Cache-Conscious Runtime Optimization for Ranking Ensembles

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

ABSTRACT
Multi-tree ensemble models have been proven to be effective for document ranking. Using a large number of trees can improve accuracy, but it takes time to calculate ranking scores of matched documents. This paper investigates data traversal methods for fast score calculation with a large ensemble. We propose a 2D blocking scheme for better cache utilization with simpler code structure compared to previous work. The experiments with several benchmarks show significant acceleration in score calculation without loss of ranking accuracy.

Categories and Subject Descriptors
H.3.3 [Information Search and Retrieval]: Retrieval Models, Search Process

Keywords
Ensemble methods; query processing; cache locality

1. INTRODUCTION
Learning ensembles based on multiple trees are effective for web search and other complex data applications (e.g., [9, 8, 10]). It is not unusual that algorithm designers use thousands of trees to reach better accuracy and the number of trees becomes even larger with the integration of bagging. For example, winning teams in the Yahoo! learning-to-rank challenge [8] have all used boosted regression trees in one form or another and the total number of trees reported for scoring ranges from 3,000 to 20,000 [11, 6, 12], or even reaches 300,000 or more combined with bagging [13].

Computing scores from a large number of trees is time-consuming. Access of irregular document attributes along with dynamic tree branching impairs the effectiveness of CPU cache and instruction branch prediction. Compiler optimization [5] cannot handle complex code such as rank scoring very well. For example, processing a 8,051-tree ensemble can take up to 3.04 milliseconds for a document with 519 features on an AMD 3.1 GHz core. Thus the scoring time per query exceeds 6 seconds to rank the top-2,000 results. It takes more time proportionally to score more documents with larger trees or more trees and this is too slow for interactive query performance. Multi-tree calculation can be parallelized; however, query processing throughput is not increased because less queries are handled in parallel. Tradeoff between ranking accuracy and performance can be played by using earlier exit based on document-ordered traversal (DOT) or scorer-ordered traversal (SOT) [7], and by tree trimming [3]. The work in [4] proposes an architecture-conscious solution called VPred that converts control dependence of code to data dependence and employs loop unrolling with vectorization to reduce instruction branch mis-prediction and mask slow memory access latency. The weakness is that cache capacity is not fully exploited and maintaining the lengthy unrolled code is not convenient.

Unorchestrated slow memory access incurs significant costs since memory access latency can be up to 200 times slower than L1 cache latency. How can fast multi-tree ensemble ranking with simple code structure be accomplished via memory hierarchy optimization, without compromising ranking accuracy? This is the focus of this paper.

We propose a cache-conscious 2D blocking method to optimize data traversal for better temporal cache locality. Our experiments show that 2D blocking can be up to 620% faster than DOT, up to 245% faster than SOT, and up to 50% faster than VPred. After applying 2D blocking on top of VPred which shows advantage in reducing branch mis-prediction, the combined solution Block-VPred could be up to 100% faster than VPred. The proposed techniques are complementary to previous work and can be integrated with the tree trimming and early-exit approximation methods.

2. PROBLEM DEFINITION
Given a query, there are \( n \) documents matching this query and the ensemble model contains \( m \) trees. Each tree is called a scorer and contributes a subscore to the overall score for a document. Following the notation in [7], Algorithm 1 shows the program of DOT. At each loop iteration \( i \), all trees are calculated to gather subscores for a document before moving to another document. In implementation, each document is represented as a feature vector and each tree can be stored in a compact array-based format [4]. The time and space cost of updating the overall score with a subscore is relatively insignificant. The dominating cost is slow memory accesses during tree traversal based on document feature values. By exchanging loops \( i \) and \( j \) in Algorithm 1, DOT becomes SOT. Their key difference is the traversal order.
Algorithm 1: Ranking score calculation with DOT.

for i = 1 to n do
  for j = 1 to m do
    Compute a subscore for document i with tree j.
    Update document score with the above subscore.

Figure 1: Data access order in DOT (a) and SOT (b).

Figure 1(a) shows the data access sequence in DOT, marked on edges between documents and tree-based scorers. These edges represent data interaction during ranking score calculation. DOT first accesses a document and the first tree (marked as Step 1); it then visits the same document and the second tree. All m trees are traversed before accessing the next document. As m becomes large, the capacity constraint of CPU cache such as L1, L2, or even L3 does not allow all m trees to be kept in the cache before the next document is accessed. The temporal locality of a document is exploited in DOT since the cached copy can be re-accessed many times before being flushed; however, there is no or minimal temporal locality exploited for trees. Similarly, Figure 1(b) marks data interaction edges and their access order in SOT. SOT traverses all documents for a tree before accessing the next tree. Temporal locality of a tree is exploited in SOT; however, there is no or minimal temporal locality exploited for documents when n is large.

VPreD [4] converts if-then-else branches to dynamic data accesses by unrolling the tree depth loop. The execution still follows DOT order, but it overloads the sparse computation of several documents to mask memory latency. Such vectorization technique also increases the chance of these documents staying in a cache when processing the next tree. However, it has not fully exploited cache capacity for better temporal locality. Another weakness is that the length of the unrolled code is quadratic to the maximum tree depth in a ensemble, and linear to the vectorization degree v. For example, the header file with maximum tree depth 51 and vectorization degree 16 requires 22,651 lines of code. Long code causes inconvenience in debugging and code extension. In comparison, our 2D blocking code has a header file of 159 lines.

3. 2D BLOCK ALGORITHM

Algorithm 2 is a 2D blocking approach that partitions the program in Algorithm 1 into four nested loops. The loop structure is named SDDS because the first (outer-most) and third levels iterate on tree-based Scorers while the second and fourth levels iterate on Documents. The inner two loops process d documents with s trees to compute subscores of these documents. We choose d and s values so that these d documents and s trees can be placed in the fast cache under its capacity constraint. To simplify the presentation of this paper, we assume $\frac{m}{n}$ and $\frac{d}{s}$ are integers. The hierarchical data access pattern is illustrated in Figure 2. The edges in the left portion of this figure represent the interaction among blocks of documents and blocks of trees with access sequence marked on edges. For each block-level edge, we demonstrate the data interaction inside blocks in the right portion of this figure. Note that there are other variations of 2D blocking structures: SDDS, DSDS and DSSD. Our evaluation finds that SDDS is the fastest for the tested benchmarks.

Algorithm 2: 2D blocking with SDDS structure.

for j = 0 to $\frac{m}{s} - 1$ do
  for i = 0 to $\frac{d}{n} - 1$ do
    for jj = 1 to s do
      for ii = 1 to d do
        Compute subscore for document $i \times d + ii$ with tree $j \times s + jj$.
        Update the score of this document.

Detailed cache performance analysis requires a study of cache miss ratio estimation in multiple levels of cache. Due to the length restriction of this paper, we use a simplified cache-memory model to illustrate the benefits of the 2D blocking scheme. This model assumes there is one level of cache which can hold d document vectors and s tree-based scorers, i.e. space usage for s and d do not exceed cache capacity. Here we estimate the total slow memory accesses during score calculation using the big O notation. The inner-most loop ii in Algorithm 2 loads 1 tree and d document vectors. Then loop jj loads another tree and still accesses the same d document vectors. Thus there are a total of $O(s)+O(d)$ slow memory accesses for loops jj and ii. In loop level i, the s trees stay in the cache and every document block causes slow memory accesses, so memory access overhead is $O(s)+O(d) \times \frac{n}{s}$. Now looking at the the outer-most loop j, total memory access overhead per query is $O(s)+O(n) = O(m+\frac{m}{n})$.

From Figure 1, memory access overhead per query in DOT can be estimated as $O(m \times n + n)$ while it is $O(m \times n + m)$ for SOT. Since term $m \times n$ typically dominates, our 2D blocking algorithm incurs s times less overhead in loading data from slow memory to cache when compared with DOT or SOT.

Vectorization in VPreD can be viewed as blocking a number of documents and the authors have reported [4] that a larger vectorization degree does not improve latency masking and for Yahoo! dataset, 16 or more degree performs about the same. The objective of 2D blocking scheme is to
Table 1: Scoring time per document per tree in nanoseconds for five algorithms. Last column shows the average scoring latency per query in seconds under the fastest algorithm marked in gray.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Leaves</th>
<th>m</th>
<th>n</th>
<th>DOT</th>
<th>SOT</th>
<th>VPred</th>
<th>2D blocking [s, d]</th>
<th>Block-VPred [s, d, v]</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yahoo!</td>
<td>50</td>
<td>7,870</td>
<td>5,000</td>
<td>186.0</td>
<td>113.8</td>
<td>47.4</td>
<td>[300, 300]</td>
<td>[367, 300, 320, 8]</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>150</td>
<td>8,051</td>
<td>2,000</td>
<td>377.8</td>
<td>150.2</td>
<td>123.0</td>
<td>[300, 300]</td>
<td>[81.9, 100, 400]</td>
<td>1.23</td>
</tr>
<tr>
<td>MSLR-30K</td>
<td>400</td>
<td>2,898</td>
<td>5,000</td>
<td>312.3</td>
<td>223.8</td>
<td>136.2</td>
<td>[300, 300]</td>
<td>[90.9, 100, 400]</td>
<td>1.25</td>
</tr>
<tr>
<td>MQ2007</td>
<td>50</td>
<td>1,647</td>
<td>5,000</td>
<td>88.3</td>
<td>41.4</td>
<td>32.6</td>
<td>[300, 300]</td>
<td>[26.6, 500, 1,000]</td>
<td>0.22</td>
</tr>
</tbody>
</table>

Figure 3: Scoring time per document per tree in nanoseconds when varying m (a) and n (b) for five algorithms, and varying s and d for 2D blocking (c). Benchmark used is Yahoo! dataset with a 150-leaf multi-tree ensemble.

A comparison of scoring time. Table 1 lists scoring time under different settings. Column 2 is the maximum number of leaves per tree. Tuple [s,d,v] includes the parameters of 2D blocking and the vectorization degree of VPred that leads to the fastest scoring time. Choices of v for VPred are the best in the tested AMD architecture and are slightly different from the values reported in [4] with Intel processors. Last column is the average scoring latency per query in seconds after visiting all trees. For example, 2D blocking is 361% faster than DOT and is 50% faster than VPred for Row 3 with Yahoo! 150-leaf 8,051-tree benchmark. In this case, Block-VPred is 62% faster than VPred and each query takes 1.23 seconds to complete scoring with Block-VPred. For a smaller tree in Row 5 (MSLR-30K), Block-VPred is 17% slower than regular 2D blocking. In such cases, the benefit of converting control dependence as data dependence does not outweigh the overhead introduced.

Figure 3 shows the scoring time for Yahoo! dataset under different settings. In Figure 3(a), n is fixed as 2,000; DOT time rises dramatically when m increases because these trees do not fit in cache; SOT time keeps relatively flat as m increases. In Figure 3(b), m is fixed as 8,051 while n varies from 10 to 100,000. SOT time increases as n grows and 2D blocking is up to 245% faster. DOT time is relatively stable. 2D blocking time and its gap to VPred are barely affected by the change of m or n. Block-VPred is 90% faster than VPred when n=5,000, and 100% faster when n=100,000. Figure 3(c) shows the 2D blocking time when varying s and d. The lowest value is achieved with s=1,000 and d=100 when these trees and documents fit in L2 cache.

Cache behavior. Linux perf tool reports L1 and L3 cache miss ratios during execution. We observed no strong correlation between L1 miss ratio and scoring time. L1 cache allows program to exploit limited spatial locality, but is too small to exploit temporal locality in our problem context. L3 miss ratio does show a strong correlation with scoring time. In our design, 2D blocking sizes (s and d) are determined based on L2 cache size. Since L2 cache is about the same size as L3 per core in the tested AMD machine, reported L3 miss ratio reflects the characteristics of L2 miss ratio.
5. CONCLUDING REMARKS

The main contribution of this work is cache-conscious design for computing ranking scores with a large number of trees and/or documents by exploiting memory hierarchy capacity for better temporal locality. Multi-tree score calculation of each query can be conducted in parallel on multiple cores to further reduce latency. Our experiments show that 2D blocking still maintains its advantage using multiple threads. In some applications, the number of top results \( n \) for each query is inherently small and can be much smaller than the optimal block size \( d \). In such cases, multiple queries could be combined and processed together to fully exploit cache capacity. Our experiments with Yahoo! dataset and 150-leaf 8,051-tree ensemble shows that combined processing could reduce scoring time per query by 12.0% when \( n=100 \), and by 48.7% when \( n=10 \).

Our 2D blocking technique is studied in the context of tree-based ranking ensembles and one of future work is to extend it for other types of ensembles by iteratively selecting a fixed number of the base rank models that can fit in the fast cache.

Acknowledgments. This work was supported in part by NSF 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.

6. REFERENCES