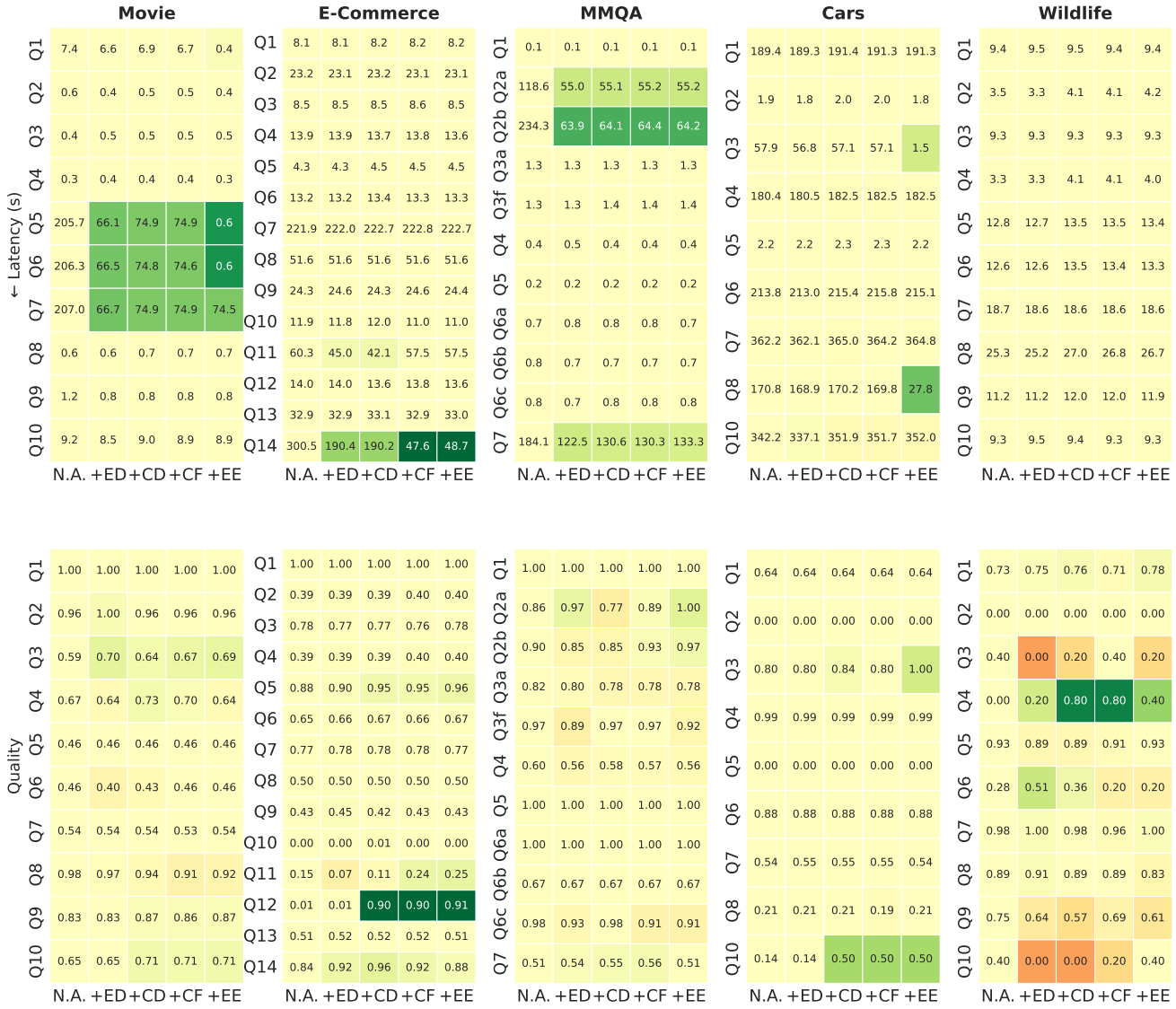

Figure 9: Plotting the query-level latency and quality changes on SemBench scenarios with BlendSQL and Gemma 4 E4B under various feature ablations.