Optimizing the DFS-based Strategy for Efficient Execution of Counting Queries in Machine Learning Applications

Pawel Bratek
Czestochowa University of Technology
Czestochowa, Poland *pawel.bratek@pcz.pl*

Lukasz Szustak
Czestochowa University of Technology
Czestochowa, Poland *lszustak@icis.pcz.pl*

Abstract

Counting queries are fundamental operations widely used in machine learning applications. This paper focuses on optimizing their execution by introducing algorithmic enhancements to the bitmap-based counting query strategy that relies on a Depth-First Search (DFS) traversal. The proposed approach is evaluated through a benchmark involving the execution of random query streams across multiple test datasets. The experimental results demonstrate a significant speedup, with execution times reduced by factors ranging from 1.26× to 2.25×. Furthermore, potential directions for further improving the performance of counting queries on modern high-performance computing (HPC) systems are discussed.

Keywords: counting queries, machine learning, DFS, HPC.

1. Introduction

Many machine learning algorithms rely on counting data records that match specific combinations of values across selected variables. This seemingly simple operation underpins a broad spectrum of learning tasks that require estimating probability distributions from data. It is particularly important in domains where statistical dependencies between variables must be quantified or structural relationships inferred. For example, counting plays a central role in learning Bayesian network structures by providing the counts needed to evaluate scoring functions and dependency measures [6]. It also supports the discovery of reliable association rules in transactional datasets [1], and underlies many classification methods [7]. Moreover, it contributes to feature representation in deep models [9] and the computation of relevance metrics in information retrieval systems [8]. The importance of counting queries lies in their computational cost, as in machine learning applications these operations may account even for more than 90% of total execution time [5]. Consequently, improving performance of counting is important research area as it may directly improve performance of many applications.

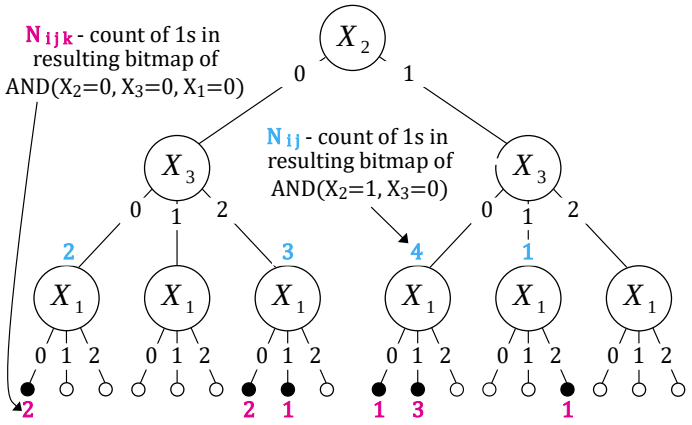
To formally introduce the concept of counting queries, we begin with basic definitions. Let $D = [D_1, D_2, \ldots, D_m]$ be a dataset containing m records of n categorical random variables $X_1, X_2, \ldots, X_n$. A simple example of a counting query is $\text{COUNT}((X_i = x_i) \wedge (X_j = x_j) \wedge \ldots)$, which returns the number of instances in D matching the specified configuration. Considering database D from Fig. 1a, the result of query $\text{COUNT}((X_1 = 0) \wedge (X_2 = 0) \wedge (X_3 = 2))$ is 2, as two observations in D satisfy the query condition. A more advanced class of queries involves executing a group of consecutive queries over the same set of variables. Let $Pa(X_i)$ (called parent set) be a subset of $\mathcal{X} - \{X_i\}$, and consider a query $\text{COUNT}(X_i \mid Pa(X_i))$. Such query produces two types of results: N_{ij} counts representing the number of observations where $Pa(X_i)$ is in configuration j, and N_{ijk} counts with additional conditioning on X_i being in state k. Fig. 2 shows N_{ijk} counts for the query $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ in contingency table form.

	X_1	X_2	X_3
D_1	1	0	2
D_2	2	1	1
D_3	0	0	2
D_4	1	1	0
D_5	0	0	0
D_6	0	1	0
D_7	0	0	2
D_8	1	1	0
D_9	0	0	0
D_{10}	1	1	0

(a)

0	1	2
0	0	1
0	1	0
0	0	1
1	0	0
1	0	0
1	0	0
0	0	1
1	0	0
1	0	0
1	0	0

(b)



(c)

Fig. 1. (**a**) Database D with three variables. (**b**) Bitmap representation of X_3 variable. (**c**) Example of executing query $\text{COUNT}(X_1 \mid \{X_2, X_3\})$ over D using the Bitmap strategy.

An interesting approach for efficient execution of counting queries, called SABNAtk, is proposed in [5]. The introduced idea is to abstract counting queries and their context, allowing N_{ij} and N_{ijk} counts to be aggregated in a streaming manner, decoupled from downstream processing. SABNAtk provides two memory-efficient strategies that outperform commonly used approaches, such as ADTrees and hash tables. The first is the Bitmap strategy, which encodes variables as bitmaps, reducing counting to logical AND operations and bit counting. The second strategy is inspired by the Radix sort and involves columnar data partitioning.

	$Pa\{X_1\}=\{X_2, X_3\}$					
	00	01	02	10	11	12
0	2		2	1		
X_1 1			1	3		
2					1	

Fig. 2. Contingency table for query $\text{COUNT}(X_1 \mid \{X_2, X_3\})$

2. Methods for Optimization of Counting Query Execution

Our previous research [3] showed that the performance of counting query strategies depends on various factors (e.g., query variables, data complexity), and therefore, no single strategy can be assumed a-priori to be optimal across all scenarios. To address this, we proposed [4] an auto-scheduling mechanism that combines the advantages of three individual strategies – Bitmap, Radix, and Contingency Table – to reduce the query stream execution time. Our approach uses online regression for classification and for each query from the stream selects the strategy with the lowest estimated cost. Using Bayesian network learning as a use case, we observed multiple speedups compared to the best possible individual strategy applied to the entire query stream.

In this work, we aim to further improve the execution efficiency of counting queries. One of the directions we consider is optimizing individual strategies within the developed auto-scheduling mechanism. Specifically, we focus on enhancing the performance of the Bitmap strategy. The core idea behind the Bitmap approach is to represent query variables as bitmaps (see Fig.1b) and reduce the counting process to performing logical AND operations on bitmaps and counting the resulting set bits. To compute the counts, the strategy performs a DFS over a tree that captures all possible configurations of the variable and its parents. A simple example of such a tree is shown in Fig.1c, where the numbers N_{ij} and N_{ijk} represent the counts produced by query $\text{COUNT}(X_1 \mid \{X_2, X_3\})$. With the memory-efficient SIMD implementation, the Bitmap strategy has proven [5] to be highly effective in many practical machine learning applications.

Algorithm 1 QUERY(X_i, Pa, F, b)

```
1: if |Pa| = 0 then
2:     N_ij ← |b|
3:     S_ijk ← 0
4:     for v ∈ [0, ..., r_i − 1] do
5:         b_v ← {p | D_i[p] = v}
6:         N_ijk ← |b ∩ b_v|
7:         if N_ijk > 0 then
8:             F(N_ijk, N_ij)
9:             S_ijk ← S_ijk + N_ijk
10:            if S_ijk = N_ij then
11:                break for
12: else
13:    X_h ← HEAD(Pa)
14:    S_ij ← 0
15:    for v ∈ [0, ..., r_h − 1] do
16:        b_v ← {p | D_h[p] = v}
17:        N_ij ← |b ∩ b_v|
18:        if N_ij > 0 then
19:            S_ij ← S_ij + N_ij
20:            QUERY(X_i, TAIL(Pa), F, b ∩ b_v)
21:            if S_ij = |b| then
22:                break for
```

However, despite these achievements, our analysis reveals that performance of the Bitmap strategy can still be improved by optimizing DFS traversal. As a result, we propose two methods that significantly reduce counting query execution time. These improvements are shown in Algorithm 1, where the original Bitmap strategy is extended with new lines, highlighting the two introduced optimizations in different colors. The first developed method (marked in magenta in Algorithm 1) aims to reduce the number of AND operations performed in the last layer of a DFS tree. The key idea behind this optimization is the observation that, for a given combination of query variables, the count N_{ij} is always equal to the sum of the corresponding N_{ijk} values, as illustrated in Fig. 3a. Therefore, by tracking the cumulative sum of computed N_{ijk} values for a given node, we can reduce the number of performed AND operations. Once this sum reaches N_{ij}, further bitmap intersections can be skipped, as they would lead only to zero values for N_{ijk} counts.

To further improve the Bitmap strategy, the second method (marked in blue in Algorithm 1) focuses on minimizing the number of visited nodes in the entire DFS tree. Unlike the first optimization, which targets only the final layer of a tree, this method generalizes a similar principle and applies it across all other layers. This approach relies on the fact that for any node at layer L, the count N_{ij} is always equal to the sum of the counts computed for its child nodes at the subsequent layer $L+1$, as shown in Fig. 3b. Therefore, once the cumulative sum of computed N_{ij} values at layer $L+1$ reaches the N_{ij} value of the parent node at layer L, the remaining nodes at $L+1$ layer can be skipped, as they would produce only zero values for N_{ij} counts. It is important to note that removing a node at any level also eliminates the entire subtree rooted at that node, including all its children. As a result, this pruning mechanism significantly reduces the number of visited nodes and improves the overall efficiency of the counting process.

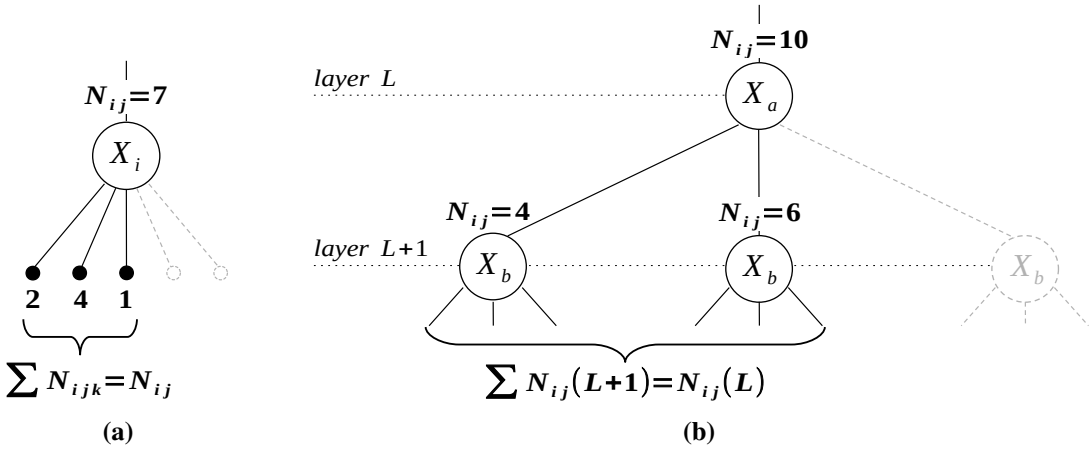


Fig. 3. (**a**) A single leaf node of the DFS tree built for an example counting query. The cumulative sum of the first three N_{ijk} values matches N_{ij}, allowing the last two bitmap intersections to be skipped. (**b**) An example internal node in the DFS tree with three children. The third child node and its subtree can be pruned as the cumulative sum of N_{ij} values at layer $L+1$ reaches N_{ij} value of the parent node at layer L.

Table 1. Speedups from optimizations in the Bitmap strategy on the random query stream benchmark

Dataset	n	Speedup			
		$m = 1$K	$m = 10$K	$m = 100$K	$m = 1$M
Child	20	1.26	1.35	1.35	1.30
Insurance	27	1.24	1.30	1.33	1.30
Mildew	35	1.48	1.62	1.76	2.25
Barley	48	1.44	1.54	1.56	1.60
Pathfinder	109	1.53	1.57	1.66	1.53

To evaluate the effectiveness of the proposed methods, we conducted a benchmark based on processing a stream of randomly sized counting queries. Tests were performed on several datasets commonly used in machine learning [2], differing in the number (n) of variables. Each dataset was evaluated in four versions with varying numbers of observations (m ranging from 1K to 1M), as dataset size is one of the key factors influencing query execution time. For each configuration, the same query stream was executed twice – once using the original Bitmap strategy and once with the optimized version. Table 1 presents the resulting speedups, calculated as the ratio of unoptimized to optimized runtime. The observed values range from 1.24× to 2.25×, with an average of 1.5×. Given the scale of real-world machine learning applications, where query volumes often reach billions, such improvements can significantly enhance efficiency.

Currently, we are working on further tailoring counting process to modern computing architectures. One direction involves developing methods that fully utilize current-generation ccNUMA systems. At the same time, we are exploring potential of RISC-V architecture for efficient query processing. These efforts aim to support the development of scalable, architecture-aware data processing techniques, increasingly important in the design of modern information systems.

References

[1] Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: ACM SIGMOD Conf. on Management of Data. pp. 207–216 (1993)

[2] Bayesian network repository. `https://www.bnlearn.com/bnrepository`

[3] Bratek, P., Szustak, L., Zola, J.: Parallelization and auto-scheduling of data access queries in ML workloads. In: Euro-Par 2021: Parallel Processing Workshops (2022)

[4] Bratek, P., Szustak, L., Zola, J.: Parallel auto-scheduling of counting queries in ML applications on HPC systems. In: Euro-Par 2023: Parallel Processing Workshops (2024)

[5] Karan, S., Eichhorn, M., Hurlburt, B., Iraci, G., Zola, J.: Fast counting in machine learning applications. In: Uncertainty in Artificial Intelligence (2018)

[6] Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (2009)

[7] Quinlan, J.: Bagging, boosting, and C4.5. In: AAAI Innovative Applications of Artificial Intelligence Conferences. pp. 725–730 (1996)

[8] Ramos, J.: Using TF-IDF to determine word relevance in document queries. In: Instructional Conference on Machine Learning. pp. 133–142 (2003)

[9] Salakhutdinov, R., Hinton, G.: Deep Boltzmann machines. In: Int. Conference on Artificial Intelligence and Statistics. pp. 448–455 (2009)