Optimizing LLM Queries in Relational Data Analytics Workloads

Shu Liu [*] [1] Asim Biswal [*] [1] Amog Kamsetty [1] Audrey Cheng [1] Luis Gaspar Schroeder [1] [2]
Liana Patel [3] Shiyi Cao [1] Xiangxi Mo [1] Ion Stoica [1] Joseph E. Gonzalez [1] Matei Zaharia [1]

Abstract

Batch data analytics is a growing application for Large Language Models (LLMs). LLMs enable users to perform a wide range of natural language tasks, such as classification, entity extraction, and translation, over large datasets. However, LLM inference is highly costly and slow: for example, an NVIDIA L4 GPU running Llama3-8B can only process 6 KB of text per second, taking about a day to handle 15 GB of data; processing a similar amount of data costs around $10K on OpenAI's GPT-4o. In this paper, we propose novel techniques that can significantly reduce the cost of LLM calls for relational data analytics workloads. Our key contribution is developing efficient algorithms for reordering the rows and the fields within each row of an input table to maximize key-value (KV) cache reuse when performing LLM serving. As such, our approach can be easily applied to existing analytics systems and serving platforms. Our evaluation shows that our solution can yield up to 3.4× improvement in job completion time on a benchmark of diverse LLM-based queries using Llama 3 models. Our solution also achieves a 32% cost savings under OpenAI and Anthropic pricing models.

1 Introduction

One of the most popular applications of large language model (LLM) batch inference is data analytics. A growing number of analytics platforms now support LLM invocations for complex analytical tasks. For instance, leading database vendors, such as AWS Redshift (aws), Databricks (dat), and Google BigQuery (goo), have integrated LLM functionality into their SQL APIs. Similarly, DataFrame libraries and programming frameworks (Chase, 2022; Patel et al., 2024) offer LLM support for querying relational (table-based) data. With these new APIs, users can write queries like the following:

```
SELECT user_id, request,
↪  support_response,
  LLM('Did {support_response} address
  ↪  {request}?', support_response,
  ↪  request) AS success
FROM customer_tickets
WHERE support_response <> NULL
```

where the LLM is invoked for each row in the customer ticket table to analyze whether the customer service requests are effectively addressed. Increasingly, analysts wish to leverage LLMs in such queries for tasks including classification, entity extraction, summarization, and translation (dat). Going forward, we will refer to queries that invoke LLMs over relational data as *LLM queries*.

Unfortunately, applying LLMs to real-world datasets (which can contain millions of rows) incurs significant computational and monetary costs. Accordingly, there has been growing research on LLM inference optimization. In particular, recent work (Kwon et al., 2023; Zheng et al., 2023; Ye et al., 2024; Juravsky et al., 2024; Gim et al., 2024) leverages prompt caching, a technique that stores the attention states of frequently reused prompt segments in GPU memory, known as key-value (KV) cache (Vaswani et al., 2023). Reusing cached state whenever a similar *prefix* of prompts appears again can significantly reduce inference latency (Zheng et al., 2023). In addition, prompt reuse also brings economic benefits. Recently, providers like OpenAI, Anthropic, and Google Gemini (OpenAI; ant, 2024; gem, 2024) have introduced prompt caching as a service, charging 2–10× less for cached prompts. Therefore, maximizing *prefix hits* in the prompt KV cache is crucial for reducing both LLM request time and monetary costs.

However, simply invoking LLMs over relational data within analytical engines and connecting to a backend inference server with a prompt cache often results in low cache hit rates. This approach fails to exploit relational workloads to fully maximize cache reuse.

In this work, we identify and present solutions to optimize relational data analytics workloads for offline LLM inference. In particular, given an LLM query, we propose **re-**

[*] Equal contribution [1] UC Berkeley [2] Technical University of Munich [3] Stanford University. Corresponding author: Shu Liu <lshu@berkeley.edu>.

Proceedings of the 6^{th} *MLSys Conference*, Santa Clara, CA, USA, 2025. Copyright 2025 by the author(s).

quest reordering at the row and field granularity of the relational data. Our key insight is that, with oracular knowledge of all requests to be sent to the LLM, we can reorder both the requests and the fields inside each request to increase the number of cache hits. In real datasets, there can be many sharing opportunities across rows and fields. For example, joining feature tables, referencing popular items, or repeating similar context in RAG queries (Lewis et al., 2021). These common patterns lead to repeating values in different fields, leaving rooms for significantly improving cache hit rates by optimizing request order and format.

Finding the optimal ordering of requests is challenging due to the exponential number of choices to order the fields and rows of data in a query. For a table with n rows and m fields, there are $n! \times (m!)^n$ potential orderings. One way to reduce this search space is to apply the same field ordering across all rows. However, as we show in Sec 3.2, this can reduce the prefix hit count by up to a factor of m compared to reordering fields on a more fine-grained, per-row basis. To support per-row field reordering, we introduce **Optimal Prefix Hit Recursion (OPHR)**, an algorithm that divides the table into smaller subtables and reorders each subtable to maximize the prefix hits. While OPHR achieves high hit rates, its complexity is exponential, which makes it impractical for large datasets. To address this challenge, we propose **Greedy Group Recursion (GGR)**, an approximate algorithm that leverages functional dependencies (such as primary and foreign key relationships from the data schema) and table statistics, which are readily available in many databases and analytics systems, to reduce the search space. In particular, functional dependencies help identify correlated fields, reducing the number of fields that need to be reordered at each step, thus decreasing the solver runtime. In addition, GGR leverages the cardinality and length statistics to efficiently approximate the greedy objective.

We implement our techniques in Apache Spark (Zaharia et al., 2012) and use vLLM (Kwon et al., 2023) as the model serving backend. Due to the lack of standard workloads in this area, we build a benchmark suite of 16 LLM queries of different types, spanning selection, projection, multi-LLM invocations, and retrieval-augmented generation (RAG) queries (Lewis et al., 2021). We evaluate these queries on recommendation and question-answering datasets such as Amazon Product Reviews, Rotten Tomatoes Movies, BIRD, Stanford Question Answering Dataset, Public Domain MusicXML, RateBeer Reviews, and Fact Extraction and VERification datasets (He & McAuley, 2016; Pang & Lee, 2005; Li et al., 2024; Rajpurkar et al., 2016; Long et al., 2024; Thorne et al., 2018). Our techniques show 1.5 – 3.4× speed-up in end-to-end query latency and reduce costs by up to 32% on proprietary model APIs, while preserving query semantics. In summary, our contributions are as follows:

- We identify significant opportunities to speed up LLM-based batch data analytics through reordering rows and fields of input tables.
- We introduce an optimal reordering algorithm (OPHR) that maximizes prefix sharing but with exponential complexity. We propose an efficient greedy algorithm (GGR) that approximates OPHR by leveraging functional dependencies and table statistics. We show that a fixed field ordering can yield as much as m (number of fields) times worse cache hits than our solution.
- We present an LLM query benchmark consisting of 16 queries and 7 real-world datasets to represent a range of retrieval and processing tasks. Our evaluation with Llama3-8B and 70B shows up to a 3.4× speedup in end-to-end query latency compared to naive orderings. With OpenAI and Anthropic prefix cache pricing models, our techniques reduce costs by up to 32%.

2 Background and Motivation

This section provides a brief overview of the inference process and the key components of the LLM architecture.

LLM inference. LLMs are made up of autoregressive Transformer models (Vaswani et al., 2023), which generate text token by token until a termination token or a length limit is reached. LLM inference consists of two stages: (i) the prefill stage, where the model processes the input prompts, and (ii) the decoding stage, where it generates output sequentially, as each token depends on all previously generated tokens through a chain of conditional probabilities. LLM inference engines (e.g., vLLM (Kwon et al., 2023), TGI (Huggingface, 2023), TensorRT-LLM (NVIDIA, 2023b)) typically batch requests continuously (Yu et al., 2022) to improve throughput. The intermediate computed state for all tokens involved is stored in memory. This token state is cached as key and value vectors in the *key-value (KV) cache*, consuming up to 800KB per token for a 13B Model (Kwon et al., 2023). A typical request (involving 2,000 tokens) can require up to 1.6 GB of memory. Despite batching (up to 32 requests), inference remains compute-intensive, with current speeds limited to 2,000 tokens/s per GPU, making LLM performance a bottleneck for many analytical tasks.

Prompt KV cache. Efficient KV cache management is critical for high LLM serving throughput. Recent work improves cache utilization by reusing tokens across requests with shared prefixes (Zheng et al., 2023). For example, if two requests share a *prefix* in prompts, the first will already have performed some computation on the input tokens and cached results in the KV cache during the prefill phase. The subsequent request can then reuse these cached values, avoiding redundant computation of the shared tokens.

Improving KV cache hit for analytics workloads. Real-world relational databases often exhibit diverse repetitive data patterns. Columnar storage systems like C-Store and Parquet (Stonebraker et al., 2018) exploit repeated values across fields for compression, while techniques like run-length encoding (RLE), multi-relational data mining, and correlation analysis (Lemire & Kaser, 2011; Džeroski, 2003; Ilyas et al., 2004) leverage diverse data relationships to optimize query execution. Relational queries also create data groupings based on access patterns. Techniques such as database cracking and multi-dimensional clustering (MDC) (Idreos et al., 2007; Chen et al., 2012), including Delta Lake Z-order (Armbrust et al., 2020), reorganize data based on query patterns to optimize performance.

These structural repetitive patterns present an opportunity for *prefix KV cache* sharing in an LLM query. In our setting, an LLM is invoked once per row in a relational table, resulting in a batch of model requests from a single LLM query. Since the full table structure and content are known in advance, we can reorder these requests to maximize shared prefixes and reduce redundant computation during inference. Our goal is to maximize the *prefix hit count* – the sum of the length of token prefixes reused from the KV cache.

Our Approach: Request Reordering. We leverage table information to enhance the KV cache hit rate. Specifically, we introduce algorithms that reorder requests of an LLM query and fields within each request to maximize prefix sharing. Our algorithm leverages functional dependencies and table statistics to reduce runtime while finding near-optimal orderings that maximize prefix hit count.

3 Problem Setup

This section introduces the problem setup of maximizing prefix hits in the prompt cache (Sec 3.1) and highlights cases where naive fixed field ordering can result in significantly lower hit rates (Sec 3.2).

3.1 Setup and Objective

In this work, we consider a generic LLM operator that takes the text of the prompt as well as a *set* of expressions listing one or more fields $\{T.a, T.b, T.c\}$ or $\{T.*\}$ of the table T. This simple design can be easily implemented in most analytics systems and enables us to dynamically reorder fields within these expressions to optimize for cache efficiency. Consider the following example query:

```sql
SELECT LLM("Summarize: ", pr.*)
FROM (
    SELECT review, rating, description
    FROM reviews r JOIN product p ON
     ↪  r.asin = p.asin
) AS pr
```

This query sends a list of rows, each with fields *review*, *rating*, and *description* from table *pr* to the LLM for a summarization task.

Objective The goal of request scheduling is to **maximize** the *prefix hit count* by optimizing the order of fields and rows of an input table with n rows and m fields. Each row may have a different field order. We represent a request schedule as a list of tuples L, where each tuple in L represents a row in the table, and the tuple elements contain the field values. We adjust the row order by rearranging the tuples in L, and adjust the field order for that row by rearranging the elements within each tuple. We pass each tuple alongside the user question to form an input request to the LLM.

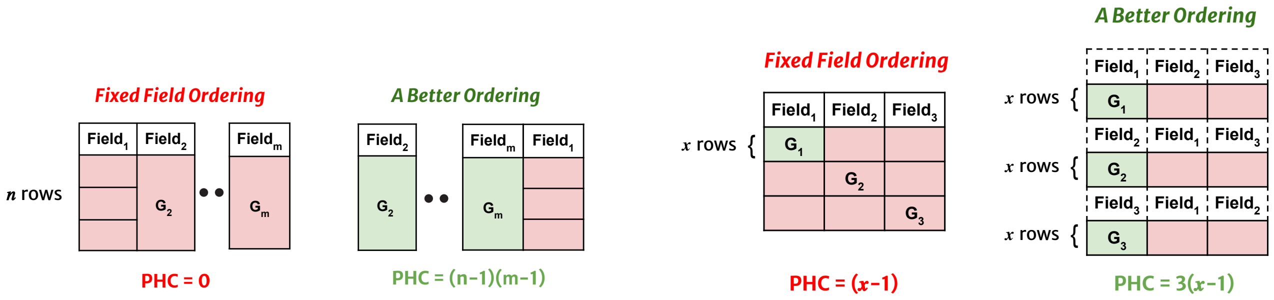
We define the *prefix hit count* (PHC) of L as the number of consecutive field cell values shared with the previous row starting from the first cell, summing over all n rows. Each cell value must exactly match the corresponding cell of the previous row (cannot be a substring), and cell values past the first must match consecutively (must be a prefix). Formally, a cell in the list of tuples is denoted as $L[r][f]$, indicating the value in tuple r at position f. Then, the PHC for a list of tuples L with n rows and m fields is given by:

$$\text{PHC}(L) = \sum_{r=1}^{n} \textit{hit}(L, r) \tag{1}$$

Here, the function $\textit{hit}(L, r)$ represents the prefix hit count for a single row r in L. For simplicity, we assume that the input list is sorted. For each row r, the function checks if the value in each field f matches any previously seen value in the same field of the previous row $r - 1$. If all previous fields match, the hit count is the sum of the squares of the lengths of the values in those fields until a mismatch occurs. The squared lengths reflect the quadratic complexity of token processing in LLM inference, where each token computation depends on every preceding token and increases computational cost quadratically with input length.

$$\textit{hit}(L, r) = \max_{0 \le c < m} \begin{cases} \sum_{f=1}^{c} \text{len}(L[r][f])^2 & \text{if } \forall f \le c, \\ & L[r][f] = \\ & L[r-1][f] \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

To simplify the design, we make two assumptions. First, we make a common assumption that at least one tuple (row) can fit into the KV cache to allow reuse. Second, we assume that a cell value only counts as a hit if it exactly matches a previously seen value – substring matches are not allowed. This is a reasonable assumption in relational databases, where exact value repetition is common and extensively leveraged by storage optimization techniques like run-length encoding (Lemire & Kaser, 2011). Column-oriented storage systems such as C-Store and Parquet (Stonebraker et al., 2018) also benefit from many exact repetitions in columnar data. These assumptions simplify design and, as shown in Sec. 6, demonstrate good real-world performance.



(a) Distinct Values in the First Field

(b) Group of Identical Values in each Field, $m = 3$

Figure 1. **Case Study of Fixed Field Ordering:** Comparing the PHC of a fixed field ordering to a better ordering in two scenarios. Green boxes denote cache hits; red boxes indicate cache misses. A box labeled G_i signifies consecutive rows share the same values in Field i; otherwise, assume values are distinct. Fig 1a shows fixed field ordering can be $(n-1)(m-1)$ worse in terms of PHC compared to an optimized ordering. Fig 1b shows fixed field ordering can be m times worse in PHC compared to an optimized ordering, where $m = 3$.

3.2 Case Study: Fixed Field Ordering

Relational data typically uses a fixed field order across rows, which can lead to lower hit rates in real-world databases with diverse data patterns (Sec 2). In fact, we show that using a fixed order can reduce the hit rate by up to m times compared to a per-row field reordering. To illustrate this, we begin with a simple example and extend it to show the potential impact of a naive fixed field ordering on prefix hit counts (PHC). First, consider a table T with n rows and m fields arranged in an arbitrary (default) order. For simplicity, we assume each value is of length one. In many cases, certain fields of an input table may contain highly unique values, like timestamps or IDs. In the worst case, suppose the first field of the table contains only unique values (Fig 1a), and the remaining $m-1$ fields contain the same value across all rows. This ordering yields 0 PHC. A more optimized ordering (Fig 1a) will place the other $m-1$ fields first, yielding a PHC of $(n-1) \times (m-1)$. Each of the $n-1$ rows has a hit after the initial cold miss, and the length of each hit is $m-1$.

Now consider a scenario where the table contains groups of consecutive rows with identical values (not necessarily in the same field). Suppose each field i has one such group with x consecutive rows of the same value, with other $n-x$ rows having distinct values, where n is the number of rows. We denote the group appearing in the Field$_i$ as G_i, so we have $G1, ..., G_m$ groups, where m is the number of fields. Now, consider a scenario where groups in consecutive fields are non-overlapping across rows, as shown in Fig 1b. With fixed field reordering, the PHC of this structure is limited to $x-1$ no matter which field is prioritized. By contrast, a better ordering would rearrange the field order for different rows to prioritize groups with shared values. Fig 1b references a table with $3x$ rows and 3 fields. A naive fixed field ordering for all rows will result in misses on two groups, each with x rows in Field$_2$ and Field$_3$. However, a better ordering will pick different Field$_j$ to prioritize for different rows, resulting in a 3 times higher hit rate of $3(x-1)$.

In the above scenario, PHC improvements from optimized field ordering can reach m times that of a fixed field ordering. For example, there can be multiple (instead of just one) such groups in each field. If each field contains roughly the same number of such groups, dynamic reordering for different rows can achieve as much as an m-fold improvement in PHC over fixed field ordering. Under the OpenAI pricing model, which charges half price for cached prompts, optimizing field order for a table with nine fields could yield 42% in cost savings compared to fixed field ordering, assuming fixed ordering has a 10% hit rate (e.g., $\frac{(x-1)}{n} = 10$). This example highlights the benefits of a more complex field reordering mechanism for different rows on PHC.

4 Recursive Request Reordering

We now introduce our algorithms that re-arrange fields to maximize prefix sharing in the KV cache. We present an optimal recursive reordering algorithm that maximizes PHC (Sec 4.1) and introduce a greedy algorithm that efficiently approximates the optimal algorithm (Sec 4.2).

4.1 Optimal Prefix Hit Recursion (OPHR)

Our Optimal Prefix Hit Maximization (OPHR) algorithm is a recursive algorithm that finds the *optimal* PHC for a given table T by considering all possible ways to split the table into a group of cells with the same value and two sub-tables. The algorithm takes as input a table T and computes the optimal PHC S along with a reordered list of tuples L. If T only has one row or field, OPHR computes PHC and trivially returns the sorted T.

In the recursive case, for each field c in T, the algorithm identifies all distinct values v in the field and the rows R_v for which the field value is v. For each distinct value v, the table is split into two sub-tables: one of T excluding rows R_v and one of R_v excluding field c. PHC for the currently selected value v is calculated as the sum of the PHC of the sub-tables and the PHC contribution of v. OPHR evaluates

Algorithm 1 Greedy Group Recursion (GGR)

```
1:  **Input:** Table $T$, Functional Dependency $FD$
2:  **Output:** Prefix Hit Count $S$, Reordered List of Tuples $L$

3:  **function** HITCOUNT($v, c, T, FD$)
4:      $R_v \leftarrow \{i \mid T[i, c] = v\}$
5:      inferred_cols $\leftarrow \{c' \mid (c, c') \in FD\}$
6:      tot_len = $\text{len}(v)^2 + \sum_{c' \in \text{inferred\_cols}} \frac{\sum_{r \in R_v} \text{len}(T[r,c'])}{|R_v|}$
7:      **return** tot_len $\times (|R_v| - 1)$, $[c]$ + inferred_cols
8:  **end function**

9:  **function** GGR($T$, $FD$)
10:     **if** $|T|_{rows} = 1$ **then**
11:         return 0, $[T[1]]$
12:     **end if**
13:     **if** $|T|_{cols} = 1$ **then**
14:         $S \leftarrow \sum_{v \in \text{distinct}(T[,1])} \text{HITCOUNT}(v, 1, T)$
15:         **Return** $S, sort([T[i] \mid i \in 1 \ldots |T|_{rows}])$
16:     **end if**
17:     $max\_HC, b\_v, b\_c, b\_cols \leftarrow -1, \text{None}, \text{None}, []$
18:     **for** $c \in \text{columns}(T)$, $v \in \text{distinct}(T[, c])$ **do**
19:         $HC, cols \leftarrow \text{HITCOUNT}(v, c, T, FD)$
20:         **if** $HC > max\_HC$ **then**
21:             $max\_HC, b\_v, b\_c, b\_cols = HC, v, c, cols$
22:         **end if**
23:     **end for**
24:     $R\_v \leftarrow \{i \mid T[i, b\_c] = b\_v\}$
25:     $A\_HC, L\_A \leftarrow \text{GGR}(T[\text{rows} \setminus R\_v, \text{cols}], FD)$
26:     $B\_HC, L\_B \leftarrow \text{GGR}(T[R\_v, \text{cols} \setminus b\_cols], FD)$
27:     $C\_HC, \_ \leftarrow \text{HITCOUNT}(b\_v, b\_c, T, FD)$
28:     $S \leftarrow A\_HC + B\_HC + C\_HC$
29:     $L \leftarrow [[b\_v] + L_A[i] \mid i \in 1 \ldots |R\_v|] + L\_B$
30:     **return** $S, L$
31: **end function**

32: **return** GGR($T$, $FD$)
```

all possible groups of distinct values in each field and selects the value that yields the maximum PHC.

Notably, the OPHR algorithm has exponential complexity with respect to the number of rows and fields due to its recursive nature and the combinatorial explosion of possible distinct value groupings (we present a more efficient algorithm in Sec 4.2).

Optimality Proof In the base case, the OPHR algorithm trivially computes the best PHC: for the single row case, the PHC is 0; for the single field case, the PHC is the sum of the squared lengths of distinct values multiplied by their occurrences minus one, which accounts for the initial miss when a value is seen the first time. Next, we prove optimality by induction. For the inductive case, assume that the OPHR algorithm is optimal for any table with $k \leq n$ rows and $l \leq m$ fields. For a table T with $n + 1$ rows and $m + 1$ fields, the algorithm iterates through each field c. For each distinct value v in field c, we split T into two sub-tables: T_A (rows not containing v), and T_B (rows containing v but excluding field c). Based on the inductive hypothesis, OPHR optimally computes PHC for both sub-tables because it is optimal for tables with fewer rows and fields. The PHC for T is the sum of PHC for T_A and T_B, plus the contribution of v. When the distinct value v is used to partition the table, its full contribution to the PHC is captured. If the table were not split based on distinct values, this contribution could be fragmented or lost due to non-contiguous groupings, leading to suboptimal PHC. Thus, the OPHR algorithm ensures optimal reordering by selecting the best from all possible configurations.

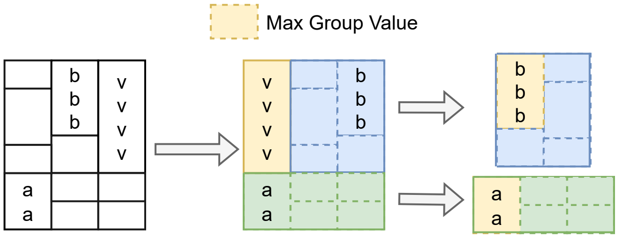


Figure 2. GGR picks the group with the maximum hit count at each step and calculates PHC as the sum of PHC of the elected group values (yellow box), the sub-table T excluding rows R_v (green box), and the sub-table of rows R_v excluding the field where the value is located in (blue box).

4.2 Greedy Group Recursion (GGR) Algorithm

Due to the computational complexity of the OPHR, we propose a Greedy Group Recursion (GGR) algorithm (Algorithm 1) that approximates OPHR. The GGR algorithm takes an input table T and returns the PHC S along with a reordered list of tuples L. It has the same base case as the OPHR algorithm if T only has one row or one field. At a high level, the GGR algorithm recursively selects the value b_v with the maximum prefix hit count (lines 3-8) at each recursion step (lines 17-23) rather than iterating through all possible distinct values in the entire table. It then prioritizes the field b_c where this b_v is in, splits the table into groups of cells of the same values and recurses on the two sub-tables (lines 24-26) and calculates the total PHC as the sum of PHC of the subtables and contributions of b_v (line 28) similar to the OPHR algorithm.

Since GGR does not iterate through all possible distinct values but instead selects the one that gives the highest hit count at each step, the number of recursive calls is significantly reduced (i.e. the maximum depth of recursion is $O(\min(n, m))$, where the algorithm reduces dimensions of the table at each recursive step). However, at each recursive step, the cost of scanning to determine distinct values can result in quadratic complexity in terms of table size.

4.2.1 *Functional Dependencies*

We leverage functional dependencies to reduce the number of fields the GGR algorithm needs to consider at each recursion step. This insight helps improve both the approximation and efficiency of the algorithm, bringing it closer to the optimal solution without the need for extensive backtracking as in the OPHR algorithm. A functional dependency (FD) is a constraint between two sets of attributes in a relation from the data. For example, let R be a relation schema and let X and Y be nonempty sets of attributes in R. We define an instance r of R that satisfies the FD $X \leftrightarrow Y$ if for every pair of tuples t_1 and t_2 in r: if $t_1.X = t_2.X$ then $t_1.Y = t_2.Y$ and vice versa. In our GGR algorithm, FDs help narrow down the fields that must be considered at each recursion step. Specifically, when a value v in field f is selected for a given row, all fields functionally dependent on f are ordered directly besides f in the final ordering for that row (lines 5-6). As an example, if $R(A, B, C)$ is a table with attributes (fields) A, B, C where we have an FD $A \leftrightarrow C$, field C is not in consideration in our recursive steps when A has already been included in the prefix.

4.2.2 Table Statistics

To further reduce the algorithm runtime, we introduce an early stopping mechanism that halts recursion by specific recursion depth (row-wise sub-table recursion, column-wise sub-table recursion) or when a threshold HITCOUNT score calculated using table statistics is not exceeded. These statistics are generally widely available, such as the number of unique entries (i.e., cardinality) and the distribution of length of values for each field. With this information, our GGR algorithm estimates a HITCOUNT score for each field c with $\texttt{HITCOUNT}(C) = \texttt{avg}(\texttt{len}(c))^2$. This score denotes the expected contribution of a field to the PHC, accounting for the average length of the values and their frequency. Using these statistics, the algorithm can prioritize fields more likely to contribute to the PHC. Additionally, we can further improve the quality of the solution by establishing a fixed field ordering for the subtables using table statistics once the recursion stops. Early termination and falling back to table statistics allows GGR to avoid scanning the table and performing recursion on real-world workloads.

4.2.3 Achieving Optimal PHC

While our GGR approximates the OPHR algorithm, it can achieve optimal PHC in certain cases. When the table has only one row or one field, GGR matches OPHR by construction. When functional dependencies are accurate and cover all the fields of a table, GGR can also identify the optimal solution. For instance, if one field A functionally determines all other fields, then GGR prioritize groups of values in A due to the accumulated HITCOUNT score (line 3 in Algorithm 1), capturing key correlations early and producing the optimal reordering. However, when fields tie in HITCOUNT, GGR may be suboptimal, as it lacks the exhaustive search used by OPHR to resolve such ties. We show more empirical results in real-world datasets comparing PHC between GGR and OPHR in Appendix D.1.

5 Implementation

We implement our algorithms in approximately 1.3K lines of Python code and evaluate them with PySpark (Armbrust et al., 2015), which is backed by Apache Spark (Zaharia et al., 2012) – a widely adopted large-scale data processing engine in industry. The *LLM operator* implements the actual LLM inference by calling a configurable LLM endpoint. We implement this function as a UDF in PySpark. It takes in a system prompt, a query prompt, and a single row of data as input (Appendix C). The row and field orders are input based on the ordering returned by the reordering function. The operator is also responsible for *prompt construction*. Specifically, it converts the user-provided question and the table row values into a prompt that an LLM can parse. We use JSON formatting to encode row values to indicate the relationship between field names and values to the LLM.

6 Evaluation

In this section, we evaluate the effectiveness of our optimizations within a constructed benchmark suite of queries. We aim to answer the following questions:

1. How does our request reordering optimization impact query latency and costs across different LLM query types and datasets?
2. How does the request reordering algorithm influence LLM accuracy for different models?
3. What is our algorithm solver time, and how does that compare to end-to-end query latency?

6.1 Evaluation Benchmark

Given the lack of standard benchmarks for LLM queries, we construct a benchmark suite to represent real-world data retrieval and processing tasks (Sec 6.1.1). We define a range of query types (Sec 6.1.2) over datasets from various sources to assess the impact of LLMs in relational analytics.

6.1.1 Datasets

Dataset	n_{rows}	n_{fields}	input$_{\text{avg}}$	output$_{\text{avg}}$	Query Type
Movies	15000	8	276	$\{2, 29, 16, 2\}$	T1-T4
Products	14890	8	377	$\{3, 107, 62, 2\}$	T1-T4
BIRD	14920	4	765	$\{2, 43\}$	T1, T2
PDMX	10000	57	738	$\{2, 72\}$	T1, T2
Beer	28479	8	156	$\{2, 38\}$	T1, T2
SQuAD	22665	5	1047	11	T5
FEVER	19929	5	1302	3	T5

Table 1. Datasets: n_{rows} and n_{fields} denote the number of rows and fields, respectively. input$_{\text{avg}}$ and output$_{\text{avg}}$ represent average input and output token lengths. Query Type is detailed in Sec 6.1.2. Since input$_{\text{avg}}$ remains consistent across query types, we report a single overall average, while output$_{\text{avg}}$ varies, with each bracketed value corresponding to a specific query type.

We build our benchmark suite on 7 commonly used recommendation and natural language processing datasets, shown in Table 1. These datasets vary in the number of rows, fields, average input/output token lengths, and appropriate query types (Sec 6.1.2). The datasets include Rotten Tomatoes Movie Reviews (Movies) (Pang & Lee, 2005), Amazon Product Reviews (Products) (He & McAuley, 2016), BIRD (Li et al., 2024)[1], Public Domain MusicXML (PDMX) (Long et al., 2024), RateBeer Reviews (Beer) (McAuley et al., 2012), Stanford Question Answering Dataset (SQuAD) (Rajpurkar et al., 2016), and Fact Extraction and Verification (FEVER) (Thorne et al., 2018). Details on the fields are in the Appendix B.

6.1.2 LLM Queries

Our evaluation consists of 16 queries across 5 query types corresponding to different real-world use cases, as shown in Table 1. We discuss each query type below and provide details on queries for each dataset in Appendix A and B.

(T1) LLM filter. Filter queries mimic SQL `WHERE` clauses and use LLMs to categorize data. This query type illustrates typical use cases in sentiment analysis, categorization, and content filtering. Given their binary or categorical focus, these queries often yield short outputs (e.g., ”Yes” or ”No”). We construct five filter queries spanning all datasets except for SQuAD and FEVER.

(T2) LLM projection. Projection queries use LLMs to summarize or interpret specific table field(s), similar to a SQL `SELECT` statement. These tasks typically produce longer outputs due to the descriptive nature of the results. We construct five projection queries spanning all datasets except SQuAD and FEVER.

(T3) Multi-LLM invocation. Multi-LLM queries involve sequential LLM calls (e.g., a filter followed by a projection), supporting tasks like multi-step data processing and combining insights. Output lengths vary by task but generally mix short and long responses. We construct two example multi-LLM invocation queries on Movies and Products datasets.

(T4) LLM aggregation. Aggregation queries incorporate LLM outputs into aggregate functions, like averaging sentiment scores given by LLMs for individual reviews. These tasks usually generate concise numeric outputs for analysis (e.g., ratings of 1 to 5), resulting in shorter output lengths similar to filter queries. We construct two example aggregation queries on Movies and Products datasets.

(T5) Retrieval-augmented generation (RAG). RAG queries involve fetching external knowledge as context, such as retrieving relevant document segments before generating answers. We evaluate FEVER and SQuAD datasets, fetching 4 contexts for FEVER and 5 contexts for SQuAD for question answering.

6.1.3 Evaluation Setup

Metrics We evaluate *end-to-end query latency* for each LLM query. We also measure the *monetary cost* of using OpenAI and Anthropic endpoints. Additionally, we hand-label a subset of the LLM filter queries to evaluate the reordering implications for query *accuracy*.

Models We run setups shown in Table 1 using Meta Llama-3-8B-Instruct (lla, 2024). For RAG queries, we use Alibaba-NLP/gte-base-en-v1.5 (Li et al., 2023) to embed the context and use Facebook Similarity Search Library (FAISS) (Johnson et al., 2019) for context retrieval. We also run Llama-3-70B-Instruct (lla, 2024) for LLM Filter queries. For cost results, we evaluate with OpenAI GPT-4o-mini and Anthropic Claude 3.5 Sonnet.

Hardware We evaluate Llama-3-8B-Instruct on a single NVIDIA L4 GPU (GCP g2-standard-4) with 24GB of GPU Memory. We also run a larger model Llama-3-70B-Instruct on 8xL4 GPUs (GCP g2-standard-48). For OpenAI and Anthropic cost experiments, we utilize their API endpoints.

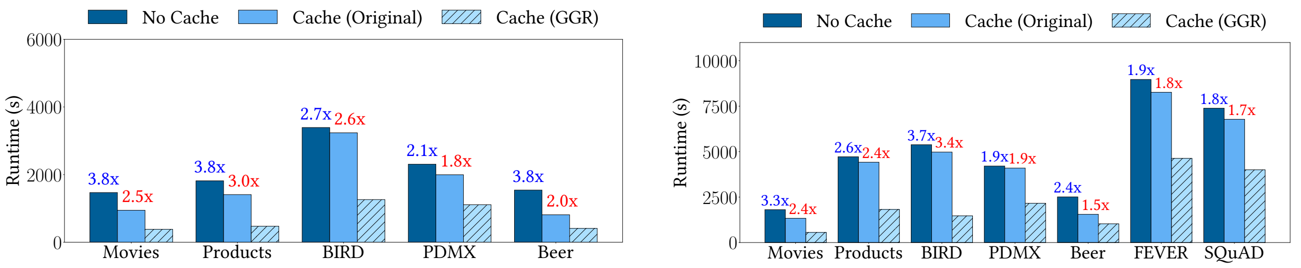
Baselines Our algorithm (*Cache (GGR)*) is compared against two baselines: one without prompt caching (*No Cache*) and another with caching enabled but without reordering (*Cache (Original)*). We do not evaluate the optimal prefix hit recursion algorithm (Sec 4.1) as it is infeasible over large tables (e.g., solving a 10-row table takes several minutes). The algorithm runtime far exceeds the LLM inference time for larger tables for the optimal algorithm.

6.2 End-to-End Benchmark Results

Overview. Fig 3 and Fig 4 show the end-to-end latency results of our techniques on LLM filter, projection, multi-LLM invocation, aggregation, and RAG queries with the Llama-3-8B-Instruct model on a single L4. Our evaluation shows that our approach can achieve 1.5 to 3.4× speedup over Cache (Original) and 1.8 to 3.8× speedup over No Cache across 16 queries. We discuss the evaluation for each query type in detail as below.

LLM filter. This query type uses an LLM operator to filter rows, often producing concise outputs of only a few tokens (see Table 1). Examples include question-answering tasks limited to ’Yes’ or ’No’ responses, or sentiment labels like ’Positive,’ ’Negative,’ or ’Neutral.’ We construct five such queries on the datasets shown in Fig 3a. Our Cache (GGR) approach achieves a 2.1 – 3.8× speed-up over No Cache by caching repeated prefixes from system prompts and input data. Cache (Original) with prompt caching enabled can achieve a modest speedup of 1.03 – 1.9× over No Cache by reusing instruction prompts and repeated values from the default input table. For queries with short decode stages, the primary benefit of prompt caching is the saved prefill computations. Our Cache (GGR) algorithm further reduces end-to-end latency by 1.8 – 3.0× over Cache (Original) through reordering rows and fields in the input table to

[1] We use Posts and Comments table joined by PostID from the BIRD dataset.



(a) Filter Queries

(b) Projection and RAG Queries

Figure 3. End-to-end Result (Filter, Selection, RAG): Our optimizations Cache (GGR) achieve 1.5 – 3.4× speed-up in end-to-end runtime over caching without reordering (Cache (Original)), and 1.8 – 3.8× over No Cache baseline.

maximize prefix reuse.

Most review datasets, such as Movies, Products, and BIRD, contain highly distinct values in the first few default fields due to the joining of reviews with metadata tables. For instance, these tables often begin with a `review_content` field. Our algorithm prioritizes fields with repeated values, like `description` and `product_title`, leading to a 57 – 74% increase in prefix hit rates and a 2.5 – 3× speed-up over the original ordering. PDMX is a dataset containing 57 fields with many unique, lengthy text entries. In this dataset, our algorithm raises the hit rate from an initial 12% to 57%, resulting in a 1.8× reduction in end-to-end latency. This lower speed-up is due to the nature of long input and 43% of cache miss from this dataset even for Cache (GGR). The Beer dataset contains some duplicated values in early fields like `review/profileName` and Cache (Original) can achieve an initial hit rate of 50%. Cache (GGR) can further increase the hit rate by an additional 30% to reach 80% and achieve a 2× speedup.

LLM projection. This query type applies the LLM to the selected data for a specific task, producing longer outputs ranging from 29 to 107 tokens (see Table 1). For example, LLMs can be used to summarize the positive aspects of movies leading to favorable ratings in the Movies dataset. As shown in Fig 3b, for datasets except for SQuAD and FEVER (i.e. RAG queries), Cache (GGR) achieves 2.4× to 3.7× speed-up over No Cache, and 1.5× to 3.4× speed-up over Cache (Original). Notice that as the output token length increases, query execution time across all baselines also grows. In cases where the decode stage dominates, benefits from prefill caching are less pronounced, leading to smaller relative performance gains than with LLM Filter queries with shorter output length. However, for datasets like BIRD and PDMX, which contain long strings, prompt caching saves memory during the decode stage, making the speedup more noticeable with longer decode times.

Multi-LLM invocation. This query type combines Filter and Selection operations, beginning with an initial LLM filter (e.g., selecting positive reviews), followed by an LLM summarization of the filtered table. Applied to the Movies and Products datasets, as shown in Fig 4, Cache (GGR) achieves a 2.7× and 2.8× speedup over the No Cache baseline for Movies and Products, respectively. Compared to Cache (Original), Cache (GGR) attains a speedup of 1.7× and 2.2×. The relative speedup compared to Cache (Original) reduces for both datasets compared to Filter and Projection queries. This is because the first LLM invocation for filtering is over distinct reviews for sentiment analysis, so Cache (Original) and Cache (GGR) performance will be similar, reducing the overall benefits. For Movies, this number reduces from 2.5× to 1.7× as the first invocation accounts for nearly half the query time; while for Products, the second invocation on Projection dominates runtime due to long decode output length (i.e., around 107), so we can still see 2.2× speed-up over Cache (Original).

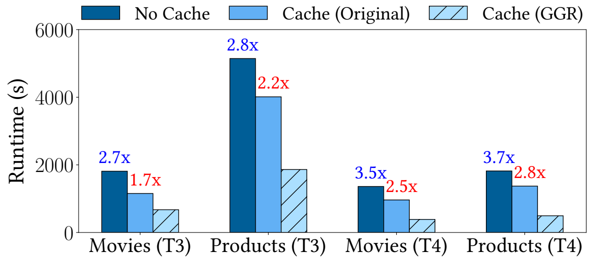


Figure 4. End-to-end Result (Multi-LLM Invocation, Aggregation): Our optimizations Cache (GGR) achieve 1.7 - 2.8× speed-up over Cache (Original), and 2.7 - 3.7× speed-up over No Cache.

Method	Movies	Prods.	BIRD	PDMX	Beer	FEVER	SQuAD
Original	35%	27%	10%	12%	50%	11%	11%
GGR	86%	83%	85%	57%	80%	67%	70%

Table 2. PHR (%) of LLM Filter and RAG queries for Original and GGR, which achieves 30 – 75% higher hit rates.

LLM aggregation. This query type uses `AVG` operator to aggregate the sentiment score on the reviews column with additional columns provided as context. We achieve a 3.5× speed-up in the Movies dataset and a 3.7× speed-up in the Products dataset over the No Cache baseline. We also achieve 2.5× speed-up on Movies and 2.8× speed-up on Products over Cache (Original). The results of this query type are similar to filtering query results, as the average

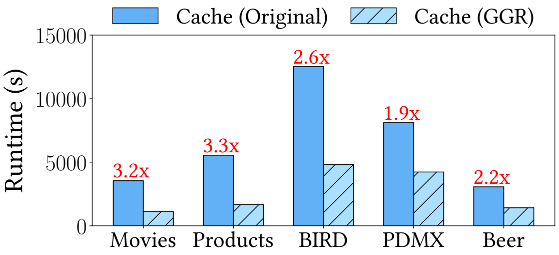


Figure 5. Cache (GGR) is able to achieve 1.9 – 3.3× speed-up over Cache (Original) for filter queries on Llama3-70B.

output length is similar.

RAG. This query is performed on a table of questions and the top four to five supporting evidence items extracted from the FEVER and SQuAD datasets. Cache (GGR) achieves a 1.9× speed-up on both FEVER and SQuAD over the No Cache baseline. We also achieve a 1.8× speed-up on FEVER and 1.7× on SQuAD over Cache (Original). In this experiment, we embed all supporting contexts for a question/claim into a vector index. We perform a K-nearest neighbor search on the vector index for each question to fetch relevant contexts. At runtime, we apply our GGR algorithm to the table of questions and contexts to maximize cache hits. Cache (GGR) can achieve 56 – 59% prefix hit rate improvements over Cache (Original), as multiple questions might share similar contexts, and Cache (GGR) can rearrange contexts to maximize prefix reuse.

Results on Different Model Sizes Fig 5 shows the evaluation of our Cache (GGR) method compared with Cache (original) on filtering queries, using Llama-3-70B-Instruct with 70B parameters. We run this model on an 8×L4 instance with tensor parallelism and measure the end-to-end query latency. Cache (GGR) achieves 1.9× to 3.3× speed-up under this setup, showing a trend similar compared to the Llama-3-8B model. We evaluate the larger model accuracy on LLM Filter queries in Sec 6.4. We also show results for the smaller 1B model in Appendix D.2.

6.3 Cost Savings on Proprietary API Endpoints

This section evaluates the cost efficiency of our GGR algorithm with closed models that support prompt caching. For OpenAI, cached prompts are offered at a 50% discount compared to uncached prompts. Anthropic beta prompt caching (ant, 2024) requires users to manually specify prompts to cache. Writing to the cache costs 25% more than the base input token price for any given model while using cached content costs only 10% of the base rate. We evaluate OpenAI GPT-4o-mini and Anthropic Claude 3.5 Sonnet, using their pricing models in our cost calculations.[2][3]

Dataset	Model	Method	PHR (%)	Cost ($)	Savings (%)
FEVER	4o-mini	Original	0.0	0.81	-
		GGR	62.2	0.55	32%
	Sonnet	Original	0.0	5.49	-
		GGR	30.6	4.33	21%

Table 3. OpenAI and Anthropic Costs: cache hit rate (HR%), cost, and savings comparison of GGR over Original for GPT-4o-mini and Claude 3.5 Sonnet in FEVER.

Dataset	PHR (%)		Est. Cost Savings (%)	
	Original	GGR	OpenAI	Anthropic
Movies	34.6	85.7	31	73
Products	26.7	83.3	33	73
BIRD	10.4	84.8	39	79
PDMX	11.8	56.6	24	48
Beer	49.9	80.1	20	55
FEVER	11.2	67.4	30	60
SQuAD	11.0	69.7	31	63

Table 4. Estimated cost savings: across datasets using PHR from Sec 6.2 and OpenAI and Anthropic's pricing model.

Since both OpenAI and Anthropic require a minimum prefix length of 1,024 tokens for caching, we duplicate each field value five times, approximating a more realistic dataset with detailed conversations and descriptions. We select the FEVER dataset for its long input length and use 1000 rows from this dataset. For Anthropic experiments, we specify cache write for only the first 1,024 tokens per request as a conservative assumption, as Anthropic does not support automatic prefix detection.

We evaluate GGR reordering on two tables submitted to the OpenAI and Anthropic APIs (each row is a request): one reordered with GGR and one in the original row and field order. Table 3 shows that GGR achieves 32% cost savings with GPT-4o-mini and 21% savings with Claude 3.5 Sonnet. The hit rate in OpenAI for GGR-reordered table is 62.2%, closely matching the hit rate (i.e., 67%) measured from our previous experiment in Table 2. Original ordering receives no cached tokens with 0% cache hits, as the shared prefix does not meet the 1,024-token minimum. The Anthropic cache hit rate is around 30.6%, two times lower than the OpenAI hit rate due to our conservative caching threshold.

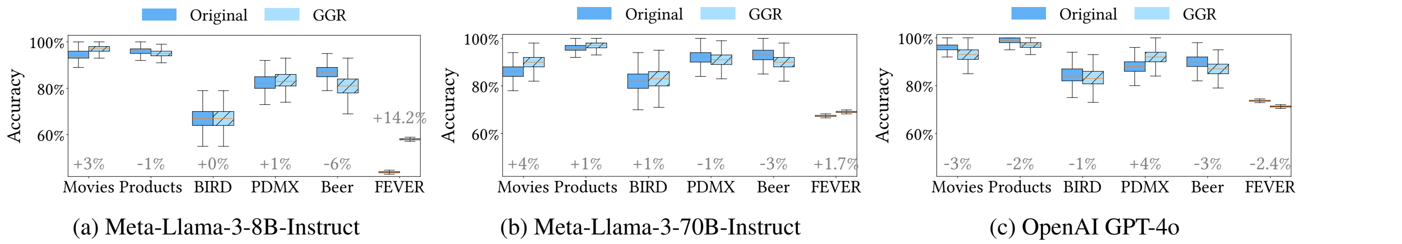
Assume that in the future, automatic prefix caching is enabled and prompts can be cached at arbitrary token lengths. We use the hit rate numbers collected from our previous experiments in Table 2 to simulate cost-saving ratios achievable by GGR, compared to the original unordered algorithm. GGR yields 20 to 39% cost savings under the OpenAI pricing model and up to 79% cost savings with Anthropic.

6.4 Impact of Reordering on Accuracy

As GGR order alters the input prompt to the LLM, we assess the impact this has on query accuracy using LLM Filter

[2] GPT-4o-mini charges $0.075/1M tokens for cached tokens versus $0.15/1M tokens for uncached tokens.

[3] Claude 3.5 Sonnet standard input tokens are priced at $3 per million tokens, cache writes at $3.75 per million, and cache reads at $0.30 per million tokens.



(a) Meta-Llama-3-8B-Instruct (b) Meta-Llama-3-70B-Instruct (c) OpenAI GPT-4o

Figure 6. Accuracy of original v.s. GGR ordering: we perform statistical bootstrapping to get a distribution of exact match accuracy measurements across 10,000 runs. The numbers indicate the difference in the median accuracy of GGR compared to the original ordering.

Solver Time (s)						
Movies	Products	BIRD	PDMX	Beer	FEVER	SQuAD
3.3	4.5	1.2	12.6	8.0	5.6	4.5

Table 5. GGR Solver time (s): GGR runs under 15 seconds for datasets with up to 30K rows and 57 fields.

queries (Sec 6.1.2) with constrained output. We also evaluate a RAG query of FEVER, excluding SQuAD due to its open-ended questions. FEVER includes ground-truth labels for all records, while 100 rows from other datasets are manually labeled. Using statistical boostrapping (Wilcox, 2003), we perform 10K runs, sampling with replacement on each run to obtain a distribution of accuracy results. Accuracy experiments are conducted with Llama-3-8B-Instruct, Llama-3-70B-Instruct, and GPT-4o models, measured as the percentage of exact matches between the LLM output and the ground truth labels.

In Fig 6, we plot the accuracy distributions across the bootstrap runs and the relative difference in median accuracy of GGR versus original ordering. The accuracy distribution of GGR ordering is within 5% accuracy of the original ordering, with the only exception being FEVER with Llama-3-8B, where the ordering with GGR performs 14.2% *better* than the original. This is due to the GGR algorithm places the "claim" field at the end of the prompt instead of at the beginning, which Llama3-8B prefers. However, the same behavior does not hold for the larger models. Overall, we can see that larger models like Llama-3-70B and GPT-4o are within 5% of accuracy difference compared with original ordering and are more robust to field reordering.

6.5 Algorithm Overhead

Latency Table 5 shows the average overheads of GGR across datasets, using a row recursion depth of four and column recursion depth of two, or an early stopping threshold of 0.1M hit count. In all cases, GGR runs in under 15 seconds – less than 0.01% of LLM query runtimes.

Memory GGR only requires the input table T (n rows, m columns) touched by the query to be loaded into memory. Recursive splitting reduces table size at each step, keeping total memory usage at $O(n \times m)$, aside from minimal stack and temporary variable overhead.

7 Related Work

Our optimizations build on recent work in LLM inference as well as prior work integrating machine learning and data management. We describe several major related areas below.

Inference-optimized systems. There has been a recent rise of dedicated systems for LLM inference, including FasterTransformer (NVIDIA, 2023a), Orca (Yu et al., 2022), vLLM (Kwon et al., 2023), and SGLang (Zheng et al., 2023). Many systems already explore developing memory-efficient GPU kernels that perform inference while leveraging shared prefixes. SGLang's RadixAttention (Zheng et al., 2023), Hydragen (Juravsky et al., 2024), and Cascade Inference (Ye et al., 2024) all implement optimized kernels. Our work builds upon prior work investigating high-throughput LLM inference and prefix caching for model serving. In addition, we leverage full workload information from batch queries to further improve performance in relational workloads.

LLMs in Relational Data Analytics Many systems support calling LLMs as operators on relational data, spanning from production database vendors like Databricks (dat), Google BigQuery (goo) and AWS Redshift (aws) to programming frameworks like LOTUS (Patel et al., 2024). While these works provide APIs for running LLMs over relational data, they do not explore how reordering data can optimize KV cache hits. There is also a line of work (Kang et al., 2017; Lu et al., 2018) that explores using cheaper models for approximate query generation. This orthogonal direction is not considered in our paper scope, as our work specifically focuses on calling LLMs as functions from inside a regular, given SQL query.

8 Conclusion

In this paper, we introduce techniques to optimize LLM invocations in relational data analytics workloads. By leveraging workload information coupled with observations about the LLM inference process, we can significantly improve end-to-end query performance and reduce costs without affecting query semantics. Our technique achieves up to 3.4× decreases in end-to-end query latency with Llama-3-8B and Llama-3-70B and also achieves up to 32% cost savings under OpenAI and Anthropic pricing models.

Acknowledgement

We thank Soujanya Ponnapalli for helpful discussions and feedback, and Jelani Nelson for academic advising. This research was supported by gifts from Accenture, AMD, Anyscale, Broadcom Inc., Google, IBM, Intel, Intesa Sanpaolo, Lambda, Mibura Inc, Samsung SDS, and SAP.

References

Large Language Models for sentiment analysis with Amazon Redshift ML (Preview) — Amazon Web Services — aws.amazon.com. https://aws.amazon.com/blogs/big-data/large-language-models-for-sentiment-analysis-with-amazon-redshift-ml-preview/. [Accessed 01-03-2024].

AI Functions on Databricks — docs.databricks.com. https://docs.databricks.com/en/large-language-models/ai-functions.html. [Accessed 01-03-2024].

LLM with Vertex AI only using SQL queries in BigQuery — Google Cloud Blog — cloud.google.com. https://cloud.google.com/blog/products/ai-machine-learning/llm-with-vertex-ai-only-using-sql-queries-in-bigquery. [Accessed 01-03-2024].

Prompt caching with claude. https://www.anthropic.com/news/prompt-caching, 2024.

Context caching. https://ai.google.dev/gemini-api/docs/caching?lang=python, 2024.

Apr 2024. URL https://ai.meta.com/blog/meta-llama-3/.

Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pp. 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2742797. URL https://doi.org/10.1145/2723372.2742797.

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

Chase, H. LangChain, October 2022. URL https://github.com/langchain-ai/langchain.

Chen, T., Zhang, N. L., Liu, T., Poon, K. M., and Wang, Y. Model-based multidimensional clustering of categorical data. *Artificial Intelligence*, 176(1):2246–2269, 2012.

Džeroski, S. Multi-relational data mining: an introduction. *ACM SIGKDD Explorations Newsletter*, 5(1):1–16, 2003.

Gim, I., Chen, G., seob Lee, S., Sarda, N., Khandelwal, A., and Zhong, L. Prompt cache: Modular attention reuse for low-latency inference, 2024. URL https://arxiv.org/abs/2311.04934.

He, R. and McAuley, J. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pp. 507–517, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee. ISBN 9781450341431. doi: 10.1145/2872427.2883037. URL https://doi.org/10.1145/2872427.2883037.

Huggingface. Text Generation Inference, 2023. URL https://huggingface.co/docs/text-generation-inference/en/index.

Idreos, S., Kersten, M. L., Manegold, S., et al. Database cracking. In *CIDR*, volume 7, pp. 68–78, 2007.

Ilyas, I. F., Markl, V., Haas, P., Brown, P., and Aboulnaga, A. Cords: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 647–658, 2004.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7 (3):535–547, 2019.

Juravsky, J., Brown, B., Ehrlich, R., Fu, D. Y., Ré, C., and Mirhoseini, A. Hydragen: High-throughput llm inference with shared prefixes, 2024.

Kang, D., Emmons, J., Abuzaid, F., Bailis, P., and Zaharia, M. Noscope: optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, aug 2017. ISSN 2150-8097.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Lemire, D. and Kaser, O. Reordering columns for smaller indexes. *Information Sciences*, 181(12):2550–2570, 2011.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., tau Yih, W., Rocktäschel, T., Riedel, S., and Kiela, D. Retrieval-augmented generation for knowledge-intensive nlp tasks, 2021.

Li, J., Hui, B., Qu, G., Yang, J., Li, B., Li, B., Wang, B., Qin, B., Geng, R., Huo, N., 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, 2024.

Li, Z., Zhang, X., Zhang, Y., Long, D., Xie, P., and Zhang, M. Towards general text embeddings with multi-stage contrastive learning. *arXiv preprint arXiv:2308.03281*, 2023.

Long, P., Novack, Z., Berg-Kirkpatrick, T., and McAuley, J. Pdmx: A large-scale public domain musicxml dataset for symbolic music processing, 2024. URL https://arxiv.org/abs/2409.10831.

Lu, Y., Chowdhery, A., Kandula, S., and Chaudhuri, S. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pp. 1493–1508, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037.

McAuley, J., Leskovec, J., and Jurafsky, D. Learning attitudes and attributes from multi-aspect reviews, 2012. URL https://arxiv.org/abs/1210.3926.

NVIDIA. Faster Transformer, 2023a. URL https://github.com/NVIDIA/FasterTransformer.

NVIDIA. TensorRT LLM, 2023b. URL https://github.com/NVIDIA/TensorRT-LLM.

OpenAI. Pricing — openai.com. https://openai.com/pricing. [Accessed 01-03-2024].

Pang, B. and Lee, L. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the ACL*, 2005.

Patel, L., Jha, S., Guestrin, C., and Zaharia, M. Lotus: Enabling semantic queries with llms over tables of unstructured and structured data, 2024. URL https://arxiv.org/abs/2407.11418.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. Squad: 100,000+ questions for machine comprehension of text, 2016.

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pp. 491–518. 2018.

Thorne, J., Vlachos, A., Christodoulopoulos, C., and Mittal, A. Fever: a large-scale dataset for fact extraction and verification, 2018. URL https://arxiv.org/abs/1803.05355.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023.

Wilcox, R. R. Bootstrap confidence interval. In *Applying Contemporary Statistical Techniques*. Academic Press, 2003.

Ye, Z., Lai, R., Lu, B.-R., Lin, C.-Y., Zheng, S., Chen, L., Chen, T., and Ceze, L. Cascade inference: Memory bandwidth efficient shared prefix batch decoding, February 2024. URL https://flashinfer.ai/2024/02/02/cascade-inference.html.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/yu.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, pp. 15–28, 2012.

Zheng, L., Yin, L., Xie, Z., Huang, J., Sun, C., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Efficiently programming large language models using sglang, 2023.

A Query Examples

Our benchmark suite incorporates a broad range of query types. We show examples of each query type as follows.

LLM filter. This query type leverages LLM for filtering data within a WHERE clause. The LLM processes and analyzes information to meet some specified criteria, such as identifying whether a movie is suitable for kids. This query type illustrates typical use cases in sentiment analysis and content filtering, which are important for application tasks, such as customer feedback analysis and content moderation.

```
SELECT t.movietitle
FROM MOVIES
WHERE LLM(
    'Given the following fields,
    ↪ determine whether the movie is
    ↪ suitable for kids. Answer ONLY
    ↪ with "Yes" or "No".',
    movieinfo,
    reviewcontent,
    reviewtype,
    movietitle
) = 'Yes'
```

LLM projection. This query type makes calls to an LLM within a SELECT statement to process information from specified database column(s). It reflects common tasks in data analytics in which the LLM is used for summarization and interpretation based on certain data attributes.

```
SELECT LLM(
    'Given the following information,
    ↪ summarize good qualities in
    ↪ this movie that led to a
    ↪ favorable rating.',
    reviewcontent, movieinfo
)
FROM MOVIES
```

Multi-LLM invocation. This query type involves multiple LLM calls in different parts of the query and addresses scenarios in which several layers of data processing or analysis are required. It represents advanced analytical tasks, such as combining different data insights.

```
SELECT LLM(
    'Given the information about a
    ↪ movie, summarize the good
    ↪ qualities that led to a
    ↪ favorable rating.',
    reviewtype,
    reviewcontent,
    movieinfo,
    genres
)
FROM MOVIES
WHERE LLM(
    'Given the following review, answer
    ↪ whether the sentiment is
    ↪ "POSITIVE" or "NEGATIVE".
    ↪ Respond ONLY with "POSITIVE" or
    ↪ "NEGATIVE", in all caps.',
    reviewcontent
) = 'NEGATIVE'
```

LLM aggregation. This query type incorporates an AVG operator that incorporates LLM outputs into further query processing. For example, one could use LLMs to assign sentiment scores to individual reviews and then aggregate these scores to calculate an average sentiment for overall customer feedback. This query type is essential for tasks that need to extract insights from complex textual data.

```
SELECT AVG(
    LLM(
        'Rate sentiment in numerical
        ↪ values from 1 (bad) to 5
        ↪ (good).',
        reviewcontent, movieinfo
    )
) AS AverageScore
FROM MOVIES
```

Retrieval-augmented generation (RAG). This query type leverages external knowledge bases for enhanced LLM processing, enriching LLM queries with a broader context. It simulates use cases where queries need to pull in relevant information from external sources, such as document databases or knowledge graphs, to provide comprehensive answers.

```
SELECT LLM(
    'Given a question and four
    ↪ supporting contexts, answer the
    ↪ provided question.',
    ↪ VectorDB.search(question, k=4),
    ↪ question)
FROM FEVER
```

B Dataset Information

We detail the fields and functional dependencies (FDs) used for each dataset as follows.

MOVIES

```
columns:
genres, movieinfo, movietitle,
productioncompany, reviewcontent,
reviewtype, rottentomatoeslink,
topcritic

FDs:
movieinfo, movietitle,
rottentomatoeslink
```

PRODUCTS

```
columns:
description, id, parent_asin,
product_title, rating, review_title,
text, verified_purchase


FDs:
parent_asin, product_title
```

BIRD

```
columns:
Body, PostDate, PostId, Text

FDs:
Body, PostId
```

PDMX

```
columns:
artistname, bestarrangement, bestpath,
bestuniquearrangement, composername,
complexity, genre, grooveconsistency,
groups, hasannotations, hascustomaudio,
hascustomvideo, haslyrics, hasmetadata,
haspaywall, id, isbestarrangement,
isbestpath, isbestuniquearrangement,
isdraft, isofficial, isoriginal,
isuserpro, isuserpublisher, isuserstaff,
license, licenseurl, metadata,
nannotations, ncomments, nfavorites,
nlyrics, notesperbar, nnotes, nratings,
ntracks, ntokens, nviews, path,
pitchclassentropy, postdate, postid,
publisher, rating, scaleconsistency,
songlength, songlengthbars,
songlengthbeats, songlengthseconds,
songname, subsetall, subsetdeduplicated,
subsetrated, subsetrateddeduplicated,
subtitle, tags, text, title, tracks,
version


FDs:
[metadata, path],
[hasannotations, hasmetadata, isdraft,
isofficial, isuserpublisher, subsetall
]
```

BEER

```
columns:
beer/beerId, beer/name, beer/style,
review/appearance, review/overall,
review/palate, review/profileName,
review/taste, review/time

FDs:
[beer/beerId, beer/name]
```

FEVER

```
-- FEVER --
columns:
claim, evidence1, evidence2,
evidence3, evidence4

FDs: []
```

SQuAD

```
columns:
question, context1, context2,
context3, context4, context5

FDs: []
```

C PROMPTS

We detail the system and user prompts for each query type and dataset as follows.

System Prompt

```
You are a data analyst. Use the provided JSON data
to answer the user query based on the specified
fields. Respond with only the answer,
no extra formatting.

Answer the below query:
{QUERY}

Given the following data:
{fields}
```

User Prompt - LLM Aggregation

```
MOVIES: Given the following fields of a movie
description and a user review, assign a sentiment
score for the review out of 5. Answer with ONLY a
single integer between 1 (bad) and 5 (good).

PRODUCTS: Given the following fields of a product
description and a user review, assign a sentiment
score for the review out of 5. Answer with ONLY a
single integer between 1 (bad) and 5 (good).
```

User Prompt - Multi-LLM Invocation

```
MOVIES/PRODUCTS: Given the following review, answer
whether the sentiment associated is 'POSITIVE' or
'NEGATIVE'. Answer in all caps with ONLY 'POSITIVE'
or 'NEGATIVE':
```

User Prompt - LLM Filter

```
MOVIES: Given the following fields, answer in one
word, 'Yes' or 'No', whether the movie would be
suitable for kids.  Answer with ONLY 'Yes' or 'No'.

PRODUCTS: Given the following fields determine if
the review speaks positively ('POSITIVE'),
negatively ('NEGATIVE'), or netural ('NEUTRAL')
about the product. Answer only 'POSITIVE',
'NEGATIVE', or 'NEUTRAL', nothing else.

BIRD: Given the following fields related to posts
in an online codebase community, answer whether the
post is related to statistics. Answer with only
'YES' or 'NO'.

PDMX: Based on following fields, answer 'YES' or
'NO' if any of the song information references a
specific individual. Answer only 'YES' or 'NO',
nothing else.

BEER: Based on the beer descriptions, does this
beer have European origin? Answer 'YES' if it does
or 'NO' if it doesn't.
```

User Prompt - LLM Projection

```
MOVIES: Given information including movie
descriptions and critic reviews, summarize the good
qualities in this movie that led to a favorable
rating. (also used in multi-invocation)

PRODUCTS: Given the following fields related to
amazon products, summarize the product, then answer
whether the product description is consistent with
the quality expressed in the review. (also used
in multi-invocation)

BIRD: Given the following fields related to posts
in an online codebase community, summarize how the
comment Text related to the post body.

PDMX: Given the following fields, provide an
overview on the music type, and analyze the given
scores. Give exactly 50 words of summary.

BEER: Given the following fields, provide an
high-level overview on the beer and review in a
20 words paragraph.
```

User Prompt - RAG

```
FEVER: You are given 4 pieces of evidence as
{evidence1}, {evidence2}, {evidence3}, and
{evidence4}. You are also given a claim as {claim}.
Answer SUPPORTS if the pieces of evidence support
the given {claim}, REFUTES if the evidence refutes
the given {claim}, or NOT ENOUGH INFO if there is
not enough information to answer. Your answer
should just be SUPPORTS, REFUTES, or NOT ENOUGH
INFO and nothing else.

SQuAD: Given a question and supporting contexts,
answer the provided question.
```

D Ablations

We present two sets of ablation experiments: one comparing the prefix hit rate (PHR) between GGR and an optimal oracle, and another examining the impact of using a smaller LLM model.

D.1 PHR of GGR v.s. OPHR

OPHR is a very expensive brute-force oracle algorithm that iterates through all possible combinations of value groups and calculates the prefix hit count. In our empirical evaluation, it is impractical to run on larger datasets.

Thus, we test the first (10, 25, 50, 100, 200) rows for each dataset and terminate OPHR runs exceeding 2 hours, reporting the result of the successful run with the most rows. For PDMX, we reduce 57 columns to 10 to enable runs on even as few as 10 rows. The PHR (prefix hit rate) and solver runtime in seconds across datasets are reported in Table 6, with the dataset labeled as {*dataset*}-{*#rows*}.

We can see that on these small samples of the datasets, our algorithm (GGR) achieves within 2% of the optimal, but can be up to *hours faster* on solver runtime.

Dataset	PHR (%)			Solver Runtime (s)	
	OPHR	GGR	Diff	OPHR	GGR
Movies-50	80.6	80.6	0%	2556	0.05
Products-25	19.7	18.5	-1.2%	357	0.06
BIRD-50	77.5	76.2	-1.3%	0.43	0.05
PDMX-25	29.4	28.6	-0.8%	822	0.05
Fever-50	7.3	6.9	-0.4%	110	0.23
Beer-10	25.7	25.6	-0.1%	1269	0.08
SQuAD-10	34.0	34.0	0%	1.6	0.05

Table 6. Comparison of Prefix Hit Rate (PHR) and solver runtime across datasets. GGR achieves near-optimal PHR while being orders of magnitude faster than OPHR.

D.2 Results of Smaller Model

To analyze the impact of using a smaller model, we run the Filter Query described in Fig. 3a with the Llama-3.2-1B model, using the same setup as with Llama-3 8B (i.e., single L4 instance), and compare the prefix hit rate and end-to-end query execution time of GGR with the default vLLM baseline (i.e. Cache Original). The results are reported in Table 7.

Metric	BIRD	Movies	PDMX
Runtime (orig/GGR)	1.5×	1.3×	1.3×
Orig PHR (%)	10.41	29.32	11.97
GGR PHR (%)	83.99	82.10	56.00
Metric	**Products**	**BEER**	
Runtime (orig/GGR)	1.4×	1.2×	
Orig PHR (%)	24.06	47.98	
GGR PHR (%)	82.10	73.93	

Table 7. Cache runtime ratio and prefix hit rate (PHR) (%) comparison between original and GGR ordering for Llama-3.2-1B.

We observe similar prefix hit rates with Llama-3.2-1B compared to our previous 8B model runs. This consistency arises from the effectiveness of GGR field reordering, which converts non-reusable field contents (0 hits) into reusable prefixes within the cache. We also observe that under the same GPU instance setup (e.g., L4 with 24 GB memory), larger models like Llama-8B (7.6 GB) exhibit larger relative performance gains from GGR compared to smaller models like Llama-1B (1.8 GB), despite seeing similar prefix hit rates. This is because prefix caching benefits from reducing computational overhead on shared prefixes and enabling larger batch sizes for LLM generation by reducing memory usage through sharing. For smaller models, the availability of ample GPU memory diminishes the relative impact of prefix caching, as larger batch sizes can be achieved without relying on caching. But for larger models, or when there is less available GPU space, prefix caching benefits become more pronounced.