Parallel Auto-Scheduling of Counting Queries in Machine Learning Applications on HPC Systems

Pawel Bratek[1(✉)], Lukasz Szustak[1], and Jaroslaw Zola[2]

[1] Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland
pawel.bratek@pcz.pl, lszustak@icis.pcz.pl
[2] University at Buffalo, Buffalo, NY 14260, USA
jzola@buffalo.edu

Abstract. We introduce a parallel mechanism for auto-scheduling data access queries in machine learning applications. Our solution combines the advantages of three individual strategies to reduce the time of query stream execution. Using bayesian network learning as a use case, we achieve several times speedup compared to the best possible strategy on two different computing servers.

1 Introduction and Problem Formulation

Counting queries are the most basic operation utilized by machine learning (ML) algorithms. Their task is to count data records with instances supporting specific configurations of the selected variables. This problem arises every time a probability distribution has to be estimated and hence applies to many applications, like evaluating a scoring function while training Bayesian network structure [5] or assessing support and confidence in association rule mining problems [1]. Other relevant application areas could be classification [6], deep learning [8] or information retrieval [7].

	X_1	X_2	X_3
D_1	2	0	1
D_2	0	1	0
D_3	1	0	1
D_4	2	0	1
D_5	1	1	0
D_6	2	0	1
D_7	0	2	1

Fig. 1. Example of database with three variables

Consider a set of n categorical random variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, where each variable X_i may be represented by different number of r_i states $[x_{i1}, \ldots, x_{ir_i}]$. Let $D = [D_1, D_2, \ldots, D_m]$ be a complete database containing m instances (observations) of $\mathcal{X}$, where each row D_i, records observed states of all n variables in $\mathcal{X}$. Given D, and a set of input query variables $\{X_i, X_j, \ldots\} \subseteq \mathcal{X}$, the counting query $\textsc{Count}((X_i = x_i) \wedge (X_j = x_j) \wedge \ldots)$ returns the number of instances of D that support the specific configuration $[x_i, x_j, \ldots]$ of variables $[X_i, X_j, \ldots]$. For instance, given the database D shown in Fig. 1, the answer to query $\textsc{Count}((X_1 = 2) \wedge (X_2 = 0) \wedge (X_3 = 1))$

D. Zeinalipour et al. (Eds.): Euro-Par 2023 Workshops, LNCS 14352, pp. 327–333, 2024.
https://doi.org/10.1007/978-3-031-48803-0_12

is 3, as there are 3 records matching the query condition. In many practical scenarios, simple counting queries COUNT are issued in batches of consecutive queries over the same set of variables. We can perceive such a batch of queries with shared context as one query $\mathrm{COUNT}(X_i, X_j, \ldots)$ in which the task is to retrieve all non-zero responses to queries COUNT for all possible configurations of query variables. For example, the result of executing query $\mathrm{COUNT}(X_2, X_3)$ is the following list of counts: $[((0,1),4), ((1,0),2), ((2,1),1)]$, where each element is in the form $((x_i, x_j), \textsc{Count}((X_2 = x_i) \wedge (X_3 = x_j)))$. Finally, let $Pa(X_i)$ (called parent set) be a subset of $\mathcal{X} - \{X_i\}$ and consider query of type $\mathrm{COUNT}(X_i \mid Pa(X_i))$, which details responses for query $\mathrm{COUNT}(Pa(X_i))$ depending on possible states of given variable X_i. Figure 2 shows the answers to an example query $\mathrm{COUNT}(X_1 \mid \{X_2, X_3\})$ in a contingency table containing only non-zero responses.

$Pa\{X_1\}=\{X_2, X_3\}$

X_1	00	01	10	11	20	21
0			1			1
1		1	1			
2		3				

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

The importance of counting queries comes from the fact that in many applications, they account for more than 90% of the total execution time [4]. Therefore, improving the performance of counting can significantly boost the performance of multiple applications. At the same time, counting is usually treated as a black-box procedure and neglected by implementing it using simple but not necessarily efficient strategies, e.g. contingency tables. While this approach may benefit from fast memory access, it often becomes impractical due to its sparse nature and prohibitive memory complexity. An alternative approach is to use a dictionary (e.g. hash table) that addresses the problem of sparsity and memory (it stores only assignments observed in D) but imposes non-trivial overheads of hashing and traversing scattered memory. More advanced techniques like ADTrees, which rely on data indexing, have limited applicability due to extensive preprocessing time and memory requirements.

A novel approach to the problem of counting, called SABNAtk, was proposed in the work [4]. SABNAtk is a framework which abstracts counting queries and their context such that the counts can be aggregated as a stream irrespective of the user-defined downstream processing. In addition to a simple contingency table strategy (`ct`), it implements two memory-efficient data traversing algorithms that outperform commonly used ADtrees and hash tables. First is the Bitmap strategy (`bv`), in which the idea is to represent query variables as bitmaps and reduce the counting process to performing logical AND operations on bitmaps and counting of resulting bits. The second is the Radix strategy (`rad`), which derives from the classic Radix sort algorithm and involves columnar data partitioning. Figure 3 shows two representative examples illustrating the performance of individual strategies depending on query size. From these plots, we can see that no single approach dominates the others, and for a given input

data, the best method depends on query properties. At the same time, choosing the proper strategy could significantly reduce the query execution time, as we noticed for the first time in [3] and presented a preliminary idea of a mechanism for selecting strategy at runtime. This work describes a developed and implemented solution, including parallel support to enable deployment in large-scale multi-threaded environments. We also demonstrate substantial results achieved in real-world machine learning application on two different HPC architectures.

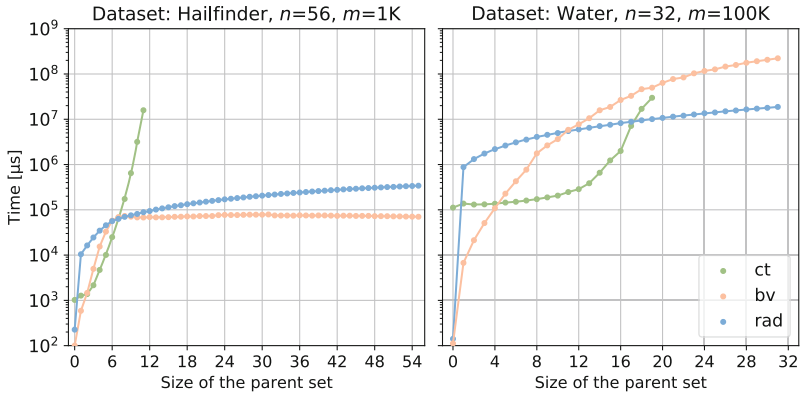

Fig. 3. The execution time of counting queries by different strategies depending on the size of the parent set (shorter is better). Each point represents an average time of one hundred randomly generated queries of a given size.

2 Proposed Approach

Our key idea is to develop a mechanism that selects the optimal strategy for each individual query from the stream based on its execution cost predicted by regression models. Apart from experimentally assessing the performance of individual strategies, we analyse their asymptotic average complexity. We consider a query $\mathrm{COUNT}(X_i \mid Pa(X_i))$ and assume that each variable of $Pa(X_i)$ derives from a multinomial distribution with K equiprobable states. This simplifying assumption is necessary for the `bv` strategy as its complexity follows directly from the properties of input data due to implementing the DFS algorithm. While the cost of computing intersections in each DFS node is constant and equal to m, the number of nodes is challenging to assess a-prior. Therefore, we analyse a single tree node and determine the probability of not removing it during traversal. Next, we use the Bernoulli scheme and properties of expectation to establish the cost of `bv` as $m \times \sum_{L=0}^{N} K^L \cdot (1 - (1 - \frac{1}{K^L})^m)$. Based on this result and complexity analysis of the two remaining strategies, we formulate functions (1) - (3) that let us model query processing time. The independent variable x of these functions is the size of the query, and the remaining parameters λ, β and γ are regression coefficients, which we want to learn by executing queries. We omit

the parameter m, as in typical deployment, it is constant due to processing a stream of queries for a given dataset. Furthermore, to enable the usage of linear regression, we model the cost of `bv` and `ct` strategies using piecewise functions dividing the domain at $x_0 = \log_\kappa(m)$. It is not necessary for the `rad` strategy, which is asymptotically linear relative to the number of query variables.

$$C_{ct}(x, \boldsymbol{\alpha}) = \begin{cases} \alpha_1 \cdot x + \alpha_2 & x \leq x_0 \\ \alpha_3 \cdot K^{\alpha_4 \cdot x} & x > x_0, \end{cases} \tag{1}$$

$$C_{bv}(x, \boldsymbol{\beta}) = \begin{cases} \beta_1 \cdot K^{\beta_2 \cdot x} & x \leq x_0 \\ \beta_3 \cdot x + \beta_4 & x > x_0, \end{cases} \tag{2}$$

$$C_{rad}(x, \boldsymbol{\gamma}) = \gamma_1 \cdot x + \gamma_2, \tag{3}$$

Since the performance of our methods depends on many factors (e.g. properties of input data, size of the query, computing platform), we train the models online, i.e. along with processing the actual stream of queries. Initially, we select strategies in a round-robin style. For each query, we measure the time of its execution and apply it to update the corresponding regression coefficient. As this is linear regression, we must calculate only mean, variance and covariance, which have stable and extremely fast incremental algorithms, e.g. [10]. When a given strategy handles some assumed number of queries, we remove it from the round-robin queue. We notice that this moment may come at a different time for each method. The reason is that depending on the input data, some approaches may be unable to process all assigned queries due to exceeding available memory (see `ct` in Fig. 3). In practice, we handle it by delegating problematic queries to the best available method. When all models (1) - (3) are trained, we start using them to estimate the performance of strategies. Specifically, for a given query, we choose an approach with the lowest predicted execution cost.

Furthermore, to enable the deployment of the developed mechanism in real-world large-scale scenarios, we implement it in a way that supports multi-threaded execution. The fact that each strategy within SABNAtk is stateless favours the execution of queries in the task-parallel model. However, the introduced mechanism must store the state (e.g. model parameters), causing potential race condition issues. Therefore, we protect all problematic procedures with mutexes, making the developed mechanism thread-safe.

Figure 4 shows two examples of applying the developed mechanism for handling a stream of random queries. From these plots, we see that our cost models quickly get an initial estimation of parameters, and as stream execution progresses, queries are handled by the strategies that offer the lowest cost of execution.

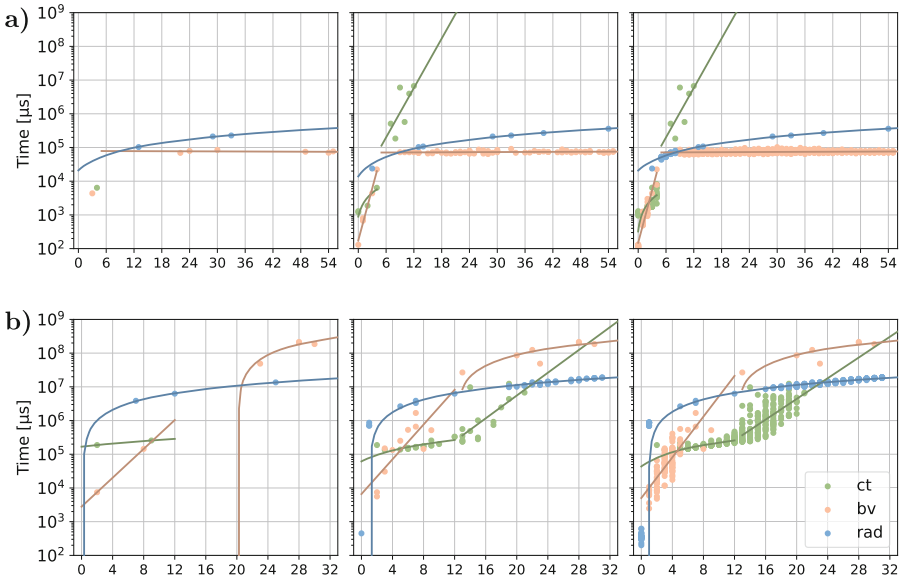

Fig. 4. The regression models for: a) Hailfinder ($n = 56$, $m = 1K$) and b) Water ($n = 32$, $m = 100K$) datasets after processing 10, 100 and 1000 queries. Each point represents the time of one query execution. X-axis is the size of the query.

3 Experimental Validation

We assess how our auto-scheduling mechanism performs in the actual real-world application related to Bayesian network structure learning. Specifically, we test the performance of the parent set assignment solver [4] using popular machine learning benchmark datasets [2] containing 100,000 instances. For given $\mathcal{X}$ and D, queries of the form $\text{COUNT}(X_i \mid Pa(X_i))$ are performed for each X_i, where Pa iterates over all possible subsets of $\mathcal{X} - \{X_i\}$, starting from empty set. Consequently, at level $i = 0, \ldots, n-1$, we have that $|Pa| = i$, and there are total $\binom{n-1}{i}$ queries to execute, creating an interesting pattern of queries that grow in size as computations progress. Internally, the solver uses MDL [9] scoring function and implements several optimizations to reduce the number of queries based on the results from previous levels. It also leverages Intel TBB to execute multiple queries in parallel.

Table 1 shows that the performance of individual strategies (`bv`, `rad` or `ct`) depends not only on the given dataset but even on the computing platform. For instance, while `bv` is the fastest strategy for the Insurance dataset running on the AMD platform, it loses with `rad` on the Intel platform and the same dataset. Our auto-scheduling mechanism (called `auto`) addresses all these issues, becoming the best approach in all test cases. At the same time, it offers significant improvement, even compared to the best possible strategy. Depending on the

dataset, speedup ranges from 5.84 to 11.26 on the AMD platform and 2.29 to 3.99 on the Intel.

Currently, we are working on adapting the counting process to modern HPC systems. Our preliminary results show that using the full potential of state-of-the-art ccNUMA architectures opens a way to enhance the speed of counting queries for the most demanding databases containing millions of records.
At the same time, we consider the problem of counting from an algorithmic perspective. In real-world applications, it is common that consecutive queries share some of the query variables. Hence, we aim to develop a query queuing and rewriting mechanism to mitigate redundant data accesses.

Table 1. The total execution time of the parent set assignment solver (in seconds)

Platform	Dataset	n	bv	rad	ct	auto
AMD EPYC 7763 2 × 64 cores 512 GB RAM	Child	20	167.13	135.97	65.44	11.20
	Insurance	27	1210.42	1461.15	∗	107.46
	Mildew	35	1788.61	382.83	∗	46.95
	Barley	48	7302.15	1256.18	∗	202.47
Intel Xeon Gold 6240 2 × 18 cores 192 GB RAM	Child	20	838.00	290.42	166.36	74.32
	Insurance	27	6197.68	3175.13	∗	796.70
	Mildew	35	9275.16	860.44	∗	248.45
	Barley	48	38489.10	3091.74	∗	973.32

∗ – strategy could not complete the test due to running out of system memory

Acknowledgements. This research was supported by the project financed under the program of the Polish Minister of Science and Higher Education under the name "Regional Initiative of Excellence" in the years 2019–2023 project number 020/RID/2018/19 the amount of financing PLN 12,000,000.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: ACM SIGMOD International Conference on Management of Data, vol. 22, issue 2, 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, pp. 525–529 (2022)
4. Karan, S., et al.: Fast counting in machine learning applications. In: Uncertainty in Artificial Intelligence (2018). arXiv: 1804.04640
5. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press, Cambridge (2009)
6. Quinlan, J.R.: Bagging, Boosting, and C4.5. In: AAAI Innovative Applications of Artificial Intelligence Conferences, pp. 725–730 (1996)

7. Ramos, J.: Using TF-IDF to determine word relevance in document queries. In: Instructional Conference on Machine Learning, pp. 133–142 (2003)
8. Salakhutdinov, R., Hinton, G.: Deep Boltzmann machines. In: International Conference on Artificial Intelligence and Statistics, pp. 448–455 (2009)
9. Schwarz, G.: Estimating the dimension of a model. Ann. Stat. **6**, 461–464 (1978)
10. West, D.H.D.: Updating mean and variance estimates: an improved method. Commun. ACM **22**, 532–535 (1979)