Play by the Type Rules: Inferring Constraints for Small Language Models in Declarative Programs

Parker Glenn, Alfy Samuel, Daben Liu
Capital One
{parker.glenn,alfy.samuel,daben.liu}@capitalone.com

Abstract

Integrating language model powered operators in declarative query languages allows for the combination of cheap and interpretable functions with powerful, generalizable reasoning. However, in order to benefit from the optimized execution of a database query language like SQL, generated outputs must align with the rules enforced by both type checkers and database contents. Current approaches address this challenge with orchestrations consisting of many LLM-based post-processing calls to ensure alignment between generated outputs and database values, introducing performance bottlenecks. We perform a study on the ability of various sized open-source language models to both parse and execute functions within a query language based on SQL, showing that small language models can excel as function executors over hybrid data sources. Then, we propose an efficient solution to enforce the well-typedness of language model functions, demonstrating 7% accuracy improvement on a multi-hop question answering dataset. We make our implementation available at https://github.com/parkervg/blendsql.

1 Introduction

Language models (LMs) are capable of impressive performance on tasks requiring multi-hop reasoning. In some cases, evidence of latent multi-hop logic chains have been observed (Yang et al., 2024; Lindsey et al., 2025). However, particularly with smaller language models which lack the luxury of over-parameterization, a two-step "divide-then-conquer" paradigm has shown promise (Wolfson et al., 2020; Wu et al., 2024; Li et al., 2024).

In tasks like proof verification, languages such as Lean have become increasingly popular as an intermediate representation (Moura and Ullrich, 2021). This program synthesis paradigm, or generation of an executable program to aid compositional reasoning, has been shown to improve performance on many math-based tasks (Olausson et al., 2023; Wang et al., 2025; Xin et al., 2024). In settings requiring multi-hop reasoning over large amounts of hybrid tabular and textual data sources, the appeal of program synthesis is two-fold: not only has synthesizing intermediate representations been proven to increase performance in certain settings (Tjangnaka et al., 2024; Shi et al., 2024; Glenn et al., 2024), but offloading logical deductions to traditional programming languages when possible allows for efficient data processing, particular in the presence of extremely large database contexts.

Existing approaches take a two-phase approach to embedding language models into typed programming languages like SQL, where a response is first generated, and an additional call to a language model is made to evaluate semantic consistency against a reference value. For example, imagine an example query with a language model function, `SELECT * FROM t WHERE city = prompt('What is the U.S. capital?')`.

A reasonable, factual generation might be "Washington D.C.". However, when integrating this output to a SQL query against a database with the "city" stored as "Washington DC", the absence of exact formatting alignment can yield unintended results that break the reasoning chains of multi-hop

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Workshop on AI for Tabular Data.

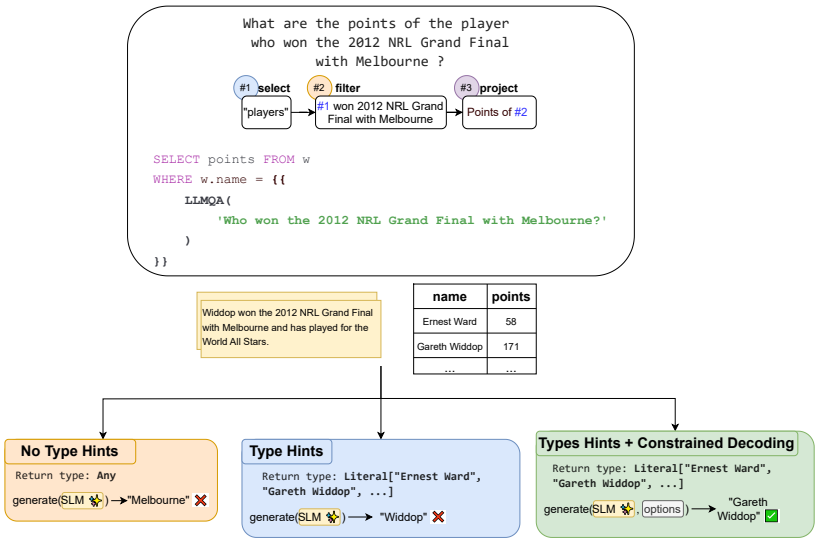

Figure 1: **Visualizing the type policies for aligning text and table values via the aggregate function LLMQA.** Even with explicit type hints included in the prompt, small language models often fail to abide by exact formatting instructions required for precise alignment to database contexts.

problems. Various approaches have been taken to solve this language model-database alignment problem: Tjangnaka et al. (2024) align unexpected LM generations to database values by prompting gpt-3.5-turbo, and Shi et al. (2024) introduce a `check()` function to evaluate semantic consistency under a given operator (e.g. =, <, >) via few-shot prompting to a LM. By taking a post-processing approach to type alignment, these additional calls to LMs introduce bottlenecks in program execution. In performance-sensitive environments such as database systems where minimizing latency is critical, this approach is suboptimal.

2 Inferring Type Constraints via Expression Context

We define three methods for handling the output of LM functions below. We use the query language BlendSQL for our experiments, where calls to language models are denoted by double-curly brackets, "{{" and "}}" (Glenn et al., 2024).

As an illustrative example, we will take the following query. We use the aggregate function LLMQA, which returns a single scalar value.

```
CREATE TABLE t(
    name TEXT,
    age INTEGER
);
INSERT INTO t VALUES('Steph Curry', 37);

/* Is Lebron James older than Steph Curry? */
SELECT {{LLMQA('How old is Lebron James?')}} > age FROM t
WHERE name = 'Steph Curry'
```

2.1 No Type Hints

By default, the language model will be prompted to return an answer to the question with no explicit type hints or coercion. After querying the language model, the final query could be:

```
SELECT 'The answer is 40.' > age FROM t WHERE name = 'Steph Curry'
```

Note that SQLite's type affinity allows for implicit coercion of certain TEXT literals to NUMERIC datatypes, such as the string "40". However, as an unconstrained language model with no explicit constraints may return unnecessary commentary with no applicable type conversion rules (e.g. "The

answer is..."), type affinity is often rendered insufficient for executing a valid and faithful query with integrated language model output, particularly for small language models (SLMs).

2.2 Type Hints

In this mode, a Python-style type hint is inserted into the prompt alongside the question. For the working query, this would be "Return type: int".

As with the "No Type Hints" setting, type affinity rules are relied on to cast inserted language model output to most SQLite datatypes. Given the desired datatype is included via instruction in the prompt, executing with a sufficiently capable instruction-finetuned language model may yield a final SQL query of:

```
SELECT '40' > age FROM t WHERE name = 'Steph Curry'
```

2.3 Type Hints & Constrained Decoding

When executed with type constraints, the process is three-fold:

1) Infer the return type of the LM-based function given Table 1, and insert the Python-style type hint into the prompt.
2) Retrieve a regular expression corresponding to the inferred return type, and use it to perform constrained decoding.
3) Cast the language model output to the appropriate native Python type (e.g. INTEGER = `int(s)`) and insert the result into the wider SQL query.

Barring any user-induced syntax errors, the output of the language model is guaranteed to result in a query that is accepted by the SQL type checker.

The resulting query in this mode would be something such as:

```
SELECT 40 > age FROM t WHERE name = 'Steph Curry'
```

Database Driven Constraints In addition to primitive types generated from pre-defined regular expressions, we also consider the LITERAL datatype as all distinct values from a column. This enables alignment between LM generations and database contents at the decoding level, in a single generation pass. We represent these type hints by inserting "Literal['a', 'b', 'c']" in our prompts.

Function Context	Inferred Signature
`f() = TRUE`	`f() → bool`
`f() > 40`	`f() → int`
`city = f()*`	`f() → Literal[ 'Washington DC' , 'San Jose' ]`
`team IN f()*`	`f() → List[Literal[ 'Red Sox' , 'Mets' ]]`
`ORDER BY f()`	`f() → Union[float, int]`
`SUM(f())`	`f() → Union[float, int]`

Table 1: **Sample of type inference rules for BLENDSQL UDFs.** Highlighted values indicate references to all distinct values of the predicate's column argument. Asterisks (*) refer to rules which only apply to the aggregate LLMQA function. In "Type Hints & Constrained Decoding" mode, each return type is used to fetch a regular expression to guide generation (e.g. `int` → `\d+`).

2.4 Hybrid Question Answering Experiments

We evaluate our program synthesis with type constraints approach on the HybridQA dataset (Chen et al., 2020), containing questions requiring multi-hop reasoning over both tables and texts from Wikipedia. Figure 1 provides an example. We evaluate our approaches on the first 1,000 examples from the HybridQA validation set. On average across the validation set, the tables have 16 rows and 4.5 columns, and the unstructured text context is 9,134 tokens. We use denotation accuracy as our primary metric (Cheng et al., 2023)[1].

[1]This metric is robust to structural differences between predictions and ground truth annotations pointing to the same semantic referent (e.g. "two" vs. "2").

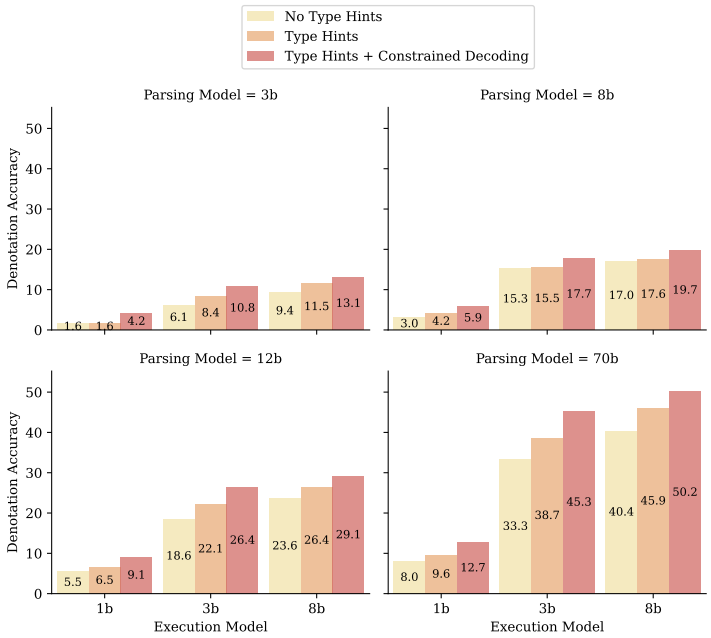

Figure 2: **Impact of various typing policies on HybridQA validation performance across model sizes.** Descriptions of typing policies can be found in Section 2

Models In order to evaluate the performance of models at various sizes, we use Llama-3.2-1B-Instruct, Llama-3.2-3B-Instruct, Llama-3.1-8B-Instruct, and Llama-3.3-70B-Instruct (Dubey et al., 2024). Additionally, we evaluate gemma-3-12b-it (Team et al., 2024).

Parsing and Execution In the parsing phase, a language model is prompted to generate a BlendSQL query given a `(question, database)` pair. We use an abbreviated version of the BlendSQL documentation[2] alongside 4 hand-picked examples from the HybridQA train split for our prompt.

We execute BlendSQL queries against a local SQLite database with the aggregate function LLMQA and scalar function LLMMAP. Full prompts for both functions can be found in Appendix 3 and 4, respectively. Additionally, we define an LLMSEARCHMAP function, which is a map function connected to unstructured article contexts via a hybrid BM25 / vector search. For both the LLMSEARCHMAP and LLMQA search, we use `all-mpnet-base-v2` (Song et al., 2020). All article text is split into sentences before being stored in the search index. We set the number of retrieved sentences (k) to 1 for the LLMSEARCHMAP function, and 10 for the LLMQA function. For all constrained decoding functionality, we use guidance (Guidance, 2023), which traverses a token trie at decoding time to mask invalid continuations given a grammar.

2.5 Results

Figure 2 shows the impact of the typing policies described in Section 2, with different combinations of parsing and execution models. In all settings, Type Hints + Constrained Decoding outperforms the rest of the policies. We see the biggest performance lift when using the 3b model to execute the functions derived from the larger 70b parameter model, where denotation accuracy rises by 6.6 points after applying type constraints. This indicates that despite occasionally failing to follow exact formatting instructions when prompted, it is still possible to efficiently extract the desired response from the model's probability distribution via the built-in type rules of SQL and constrained decoding.

3 Conclusion

In this work, we propose an efficient decoding-level approach for aligning the generated outputs of language model functions with database contents. Additionally, we present evidence that small language models can excel as function executors on a complex multi-hop reasoning dataset when

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

given appropriate constraints. This approach, while initially developed in a SQL-like language, can be extended to any typed declarative programming language.

References

Sohee Yang, Elena Gribovskaya, Nora Kassner, Mor Geva, and Sebastian Riedel. Do large language models latently perform multi-hop reasoning? *arXiv preprint arXiv:2402.16837*, 2024.

Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. On the biology of a large language model. *Transformer Circuits Thread*, 2025. URL https://transformer-circuits.pub/2025/attribution-graphs/biology.html.

Tomer Wolfson, Mor Geva, Ankit Gupta, Matt Gardner, Yoav Goldberg, Daniel Deutch, and Jonathan Berant. Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 8:183–198, 2020.

Zhuofeng Wu, He Bai, Aonan Zhang, Jiatao Gu, VG Vydiswaran, Navdeep Jaitly, and Yizhe Zhang. Divide-or-conquer? which part should you distill your llm? *arXiv preprint arXiv:2402.15000*, 2024.

Xiang Li, Shizhu He, Fangyu Lei, JunYang JunYang, Tianhuang Su, Kang Liu, and Jun Zhao. Teaching small language models to reason for knowledge-intensive multi-hop question answering. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7804–7816, 2024.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.

Theo X Olausson, Alex Gu, Benjamin Lipkin, Cedegao E Zhang, Armando Solar-Lezama, Joshua B Tenenbaum, and Roger Levy. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. 2023.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*, 2025.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv e-prints*, pages arXiv–2405, 2024.

Shicheng Liu Jialiang Xu Wesley Tjangnaka, Sina J Semnani Chen Jie Yu, and Monica S Lam. Suql: Conversational search over structured and unstructured data with large language models. 2024.

Qi Shi, Han Cui, Haofeng Wang, Qingfu Zhu, Wanxiang Che, and Ting Liu. Exploring hybrid question answering via program-based prompting. *arXiv preprint arXiv:2402.10812*, 2024.

Parker Glenn, Parag Dakle, Liang Wang, and Preethi Raghavan. Blendsql: A scalable dialect for unifying hybrid question answering in relational algebra. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 453–466, 2024.

Wenhu Chen, Hanwen Zha, Zhiyu Chen, Wenhan Xiong, Hong Wang, and William Wang. Hybridqa: A dataset of multi-hop question answering over tabular and textual data. *arXiv preprint arXiv:2004.07347*, 2020.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. Binding language models in symbolic languages. In *International Conference on Learning Representations (ICLR 2023)(01/05/2023-05/05/2023, Kigali, Rwanda)*, 2023.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407, 2024.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*, 2024.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding. *Advances in neural information processing systems*, 33: 16857–16867, 2020.

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

Lark. Lark is a parsing toolkit for python, built with a focus on ergonomics, performance and modularity. https://github.com/lark-parser/lark. Accessed: 2025-08-27.

Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. Semantic operators: A declarative model for rich, ai-based data processing. *arXiv preprint arXiv:2407.11418*, 2024.

The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL https://doi.org/10.5281/zenodo.3509134.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023.

A Appendix

A.1 Exploring Training-Free Approaches

Context-Free Grammar Guide Despite the efficiency of executing program-based solutions for question answering tasks, the implementation of a parsing step allows for potential execution errors. These execution errors may be due to syntax (e.g. subquery missing a parentheses), or semantics only noticeable at runtime (e.g. referencing a non-existent column). We design a context-free grammar to guide BlendSQL parsing at generation time to solve for many syntactic errors. The grammar is implemented via Lark (Lark), and we leverage guidance to translate the grammar into an optimized constrained decoding mask at generation time (Guidance, 2023). This grammar ensures that generated BlendSQL queries meet certain conditions, such as having balanced parentheses, and specialized functions are used in the correct context (e.g. LLMMAP must receive a quoted string and table reference as arguments[3]). However, the context-free grammar is unable to verify semantic constraints, such as ensuring that the table passed to LLMMAP exists within the current database.

Shown in Figure 6, despite the CFG preventing many syntax errors that would otherwise have occurred, the downstream denotation accuracy is not consistently improved. Specifically, smaller models that are more prone to simple synactic mistakes benefit more from the CFG guide, whereas the large Llama-3.3-70b-Instruct is actually harmed by the constraints. We hypothesize this may be due to errors in the Lark CFG or guidance's use of fast-forward tokens[4], though leave deeper exploration of this to future work.

[3]Since these aren't semantic constraints, this really only enforces that it *looks like* a table reference

[4]https://github.com/guidance-ai/llguidance/blob/main/docs/fast_forward.md

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'.

Question: {{question}}

Output datatype: {{return_type}}

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


Answer:
```

Figure 3: **Prompt for the LLMQA function.**

LLMMAP Prompt

```
Complete the docstring for the provided Python function.
The output should correctly answer the question provided for each input value.
On each newline, you will follow the format of f({value}) == answer.

def f(s: str) -> bool:
    """Is an NBA team?
    Args:
        s (str): Value from the "w.team" column in a SQL database.

    Returns:
        bool: Answer to the above question for each value `s`.

    Examples:
        ```python
        # f() returns the output to the question 'Is an NBA team?'
        f("Lakers") == True
        f("Nuggets") == True
        f("Dodgers") == False
        f("Mets") == False
        ```
        """
        ...

def f(s: str) -> {{return_type}}:
    """{{question}}
    Args:
        s (str): Value from the {{table_name}}.{{column_name}} in a
            SQL database.

    Returns:
        {{return_type}}: Answer to the above question for each value `s`.

    Examples:
        ```python
        # f() returns the output to the question '{{question}}'
        f({{value}}) =
```

Figure 4: **Prompt for the LLMMAP function.** The instruction and few-shot example(s) are prefix cached, enabling quick batch inference over the sequence of database values.
```

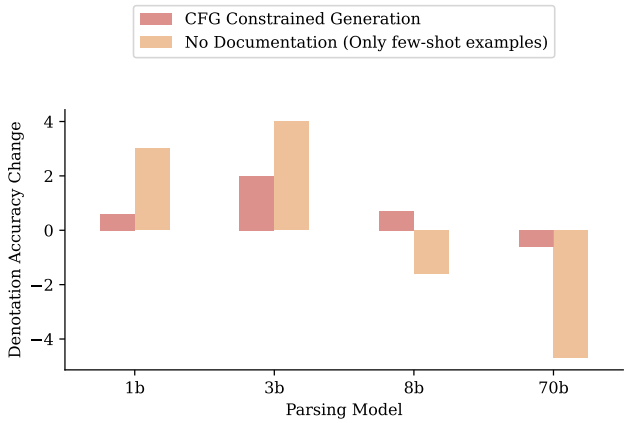

Figure 5: **Impact of ablations for different parsing models.** While larger models show decreased performance when removing descriptive documentation, smaller models exhibit moderate gains. Results shown use a Llama-3.1-8b-Instruct as a function executor in the "Type Hints + Constrained Decoding" setting, described in Section 2.3.

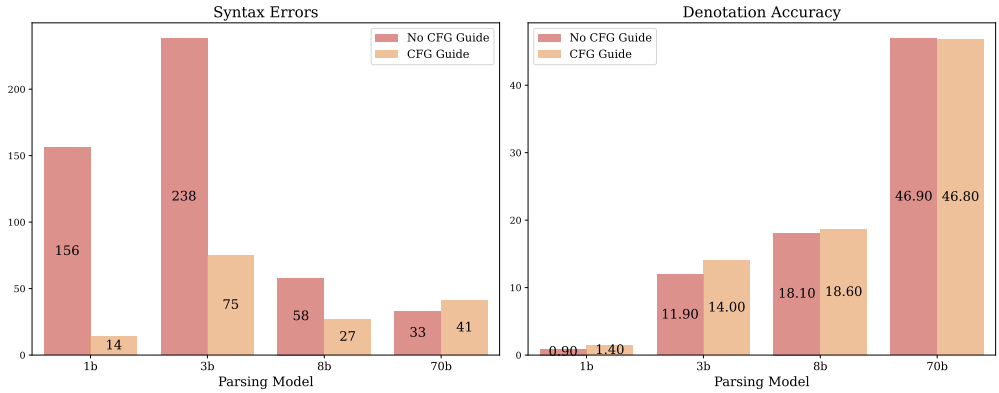

Figure 6: **Decreasing syntax errors isn't strongly correlated with improved downstream performance**. The real difficulty of semantic parsing lies in the semantic alignment, not shallow syntactic grammaticality. Results shown use a Llama-3.1-8b-Instruct as a function executor in the "Type Hints + Constrained Decoding" setting, described in Section 2.3.

Error Type	**Count**
Empty LLMQA Context	48
Generic SQLite Syntax	13
BlendSQL Column Reference Error	13
Hallucinated Column	11
Tokenization Error	6
Hallucinated Table	4
F-String Syntax	1
Misc.	1

Table 2: **Categorization of execution errors raised by programs generated by Llama-70-Instruct.** Results shown are from 1000 examples from the HybridQA validation set.

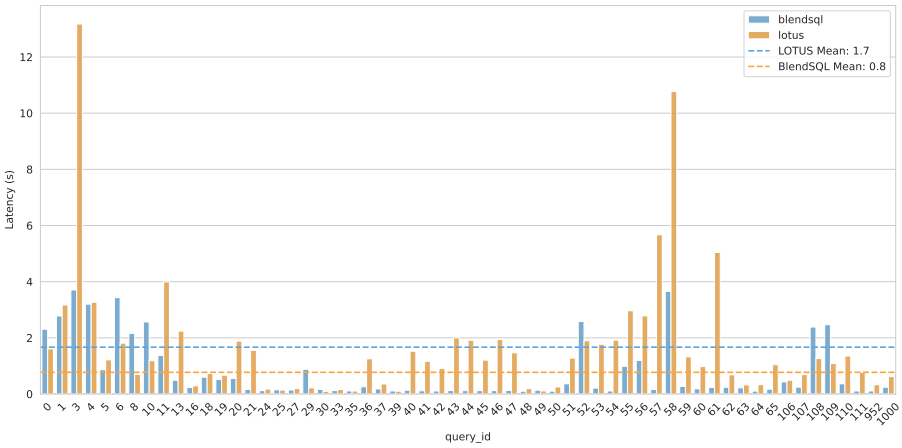

Figure 7: **Sample level latency of declarative LLM programs across question types on TAG-Benchmark.** Results shown are averaged across 5 runs on an RTX 5080.

A.2 Runtime Benchmark

We validate both the efficiency and expressivity of BlendSQL as an intermediate representation by comparing against LOTUS (Patel et al., 2024) on the TAG-benchmark questions. LOTUS is a declarative API for data processing with LLM functions, whose syntax builds off of Pandas (pandas development team, 2020). TAG-Bench is a dataset built off of BIRD-SQL dataset (Li et al., 2023) for text-to-SQL. The annotated queries span 5 domains from BIRD, and each requires reasoning beyond what is present in the given database. For example, given the question "How many test takers are there at the school/s in a county with population over 2 million?", a language model must apply a map operation over the `County` column to derive the estimated population from its parametric knowledge. The average size of tables in the TAG-Bench dataset is 53,631 rows, highlighting the need for efficient systems.

Figure 7 shows the sample-level latency of LOTUS and BlendSQL programs on 60 questions from the TAG-Bench dataset. Using the same quantized Llama-3.1-8b and 16GB RTX 5080, latency decreases by 53% from 1.7 to 0.76 seconds, highlighting the efficiency of BlendSQL, in addition to the expressivity of the two simple map and reduce functions.

Both systems are evaluated on the same RTX 5080 16GB GPU. The max context length is set to 8000 for all evaluations.

BlendSQL Setup We use blendsql==0.0.48 for our runtime experiment. The latency of the programs at tag_queries.py are measured. We use llama-cpp-python version 0.3.16, pointing to llama.cpp@4227c9be4268ac844921b90f31595f81236bd317. The Q4_K_M quant from bartowski/Meta-Llama-3.1-8B-Instruct-GGUF model is used.

LOTUS Setup We use lotus-ai==1.1.3 for our runtime experiment. The latency of the programs written by the authors at hand_written.py are used. Generation is performed using ollama version 0.6.7, which uses llama.cpp@e54d41befcc1575f4c898c5ff4ef43970cead75f as its backend. The Q4_K_M quant, referenced by ollama via llama3.1:8b, is used.

B Example Programs

Below contains a short sample of BlendSQL queries generated by Llama-3.3-70b-Instruct.

```
/* What is the difference in time between José Reliegos of Spain and the
    person born 5 September 1892 who competed at the 1928 Olympics ? */
SELECT
    CAST(REPLACE("time", ':', '.') AS REAL) -
    (SELECT CAST(REPLACE("time", ':', '.') AS REAL)
     FROM w
     WHERE athlete = {{
        LLMQA(
            'Who was born on 5 September 1892 and competed at the 1928
    Olympics?'
        )
    }})
FROM w
WHERE athlete = 'josé reliegos'

/* Which # 1 ranked gymnast is the oldest ? */
WITH t AS (
    SELECT gymnasts FROM w
    WHERE rank = 1
) SELECT gymnasts FROM t
ORDER BY {{LLMSearchMap('What year was {} born?', t.gymnasts)}} ASC LIMIT 1

/* What city is the university that taught Angie Barker located in ? */
SELECT {{
    LLMQA(
        'In what city is {}?',
        (SELECT institution FROM w WHERE name = 'angie barker')
    )
}}

/* In which city is this institute located that the retired American
    professional basketball player born on November 23 , 1971 is affiliated
    with ? */
SELECT {{
    LLMQA(
        'In which city is {} located?',
        (
            SELECT "school / club team" FROM w
            WHERE player = {{
                LLMQA(
                    'What is the name of the retired American professional
                    basketball player born on November 23, 1971?'
                )
            }}
        )
    )
}}

/* How many players whose first names are Adam and weigh more than 77.1kg?
    */
SELECT COUNT(*) FROM Player p
WHERE p.player_name LIKE 'Adam%'
AND p.weight > {{LLMQA('What is 77.1kg in pounds?')}}

/* Of the 5 racetracks that hosted the most recent races, rank the
    locations by distance to the equator. */
WITH recent_races AS (
    SELECT c.location FROM races ra
    JOIN circuits c ON c.circuitId = ra.circuitId
    ORDER BY ra.date DESC LIMIT 5
) SELECT * FROM VALUES {{
    LLMQA(
        'Order the locations by distance to the equator (closest ->
    farthest)',
        options=recent_races.location,
        quantifier='{5}'
    )
}}
```