A Comparison of Cache Blocking Methods for Fast Execution of Ensemble-based Score Computation

Xin Jin, Tao Yang, Xun Tang
University of California at Santa Barbara, CA 93106, USA
{xin_jin, tyang, xtang}@cs.ucsb.edu

ABSTRACT

Machine-learned classification and ranking techniques often use ensembles to aggregate partial scores of feature vectors for high accuracy and the runtime score computation can become expensive when employing a large number of ensembles. The previous work has shown the judicious use of memory hierarchy in a modern CPU architecture which can effectively shorten the time of score computation. However, different traversal methods and blocking parameter settings can exhibit different cache and cost behavior depending on data and architectural characteristics. It is very time-consuming to conduct exhaustive search for performance comparison and optimum selection. This paper provides an analytic comparison of cache blocking methods on their data access performance with an approximation and proposes a fast guided sampling scheme to select a traversal method and blocking parameters for effective use of memory hierarchy. The evaluation studies with three datasets show that within a reasonable amount of time, the proposed scheme can identify a highly competitive solution that significantly accelerates score calculation.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

Keywords

Ensemble methods; query processing; cache locality

1. INTRODUCTION

Ensemble-based machine learning techniques have been proven to be effective for dealing data-intensive applications with complex features and document ranking is a representative application benefiting from use of the large number of ensembles. For example, in the Yahoo! learning-to-rank challenge [7], all winners have used some forms of gradient boosted regression trees, e.g. [8]. The total number of trees reported for ranking can be upto 3,000 to 20,000 [10, 5, 11], or even 300,000 or more using bagging method [13]. Ranking for large ensembles is expensive. As reported in [14], it takes more than 6 seconds to rank the top-2000 results for a query processing a 8,051-tree ensemble and 519 features per document on an AMD 3.1 GHz core. If such an algorithm is used to compute scores for a large number of vectors in applications such as classification, the total job is also very time consuming.

The previous work addressed the speedup of runtime execution for ensemble-based ranking in several aspects including tree trimming [3] for a tradeoff of ranking accuracy and performance, earlier exit [6], and loop unrolling [4], and ensemble restructuring for a tree-based model [12]. Memory access can be 100x slower than L1 cache and un-orchestrated slow memory access incurs significant cost, dominating the entire computation. The work shown in [14, 12] proposes a cache-conscious blocking method for better cache locality. However, there are other block methods to select and it is an open problem how to identify the best cache blocking method and parameter settings given different data and architecture characteristics. Experimentally determining this choice can be extremely time-consuming and the comparative result may not be valid any more with a change of underlying feature vector structure or architecture. This paper provides an analysis of multiple blocking methods with different data traversal orders, which provides better insight on program execution performance and leads a fast approximation to select the optimized structure.

Here, we consider the fast computation of ensemble-based scoring that aggregates and derives final scores for n feature vectors using m ensembles. For testing and comparing performance in ranking q sampled queries, the time cost for searching through all combinations can be as high as $O(m^2 * n^2 * q)$. The main contribution of this work is to develop an analytic framework to compare memory access performance of data traversal under multi-level caches to find the fastest program execution with effective use of memory hierarchy. Our scheme results in a much smaller complexity with $O(m * n * q)$. Our experiments with three datasets corroborate the effectiveness of search cost reduction while the guided approximation identifies a highly competitive blocking choice. We also demonstrate the use of this scheme with QuickScorer [12] and for batched query processing.

The rest of the paper is organized as follow. Next, we describe the background information and related work. Section 3 discusses the design considerations. Section 4 gives a comparative analysis on different blocking methods. Section

SIGIR '16, July 17 - 21, 2016, Pisa, Italy
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4069-4/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2911451.2911520

5 presents evaluation results. Finally, Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

Given n feature vectors and an ensemble model that contains m scorers, these vectors and scorers fit in memory. The ensemble computation calculates a score for each feature vector and each scorer contributes a subscore to the overall score for a vector. For example, for ranking a document set with an additive regression tree model [8, 5], each document is represented as a feature vector and each tree can be stored in a compact array-based format [4]. Following the notation in [6], Algorithm 1 shows the SD method with the two-loop standard execution order. At each out loop iteration i, all scorers are used to gather subscores for a vector before moving to another vector. The dominating cost is slow memory accesses when scorers read feature vector values and update partial values.

Algorithm 1: SD standard method for score calculation.

```
for $i$ = 1 to $n$ do
    for $j$ = 1 to $m$ do
        Update score for vector $i$ with scorer $j$.
```

Tang et. al [14] proposed a 2D cache blocking structure called SDSD as depicted in Algorithm 2 which partitions the program in Algorithm 1 into four nested loops. The inner two loops process d feature vectors with s trees. To simplify the presentation, we assume n/d and m/s are integers. By fitting the inner block in fast cache, this method can be much faster than SD. There are other possible cache blocking methods with different data traversal orders and it is an unanswered question on how to choose among them. Also, in [14] there is no cost analysis on how to set a proper parameter for the size of blocking in terms of s and d values. While choices of their values can be restricted to fit in the fast cache, they can still be fairly large. For example, s and d can still reach upto 3,276 and 11,440 respectively in some of our experiments shown in Section 5. Assume m and n are smaller than these upper bound numbers, s ranges from 1 to m and d ranges from 1 to n and there are $m * n$ combinations to compare as they all fit in different levels of cache. Since running each test query takes $O(n * m * q)$, the total cost is $O(m^2 * n^2 * q)$. For instance, given $n = 10,000$, $m = 3,000$, $q = 1000$, the total time takes over 1,141 years with one core, assuming it takes 40 nanoseconds to compute a partial score for a vector with a scorer. If we sample each of s and d values with step gap 100, the total one-core time is over 41 days without knowing if such sampling finds a solution competitive to the optimum. While running such a sampling can be fully parallelized, we still need a faster scheme with well-guided approximation.

Algorithm 2: 2D blocking with SDSD structure.

```
for $j = 0$ to $\frac{m}{s} - 1$ do
    for $i = 0$ to $\frac{n}{d} - 1$ do
        for $jj = 1$ to $s$ do
            for $ii = 1$ to $d$ do
                Update score for vector $i \times d + ii$ with
                scorer $j \times s + jj$.
```

There are other performance speedup techniques proposed in the previous work to speedup fast ranking score computation, which can be summarized into two categories. The first category is to achieve a tradeoff between ranking efficiency and accuracy. In [6], an early exit optimization was developed to reduce scoring time while retaining a good ranking accuracy. In [16, 17], ranking is optimized to seek the tradeoff between efficiency and effectiveness. Asadi et.al [3] considered the fact that compact, shallow, and balanced trees yield faster computation and generated such trees with trimming technique. The second category is to improve efficiency given a fixed model. The work in [4] proposed an architecture-conscious solution called VPred that converts control dependence of code to data dependence and employs loop unrolling with vectorization. Lucchese et.al proposed the QuickScorer (QS) algorithm [12] which traverses multiple trees in an interleaved manner and accelerates with bit-wise operations. They propose a block-wise variant of QS (called BWQS) by partitioning trees into blocks and applying QS to each block of trees. Given different dataset characteristics, it is an open problem how to find the optimal partitioning. Also there are other ways to arrange blocking and our work is complementary and can be used to compare different options.

3. DESIGN CONSIDERATION AND COST MODEL

There are six ways of loop blocking depending on the order of data traversal: DSD, SDS, DSDS, DSSD, SDDS, and SDSD. Following the naming in [15], symbol D here stands for a loop control over feature vectors and S stands for a loop control over scorers. For example, DSDS means that feature vector traversal is controlled by the outermost and the third outermost loops while scorer traversal is controlled by the second and the innermost loops. The inner two loops access d vectors and s scorers.

Figure 1 illustrates the execution and data traversal order of these methods. Figure 1(a) shows that DSD initially visits one scorer and d vectors. Then it visits another scorer and the same d vectors. Figure 1(b) depicts that SDS initially visits one vector and s scorers. Then it visits another vector and the same s scorers. Figure 1(c) illustrates DSDS which visits scorers and vectors block by block and row by row. Figure 1(f) illustrates SDSD which visits scorers and vectors block by block and column by column.

Our objective is to compare these blocking methods and find a value for s and d to minimize the time cost of score computation under a constraint $1 \leq s \leq m,\ 1 \leq d \leq n$. We have the following considerations.

- For DSD, when the inner most loop uses $d = 1$, it becomes a special case DS which is the same as the traditional loop structure DS shown in Algorithm 1. For SDS, when the inner loop uses $s = 1$, it becomes a special case SD.

- The traversal order of DSSD during execution is the same as that of DSD as illustrated in Figure 1(a) and (d). Thus DSD can represent both during our analysis. Similarly, the traversal order of SDDS is the same as that of SDS as shown in Figure 1(b) and (e). As a result of the above argument, the six types of control are reduced to four.

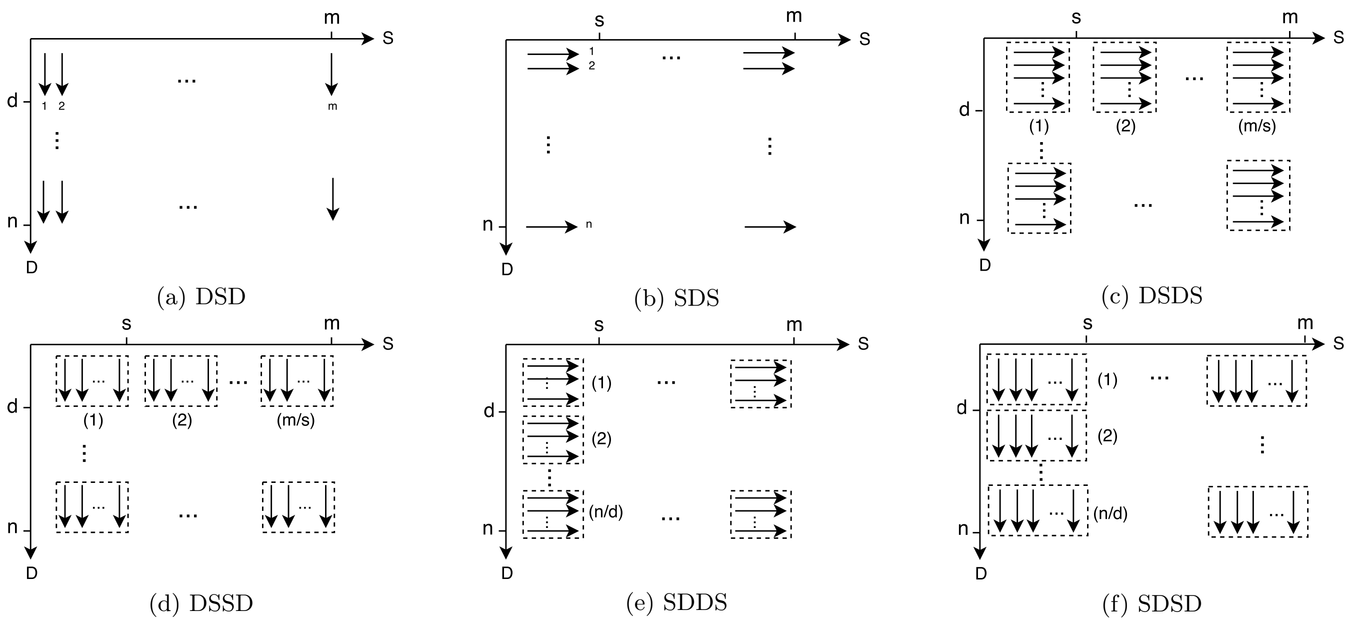

(a) DSD (b) SDS (c) DSDS (d) DSSD (e) SDDS (f) SDSD

Figure 1: Data traversal order of cache blocking methods during execution

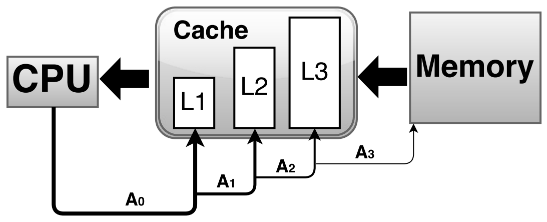


Figure 2: Data access flow of CPU with memory hierarchy.

The following parameters are used in assessing the average memory access cost of processing n feature vectors with m scorers. We assume that CPU has three levels of caches: L1, L2, and L3 and the three level setting is popular in the currently available processors from Intel and AMD. Let δ_1 be the read or write cost of accessing L1 and cost for accessing other cache is δ_1 multiplied by a constant ratio. Namely $c_2\delta_1$ is the cost of accessing L2, $c_3\delta_1$ is the cost of accessing L3, and $c_4\delta_1$ is the cost of accessing memory.

Our analysis separates the cost for accessing feature vectors and scorers. Without losing the generality, let A_i be the total amount of data access to feature vectors at cache level $i+1$ while ηA_i be the total amount of accesses to scorers at cache level $i+1$ and η is the average frequency ratio between access of feature vectors and scorers during computation.

The total data access cost is the summation of the cost of accessing each level of memory hierarchy:

$$\begin{aligned}Cost = & A_0\delta_1(1+\alpha_D c_2+\alpha_D\beta_D c_3+\alpha_D\beta_D\gamma_D c_4)\\ & +\eta(A_0\delta_1(1+\alpha_S c_2+\alpha_S\beta_S c_3+\alpha_S\beta_S\gamma_S c_4)).\end{aligned}$$

where α_S, β_S, γ_S, α_D, β_D, and γ_D are the miss rates of L1, L2 and L3 to access scorers and feature vectors respectively. Data accesses flow from CPU to memory for feature vectors is illustrated in Figure 2. $A_1 = A_0\alpha_D$ is the total number of feature data access to L2 due to their misses to L1; $A_2 = A_0\alpha_D\beta_D$ is the total number of feature data access to L3. $A_3 = A_0\alpha_D\beta_D\gamma_D$ is the total number of data access to memory.

Then the time cost divided by $A_0\delta_1$ is defined as

$$T = \frac{Cost}{\delta_1 A_0} = \eta T_S + T_D$$

where $T_D = 1+\alpha_D c_2+\alpha_D\beta_D c_3+\alpha_D\beta_D\gamma_D c_4$ and $T_S = 1+\alpha_S c_2+\alpha_S\beta_S c_3+\alpha_S\beta_S\gamma_S c_4$.

Since A_0 and δ_1 are constants, in the rest of the analysis, we focus on computing the above data access cost ratio T.

Notice that once data is brought from memory hierarchy, the arithmetic computing cost of all four methods is the same. Thus we just need to analyze and compare the data access cost ratio T for the four traversal methods. In practice, data access cost often weights more than arithmetic cost.

Due to the restriction on the paper length, we first present the analysis of cache performance for DSD and then list the result of DSDS, SDSD, and SDS as the case subdivision and cost derivation process are similar. Finally we describe an approximate scheme to select the best structure by taking advantages of the derived data access cost ratio for the four methods.

4. COST ANALYSIS AND COMPARISON

4.1 Time cost for DSD

4.1.1 Cases under consideration

Algorithm 3: The program structure of DSD method.

```
for all vector blocks do
    for i in all scorers do
        for j in a vector block do
            Update score for vector j with scorer i.
```

Algorithm 3 lists the program control structure of DSD and a vector block contains d vectors. Once a scorer s_i is loaded to cache, it will be used by d vectors in the inner most loop. Then the next scorer s_{i+1} will go through the same d vectors. If we choose d properly such that d vectors fit in cache, we do not need to load them from memory for each scorer. Figure 3 illustrates how the cost of score computation could change when value d increases from 1 to n. The impact of d value on the cost is segmented with respect to the size of L1, L2, and L3. When d is small, the d vectors can fit in L1 cache, and there is an advantage of reusing these d vectors within L1 cache. Thus d should be as large as possible. When d value becomes too big, the benefit of leveraging L1 cache decreases because d vectors may not fit in L1 any more and therefore the access cost can increase with larger d value. We can reason similarly when d vectors fit or do not fit in L2 and L3 caches.

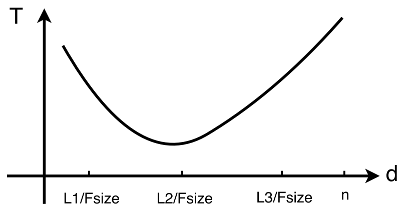


Figure 3: Performance under different values of d.

We will clarify the tradeoff of increasing d value when we derive a more concrete analysis. Let $Fsize$ be the average data size of each feature vector. Without introducing more symbols, we also let $L1, L2$ and $L3$ represent the size of L1 cache, L2 cache and L3 cache respectively in a formula expression. To assess the impact of increasing d values, we divide the increasing range into four parts as illustrated in Figure 3.

- d vectors fits in L1 cache. Namely $d \leq \frac{L1}{Fsize}$
- d vectors do not fit in L1 cache, but fit in L2 cache. $\frac{L1}{Fsize} < d \leq \frac{L2}{Fsize}$
- d vectors do not fit in L2 cache, but fit in L3 cache. $\frac{L2}{Fsize} < d \leq \frac{L3}{Fsize}$
- d vectors exceed L3 cache and but fit in memory. $\frac{L3}{Fsize} < d \leq n$.

The cache access behavior of inner most loop in Algorithm 3 is affected by the average size of each scorer. For example, a larger scorer footprint leaves little space for L1 to host feature vectors. Figure 4 illustrates that we need to consider the following four scenarios and for each scenario, we need to further consider the four d range cases discussed above. Let $Ssize$ represent the average data size of each scorer and the four scenarios corresponding to the root branches in Figure 4 are defined as follows.

- **Scenario 1:** $Ssize \leq L1$. When a scorer can fit in L1 cache, there are four cases for the d vectors: the vector block fits in L1 cache, L2 cache, L3 cache or memory.
- **Scenario 2:** $L1 < Ssize \leq L2$. When a scorer size is between L1 and L2 cache sizes, the vector block can only fit in a higher level of cache (say L2 or L3 cache). Otherwise, old vectors will be kicked out from L1 cache by the scorer and the vector block could not stay in L1 cache. Thus there are only three cases for the d vectors as depicted in the second root branch of Figure 4: the vector block fits in L2 cache, L3 cache or memory.
- **Scenario 3:** $L2 < Ssize \leq L3$. When a scorer size is inbetween L2 and L3 cache size, there are two cases for the d vectors: the vector block fits in L3 cache or memory.
- **Scenario 4:** $Ssize > L3$. When a scorer cannot fit in L3 cache, the d feature vectors will not be able to fit in L3 cache also and they can only fit in memory.

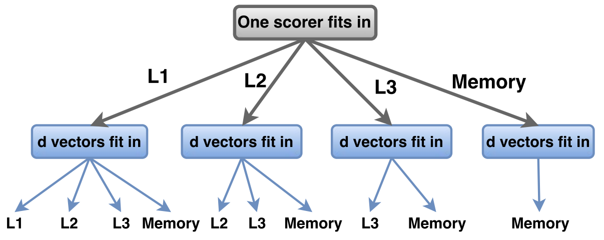


Figure 4: Range cases of d considered under different scenarios for DSD.

To simplify the analysis, we assume that m and n are sufficiently large so that m scorers do not fit in L3 cache, and also n feature vectors do not fit in L3 cache.

4.1.2 DSD under Scenario 1

Under Scenario 1, we first compute T_S as follows. Since each scorer is loaded once for the inner most and will reused d times for computing the subscores for d vectors in the inner most loop. Then the L1 cache miss ratio $\alpha_S \approx 1/d$. If there is an L1 cache miss for a scorer, L2 cache miss and L3 cache miss can occur with a high chance because the unseen new scorer has not been used ever and thus it is fetched from memory. Thus $\beta_S \approx 1$ and $\gamma_S \approx 1$.

$$T_S \approx 1 + \frac{c_2}{d} + \frac{1}{d}(c_3 + c_4) \approx 1 + \frac{c_4}{d}$$

We shall estimate T_D under 4 different ranges of d values following Figure 3. We call these 4 range cases under DSD as DSD_i where $1 \leq i \leq 4$. The total cost ratio of accessing scorers for DSD is

$$T_{DSD_i} = \frac{Cost}{\delta_1 A_0} = \eta T_S + T_D \approx \eta + \frac{\eta c_4}{d} + T_D$$

where

$$T_D = 1 + c_2\alpha_i + \alpha_i\beta_i(c_3 + c_4\gamma_i) \tag{1}$$

and α_i, β_i and γ_i are cache miss rates for accessing feature vectors under range case i. Note that in differentiating these miss rate of different cases, we use script "i" instead of "D, i" in order to simplify the presentation. Table 1 summarizes the cost of DSD for Scenario 1 when each scorer fits in L1 cache on average.

Range Case DSD_1: d vectors fit in L1. Once a scorer is loaded to L1, the inner most loop load d feature vectors to L1 and these vectors stay and will be available in L1 when a new scorer is fetched to L1. Given m scorers, each feature vector in L1 is accessed m times and there is one 1 miss initially and the rest of $m-1$ accesses will hit L1. Thus the L1 cache miss ratio with respect to feature vectors is $\alpha_1 \approx 1/m$. If there is an L1 cache miss for a feature vector, there must be an L2 cache miss and L3 cache miss. Thus, $\beta_1 \approx 1$ and $\gamma_1 \approx 1$. Plugging into Equation 1, we get the total cost ratio:

$$T_D \approx 1 + \frac{c_2}{m} + \frac{1}{m} \cdot (c_3 + c_4).$$

Range case DSD_2**:** d vectors fits in L2. Once a scorer is loaded to L1, the inner most loop can load a feature vector and keep it at least at L2 when a new scorer is loaded. Given there are m scorers, $A_2/A_0 \approx 1/m$. Namely $\alpha_2\beta_2 = A_2/A_0 \approx 1/m$. Since L2 can hold d vectors needed for inner most loop, $A_2 \approx A_3$. Thus $\gamma_2 = A_3/A_2 \approx 1$.

$$T_D \approx 1 + \alpha_2 \cdot c_2 + \frac{1}{m} \cdot (c_3 + c_4).$$

Range case DSD_3**:** d vectors fit in L3. Once a scorer is loaded to L1, the inner most loop can load a feature vector and keep it at least at L3 when a new scorer is loaded. Given there are m scorers, $A_3/A_0 \approx 1/m$. Namely $\alpha_3\beta_3\gamma_3 = A_3/A_0 \approx 1/m$.

$$T_D \approx 1 + \alpha_3 \cdot c_2 + \alpha_3\beta_3 \cdot c_3 + \frac{1}{m} \cdot c_4.$$

Range case DSD_4**:** d vectors donot not fit in L3. In this case, $A_0 \approx A_1 \approx A_2 \approx A_3$. In this case, actually we put n documents in the inner loop. It's obvious to see that L1, L2 and L3's cache miss ratio are all 1 because comparing to the memory size, even L3 cache size is too small.

$\alpha_4 \approx 1$, $\beta_4 \approx 1$ and $\gamma_4 \approx 1$.

$$T_D \approx 1 + c_2 + c_3 + c_4.$$

Cases	d vectors fit in	$T_{SDS_i} = \eta T_s + T_D \approx$
DSD_1	L1	$\eta + \eta\frac{c_4}{d_1} + 1 + \frac{c_2+c_3+c_4}{m}$
DSD_2	L2	$\eta + \eta\frac{c_4}{d_2} + 1 + \alpha_2 c_2 + \frac{c_3+c_4}{m}$
DSD_3	L3	$\eta + \eta\frac{c_4}{d_3} + 1 + \alpha_3 c_2 + \alpha_3\beta_3 c_3 + \frac{c_4}{m}$
DSD_4	memory	$\eta + \eta\frac{c_4}{d_4} + 1 + c_2 + c_3 + c_4$

Table 1: Cost of DSD when 1 scorer fits in L1.

4.1.3 Other Scenarios of DSD

For Scenario 2 when a scorer fits in L2 on average, there are only 3 range cases: DSD_2, DSD_3, and DSD_4. L1 miss rate in T_D becomes 1 and T_S adds c_2 as $T_S \approx 1 + c_2 + \frac{c_4}{d}$. For Scenario 3 where a scorer fits in L3, there are only 2 possible cases to consider: DSD_3 and DSD_4. L1 and L2 miss rates in T_D become 1 and T_S adds c_3 as $T_S \approx 1+c_3+\frac{c_4}{d}$. For Scenario 4 where a scorer fits memory only, there is one case to consider: DSD_4. Its T_D does not change while $T_S \approx 1 + c_4$.

4.2 Cost Comparison of the Four Methods

The data access cost for SDS, DSDS, and SDSD is listed in Appendix A. There is a total of 28 range cases considered in these four methods: DSD_i, SDS_i, DSD_iS_j, and SDS_iD_j where $1 \le i, j \le 4$. The cost results of these 28 cases can be used in the following two aspects.

- Identify the approximated optimum selection from 28 cases instead of exhaustive search of all combinations. The selection algorithm goes through 28 cases and runs the average time performance sampling through a benchmark of q queries.

  From Tables 1, Tables 8, Tables 9, and Tables 10, increasing d or s under its range limit decreases the time cost. Thus d and s should be chosen to be as large as possible. On the other hand, scorers and vectors share each level of cache and we set a constraint that vectors and scorers accessed in the inner most loop of DSD and SDS or in the two inner most loops of DSDS and SDSD fit in the corresponding cache. For example, as a midpoint approximation, we let d vectors occupy upto half of each cache level and s scorers occupy upto half of each cache level. Namely, for $1 \le i \le 3$, d and s are chosen for each range case as: $d_i = \frac{0.5Li}{Fsize}$, $s_i = \frac{0.5Li}{Ssize}$. As all n vectors and m scorers fit in memory, $d_4 = n$ and $s_4 = m$.

- Narrow the search scope when characteristics of a dataset or targeted machine architecture is given because many of 28 cases may be eliminated. That facilitates the reduction of search space and allows more sampling points at each range case selected as long as time complexity permits. For example, the above midpoint sampling for each range case after elimination can be expanded as follows. Since each scorer may only access a fraction of a feature vector and a fraction of the scorer data structure for computation, we choose sampling points as $d_i = \frac{0.5Li}{\mu Fsize}$, $s_i = \frac{0.5Li}{\mu Ssize}$. Coefficient μ represents the average data usage of a vector or a scorer during computation and for example, we sample more points with μ as 1, 0.75, 0.5, and 0.25.

To illustrate the second point above, we show that the following proposition is true and can narrow the search scope from 28 to 4 range cases.

Proposition 1. *When each feature vector fits in L1 and each score fits in L1 on average, and* $\frac{\eta c_4}{d_2} \ll 1$, $\frac{c_4}{s_2} \ll 1$, *the candidates with the lowest access cost are among range cases* DSD_2, DSD_2S_1, SDS_2D_2, *and* SDS_2D_1.

A proof is listed in Appendix B. Yahoo!, MS, and MQ datasets discussed in Section 5 fall into the condition of this proposition when each regression tree used is not too big (e.g. containing upto 50 leaves). The range of d and s values for these datasets is listed in Table 3 and Table 2 of Section 5. When a regression tree contains 150 leaves, ratio c_4/s_2 is getting close to 1, cases SDS_3D_1 and SDS_3D_2 can be competitive as a best candidate. Thus with such a condition, we can search for 6 cases instead of 28 cases.

In summary, a guided sampling scheme conducts the following steps. 1) Identify data and architecture parameters. When possible, apply Proposition 1 or its variation to eliminate some of 28 range cases from the cost analysis of DSD, SDS, DSDS, and SDSD. 2) For each of selected range cases, choose blocking factor d_i and s_i under a constraint that vectors and scorers accessed in the inner most loop of DSD and SDS or in the two inner most loops of DSDS and SDSD fit in the corresponding level of cache. One approach is to choose $d_i = \frac{0.5Li}{\mu Fsize}$ $s_i = \frac{0.5Li}{\mu Ssize}$ with a number of sampled μ values. 3) Run and collect the average query response time with m

scorers and n vectors from each sampled case. Select the case and parameter setting with the lowest response time. The total complexity of this scheme with q test queries is $O(m * n * q)$.

4.3 Discussions

Integration with the QuickScorer method. When the ensemble computation uses the original computing algorithm for gradient boosted regression trees (e.g. [8, 5]), the main data structure of each scorer is a tree. To use the BWQS algorithm [12], we treat each scorer as the application of QS on a block of trees. The following parameters are involved: the size of a scorer changes when different partitioning is adopted while the number of inner-loop scorers (s) and inner-loop vectors (d) can vary too for different blocking methods. Thus we add a partitioning search loop on the top of the aforementioned comparison and sampling scheme to select the best partitioning.

Batched query processing. When the ensemble score computation is used for query processing where n is small, d value of the inner most loop limited by n can be insufficient to explore the cache locality and the effectiveness of blocking degrades. When batch processing is allowed, we can boost the cache utilization by processing feature vectors from multiple queries in fast cache, which essentially raises n values. One application of such batched processing is to conduct an offline experiment to assess the ranking performance of an algorithm in answering a large number of queries and there is no need to output ranking results immediately.

For an online ranking application, the ranking results need to be produced promptly. While reaching a high throughput, batching a large number of queries can increase the average waiting time of batched queries and affect the response time. With this constraint in mind, we set a limit on the largest waiting time allowed in choosing a batch size for a higher throughout with a modest increase of response time.

5. EVALUATIONS

5.1 Settings

This section provides an experimental comparison of different cache blocking methods and validates the effectiveness of the selected method with unoptimized ones. The evaluation tasks are listed as follows: (1) Illustrate the fast comparison of the 28 range cases for using DSD, SDS, SDSD and DSDS with guided sampling. (2) Integrate our cache blocking selection algorithm with the QS algorithm [12] for tree-based ranking. (3) Assess the batched query processing in improving the throughput when n is small.

We implement the blocking methods using C compiled with GCC optimization flag -O3. Experiments are conducted on a Linux CentOS 6.6 server with 8 cores of 3.1GHz AMD Bulldozer FX8120 and 16GB memory. FX8120 has 16KB of L1. We set $L2 \approx 1MB$ as 2MB L2 cache is shared by two cores. Its 8MB L3 cache is shared by 8 cores and since L3 hosts tree data useful for multiple queries, we set $L3 \approx 2MB$. The cache line is of size 64 bytes. For AMD Bulldozer, c_2 is around 7.3, c_3 is around 25.1, and c_4 is around 80.9. We have also conducted experiments in a 24-core Intel Xeon E5-2680v3 2.5 GHz server with $L1 = 32KB$, $L2 = 256KB$,, and $L3 \approx 2.5MB$ per core. The Intel results are similar and thus we mainly report the AMD numbers.

The following learning-to-rank datasets are used as evaluation benchmarks. (1) Yahoo! dataset [7] with 700 features per document feature vector. (2) MSLR-30K dataset [2] with 136 features per document vector. (3) MQ2007 dataset [1] with 46 features per document vector. Table 2 shows the range of d values when fitting d vectors in different cache levels for these 3 datasets.

d vectors fit in	Yahoo!	MS	MQ
L1	$d \leq 5$	$d \leq 30$	$d \leq 89$
L2	$d \leq 373$	$d \leq 1928$	$d \leq 5720$
L3	$d \leq 747$	$d \leq 3856$	$d \leq 11440$
Memory	$d \leq n$	$d \leq n$	$d \leq n$

Table 2: The vector counts for fitting in differnt cache levels.

We use LambdaMART [5] for ranking with additive tree ensembles and derive tree ensembles using the open-source jforests [9] package. To assess score computation in presence of a large number of trees, we have also used a bagging method [13] to combine multiple ensembles and each ensemble contains additive boosting trees. Because the size of a scorer affects the cache performance and parameter choices, we generate the size of each tree with several settings: 10 leaves per tree, 50 leaves per tree, and 150 leaves per tree. Table 3 shows the range of s values when fitting s scorers in different cache levels under three choices of the regression tree size. The η value is about 1 because the basic access operation of a scorer is to fetch 1 tree node and then a document feature. Each of them fits in one cache line. The default total number of trees used is about 20,000 for Yahoo! dataset, 10,000 for MS, and 4,000 for MQ. We also use other numbers of trees in our experiments. When using the QS method [12], each scorer is a meta tree merged from multiple trees and η value is around 4 because the basic access operation of a scorer fetches elements from 4 data structures and then a document feature.

s scorers fit in	10 leaves	50 leaves	150 leaves
L1	$s \leq 25$	$s \leq 5$	$s = 1$
L2	$s \leq 1638$	$s \leq 327$	$s \leq 109$
L3	$s \leq 3276$	$s \leq 655$	$s \leq 218$
Memory	$s \leq m$	$s \leq m$	$s \leq m$

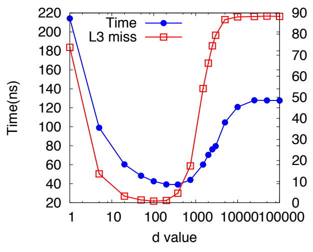
Table 3: The tree counts for fitting different cache levels.

The above data sets contain 23 to 120 documents per query with labeled relevancy judgment. In practice, a search system with a large dataset ranks thousands or tens of thousands of top results after a preliminary selection. To evaluate the score computation in such a setting, we synthetically generate more matched document vectors for each query. In this process, we generate relatively more vectors that bear similarity to those with low labeled relevance scores, because a large percentage of matched results per query are less relevant in practice. The number of vectors per query including synthetically generated vectors varies from 3,000 to 10,000 for Yahoo! dataset, from 2,000 to 6,000 for MS, and from 1,000 to 4,000 for MQ.

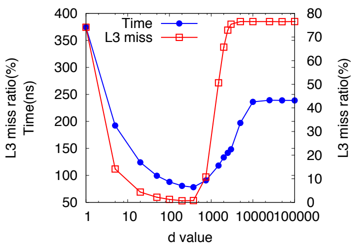
Metrics. We mainly report the average time of computing a subscore for each vector under one tree. With n matched vectors scored using an m-tree model, this scoring time multiplied by n and m is the scoring time per query. The throughput is the number of feature vectors scored per second. The number reported here is measured in a multi-core environment where each query is executed in a single core.

5.2 A comparison of cache blocking methods

Table 4 shows the score computing time of a vector per tree in nanoseconds under different cache blocking cases for Yahoo!, MS and MQ datasets. “Y! 10” means Yahoo! dataset and each regression tree has 10 leaves. Row 2 is the scoring time of DS without cache blocking. The cost of all 28 cases under 4 cache blocking methods using guided sampling are listed, starting from Row 3. Under Proposition 1, our scheme searches the optimum only from four cases DSD_2, DSD_2S_1, SDS_2D_1, and SDS_2D_2. The corresponding entries in this table are marked in a gray color. For Yahoo! dataset with 150 leaves per tree, as we discussed in Section 4.2, extra two cases SDS_3D_1, and SDS_3D_2 are also compared and thus marked in a gray color. For each column from column 2, entry marked ‘$\star$’ indicates the smallest value is found and this entry is considered to be highly competitive. Our comparison scheme selects DSD_2 as the best range case with $d = 373$ for Yahoo! dataset under all three tree size settings, $d = 1928$ for MS 50 leaves case and $d = 5720$ for MQ 10 leaves case. It selects SDS_2D_2 with $d = 1928$ and $s = 1638$ for MS 10 leaves case.



(a) Yahoo! 50 leaves



(b) Yahoo! 150 leaves

Figure 5: Time cost and cache miss of DSD as d varies.

The running cost of the above guided sampling in CPU hours with one core is shown the second row of Table 5 and can be completed within about 10 hours using a 8-core server. We have also conducted exhaustive search with greatly-increased sampling points to obtain an estimated optimum solution. The best cases identified in the estimated optimum are listed in the third row of Table 5 and exactly match what has been selected by our guided sampling scheme. The fourth row of the Table 5 shows the sample error which is the cost difference ratio between the optimum solution and the solution approximated by our scheme. The difference is within 2.2%. The fifth and sixth rows are the best cases and difference ratio obtained on the Intel machine. The error is within 2.4% while all best cases of the estimated optimum match those of the approximated solution. The above result shows that our guided sampling can find a highly competitive blocking solution within reasonable hours using a modest server and such a solution can result in upto 6.57x response time reduction compared to DS without cache blocking.

Impact of blocking size on time cost and cache miss rate. In Section 4, we have used Figure 3 to illustrate the correlation between data access time cost and blocking

	Y! 10	Y! 50	Y! 150	MS 10	MS 50	MQ 10
DS	59.83	214.29	374.56	59.67	206.98	34.89
DSD_1	33.04	99.00	193.95	17.34	47.79	9.82
DSD_2	14.09$\star$	39.11 $\star$	78.11$\star$	14.37	31.49$\star$	8.52$\star$
DSD_3	25.51	71.45	143.55	25.95	49.72	13.84
DSD_4	54.06	126.32	241.47	42.37	77.75	20.13
DSD_1S_1	41.56	125.44	237.35	24.46	63	13.67
DSD_2S_1	21.55	50.41	84.87	17.31	38.61	11.4
DSD_3S_1	25.71	77.75	141.07	18.1	40.38	11.6
DSD_4S_1	40.13	120.84	235.36	19.93	52.93	11.83
DSD_2S_2	29.77	72.42	123.3	30.38	67.33	19.5
DSD_3S_2	30.49	72.79	124.16	31.25	67.71	19.88
DSD_4S_2	29.73	72.74	123.07	31.35	68.45	19.85
DSD_3S_3	39.58	79.73	131.6	40.73	76.2	22.39
DSD_4S_3	46.13	105.91	157.47	46.2	100.58	28.16
DSD_4S_4	59.81	217.14	375.24	59.09	210.08	34.95
SDS_1D_1	60.67	149.17	267.69	23.67	62.61	10.99
SDS_2D_1	23.75	58.28	102.93	16.28	40.53	9.52
SDS_3D_1	27.08	72.31	130.48	16.85	43.22	9.6
SDS_4D_1	31.58	98.72	192.1	17.47	48.65	9.83
SDS_2D_2	14.5	39.75	82.26	13.21$\star$	32.23	8.68
SDS_3D_2	15.71	42.68	88.05	15.09	37.30	10.12
SDS_4D_2	15.55	45.46	88.45	15.07	34.52	10.03
SDS_3D_3	20.76	59.03	120.15	21.55	50.02	13.04
SDS_4D_3	26.28	73.81	143.4	26.28	52.29	13.93
SDS_4D_4	54	131.58	250.17	42.46	81.7	20.13
SDS_1	42.96	113.73	240.89	21.39	50.83	12.62
SDS_2	30.91	72.22	122.22	31.65	67.28	20.27
SDS_3	46.11	107.19	156.67	46.56	100.75	28.78
SDS_4	59.83	214.29	374.56	59.67	206.98	34.89

Table 4: Scoring time of one vector per tree in nanoseconds for different cache blocking range cases.

size in deriving the cost for DSD. Figure 5 shows the experimental result to validate. This figure shows time cost curve of DSD and L3 miss rate measured using Linux tool perf when d value varies for Yahoo! dataset with 50 leaves (a) and 150 leaves (b) per tree. When d is too small, cache is not fully utilized and the cost of T_S for DSD derived in Section 4.1.2 is large. When d is too big which falls into case DSD_3 or DSD_4, the cost curve matches the analysis in Appendix A that $T_{DSD_4} > T_{DSD_3} > T_{DSD_2}$.

There is also a correlation between the overall time cost and L3 cache miss when d varies. For small d, the coefficient for L3 cost c_3 in T_S is big. For large d, there is more L3 cache miss, making T_D bigger as shown in Section 4.

Impact of m and n values on time cost. Figure 6 shows the time cost per tree per document when m changes from 2,000 to 20,000 for all datasets. In this experiment, we generate extra trees for MS and MQ datasets. It shows that with sufficiently large value of m, the cache behavior does not change much and the processing cost is about the same for different m values.

	Y! 10	Y! 50	Y! 150	MS 10	MS 50	MQ 10
Comp. time	10.67h	27.09h	81.86h	2.719h	6.349h	0.424h
Best AMD	DSD_2	DSD_2	DSD_2	SDS_2D_2	DSD_2	DSD_2
Error AMD	0.21%	0.15%	0.10%	0.61%	2.2%	0.47%
Best Intel	DSD_2	SDS_2D_2	SDS_2D_2	DSD_2	SDS_2D_2	DSD_2
Error Intel	1.4%	2.4%	0.64%	0.18%	0.52%	0.14%

Table 5: CPU hours for comparison, sampling errors, and best cases.

Figure 7 shows the time cost per tree per vector when n changes from 1 to 100,000. When n is smaller than 100, the performance drops significantly and cache is not fully utilized. When n is larger than 1000, the cost becomes stable

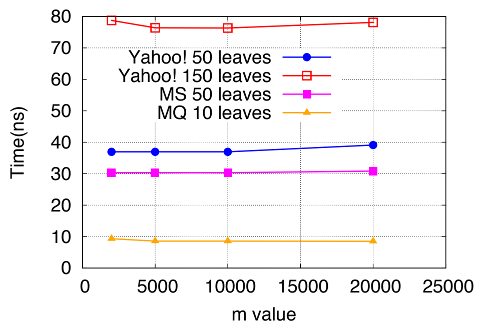


Figure 6: Scoring time per vector per tree when m changes.

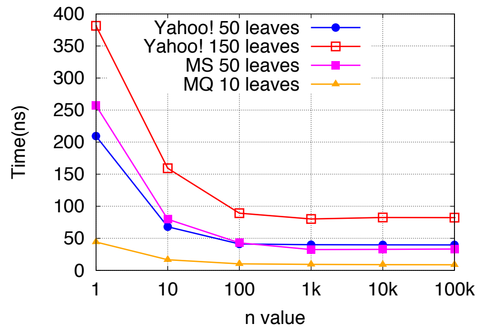


Figure 7: Scoring time per vector per tree when n changes.

and there is no much reduction. We will discuss the experiment results when batched query processing is allowed shortly.

5.3 Selective cache blocking for QuickScorer

We integrate our scheme with the BWQS algorithm [12] as follows. Step 1: Given m trees and let τ be the number of trees that will be merged to use the QS method. The number of scorers is $m' = m/\tau$. Step 2: Given m' scorers and n vectors, use our scheme to find the best blocking method and parameters. Step 3: Repeat Step 1 and Step 2 for a different sampling choice of τ. Step 4: The τ with the smallest time cost yields the best overall performance.

Figure 8 shows the BWQS results under the different number of scorers m' where m is fixed as 20,000 and m' varies from 1 to 20,000. Notice that when m' is small, each scorer contains many trees and does not fit in L1 or even L2. The

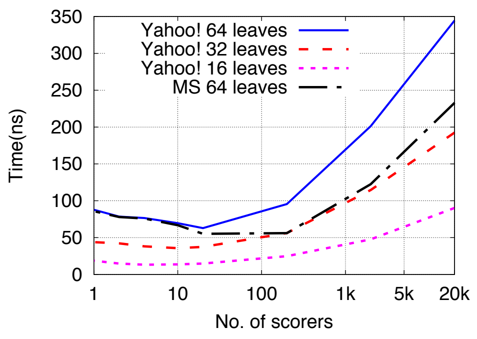


Figure 8: Scoring time of a vector per tree when varying the number of BWQS scorers.

Time(ns)	Tree scorers	BWQS scorers
Yahoo! 10	14.09	11.24
Yahoo! 50	39.11	52.88
Yahoo! 150	78.11	2869.29
MS 50	31.49	44.64
MQ 10	8.52	7.39

Table 6: Use of the comparison and selection scheme with BWQS scorers and with the original regression tree scorers.

best m' found for Y!64, Y!32, Y!16 and MS64 are 20, 10, 4 and 200, respectively. For Y!64, when $m' = 20$, case DSD_3 with $d = 75$ reaches the best performance with 62.89ns scoring time. If d is chosen as 1, the scoring time would be 89.33ns. Here the constraint to derive d value is explained as follow. Let F be the number of features in each document vector and L be the number of leaves in each tree. Following [12], the size of d document vectors is $4dF$ bytes and the QS data structure is composed of 6 parts: the result bit vectors with size $d\tau \cdot \frac{L}{8}$, the thresholds with size $4\tau L$, the offsets with size $4F$, the tree ids with size $4\tau L$, the bitvectors with size $\frac{\tau \cdot L^2}{8}$, and leaves with size $4\tau L$. The total size of d vectors and QS data structure is $4F(d+1) + (d \cdot \frac{L}{8} + (12 + \frac{L}{8})L) \cdot \tau$, which needs to fit in L3 cache because one BWQS scorer may not fit in $L2$. For Yahoo! dataset with $F = 700$, when $L = 64$ and $\tau = 1000$, we can derive $d = 75.4$ with $L3 \approx 2MB$.

Table 6 shows the scoring time per vector per tree when applying cache blocking with our comparison and selection scheme to the original tree scorer and to the BWQS scorer. This comparison shows that when the number of leaves per tree is small(10), BWQS performs better. When the number of leaves per tree increases to 150, BWQS becomes fairly slow. Our explanation is listed as follows. The core QS scheme has a complexity sensitive to the number of tree nodes detected as “false” nodes because bit-wise operations need to be conducted for all such nodes. When the number of “false” nodes is large and linear to L, the overall time cost grows at linearly to increasing of L for small L. When $L > 64$, the bit operation has to be carried by multiple 64-bit instructions and there is additional overhead for managing this complexity. For a large tree with many false nodes, QuickScorer can become very expensive. On the other other hand, the original regression tree algorithm has a complexity logarithmically proportional to L. It should be mentioned that the scoring time per vector per tree reported here seems to be slower than what was reported in [12] for $L = 64$. That can be caused by a difference in dataset characteristics, our code implementation, and test platform. We will investigate this issue in the future work.

5.4 Batched Query Processing

We illustrate the benefit of batched query processing when n is small. Table 7 shows the throughput under different batch sizes when ranking only 10 document vectors ($n = 10$). The throughput is defined as the number of queries processed per second. The last row shows the throughput when DS without cache blocking is used. When batch size is 10, for MS 50, the average processing time is reduced from 79.79ns to 42.93ns. When the batch size becomes much bigger, the benefit is not significant any more while there is an increase of waiting time. Thus a modest batch size is sufficient in

Batch size	Y! 50	Y! 150	MS 50	MQ 10
10k	125.79	60.78	310.27	2880.2
1k	125.60	60.64	304.88	2805.8
100	125.00	60.37	308.45	2673.8
10	121.54	56.03	232.94	2460.6
1	73.53	31.40	125.33	1505.1
DS	23.87	13.12	38.89	565.6

Table 7: Throughput under different batch size when $n = 10$.

this case to reach upto 1.86x throughput performance improvement.

6. CONCLUSIONS

The main contribution of this paper is a fast comparison and selection scheme to find an optimized cache blocking method with guided sampling. Our analysis estimates the data access cost of different methods approximately, which provides a foundation to select sampling points in comparing different methods and in narrowing search space.

The evaluation studies with 3 datasets show that different blocking methods and parameter values can exhibit different cache and cost behavior and our guided sampling can identify a highly competitive solution among DSD, SDS, DSDS, and SDSD methods in a reasonable amount of hours using a modest multi-core server. The difference between the selected solution and the estimated optimum is within 2.4% and the response time of this solution can be 6.57x faster than DS without cache blocking. The analytic cost analysis shows that the search space for datasets such as Yahoo!, MS, and MQ can be greatly narrowed by taking advantages of data and architectural characteristics. When the number of feature vectors per query is small, cache utilization is affected and if allowed, batched query processing can bring upto 1.86x performance improvement. The evaluation demonstrates that our scheme can be used to find the optimized partitioning for QuickScorer.

Acknowledgments. We thank the anonymous referees for their thorough comments. This work is supported in part by NSF IIS-1528041 and IIS-1118106. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

APPENDIX

A. COST ANALYSIS FOR SDS, DSDS, AND SDSD

Similar to the analysis of DSD in Section 4, this appendix section estimates the data access cost of SDS, DSDS, and SDSD. We skip the details on how the results are derived and summarize them in the following tables. We use a similar naming of range cases for these 3 methods: SDS_i, DSD_iS_j, and SDS_iD_j where $1 \leq i, j \leq 4$. Notation D_i in each range case name means that d vectors fit in cache level i, but not $i-1$. Level $i = 4$ refers to memory. Notation S_i means that s scorers fit in cache level i, but not $i-1$.

Table 8 lists the access cost ratio of SDS for the scenario when a feature vector fits in L1. For the scenario when one vector fits in L2 only, there are 3 range cases to consider: SDS_2, SDS_3, and SDS_4. Formula T_S is the same while T_D adds c_2 as: $T_D = 1 + c_2 + \frac{c_4}{s}$. For the scenario when a vector fits in L3 only, there are only 2 cases to consider: SDS_3, and SDS_4. Formula T_S is the same while T_D adds c_3 as $T_D = 1 + c_3 + \frac{c_4}{s}$. For the scenario when a vector fits in memory only: there is one case to consider: SDS_4. T_S is the same while T_D changes as $T_D = 1 + c_4$.

Cases	s scorers fit in	$T_{SDS_i} = T_D + \eta T_S \approx$
SDS_1	L1	$1 + \frac{c_4}{s_1} + \eta + \eta\frac{c_4}{n}$
SDS_2	L2	$1 + \frac{c_4}{s_2} + \eta + \eta c_2$
SDS_3	L3	$1 + \frac{c_4}{s_3} + \eta + \eta c_3$
SDS_4	memory	$1 + \frac{c_4}{s_4} + \eta + \eta c_4$

Table 8: Cost of SDS when 1 feature vector fits in L1.

Table 9 lists the data access cost ratio of DSDS for the scenario when a feature vector fits in L1. For the scenario when one vector fits in L2, there are only 6 range cases to consider: DSD_2S_2, DSD_3S_2, DSD_4S_2, DSD_3S_3, DSD_4S_3 and DSD_4S_4. Formula T_S does not change while T_D adds an extra c_2 term. For the scenario when one vector fits in L3, there are only 3 cases to consider: DSD_3S_3, DSD_4S_3 and DSD_4S_4. Formula T_S does not change while T_D adds an extra c_3 term. For the scenario when one vector fits in memory only, there is only one case to consider: DSD_4S_4. Formula T_S does not change while T_D adds an extra c_4 term.

Cases	d vectors fit in	s scorers fit in	$T_{DSD_iS_j} = T_D + \eta T_S \approx$
DSD_1S_1	L1	L1	$1 + \frac{c_4}{m} + \eta + \eta\frac{c_4}{d_1}$
DSD_2S_1	L2	L1	$1 + \frac{c_2}{s_1} + \eta + \eta\frac{c_4}{d_2}$
DSD_3S_1	L3	L1	$1 + \frac{c_3}{s_1} + \eta + \eta\frac{c_4}{d_3}$
DSD_4S_1	memory	L1	$1 + \frac{c_4}{s_1} + \eta + \eta\frac{c_4}{d_4}$
DSD_2S_2	L2	L2	$1 + \frac{c_2}{s_2} + \eta + \eta c_2 + \eta\frac{c_4}{d_2}$
DSD_3S_2	L3	L2	$1 + \frac{c_3}{s_2} + \eta + \eta c_2 + \eta\frac{c_4}{d_3}$
DSD_4S_2	memory	L2	$1 + \frac{c_4}{s_2} + \eta + \eta_2 + \eta\frac{c_4}{d_4}$
DSD_3S_3	L3	L3	$1 + \frac{c_3}{s_3} + \eta + \eta c_3 + \eta\frac{c_4}{d_3}$
DSD_4S_3	memory	L3	$1 + \frac{c_4}{s_3} + \eta + \eta c_3 + \eta\frac{c_4}{d_4}$
DSD_4S_4	memory	memory	$1 + \frac{c_4}{s_4} + \eta + \eta c_4$

Table 9: Cost of DSDS when 1 feature vector fits in L1.

Table 10 lists the data access time ratio for SDSD when a feature vector fits in L1. Symbol $\alpha_{i,j}$ denotes L1 cache miss rate in the corresponding case SDS_iD_j; Symbol $\beta_{i,j}$ is the corresponding L2 miss rate.

For the scenario 2 when one scorer fits in L2, there are 6 range cases to consider: SDS_2D_2, SDS_3D_2, SDS_4D_2, SDS_3D_3, SDS_4D_3 and SDS_4D_4. α in T_D becomes 1 and T_S adds an extra c_2 term. For the scenario when one scorer fits in L3, there are 3 cases to consider: SDS_3D_3, SDS_4D_3 and SDS_4D_4. α and β in T_D becomes 1 and T_S adds an extra c_3 term. For the scenario when one scorer does not fit L3, there is only one case to consider: SDS_4D_4. Its T_D does not change while T_S adds an extra c_4 term.

B. PROOF FOR PROPOSITION 1

For each of DSD, SDS, DSDS, and SDSD methods, we eliminate its range cases that do not qualify for the best candidate as follows.

Cases	$T_{SDS_iD_j} = \eta T_S + T_D \approx$
SDS_1D_1	$\eta + \eta\frac{c_4}{n} + 1 + \frac{c_4}{s_1}$
SDS_2D_1	$\eta + \eta\frac{c_2}{d_1} + 1 + \frac{c_4}{s_2}$
SDS_3D_1	$\eta + \eta\frac{c_3}{d_1} + 1 + \frac{c_4}{s_3}$
SDS_4D_1	$\eta + \eta\frac{c_4}{d_1} + 1 + \frac{c_4}{s_4}$
SDS_2D_2	$\eta + \eta\frac{c_2}{d_2} + 1 + \alpha_{2,2}c_2 + \frac{c_4}{s_2}$
SDS_3D_2	$\eta + \eta\frac{c_3}{d_2} + 1 + \alpha_{3,2}c_2 + \frac{c_4}{s_3}$
SDS_4D_2	$\eta + \eta\frac{c_4}{d_2} + 1 + \alpha_{4,2}c_2 + \frac{c_4}{s_4}$
SDS_3D_3	$\eta + \eta\frac{c_3}{d_3} + 1 + \alpha_{3,3}c_2 + \alpha_{3,3}\beta_{3,3}c_3 + \frac{c_4}{s_3}$
SDS_4D_3	$\eta + \eta\frac{c_4}{d_3} + 1 + \alpha_{4,3}c_2 + \alpha_{4,3}\beta_{4,3}c_3 + \frac{c_4}{s_4}$
SDS_4D_4	$\eta + \eta\frac{c_4}{d_4} + 1 + c_4$

Table 10: Cost of SDSD when one scover fits in L1.

- For DSD cases listed in Table 1, case DSD_4 is excluded from the best case list as term c_4 in DSD_4 cost dominates the weight. Thus $T_{DSD_4} > T_{DSD_3}$, $T_{DSD_4} > T_{DSD_2}$, and $T_{DSD_4} > T_{DSD_1}$. We also drop DSD_1 because $T_{DSD_1} > T_{SDS_2D_1}$.

  Now we compare DSD_2 and DSD_3. Since $\frac{\eta c_4}{d_2} \ll 1$, $\frac{\eta c_4}{d_3} \ll 1$, and these two terms can be dropped approximately from the cost expressions. Note that $\alpha_3\beta_3 = \frac{1}{m\gamma_3} \geq \frac{1}{m}$, and c_3 is much larger than c_2. Also $\alpha_2 < \alpha_3$ since the inner most loop of DSD_2 accesses less vectors. This makes $T_{DSD_3} > T_{DSD_2}$.

- Now we compare SDS cases listed in Table 8. Since $\frac{c_4}{s_2} \ll 1$, this leads to $\frac{c_4}{s_3} \ll 1$, and $\frac{c_4}{s_4} \ll 1$, and $\frac{c_4}{m} \ll 1$. Therefore $T_{SDS_4} > T_{SDS_3} > T_{SDS_2}$. Thus we drop cases SDS_3 or SDS_4.

  We drop case SDS_2 because $T_{SDS_2} > T_{DSD_2S_2}$. We also drop case SDS_1 because $T_{SDS_1} > T_{DSD_2S_1}$ given $\frac{\eta c_4}{d_2} \ll 1$.

- For DSDS, since $\frac{\eta c_4}{d_2} \ll 1$, $T_{DSD_2S_1} \approx 1+\eta+\frac{c_2}{s_1}$. Then $T_{DSD_2S_1} < T_{DSD_3S_1}$ and $T_{DSD_2S_1} < T_{DSD_4S_1}$. Thus we drop cases DSD_2S_1 and DSD_4S_1.

  Since $\frac{c_4}{s_2} \ll 1$, $\frac{c_2}{s_2} \ll 1$. Then $T_{DSD_2S_2} \approx 1 + \eta + \eta c_2$. Then $T_{DSD_2S_2}$ is smaller than any of $T_{DSD_3S_2}$, $T_{DSD_4S_2}$, $T_{DSD_3S_3}$, $T_{DSD_4S_3}$, and $T_{DSD_4S_4}$. We drop all cases in DSDS except DSD_1S_1, DSD_2S_1, and DSD_2S_2.

  Since $T_{DSD_2S_2} > 1 + \eta + \eta\frac{c_2}{d_1} \approx T_{SDS_2D_1}$ given $\frac{\eta c_4}{s_2} \ll 1$, we can further drop case DSD_2S_2. Then $T_{DSD_1S_1} > T_{DSD_2S_1}$ and we can further drop case DSD_1S_1.

- For SDSD, we drop case SDS_1D_1 because $T_{SDS_1S_1} \approx 1 + \eta + \frac{c_4}{s_1} > T_{DSD_2S_1}$ given $\frac{\eta c_4}{d_2} \ll 1$. We drop Case SDS_3D_1 because $T_{SDS_3S_1} \approx 1+\eta+\eta\cdot\frac{c_3}{d_1} > T_{SDS_2S_1}$. We drop Case SDS_4D_1 because $T_{SDS_4S_1} \approx 1+\eta+\eta\cdot\frac{c_4}{d_1} > T_{SDS_2S_1}$.

  Note that $T_{SDS_2S_2} \approx 1 + \eta + \alpha_{2,2}c_2$ because $\frac{\eta c_4}{s_2} \ll 1$ and $\frac{\eta c_2}{d_2} < \frac{\eta c_4}{d_2} \ll 1$. Then we drop SDS_4D_4 because $T_{SDS_2D_2} < T_{SDS_4D_4}$. Also, $T_{SDS_2D_2}$ is smaller than any of $T_{SDS_3D_2}$, $T_{SDS_4D_2}$, $T_{SDS_3D_3}$, and $T_{SDS_4D_3}$. That is because $\alpha_{2,2} < \alpha_{3,2}, \alpha_{4,2}, \alpha_{3,3}, \alpha_{4,3}$ due to the fact that s scorers fits in the smaller cache in Case SDS_2D_2 than other cases compared here. Thus only SDS_2D_2 and SDS_2D_1 qualify for the best candidates.

C. REFERENCES

[1] Lector 4.0 datasets. http://research.microsoft.com/en-us/um/beijing/projects/letor/letor4dataset.aspx.

[2] Microsoft learning to rank datasets. http://research.microsoft.com/en-us/projects/mslr/.

[3] Nima Asadi and Jimmy Lin. Training Efficient Tree-Based Models for Document Ranking. In *ECIR*, pages 146–157, 2013.

[4] Nima Asadi, Jimmy Lin, and Arjen P De Vries. Runtime Optimizations for Tree-Based Machine Learning Models. *IEEE TKDE*, pages 1–13, 2013.

[5] Christopher J. C. Burges, Krysta Marie Svore, Paul N. Bennett, Andrzej Pastusiak, and Qiang Wu. Learning to rank using an ensemble of lambda-gradient models. In *J. of Machine Learning Research*, pages 25–35, 2011.

[6] B. Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. Early exit optimizations for additive machine learned ranking systems. WSDM '10, pages 411–420, 2010.

[7] Olivier Chapelle and Yi Chang. Yahoo! Learning to Rank Challenge Overview. *J. of Machine Learning Research*, pages 1–24, 2011.

[8] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[9] Yasser Ganjisaffar, Rich Caruana, and Cristina Lopes. Bagging Gradient-Boosted Trees for High Precision, Low Variance Ranking Models. In *SIGIR*, pages 85–94, 2011.

[10] Pierre Geurts and Gilles Louppe. Learning to rank with extremely randomized trees. *J. of Machine Learning Research*, 14:49–61, 2011.

[11] Andrey Gulin, Igor Kuralenok, and Dmitry Pavlov. Winning the transfer learning track of yahoo!'s learning to rank challenge with yetirank. *J. of Machine Learning Research*, 14:63–76, 2011.

[12] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *SIGIR*, pages 73–82, 2015.

[13] Dmitry Yurievich Pavlov, Alexey Gorodilov, and Cliff A. Brunk. Bagboo: a scalable hybrid bagging-the-boosting model. In *CIKM*, pages 1897–1900, 2010.

[14] Xun Tang, Xin Jin, and Tao Yang. Cache-conscious runtime optimization for ranking ensembles. SIGIR '14, pages 1123–1126, 2014.

[15] Jiancong Tong, Gang Wang, and Xiaoguang Liu. Latency-aware strategy for static list caching in flash-based web search engines. In *CIKM*, pages 1209–1212, 2013.

[16] Lidan Wang, Jimmy Lin, and Donald Metzler. Learning to efficiently rank. SIGIR '10, pages 138–145, 2010.

[17] Lidan Wang, Jimmy Lin, and Donald Metzler. A cascade ranking model for efficient ranked retrieval. SIGIR '11, pages 105–114, 2011.