IMPROVING OPTIMALITY AND SPEED OF GREEDY GROUP RECURSION ALGORITHM

Florin Dobrian [1] **Oleg Puzyrko White** [1]

ABSTRACT

Recently (Liu et al., 2025) has presented efficient algorithm - Greedy Group Recursion (GGR) - 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. In this paper, we propose several adjustments to GGR algorithm that can improve optimality of the solution and reduce its execution time.

1 INTRODUCTION

There has been growing research on LLM inference optimization. In particular, recent work (Liu et al., 2025; Cheng et al., 2025) presents solutions to optimize relational data analytics workloads for offline LLM inference. It proposes 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.

1.1 Greedy Group Recursion (GGR) Algorithm

The GGR algorithm is described in detail in (Liu et al., 2025). Let us briefly outline its main points.

The pseudocode specification of GGR is in Algorithm 1 listing and is taken from the original paper but with several critical corrections (and multiple little typo fixes).

GGR algorithm optimizes LLM queries by finding a $(n \times m)$ table rearrangement that maximizes the LLM's KV cache prefix hit count (PHC). It achieves this goal by reordering the rows and the fields within each row to **maximize** the *prefix hit count*. Each row may have a different field order. The rearranged table is represented as a list of tuples L, where each tuple in L corresponds to a row, and the tuple elements contain the column values. A cell in the list of tuples is denoted as $L[r][f]$, indicating the value in tuple r at position f.

The hit count of a single cell $L[r][c]$ is non-zero **only** if its value is the same as in the previous row $L[r][c] = L[r-1][c]$ **and** all preceding fields have the same property $\forall f \leq c, L[r][f] = L[r-1][f]$. Then its hit count is computed as the square of the value's string length:

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

The squared string length reflects the quadratic complexity of token processing in LLM inference, where each token computation depends on every preceding token and increases computational cost quadratically with the input length.

The hit count for a single row r in L is then just a sum of hit counts for all cells in a row:

$$hit(L,r) = \sum_{c=1}^{m} hit(L,r,c) \tag{2}$$

The PHC for a list of tuples L with n rows and m fields is given by:

$$\text{PHC}(L) = \sum_{r=1}^{n} hit(L,r) \tag{3}$$

The PHC(L) is the objective function of a table reordering algorithm: it tries to find output L with maximum PHC(L) value.

The greedy part of GGR is based on finding a distinct value b_v (in the corresponding column b_c) with the highest hit count and then rearranging the table so that all cells with that value appear in consecutive rows and in the first field of the output L. GGR then uses the group of b_v, b_c cells to split the table into two smaller sub-tables and recursively run GGR on each of them.

Compared to the brute-force algorithm that requires $n! * (m!)^n$ potential orderings, GGR significantly reduces the search space by selecting the highest-hit value and then reducing the dimensions of the table at each recursive step with the maximum depth of recursion $O(\min(n, m))$. GGR

[1]Data Analytics Group. Corresponding author: Oleg P. White <oleg.p.white@gmail.com>.

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: inferred_vals $\leftarrow \{T[R_v[1], c'] \mid c' \in \text{inferred\_cols}\}$
7: tot_len $\leftarrow \text{len}(v)^2 + \sum_{c' \in \text{inferred\_cols}} \left(\frac{\sum_{r \in R_v} \text{len}(T[r, c'])}{|R_v|} \right)^2$
8: $HC \leftarrow \text{tot\_len} \times (|R_v| - 1)$
9: $cols \leftarrow [c] + \text{inferred\_cols}$
10: $vals \leftarrow [v] + \text{inferred\_vals}$
11: **return** $HC, cols, vals$
12: **end function**

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

38: **return** GGR(T, FD)

achieves close-to-perfect PHC output with fast practical execution time.

We can follow the pseudocode specification of GGR in Algorithm 1 listing.

At each recursive step, the GGR algorithm scans the table (lines 21–29) to find all distinct values with corresponding hit counts (lines 3–12). It then selects the highest-hit value b_v (in the column b_c) and splits the table into two sub-tables - one with all the rows R_v containing b_v value but excluding the column b_c and its FD-associated columns (line 31: $T[R\_v, \text{cols} \setminus b\_cols]$), and another sub-table with the remaining rows (line 32: $T[\text{rows} \setminus R\_v, \text{cols}]$). See Figure 1 for illustration of this key step. In the final step, GGR recurses on the two sub-tables (lines 31–32) and calculates

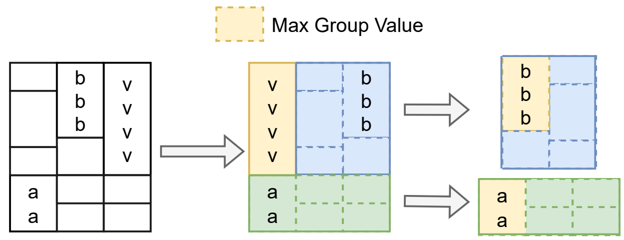

Figure 1. GGR picks the group with the maximum hit count at each step and calculates PHC as the sum of PHC of the elected group values (yellow box), the sub-table T excluding rows R_v (green box), and the sub-table of rows R_v excluding the field where the value is located in (blue box).

the total PHC as the sum of PHCs computed for each sub-table and of hit count computed for b_v, b_c (line 34).

REFERENCES

Cheng, A., Liu, S., Pan, M., Li, Z., Agarwal, S., Cemri, M., Wang, B., Krentsel, A., Xia, T., Park, J., Yang, S., Chen, J., Agrawal, L., Naren, A., Li, S., Ma, R., Desai, A., Xing, J., Sen, K., Zaharia, M., and Stoica, I. Let the barbarians in: How ai can accelerate systems performance research, 2025. URL https://arxiv.org/abs/2512.14806.

Liu, S., Biswal, A., Kamsetty, A., Cheng, A., Schroeder, L. G., Patel, L., Cao, S., Mo, X., Stoica, I., Gonzalez, J. E., and Zaharia, M. Optimizing llm queries in relational data analytics workloads, 2025. URL https://arxiv.org/abs/2403.05821.