
IMPROVING OPTIMALITY AND EFFICIENCY OF GREEDY GROUP RECURSION ALGORITHM

Florin Dobrian¹ Oleg Puzyrko White¹

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. 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 .

1.1.1 Prefix Hit Count Calculation

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 $\forall f \leq$

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

$c, L[r][f] = L[r - 1][f]$. Then its hit count is computed as the square of the value's string length:

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

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:

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

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

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

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

1.1.2 Selecting the highest-hit distinct value

The greedy part of GGR is based on finding a distinct value v (in the corresponding column c) with the highest hit count among all distinct values in the table T . If $R_v = R(v, c, T)$ are the rows with value v in the column c :

$$R(v, c, T) = \{i \mid T[i, c] = v\}, \quad (4)$$

then hit count of v, c is given by:

$$\text{HitCount}(v, c, T) = \text{len}(v)^2 \times (|R(v, c, T)| - 1) \quad (5)$$

Algorithm 1 Greedy Group Recursion (GGR)

```

1: Input: Table  $T$ , Functional Dependency  $FD$ 
2: Output: Prefix Hit Count  $HC$ , 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:    $inferred\_vals \leftarrow \{T[R_v[1], c'] \mid c' \in inferred\_cols\}$ 
7:    $tot\_len \leftarrow \text{len}(v)^2 + \sum_{v' \in inferred\_vals} \text{len}(v')^2$ 
8:    $HC \leftarrow tot\_len \times (|R_v| - 1)$ 
9:    $cols \leftarrow [c] + inferred\_cols$ 
10:   $vals \leftarrow [v] + inferred\_vals$ 
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  $S, \text{sort}([T[i] \mid i \in 1 \dots |T|_{rows}])$ 
20:   end if
21:    $b\_v, b\_c \leftarrow \text{None}, \text{None}$ 
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:    $HC_A, L_A \leftarrow \text{GGR}(T[\text{rows} \setminus R_v, cols], FD)$ 
32:    $HC_B, L_B \leftarrow \text{GGR}(T[R_v, cols \setminus b\_cols], FD)$ 
33:    $HC_C \leftarrow max\_HC$ 
34:    $HC \leftarrow HC_A + HC_B + HC_C$ 
35:    $L \leftarrow [b\_vals + L_B[i] \mid i \in 1 \dots |R_v|] + L_A$ 
36:   return  $HC, L$ 
37: end function

38: return GGR( $T, FD$ )

```

GGR then places all v, c cells in consecutive rows and in the first field of the output L and splits remaining cells of the table T into the sub-table of rows R_v with excluded column c : $T[R_v, \text{cols} \setminus [c]]$, and the sub-table of rows excluding R_v : $T[\text{rows} \setminus R_v, \text{cols}]$. GGR then recursively runs on each of two sub-tables. See Figure 1 for illustration of this key step.

Compared to the brute-force algorithm that requires $n! \times (m!)^n$ 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 achieves close-to-perfect PHC output with fast practical execution time.

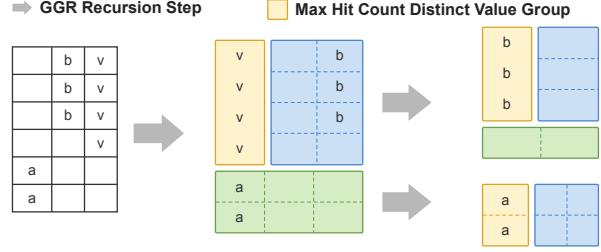


Figure 1. GGR picks the group with the maximum hit count at each step and calculates PHC as the sum of PHC of the elected group values (yellow box), the sub-table T excluding rows R_v (green box), and the sub-table of rows R_v excluding the field where the value is located in (blue box). In the input table T (white box on the left), $\text{HitCount}(v) > \text{HitCount}(b) > \text{HitCount}(a)$ and all other distinct values have hit count zero.

1.1.3 Functional Dependencies

GGR algorithm leverages a table's functional dependencies (FD) to reduce the number of fields it needs to consider at each recursion step.

We say that two columns A and B are in functional dependency $A \leftrightarrow B$ in a table T if, for any two rows r_1 and r_2 , $T[r_1, A] = T[r_2, A]$ implies $T[r_1, B] = T[r_2, B]$ and vice versa.

In other words, for any distinct value a in the column A , there is a distinct value b in the column B such that they both are on the same rows. I.e., for $R_a \leftarrow \{i \mid T[i, A] = a\}$ and $R_b \leftarrow \{i \mid T[i, B] = b\}$ we have $R_a = R_b$.

FD rules can be specified as a list of disjoint sets containing column indices. For example, for the table with columns A, B, C, D, E, F where $A \leftrightarrow B$ and $C \leftrightarrow D \leftrightarrow E$, the FD rules can be written as $[[A, B], [C, D, E]]$.

GGR takes advantage of FD by combining distinct values from columns in the same FD rule into one block of value groups. See Figure 2 for illustration of GGR iteration in the presence of FD rule.

1.1.4 GGR Specification

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: $T[\text{rows} \setminus R_v, \text{cols}]$). In the final step, GGR recurses on the two sub-tables (lines 31–32) and calculates the total PHC as the sum of PHCs computed

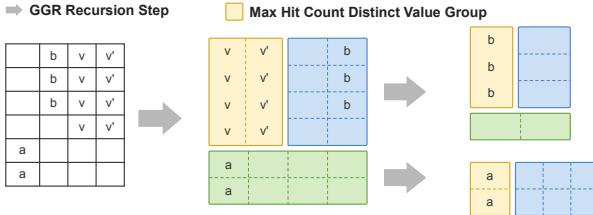


Figure 2. Similar to Figure 1 but with two distinct values v (in column c) and v' (in column c') bound by FD rule $FD = [c, c']$. In the input table T (white box on the left), $\text{HitCount}(v) + \text{HitCount}(v') > \text{HitCount}(b) > \text{HitCount}(a)$ and all other distinct values have hit count zero.

for each sub-table and of hit count computed for b_v, b_{-c} (line 34).

There are two recursion termination conditions:

- When a table contains only a single row. Then PHC is zero and the table is not rearranged. (Lines 14–16.)
- When a table contains only a single column. Then the table with sorted column values is returned and PHC is trivially computed. (Lines 17–20.)

1.1.5 Optimality of GGR Algorithm

While GGR approximates the ideal optimal rearrangement of rows and columns, it can achieve *proven* optimal PHC in certain cases.

The first case is a trivial one – when hit counts of all distinct values (with FD accounted for) are zero. It means that every distinct value in the table appears in a single row only. In this case non-zero PHC cannot be achieved by any table rearrangement. This case also covers one of the recursion termination conditions – a table with a single row.

The second case is when all columns of a table are bound by an FD rule. In other words, any column functionally determines all other columns. In this case for each distinct value GGR combines all columns to one group of values, capturing all key correlations and producing the optimal reordering. This case also covers the recursion termination conditions – a table with a single column.

However, there are cases when GGR clearly produces sub-optimal solutions.

One case is when distinct values tie in max hit count. See an example of this case with GGR producing suboptimal reordering in Figure 3.

Another case is when a distinct value in one column correlates with a distinct value in another column. This is not the case of FD-bound columns (even though it might sound similar), but, rather, the case of data heuristics when two distinct values in two columns happen to share the same

GGR suboptimal example 1

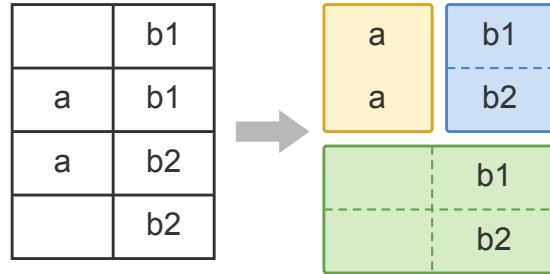


Figure 3. Example of a table with max hit count ties and sub-optimal GGR output. $\text{HitCount}(a) = \text{HitCount}(b1) = \text{HitCount}(b2)$

GGR suboptimal example 2

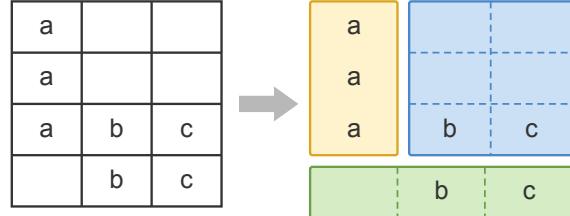


Figure 4. Example of a table with correlated distinct values b and c , and suboptimal GGR output. $\text{HitCount}(a) = \text{HitCount}(b) + \text{HitCount}(c)$ and $R_b = R_c$

rows. See an example of this case with GGR producing suboptimal reordering in Figure 4.

1.2 Reference Implementation of GGR Algorithm

The reference implementation of GGR specification from Algorithm 1 can be found in (Dobrian & White, 2026).

2 ADJUSTMENTS TO GGR ALGORITHM

The pseudocode specification of Adjusted GGR can be found in Algorithm 2 listing.

2.1 Early Termination

2.2 Faster Table Scan with FD and Table Statistics

2.3 K Top Hit Counts in One Column

There could be a case when some column A has a distinct value a_1 with max hit count, but then it has another distinct value a_2 with the second-highest hit count. In general, the column might have K distinct values with the top hit counts (some of them can be ties).

This is a typical case for tables with a very low cardinality metadata column that can easily contain distinct values with (almost) all top hit counts. Consider an example of a

Algorithm 2 Adjusted Greedy Group Recursion (AGGR)

```

1: Input: Table  $T$ , Functional Dependency  $FD$ 
2: Output: Prefix Hit Count  $HC$ , 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:    $inferred\_vals \leftarrow \{T[R_v[1], c'] \mid c' \in inferred\_cols\}$ 
7:    $tot\_len \leftarrow \text{len}(v)^2 + \sum_{v' \in inferred\_vals} \text{len}(v')^2$ 
8:    $HC \leftarrow tot\_len \times (|R_v| - 1)$ 
9:    $cols \leftarrow [c] + inferred\_cols$ 
10:   $vals \leftarrow [v] + inferred\_vals$ 
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  $S, \text{sort}([T[i] \mid i \in 1 \dots |T|_{rows}])$ 
20:   end if
21:    $b\_v, b\_c \leftarrow \text{None}, \text{None}$ 
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:    $HCA, LA \leftarrow \text{GGR}(T[\text{rows} \setminus R_v, cols], FD)$ 
32:    $HCB, LB \leftarrow \text{GGR}(T[R_v, cols \setminus b\_cols], FD)$ 
33:    $HCC \leftarrow max\_HC$ 
34:    $HC \leftarrow HCA + HCB + HCC$ 
35:    $L \leftarrow [b\_vals + LB[i] \mid i \in 1 \dots |R_v|] + LA$ 
36:   return  $HC, L$ 
37: end function

38: return GGR( $T, FD$ )

```

“Device Type” column with three distinct values “Android Device”, “iOS Device”, and “Web Browser” – all with the top hit counts.

Another example of this case is when most columns have zero hit count distinct values, but one column has lots of distinct values with low but non-zero hit counts.

We can exploit cases like this by splitting the table into $K+1$ sub-tables in one iteration rather than going through K separate iterations. It also accelerates the reduction of sub-table size benefitting the table scan in subsequent recursions. See Figure 5 for a description of this adjustment.

We can characterize this approach as an extension of the “greedy” strategy to selecting multiple distinct values in one column in one iteration.

→ AGGR Recursion Step

Top Hit Count Value Groups

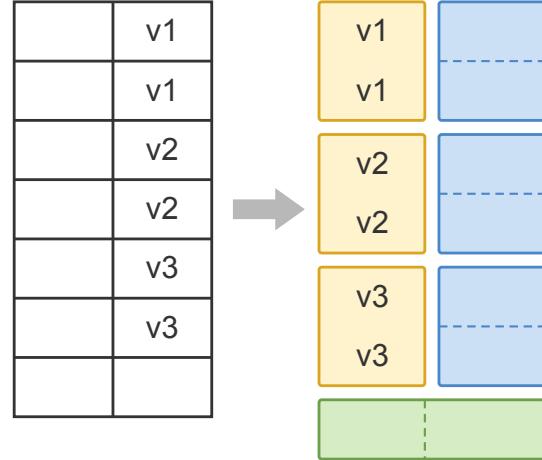


Figure 5. If a column V contains K distinct values v_1, v_2, \dots, v_K with the highest hit counts in the table, Adjusted GGR picks all K top hit count groups in one recursion step and splits the table into $K+1$ sub-tables: one sub-table T excluding rows $\sum_{i=1}^K R_{vi}$ (green box), and K sub-tables of rows R_{vi} for $i \in 1 \dots K$ excluding the column V (blue boxes).

Note that this adjustment does not improve the output optimality but reduces the number of iterations, in some cases significantly.

2.4 Max Hit Count Ties Across Columns

2.5 Multiple Columns with the same number of Max Hit Count Ties

2.6 Ad-hoc heuristics for correlated distinct values

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>.

Dobrian, F. and White, O. Reference implementation of GGR algorithm in Python, 2026. URL <https://github.com/whitechno/llm-sql-phc>.

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>.