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 PROBLEM SETUP

This section introduces the problem setup of maximizing prefix hits in the prompt cache (Sec 1.1) and highlights cases where naive fixed field ordering can result in significantly lower hit rates (Sec 1.2).

1.1 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:

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

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.

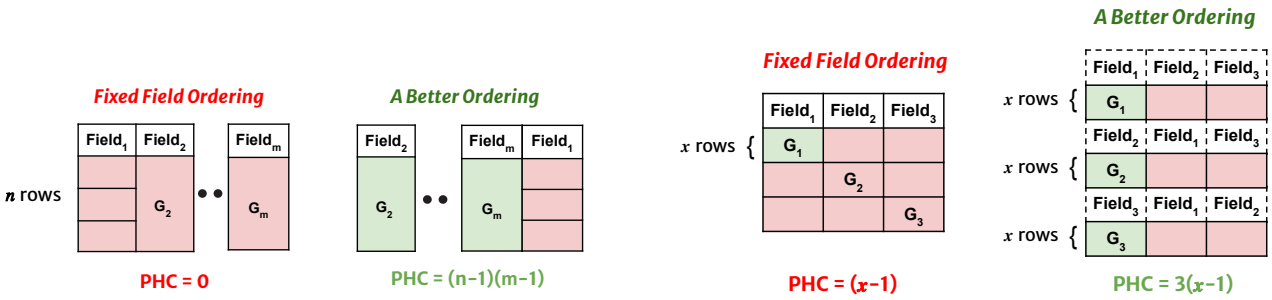
We define the *prefix hit count* (PHC) of L as the number of consecutive field cell values shared with the previous row starting from the first cell, summing over all n rows. Each cell value must exactly match the corresponding cell of the previous row (cannot be a substring), and cell values past the 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:

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

Here, the function $\mathit{hit}(L, r)$ 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 oc-

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

Proceedings of the 6th MLSys Conference, Santa Clara, CA, USA, 2025.



(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 $m = 3$.

curs. 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.

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

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. **??**, demonstrate good real-world performance.

1.2 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 **??**). 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.

Algorithm 1 Greedy Group Recursion (GGR)

Input: Table T, Functional Dependency FD
Output: Prefix Hit Count S, Reordered List of Tuples L

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

function GGR(T, FD)
  if $|T|_{rows} = 1$ **then**
    return 0, $[T[1]]$
  end if
  if $|T|_{cols} = 1$ **then**
    $S \leftarrow \sum_{v \in \text{distinct}(T[,1])}$ HITCOUNT$(v, 1, T)$
    Return $S, sort([T[i] \mid i \in 1 \dots |T|_{rows}])$
  end if
  $max\_HC, b\_v, b\_c, b\_cols \leftarrow -1$, None, None, []
  for $c \in$ columns(T), $v \in$ distinct$(T[, c])$ **do**
    $HC, cols \leftarrow$ HITCOUNT(v, c, T, FD)
    if $HC > max\_HC$ **then**
      $max\_HC, b\_v, b\_c, b\_cols = HC, v, c, cols$
    end if
  end for
  $R\_v \leftarrow \{i \mid T[i, b\_c] = b\_v\}$
  $A\_HC, L\_A \leftarrow$ GGR$(T[\text{rows} \setminus R_v, \text{cols}], FD)$
  $B\_HC, L\_B \leftarrow$ GGR$(T[R_v, \text{cols} \setminus b_cols], FD)$
  $C\_HC, \_ \leftarrow$ HITCOUNT(b_v, b_c, T, FD)
  $S \leftarrow A\_HC + B\_HC + C\_HC$
  $L \leftarrow [[b\_v] + L_A[i] \mid i \in 1 \dots |R\_v|] + L\_B$
  return S, L
end function

return GGR(T, FD)

2 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 2.1) and introduce a greedy algorithm that efficiently approximates the optimal algorithm (Sec 2.2).

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

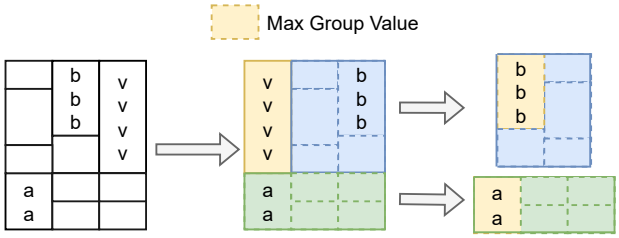


Figure 2. GGR picks the group with the maximum hit count at each step and calculates PHC as the sum of PHC of the elected group values (yellow box), the sub-table T excluding rows R_v (green box), and the sub-table of rows R_v excluding the field where the value is located in (blue box).

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 2.2).

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.

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

REFERENCES

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

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.