RapidScorer: Fast Tree Ensemble Evaluation by Maximizing Compactness in Data Level Parallelization

Ting Ye
Microsoft
Vancouver, BC, Canada
tiy@microsoft.com

Hucheng Zhou*
Alibaba Group
Hangzhou, Zhejiang, China
hucheng.zhc@alibaba-inc.com

Will Y. Zou
Microsoft
Sunnyvale, CA, USA
will.zou@microsoft.com

Bin Gao
Microsoft
Redmond, WA, USA
bingao@microsoft.com

Ruofei Zhang
Microsoft
Sunnyvale, CA, USA
bzhang@microsoft.com

ABSTRACT

Relevance ranking models based on additive ensembles of regression trees have shown quite good effectiveness in web search engines. In the era of big data, tree ensemble models grow large in both tree depth and ensemble size to provide even better search relevance and user experience. However, the computational cost for their scoring process is high, such that it becomes a challenging issue to apply the big tree ensemble models in a search engine which needs to answer thousands of queries per second. Although several works have been proposed to improve the scoring process, the challenge is still great especially when the model size grows large. In this paper, we present RapidScorer, a novel framework for speeding up the scoring process of industry-scale tree ensemble models, without hurting the quality of scoring results. RapidScorer introduces a modified run length encoding called `epitome` to the bitvector representation of the tree nodes. `Epitome` can greatly reduce the computation cost to traverse the tree ensemble, and work with several other proposed strategies to maximize the compactness of data units in memory. The achieved compactness makes it possible to fully utilize data parallelization to improve model scalability. Experiments on two web search benchmarks show that, RapidScorer achieves significant speed-up over the state-of-the-art methods: V-QuickScorer, ranging from 1.3x to 3.5x; QuickScorer, ranging from 2.1x to 25.0x; VPred, ranging from 2.3x to 18.3x; and XGBoost, ranging from 2.6x to 42.5x.

ACM Reference Format:
Ting Ye, Hucheng Zhou, Will Y. Zou, Bin Gao, and Ruofei Zhang. 2018. RapidScorer: Fast Tree Ensemble Evaluation by Maximizing Compactness in Data Level Parallelization. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219819.3219857

*This work was completed during this author's working at Microsoft Research.

KDD '18, August 19–23, 2018, London, United Kingdom

ACM ISBN 978-1-4503-5552-0/18/08…$15.00
https://doi.org/10.1145/3219819.3219857

1 INTRODUCTION

Decision tree ensembles are intensively used models in web services, as they can find complex correlations across features, resulting in good performance in search ranking and ad click prediction. For example, Gradient-Boosting Decision Trees (GBDT) [14], and LambdaMART [4] achieved the best results in the Kaggle competition [12, 18] and Yahoo! Learning to Rank Challenge [6]. In industry, tree ensembles have been implemented as the search ranking models in Microsoft Bing [20, 35], AltaVista, Yahoo! and Yandex [30], and as the advertising model in Facebook [15].

There are two challenges in serving online tree ensembles. The first is the demand of designing highly efficient tree ensemble traversal algorithms for the scoring process, as web services often impose strict requirements on model latency [33, 34]. The second is model scalability, as web service systems need to support fast growing tree ensemble models, with deeper trees and larger ensemble size, in the era of big data [11, 25, 34].

Some existing works leverage boolean-vector operations [23], vectorization [2, 31], and Single Instruction Multiple Data (SIMD) data-level parallelism [24] to address the aforementioned challenges. For example, as the conventional root-to-leaf tree traversal algorithm suffers from the CPU inefficiency due to the forced sequential computation [2, 10, 23, 24], the recent approach QuickScorer [10, 23] reformulates the sequential computation by performing an interleaved tree traversal with logical bitwise operations. The computation operands are boolean-vectors of length Λ, the maximum number of leaves per tree. The upgraded model, V-QuickScorer [24], further exploits CPU SIMD extensions to vectorize scoring process, such that multiple samples (e.g., documents or ads) can be evaluated simultaneously in the ensemble tree traversal. These algorithms have offered considerable speed-up of tree ensembles evaluations.

However, the above boolean-vector based algorithms iteratively compute on vectors of length Λ and thus their complexity scales linearly with the maximum number of leaves per tree, i.e. $O(\Lambda)$. With deeper trees, the CPU execution is more expensive and the parallelization is more difficult. The running time of boolean-vector methods can be extremely high for industry-scale tree ensembles. For example, the number of leaves in a single tree can commonly be more than 200 for the tree ensembles in Bing Ads to do click

prediction as illustrated by Table 1. The top-performing models [8, 33] for KDD Cup and Yahoo! ranking challenges have to deal with even larger numbers of leaves to achieve better performance.

To better address the aforementioned challenges, we present RapidScorer for tree ensembles evaluation that is both fast at traversal speed and scales up to deeper and larger tree ensembles. Specifically, we introduce a modified run length encoding called epitome to the bitmask representation of the tree ensemble models. The epitome data structure brings two benefits to the tree ensembles evaluation, addressing the two challenges respectively.

First, using epitome, we utilize the hierarchical structure in the trees to epitomize memory usage per node to constant scale. The data-flow and layout are redesigned with modern CPU architecture, so that the epitomization can help improve the tree ensemble traversal with significant speed-up for both small and large tree ensembles. As a result, we can reduce the complexity per node from $O(\Lambda)$ to $O(\sqrt{\Lambda})$ *w.r.t.* the maximum number of leaves Λ.

Second, the epitome structure for CPU operations works well with the bit-wise memory usage in SIMD registers. SIMD extensions have shown excellent competence for parallelizing computation for web serving models [24, 31]. SIMD extensions, namely Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX), exploit wide registers of 128 and 256 bits, so the data inserted in parallel are processed with a single instruction. In tree ensembles, SIMD can be used to implement data-level parallelism [1, 9] across the ensemble and across data samples. In RapidScorer, SIMD can coordinate well with epitome structure to speed up the calculation by one order of magnitude. Specifically, we can vectorize all steps by leveraging SIMD and epitome to evaluate multiple samples in parallel. As a result, RapidScorer can maximize the compactness of data units and fully utilize the SIMD data-level parallelization to further speed up the tree ensembles evaluation.

In our experiments, we tested RapidScorer against existing works QuickScorer [10, 23], V-QuickScorer [24], VPred [2], and XGBoost [7], for both small scale ensembles and industry scale ensembles. The experiments were performed on a public dataset MSN [27] for search ranking and a production dataset in Bing Ads for ad click prediction. According to the results, RapidScorer achieves speed-up over V-QuickScorer ranging from 1.3x to 3.5x, over QuickScorer ranging from 2.1x to 25.0x, over VPred ranging from 2.3x to 18.3x, and over XGBoost ranging from 2.6x to 42.5x.

To sum up, our contributions are listed as below. (i) We introduced a modified run length encoding called epitome to the bitmask representation of the tree ensemble models, and proposed RapidScorer for fast tree ensemble evaluation. (ii) With epitome, RapidScorer for tree ensembles evaluation can reduce per-node CPU complexity from $O(\Lambda)$ to $O(\sqrt{\Lambda})$ *w.r.t.* the number of leaves. (iii) With epitome, RapidScorer can make the SIMD register occupation significantly compact. The compactness brings benefits to the design of data flow, memory layout, and vectorization of the SIMD instructions. (iv) In RapidScorer, the equivalent nodes from different trees can be merged together as one node since the node processing is order insensitive, which further reduces memory usage and improves computing performance.

2 RELATED WORK

The optimization for efficiently scoring documents by means of regression tree ensembles has been well studied in information retrieval. Several previous works [2, 5, 10, 23, 24] have provided strategies to speed up tree traversal without losing quality.

In the scoring process of tree ensembles, a naïve implementation is traversal from each tree root all the way down to a specific leaf, like the example in Figure 1. However, this strategy induces frequent *control hazards*, i.e., the next instruction to be executed remains unknown until the boolean test of current node is performed. In addition, due to the unpredictable of the tree nodes traversal, it is hard for cache to pre-fetch the next correct node to be visited, which results in low *cache hit ratio*. Therefore, the instruction pipeline of the processor will be stalled as it always needs to wait for the result of boolean test and the fetching of the next correct node, which makes the tree traversal very inefficient.

Asadi *et al.* [2] proposed Pred and its vectorized version VPred, to rearrange the traversal computation with the goal of converting *control hazards* into *data hazards*. Pred unrolls the traversal of a d-depth tree with the same d operations. Thus, *control hazards* are removed as the next instruction is always known, but *data hazards* are introduced as the next instruction requires the results of the current one. VPred reduces *data hazards* by operating on multiple samples simultaneously in an interleaved way, but it always runs d steps, even if a document might reach an exit leaf very earlier.

QuickScorer [10, 23] and V-QuickScorer [24] are more recent approaches which transform sequential tree evaluation into a more cache friendly process. We will describe them in details in Section 3 as they are closely related to our work.

Different with the above works that aim at speeding up tree ensemble evaluation with no quality loss, some recent approaches approximate the results to reduce scoring time. For example, Lucchese *et al.* [22] proposed several pruning strategies to remove trees in the ensemble and fine-tuned the weights of the remaining trees according to some quality metrics. Cambazoglu *et al.* [5] proposed to early terminate the low-score samples before the traversal of the whole tree ensemble.

It is also worth mentioning the *oblivious decision trees* [19], a popular variant of standard decision trees, which enforces the learned trees of an ensemble to be *oblivious*, i.e., the trees must be balanced and all branching nodes at the same level of each tree have to perform the same test. Several approaches address on *oblivious decision trees* and achieve noticeable speed-up, such as BDT [21], CatBoost [26], and Oblivious [10].

3 BACKGROUND

Usually, a tree ensemble $\mathcal{T}$ is a predictive model composed of a weighted combination of multiple decision trees, i.e., $\mathcal{T} = \{T_0, T_1, ..., T_{|\mathcal{T}|-1}\}$ with weights $\{w_0, w_1, ..., w_{|\mathcal{T}|-1}\}$. Each tree $T_h = (N_h, L_h)$ is composed of $|L_h| - 1$ internal nodes (referred to as *nodes*) $N_h = \{n_0^h, n_1^h, ..., n_{|L_h|-2}^h\}$ and $|L_h|$ leaf nodes (referred to as *leaves*) $L_h = \{l_0^h, l_1^h, ..., l_{|L_h|-1}^h\}$. Each node $n_i^h \in N_h$ contains four fields: the specific splitting feature with id ϕ_i^h, the corresponding splitting threshold $\theta_i^h \in \mathbb{R}$, and the pointers to left and right child. Each leaf $l_j^h \in L_h$ has a score value $s_j^h \in \texttt{leafvalue}[T_h]$, where $\texttt{leafvalue}[T_h]$ is

the array containing the score contribution of T_h. As different trees may have different number of leaves, a parameter Λ is set to be the maximum number of leaves for each T_h in $\mathcal{T}$, and thus $|L_h| \leq \Lambda$.

In tree ensemble evaluation, each sample is represented by a real-valued vector $\mathbf{x}$ of features, i.e., $\mathbf{x} = (x_0, x_1, ..., x_{|F|-1})^T$ where $F = \{f_0, f_1, ..., f_{|F|-1}\}$ is the set of features and x_k stores the value of feature f_k. The evaluation of tree T_h on sample $\mathbf{x}$ returns the score $s_h(\mathbf{x})$ at the exit leaf that $\mathbf{x}$ falls into. This leaf is found by starting at the root node and following a path determined by the decision criteria at each node. That is, if $x_{\phi_i^h} \leq \theta_i^h$ at node n_i^h is true (TRUE node), it falls into the left child; otherwise (FALSE node), it falls into the right child. For example, in Figure 1[1], if $\mathbf{x}$ satisfies $x_2 > \theta_0^h, x_3 \leq \theta_2^h$, and $x_0 \leq \theta_4^h$, then the boolean tests in n_0^h, n_2^h, and n_4^h will be FALSE, TRUE, and TRUE respectively. Thus, the traversal turns right at n_0^h (FALSE node), turns left at n_2^h (TRUE node), turns left at n_4^h (TRUE node), and finally reaches exit leaf l_3^h.

The tree traversal process is repeated for all the trees in the ensemble $\mathcal{T}$, and the final score for $\mathbf{x}$ is calculated as the weighted sum over the contributions of all the trees in $\mathcal{T}$.

3.1 QuickScorer

QuickScorer [10, 23] is a state-of-the-art approach which transforms sequential tree evaluation into a cache friendly process. This algorithm establishes tree traversal by applying bitwise AND to boolean representations of FALSE nodes, i.e., nodemask.

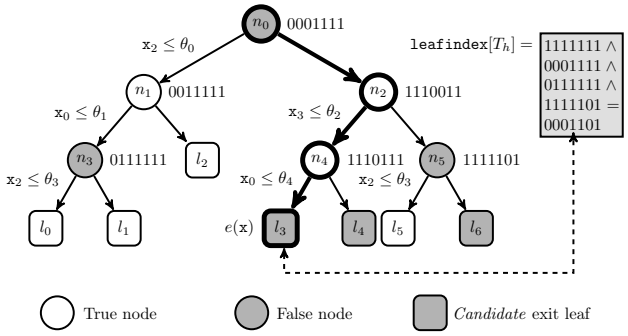


Figure 1: Tree traversal example of QuickScorer.

Algorithm 1 illustrates the details of QuickScorer. In the scoring process of a sample $\mathbf{x}$, for each tree $T_h \in \mathcal{T}$, QuickScorer maintains a bitvector leafindex $[T_h]$, composed of Λ bits, one per leaf, to indicate the possible exit leaf candidates with corresponding bits equal to 1. Initially, all bits in leafindex $[T_h]$ are set to 1. For each node n_i^h, the children pointers are replaced by a bitmask nodemask$[n_i^h]$, which acts as a mask to encode the set of unreachable leaves when n_i^h is a FALSE node. For example, in Figure 1, the bitmask 0001111 in FALSE node n_0^h means, as $x_2 > \theta_0^h$, the leaves $\{l_0^h, l_1^h, l_2^h\}$ would not be visited by $\mathbf{x}$. Thus, the bitwise AND operation between leafindex $[T_h]$ and nodemask$[n_0^h]$ corresponds to the removal of the leaves in the left subtree of n_0^h from

[1]To simplify the notations, we drop the superscript h in the figures.

Algorithm 1 The QuickScorer Algorithm

```
1:  function QuickScorer(x, T):
2:      for h ∈ 0, 1, ..., |T| − 1 do
3:          leafindex[T_h] ← {11...11}
4:      for k ∈ 0, 1, ..., |F| − 1 do            // Mask Computation
5:          currentnode = n_i^h ← offset[f_k]
6:          endnode ← offset[f_{k+1}]
7:          while x_k > θ_i^h && currentnode < endnode do
8:              leafindex[T_h] ← leafindex[T_h] ∧ nodemask[n_i^h]
9:              currentnode = n_{i'}^{h'} ← next node of n_i^h for f_k
10:     s(x) ← 0
11:     for h ∈ 0, 1, ..., |T| − 1 do            // Score Computation
12:         j ← index of leftmost bit equal to 1 in leafindex[T_h]
13:         s(x) ← s(x) + w_h s_j^h
14:     return s(x)
```

the set of exit leaf candidates. QuickScorer proves that, after updating leafindex $[T_h]$ on all FALSE nodes, the exit leaf is exactly the one corresponding to the position of the leftmost bit equal to 1 in leafindex $[T_h]$. Finally, the score of the exit leaf can be obtained as the score s_3^h. Figure 1 shows how the initial bitvector leafindex $[T_h]$ is updated by using bitwise AND operations.

To efficiently identify all the FALSE nodes in $\mathcal{T}$, QuickScorer processes the nodes of all the trees *feature by feature* in ascending order of their predicate thresholds. As bitwise AND operation is insensitive to the order of variables, QuickScorer has a nice property that the updating process of leafindex $[T_h]$ is insensitive to the node processing order. This property allows it to perform an interleaved traversal of the trees in a cache-aware fashion. Specifically, QuickScorer groups all nodes of all trees that have the same splitting feature f_k together and adopts an array, offset $[f_k]$, to mark the starting node of each feature. Within each feature, nodes are sorted according to their thresholds in ascending order and each node is represented in a triplet: $\left(\theta_i^h, h, \text{nodemask}[n_i^h]\right)$.

In addition, Lucchese *et al.* [10, 23] presented two optimization strategies for specific conditions. One is *blocking* for large tree ensembles, which splits the tree ensemble into disjoint blocks. The other is *tree reversing* for trees that contain more FALSE nodes than TRUE nodes on average over a collection of training documents.

3.2 V-QuickScorer

V-QuickScorer [24], QuickScorer's vectorized extension, further exploits SIMD extensions to vectorize the tree scoring process in QuickScorer by evaluating multiple input samples simultaneously. SIMD extensions, including Streaming SIMD Extensions (SSE) [32] and Advanced Vector Extensions (AVX) [13], are sets of instructions exploiting wide registers of 128 and 256 bits.

Specifically, during the mask computation step (line 7-9 in Algorithm 1), multiple input samples can be tested against a given node and their corresponding leafindex $[T_h]$ can be updated in parallel. For example, suppose Λ is 32, then line 8 in Algorithm 1 can be extended to support the parallelism of $\frac{128}{\Lambda} = 4$ and $\frac{256}{\Lambda} = 8$ samples by adopting one SSE and AVX instructions, respectively. Similarly, the score calculation of multiple input samples (line 13) can also be computed simultaneously.

Table 1: Validation AUC scores and gains of GBDT for ad click prediction in Bing Ads with various leaf numbers.

Λ	32	64	128	256	400
AUC	0.8385	0.8392	0.8397	0.8402	0.8408
AUC Gain	+0.00%	+0.08%	+0.14%	+0.20%	+0.07%

3.3 Limitations

As mentioned by Jin *et al.* [17], the core QuickScorer and V-QuickScorer scheme has a complexity sensitive to the maximum number of leaves Λ. These algorithms are efficient when Λ is less than 64, i.e., a typical machine word of modern CPUs (64 bits). When $\Lambda > 64$, the AND operation has to be carried by multiple 64-bit instructions (line 8 in Algorithm 1), thus QuickScorer can become very expensive. While in industrial settings where more complex models are needed to fit the web-scale training data, Λ can commonly be much more than 64. For instance, Table 1 illustrates a case used in Bing Ads for click prediction, where $\Lambda = 400$ achieves best AUC score. Things will be even worse for V-QuickScorer, in which the number of parallelism v heavily relies on Λ, i.e., $v = \frac{128}{\Lambda}$ for SSE and $v = \frac{256}{\Lambda}$ for AVX. We can see that the above limitation is caused by the data structures `leafindex` and `nodemask`. Our proposed algorithm will redesign the data structure to break through the limitations in QuickScorer and V-QuickScorer.

4 DATA STRUCTURE IN RAPIDSCORER

In order to efficiently improve the traversal speed of the tree ensembles, we propose a novel algorithm called RapidScorer. Our algorithm takes the advantages of the tree traversal of QuickScorer and V-QuickScorer, but completely avoids their disadvantages by presenting three strategies. In Section 4.1, we present a novel tree-size insensitive encoding method for tree nodes, which provides a much more compact memory footprint, faster bitwise AND operations, and better flexibility to further support parallelism of multiple samples simultaneously. In Section 4.2, we present the equivalent nodes merging, in which the nodes with same splitting feature and threshold are grouped together so that we only need to take an one-time boolean test for those nodes. In Section 4.3, we design a novel layout for the efficiency of RapidScorer for data-level parallelization.

4.1 Epitome Structure on Nodes

As discussed in Section 3, previous tree ensemble traversal algorithms [10, 23, 24] represent the nodes of decision trees by boolean vectors like the `nodemask` structure. This data structure offers a technical framework to use boolean representation in tree ensemble evaluation. However, for large trees, this representation is redundant and inefficient, which hurts the calculation efficiency in the update of `leafindex`. For example, given a larger decision tree with 400 leaves, the length of $\texttt{nodemask}[n_i^h]$ is 400, which is much longer than that in Figure 1 with 7 leaves. Meanwhile, using trees with larger number of leaves in GBDT usually leads to better model performance. As illustrated in Table 1, we applied an ensemble in Bing Ads with 1,200 trees on 256 features for ad click prediction (click or non-click binary classification). When Λ is changed from 32, 64, 128, 256 to 400, we can see the AUC score is increased.

To improve the calculation efficiency for trees with large number of leaves, our first strategy is to encode the node representation into a new data structure.

4.1.1 *The Number of* **0***s in* `nodemask` *on Decision Trees.*

An important fact for the update process of `leafindex` is: only bit-**0** matters while bit-**1** does not change the bitwise AND operation result, i.e., given arbitrary bit y, $\mathbf{0} \wedge y = \mathbf{0}$, $\mathbf{1} \wedge y = y$. This fact inspires that we only need to focus on bit-**0**s in `nodemask` for the update of `leafindex`. We further notice that, averagely, bit-**0**s only occupy a small percentage of the total bits in `nodemask`. For example, in Figure 1, the number of bit-**0**s is only 1 in these nodes: n_3^h, n_4^h, n_5^h. In the following theorem, we will prove that, averagely, the percentage of bit-**0**s in `nodemask` is approximately $\sqrt{\pi L_h}/2L_h$.

Theorem 1: For decision trees with L leaves, the average number of bit-**0**s (referred to as *#0s*) in the `nodemask` of each node is $O(\sqrt{L})$.

Proof. We assume the appearance probabilities of binary trees in different shapes are the same[2]. Given L leaves (i.e., $m = L - 1$ nodes), according to a proved theorem, *Theorem 2* [16], the number of binary trees is C_m, where C_m is a Catalan number,

$$C_m = \frac{1}{m+1}\binom{2m}{m} = \frac{(2m)!}{(m+1)!m!} = \prod_{k=2}^{m}\frac{m+k}{k}, \quad \textbf{for } m \geq 0. \tag{1}$$

We first calculate the sum of *#0s* (S_m) across total nodes of these C_m trees, then obtain the average *#0s* per node (P_m) by $P_m = \frac{S_m}{C_m \cdot m}$.

For $m = 0$: S_0 is 0 as there is no node.

For $m \geq 1$: a tree T with m nodes has a root node n_0 with left and right subtrees, T_l, T_r. Since n_0 is an node, T_l and T_r must have $m-1$ nodes together, i.e., given arbitrary integer q where $1 \leq q \leq m$, if T_l has $q-1$ nodes then T_r has $m-q$ nodes. Thus, according to *Theorem 2*, the number of trees for T_l and T_r are C_{q-1} and C_{m-q} respectively. S_m can be regarded as a summation of three parts (i.e., the root node n_0; the left subtree of n_0, T_l; the right subtree of n_0, T_r), which results in a recursive formula:

$$S_m = \sum_{q=1}^{m} C_{q-1} \cdot C_{m-q} \cdot q + C_{q-1} \cdot S_{m-q} + C_{m-q} \cdot S_{q-1} \tag{2}$$

where $C_{q-1} \cdot C_{m-q} \cdot q$ is the summation of *#0s* for root node n_0; $C_{m-q} \cdot S_{q-1}$ is the summation of *#0s* for all nodes in T_l; $C_{q-1} \cdot S_{m-q}$ is the summation of *#0s* for all nodes in T_r.

Solving Equation 2, we can get S_m as:

$$S_m = 2^{2m-1} - \frac{m+1}{2} \cdot C_m \tag{3}$$

Thus, the average *#0s* per node (P_m) can be obtained by:

$$P_m = \frac{S_m}{C_m \cdot m} = \frac{2^{2m-1}}{C_m \cdot m} - \frac{m+1}{2m}, \quad \textbf{for } m \geq 1. \tag{4}$$

P_m can be approximated by Stirling's approximation [29], which is an approximation for factorials in mathematics:

$$m! \sim \sqrt{2\pi m}\left(\frac{m}{e}\right)^m \tag{5}$$

[2]In practice, the trees usually tend to be balanced. This property would make a better result for the *#0s*, which is $O(\log L)$. We omit its proof due to space limit.

$$P_m = \frac{2^{2m-1}}{m \cdot C_m} - \frac{m+1}{2m} = \frac{2^{2m-1}}{m} \cdot \frac{(m+1)(m!)^2}{(2m)!} - \frac{m+1}{2m}$$
$$\sim \frac{2^{2m-1}}{m} \cdot \frac{2\pi m(m+1)\left(\frac{m}{e}\right)^{2m}}{\sqrt{4\pi m}\left(\frac{2m}{e}\right)^{2m}} - \frac{m+1}{2m} \quad (6)$$
$$= \frac{\sqrt{\pi m}}{2} + \frac{\sqrt{\pi m} - m - 1}{2m} < \frac{\sqrt{\pi m}}{2} < \frac{\sqrt{\pi L}}{2} = O\left(\sqrt{L}\right)$$

□

4.1.2 Epitome Data Structure.

Inspired by *Theorem 1*, we introduce a novel data structure called `epitome` to encode the decision tree nodes, which allows performing `AND` operations only on bit-**0**s. Epitome is a modified run-length encoding [28], which has two aspects: one is the compact data encoding, and the other is the efficient operator computation.

For compact data encoding, we replace the `nodemask` of a node by `epitome`, which contains the first and the last bytes that contain bit-**0**s, together with their positions. The four bytes are defined as: `fb`, the first byte that contains bit-**0**s; `fbp`, the byte position of `fb`; `eb`, the last byte that contains bit-**0**s; and `ebp`, the byte position of `eb`. We use a quadruple `ep` = {`fb`, `fbp`, `eb`, `ebp`} to denote `epitome`.

The `AND` operation between `leafindex` and `epitome` is given in Algorithm 2. For ease of understanding, we use a two-dimensional array to describe `leafindex`. In implementation, we only need a one-dimensional array, as the lengths of all `leafindex` $[T_h]$ are the same, and thus the byte index such as `ep.fbp` will be replaced by a global index. Algorithm 2 only does logic `AND` on the bytes with bit-**0**s. *Thus, the epitomization procedure reduces the average CPU complexity from* $O(\Lambda)$ *to* $O(\sqrt{\Lambda})$ *at each node.*

Algorithm 2 Epitome `AND` operators

1: **function** EPITOME_AND(ep, `leafindex` $[T_h]$):
2: `leafindex` $[T_h]$ `[ep.fbp]` $\leftarrow$ `leafindex` $[T_h]$ `[ep.fbp]` $\wedge$ `ep.fb`
3: `leafindex` $[T_h]$ `[ep.fbp + 1 : ep.ebp]` $\leftarrow$ $00\ldots0$ $bytes$
4: `leafindex` $[T_h]$ `[ep.ebp]` $\leftarrow$ `leafindex` $[T_h]$ `[ep.ebp]` $\wedge$ `ep.eb`
5: return `leafindex` $[T_h]$

Here, we describe an optimization which further reduce the cost of updating `leafindex` $[T_h]$, `epitome_short`, which is a simplified `epitome`. It contains a tuple `epS` = {`fb`, `fbp`}, which is used when bit-**0**s can be covered by single byte block. That is, `eb` and `ebp` of a node can be omitted since `fb` and `fbp` can cover all of bit-**0**s. These nodes only need to run line 2 in Algorithm 2. In implementation, as the node processing is order-insensitive, the `epitome_short` nodes can be grouped and processed separately. In the decision tree hierarchy, `epitome_short` can be applied to most of the nodes. Empirically, Table 2 lists the distribution of nodes represented by `epitome` in the tree ensembles (20,000 trees with different Λ) trained on a public search ranking dataset [27]. Columns `epS` and `ep` report the percentages of nodes which can be represented by `epitome_short` and `epitome`, respectively. "Avg ins" means the average number of machine instructions using `epitome`. Table 2 shows that the percentage of nodes using `epitome_short` dominates in all cases.

In section 5, we will show that `epitome` works well with SIMD instruction, which can support 16 or 32 `epitomes` in one instruction.

Table 2: The distribution of nodes represented by `epitome`.

Λ	8	16	32	64	128	256	400
epS	100%	83%	75%	71%	68%	67%	66%
ep	0%	17%	25%	29%	32%	33%	34%
avg ins	1	1.34	1.49	1.59	1.64	1.66	1.67

4.2 Equivalent Nodes Merging

The second strategy is equivalent nodes merging. In the tree ensemble, there are a large number of nodes with the same splitting feature and threshold, though usually belong to different trees. This is because the upper bound of the total number of nodes $|\mathcal{T}|(\Lambda - 1)$ is usually much larger than the number of features $|F|$. In our algorithm, we regard the nodes with the same splitting feature and threshold (no matter in the same tree or not) as *equivalent nodes*, which can be grouped together as an *unique* merged node, to reduce the overall cost in detecting the `FALSE` nodes.

To perform the equivalent nodes merging, we present a new structure `eqnode` to represent all these nodes. That is, `eqnode` = $(\theta, u,$ `treeids`, `epitomes`$)$, where θ is the feature threshold of these nodes, u is the number of nodes before merging, `treeids` is an array which stores the corresponding tree id h each node belongs to, and `epitomes` is an array which stores the corresponding `epitome` of each node. With `eqnode`, `FALSE` nodes detection can be accomplished by just one-time processing for all the nodes in the group.

We computed the statistics on the numbers of nodes in the tree ensembles used in Bing Ads for ad click prediction. The results are shown in Table 3, where *#unique* is the number of unique nodes after merging, *#original* is the number of nodes before merging, and *Ratio* is the ratio of *#unique* over *#original*. The results show that there are a significant number of equivalent nodes. In addition, as Λ increases, the percentage of unique nodes decreases. For example, in the ensemble of 3,000 trees with 400 leaves each tree, only 0.93% unique nodes need processing. This implies that redundancy grows with the number of leaves, and we can save more computations by leveraging equivalent nodes merging with larger number of leaves.

Table 3: Statistics on number of nodes in tree ensembles before and after equivalent nodes merging.

Λ	1,000 trees			3,000 trees		
	#unique	*#original*	*Ratio*	*#unique*	*#original*	*Ratio*
8	3,565	7,000	50.93%	6,959	21,000	33.14%
16	5,686	15,000	37.91%	8,942	45,000	19.87%
32	7,726	31,000	24.92%	10,043	93,000	10.80%
64	9,218	63,000	14.63%	10,575	189,000	5.60%
128	9,993	127,000	7.87%	10,837	381,000	2.84%
256	10,382	255,000	4.07%	11,016	765,000	1.44%
400	10,707	399,000	2.68%	11,152	1,197,000	0.93%

4.3 Layout for Boolean Vector Tree Traversal

In the third strategy, we redesign the layout of required data structures in memory, since it is crucial for the efficiency of our RAPIDSCORER algorithm. We leverage the layout in QUICKSCORER with our `epitome` encoding and `eqnode` structure. Specifically, the arrays of

nodes, offset, leafindexes, and leafvalues are all juxtaposed one after the other as illustrated in Figure 2, to represent the data structure of the tree ensemble $\mathcal{T}$.

- nodes is a global array of eqnode structures, where each item groups nodes with the same feature and threshold. As these nodes may belong to different trees and may have different nodemasks, we keep their original tree ids as treeids and transform their corresponding nodemasks into epitomes. Suppose the number of eqnodes with the same splitting feature f_k is denoted by $|f_k|$. These $|f_k|$ eqnodes are grouped together and sorted by their thresholds in an ascending way.
- offset is an auxiliary array to mark the starting position of each feature in the global array of nodes, since different features may have different numbers of eqnodes.
- leafindexes is a global array juxtaposed by the boolean vectors of leafindex $[T_h]$. To perform multiple samples parallelism, leafindex $[T_h]$ has v copies, each corresponding to one sample. To make full use of SIMD, we propose a new structure called ByteTransposition for leafindex $[T_h]$ in Section 5.2.
- leafvalues is a global array to store the values for leaves of each tree grouped by their tree ids, i.e., a group of leafvalue $[T_h]$.

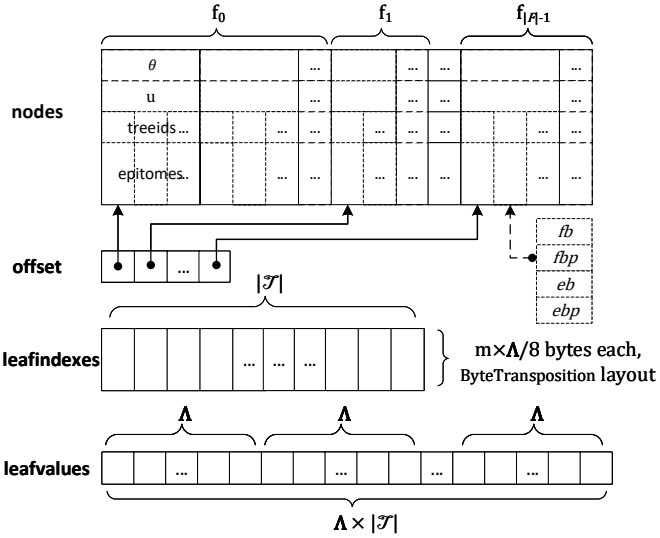


Figure 2: Arrays used by RapidScorer.

With this layout, the data flow runs through the features to find FALSE nodes. For each FALSE node n_i^h, leafindex $[T_h]$ is updated by epitome AND operation. The algorithm eventually finds the indexes of leftmost TRUE bits in leafindex $[T_h]$ for v samples (feature vectors).

5 VECTORIZATION WITH SIMD

In this section, we will show that SIMD can coordinate well with the strategies in Section 4 to speed up the calculation by one order of magnitude. The basic idea of vectorization with SIMD is to group a set of data together which have the same operations. Following this idea, we do vectorization on multiple samples, which exactly meets the scenario of scoring multiple documents for a given query. We first show the complete RapidScorer Algorithm in its logical flow, and then dive into the details for each vectorization step. We will present its three advantages over V-QuickScorer, corresponding to the three major steps in RapidScorer.

Algorithm 3 The RapidScorer Algorithm

Input:
- χ: combination of feature vectors $\{\mathbf{x}\}$ of v samples
- $\mathcal{T}$: ensemble of binary decision trees, with nodes, offset, leafindexes, and leafvalues

Output:
- $\overrightarrow{s}$: final score vector of χ

```
1:  function RapidScorer(χ, 𝒯):
2:      for h ∈ 0, 1, ..., |𝒯| − 1 do
3:          leafindex[T_h] ← 11...1
4:      for k ∈ 0, 1, ..., |F| − 1 do
5:          // Vectorized_FalseNodeDetection: line 6-8
6:          x_k ← (χ_kv, ..., χ_(k+1)v−1)^T
7:          for i ∈ 0, 1, ..., |f_k| − 1 do
8:              η ← x_k ≤ nodes[k][i].θ
9:              if η ≠ FF_hex then          // FF_hex = 11111111
10:                 for q ∈ 0, 1, ..., nodes[k][i].u − 1 do
11:                     h ← nodes[k][i].treeids[q]
12:                     p ← nodes[k][i].epitomes[q]
13:                     Vectorized_AND(p, leafindex[T_h], η)
14:             else
15:                 break
16:     s ← 0
17:     for h ∈ 0, 1, ..., |𝒯| − 1 do
18:         c ← Vectorized_FindLeafIndex(leafindex[T_h])
19:         s ← s + w_h · leafvalue[T_h][c]
20:     return s
```

5.1 The RapidScorer Algorithm

The complete RapidScorer algorithm is illustrated in Algorithm 3. The inputs are: the feature vector χ of v samples and the ensemble of trees $\mathcal{T}$. χ is an array that combines all feature vectors $\{\mathbf{x}\}$ of the v samples. To perform vectorization, v should be a constant that depends on the register width, i.e., $v = \lceil \frac{r}{\lambda} \rceil$, where r is the maximum register width and λ is the minimum processing unit for each sample. In our experiments, λ is set to 8, length of a byte, and r is 128 (SSE) or 256 (AVX)[3]. In the storage of multiple samples, feature values from the same feature are placed contiguously, i.e., $\chi_{kv}, \cdots, \chi_{(k+1)v}$ store the values of the kth feature for the v samples.

The scoring process in RapidScorer runs as follows. At first, RapidScorer initializes leafindexes to be all **1**s (line 2-3). Then, it loops over all the features in F (line 4) to identify whether a node is a FALSE node for each sample (line 6-15). The information of FALSE nodes will be stored in an indicator vector $\overrightarrow{\eta}$[4] (line 8). If $\overrightarrow{\eta}$ does not equal to $\overleftarrow{\mathtt{FF}_{hex}}$[5] (line 9), which means that the current node is still a FALSE node for some samples in the current feature f_k,

[3]Note that r can be 512 if AVX-512 is supported. It can also be 64 or less if the machine does not support SIMD instructions.

[4]A symbol with a right arrow on head like $\overrightarrow{\eta}$ means a vector containing v different items of this symbol, belonging to the v samples in the parallelization batch.

[5]A symbol with a left arrow on head like $\overleftarrow{\mathtt{FF}_{hex}}$ means a vector containing v identical items of this symbol, belonging to the v samples in the parallelization batch.

then RapidScorer will iterate over all the same-threshold nodes and do vectorized AND operation (see Algorithm 4) on each node to update $\texttt{leafindex}[T_h]$[6] (line 10-13); otherwise it will break (line 14-15) and move to the next feature f_{k+1}. After finishing updating $\texttt{leafindex}[T_h]$, RapidScorer inspects all items in $\texttt{leafindex}[T_h]$ (line 17) to identify the indexes of the exit leaves of the v samples (line 18), and uses these indexes to obtain the values stored in $\texttt{leafvalue}[T_h]$ (line 19). Finally, the output scores $\overrightarrow{s}$ will be updated by the obtained values (line 19).

5.2 Vectorizations

We dive into the details in each vectorization step to explain how we vectorize all operations to make full use of the SIMD performance, so that RapidScorer is able to support 16 ways or 32 ways data parallelism with SSE or AVX. We will describe three major steps that contribute to the speed-up of RapidScorer: vectorized FALSE node detection based on equivalent nodes merging, vectorized AND operation based on epitome, and vectorized index of exit leaves.

Step 1 is to detect FALSE nodes. Based on equivalent nodes merging, we utilize SIMD to parallelize the process of detecting FALSE nodes, called Vectorized_FalseNodeDetection. In this step, multiple samples can be tested against a given node predicate, p. The testing results will be stored in a vector $\overrightarrow{\eta}$. As the feature values may vary from different samples, given testing condition in line 8, p could be FALSE node for some samples but TRUE node for the others. If all tests of p on v samples result in TRUE, i.e., we do not have any FALSE nodes, then the next feature is processed; otherwise, $\texttt{leafindex}[T_h]$ is updated by vectorized AND operation. To ensure logical AND is only performed on FALSE nodes, $\overrightarrow{\eta}$ will be used as a mask in step 2.

Step 2 is to update tree vectors for the samples by AND operation. As the operations in Algorithm 2 are all byte-based operations, they can be vectorized perfectly with SIMD. Given the current node p, the test indicator vector $\overrightarrow{\eta}$, and tree vectors $\texttt{leafindex}[T_h]$, the vectorized logical AND is described in Algorithm 4, where only line 2 is required if p is an epitome_short node. The correctness of Algorithm 4 can be verified easily, so we omitted it to save space. To ensure that the memory access for vectorized AND operation of v samples is sequential, we reorder the bytes in $\texttt{leafindex}[T_h]$ such that the bytes corresponding to the same byte index of v samples are located contiguously. We named this layout as ByteTransposition layout, which is illustrated in the top table of Figure 3.

Step 3 is to find exit leaf indexes for v samples, i.e. Vectorized_FindLeafIndex. The task is to quickly find the leftmost TRUE bits from $\texttt{leafindex}[T_h]$. We present a novel strategy with SIMD, which first finds the byte-wise position, and then finds the bit-wise position precisely. Figure 3 shows the process of vectorizing the exit leaf index finding. The $\texttt{leafindex}[T_h]$ for v samples are located in the ByteTransposition layout. Each single row fits into the size of one SIMD register. In **Step 3.a**, the algorithm operates on all v tree vectors, and returns two separate results. One is the first byte that contains bit-**1**, shown as the row denoted by $\overrightarrow{b}$ in the left branch. The other is the index of the first byte (starting with 00_{hex}) that contains bit-**1**, shown as the row denoted by $\overrightarrow{c_1}$ in the right branch. For example, in $\texttt{leafindex}[T_h]_2 = (00, 69, 07...)^T_{hex}$, we first inspect the first byte, 00_{hex}, which does not have bit-**1**, then move to the second byte 69_{hex}, which has bit-**1**. Thus, 69_{hex} is recorded into $\overrightarrow{b}$, and its position index 01_{hex} is written into $\overrightarrow{c_1}$. In **Step 3.b**, the algorithm finds the index of the first bit-**1** at each byte, denoted by $\overrightarrow{c_2}$. Either in Step 3.a or Step 3.b, an auxiliary indicator vector is adopted to record whether the first non-zero byte or bit-**1** has been found, which is similar to $\overrightarrow{\eta}$ in Algorithm 4. In **Step 3.c**, the final bit-wise indexes for the leftmost TRUE, denoted by $\overrightarrow{c}$, is simply computed by $\overrightarrow{c} \leftarrow \overrightarrow{c_1} \times 8 + \overrightarrow{c_2}$. Thus, we get the exit leaf indexes of the v samples for T_h simultaneously.

Algorithm 4 Vectorized logical AND operation

```
1: function Vectorized_AND(p, leafindex[T_h], η):
2:     leafindex[T_h][p.fbp] ← leafindex[T_h][p.fbp] ∧ η ∨ p.fb
3:     if p.fbp ≠ p.ebp then
4:         for k = p.fbp + 1 to p.ebp − 1 do
5:             leafindex[T_h][k] ← η ∧ leafindex[T_h][k]
6:         leafindex[T_h][p.ebp] ← leafindex[T_h][p.ebp] ∧ η ∨ p.eb
```

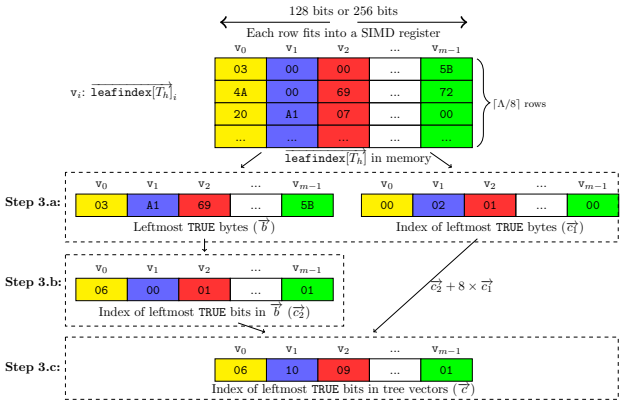


Figure 3: Illustration of vectorizing leaf indexing strategy.

6 EXPERIMENTAL EVALUATION

In this section, we conduct a set of quantitative experiments to compare the performance of RapidScorer with several state-of-the-art baseline algorithms, in a public dataset for search ranking and a production dataset for ad click prediction in search advertising.

6.1 Experiment Setup

6.1.1 Datasets. The experiments were conducted on two datasets: the MSN[7] dataset and the AdsCTR dataset. The MSN dataset is a public dataset with 136 features extracted from query-url pairs. There are 723,412 samples for training, 235,259 samples for validation, and 241,521 samples for testing. We trained GBDT models on this dataset using an open source tool, XGBoost [7][8], with 8, 16, 32,

[6] $\texttt{leafindex}[T_h]$ represents $\overrightarrow{\texttt{leafindex}[T_h]}$ in paragraphs of Section 5.

[7] https://www.microsoft.com/en-us/research/project/mslr/

[8] XGBoost is widely used by data scientists, and is behind many winning solutions of various machine learning challenges. https://github.com/dmlc/xgboost

Table 4: Per-sample scoring time in μs of RapidScorer(RS), V-QuickScorer(vQS), QuickScorer(QS), VPred, and XGBoost(XB). Orange numbers highlight the best algorithm; blue numbers highlight the best previous results.

Method	Λ	Number of trees							
		1,000		5,000		10,000		20,000	
		MSN	AdsCTR	MSN	AdsCTR	MSN	AdsCTR	MSN	AdsCTR
RS(AVX)		**0.9 (-)**	**1.4 (-)**	**4.6 (-)**	**6.3 (-)**	**9.1 (-)**	**11.8 (-)**	**18.1 (-)**	**23.1 (-)**
RS(SSE)		1.1 (1.2x)	1.8 (1.3x)	5.2 (1.1x)	7.6 (1.2x)	10.3 (1.1x)	13.5 (1.1x)	20.6 (1.1x)	26.4 (1.1x)
QS	8	2.4 (2.7x)	4.9 (3.4x)	11.1 (2.4x)	18.6 (3.0x)	22.2 (2.4x)	36.0 (3.0x)	44.8 (2.5x)	64.5 (2.8x)
VPred		7.8 (8.8x)	8.9 (6.2x)	38.9 (8.4x)	44.4 (7.1x)	76.0 (8.3x)	88.8 (7.5x)	153.3 (8.4x)	177.2 (7.7x)
XB		25.9 (28.9x)	30.9 (21.5x)	122.7 (26.4x)	178.9 (28.5x)	261.1 (28.6x)	359.9 (30.4x)	604.6 (33.3x)	760.9 (32.9x)
RS(AVX)		**1.4 (-)**	**2.1 (-)**	**6.8 (-)**	**9.5 (-)**	**12.8 (-)**	**17.6 (-)**	**21.7 (-)**	**35.1 (-)**
RS(SSE)		1.5 (1.1x)	2.6 (1.3x)	7.8 (1.1x)	11.5 (1.2x)	15.0 (1.2x)	22.0 (1.2x)	24.5 (1.1x)	43.3 (1.2x)
QS	16	3.1 (2.2x)	7.5 (3.6x)	16.5 (2.4x)	29.0 (3.1x)	33.2 (2.6x)	53.7 (3.0x)	73.6 (3.4x)	105.9 (3.0x)
VPred		16.2 (11.4x)	16.0 (7.6x)	80.3 (11.8x)	79.8 (8.4x)	161.5 (12.7x)	160.5 (9.1x)	319.7 (14.7x)	326.8 (9.3x)
XB		38.3 (27.1x)	49.4 (23.5x)	185.4 (27.2x)	236.2 (24.9x)	423.6 (33.2x)	475.0 (26.9x)	921.8 (42.5x)	976.5 (27.9x)
RS(AVX)		**2.5 (-)**	**3.4 (-)**	**11.9 (-)**	**15.2 (-)**	**25.0 (-)**	**30.4 (-)**	**48.3 (-)**	**58.8 (-)**
RS(SSE)		2.7 (1.1x)	4.1 (1.2x)	13.4 (1.1x)	17.9 (1.2x)	28.4 (1.1x)	35.6 (1.2x)	54.4 (1.1x)	69.9 (1.2x)
vQS(AVX)	32	3.2 (1.3x)	5.1 (1.5x)	17.8 (1.5x)	23.0 (1.5x)	39.3 (1.6x)	49.4 (1.6x)	81.0 (1.7x)	110.0 (1.9x)
QS		5.1 (2.1x)	11.5 (3.4x)	27.2 (2.3x)	49.0 (3.2x)	62.6 (2.5x)	94.6 (3.1x)	168.6 (3.5x)	229.5 (3.9x)
VPred		31.2 (12.6x)	30.7 (9.0x)	164.9 (13.8x)	158.4 (10.4x)	345.8 (13.8x)	332.1 (10.9x)	720.8 (14.9x)	736.3 (12.5x)
XB		47.8 (19.4x)	63.7 (18.7x)	264.9 (22.2x)	303.8 (20.0x)	560.8 (22.4x)	636.9 (20.9x)	1343.9 (27.8x)	1253.1 (21.3x)
RS(AVX)		**4.3 (-)**	**6.0 (-)**	**20.8 (-)**	**28.2 (-)**	**40.2 (-)**	**56.8 (-)**	**88.1 (-)**	**124.8 (-)**
RS(SSE)		5.3 (1.3x)	6.8 (1.1x)	25.4 (1.2x)	32.0 (1.1x)	45.8 (1.1x)	64.7 (1.1x)	98.6 (1.1x)	141.2 (1.1x)
vQS(AVX)	64	7.3 (1.7x)	12.8 (2.1x)	40.2 (1.9x)	61.4 (2.2x)	139.6 (3.5x)	144.1 (2.5x)	304.2 (3.5x)	361.9 (2.9x)
QS		10.4 (2.4x)	19.7 (3.3x)	57.8 (2.8x)	99.1 (3.5x)	154.6 (3.8x)	220.7 (3.9x)	435.9 (4.9x)	514.5 (4.1x)
VPred		59.9 (14.0x)	56.5 (9.4x)	338.7 (16.3x)	320.0 (11.3x)	736.4 (18.3x)	683.0 (12.0x)	1284.8 (14.6x)	1372.8 (11.0x)
XB		60.6 (14.2x)	78.2 (13.0x)	366.3 (17.6x)	387.1 (13.7x)	821.1 (20.4x)	778.3 (13.7x)	2280.3 (25.9x)	1899.2 (15.2x)
RS(AVX)		**6.8 (-)**	**11.0 (-)**	**33.8 (-)**	**55.5 (-)**	**75.5 (-)**	**124.2 (-)**	**230.5 (-)**	**366.9 (-)**
RS(SSE)		8.9 (1.3x)	12.4 (1.1x)	43.6 (1.3x)	62.2 (1.1x)	96.5 (1.3x)	139.7 (1.1x)	290.2 (1.3x)	415.0 (1.1x)
QS	128	39.2 (5.8x)	64.4 (5.8x)	268.4 (7.9x)	430.5 (7.8x)	618.9 (8.2x)	1086.9 (8.8x)	2083.6 (9.0x)	3143.8 (8.6x)
VPred		77.6 (11.5x)	79.5 (7.2x)	441.8 (13.1x)	436.9 (7.9x)	921.3 (12.2x)	952.2 (7.7x)	2154.3 (9.3x)	2114.1 (5.8x)
XB		85.0 (12.6x)	89.8 (8.1x)	515.4 (15.2x)	504.1 (9.1x)	1156.0 (15.3x)	926.1 (7.5x)	3112.4 (13.5x)	2805.5 (7.6x)
RS(AVX)		**12.3 (-)**	**21.2 (-)**	**70.1 (-)**	**119.1 (-)**	**184.7 (-)**	**282.3 (-)**	**595.8 (-)**	**793.1 (-)**
RS(SSE)		16.4 (1.3x)	24.1 (1.1x)	88.5 (1.3x)	139.2 (1.2x)	228.2 (1.2x)	341.1 (1.2x)	710.1 (1.2x)	913.6 (1.2x)
QS	256	148.8 (12.1x)	182.2 (8.6x)	1226.9 (17.5x)	1817.7 (15.3x)	3167.0 (17.1x)	4276.2 (15.2x)	6937.5 (11.6x)	9042.6 (11.4x)
VPred		91.0 (7.4x)	96.7 (4.6x)	529.9 (7.6x)	558.6 (4.7x)	1203.7 (6.5x)	1264.9 (4.5x)	2645.9 (4.4x)	2715.1 (3.4x)
XB		117.5 (9.6x)	101.0 (4.8x)	675.4 (9.6x)	524.6 (4.4x)	1356.9 (7.3x)	1268.6 (4.5x)	4570.0 (7.7x)	3934.2 (5.0x)
RS(AVX)		**25.2 (-)**	**36.0 (-)**	**196.8 (-)**	**269.0 (-)**	**475.2 (-)**	**768.0 (-)**	**1260.6 (-)**	**1799.2 (-)**
RS(SSE)		34.0 (1.3x)	41.6 (1.2x)	240.8 (1.2x)	319.5 (1.2x)	581.5 (1.2x)	887.5 (1.2x)	1438.0 (1.1x)	2001.0 (1.1x)
QS	400	536.0 (21.3x)	727.0 (20.2x)	4847.1 (24.6x)	6231.3 (23.2x)	11858 (25.0x)	13198 (17.2x)	23465 (18.6x)	28628 (15.9x)
VPred		123.4 (4.9x)	125.6 (3.5x)	729.6 (3.7x)	743.7 (2.8x)	1798.0 (3.8x)	1953.4 (2.5x)	3915.8 (3.1x)	4130.5 (2.3x)
XB		139.1 (5.5x)	111.6 (3.1x)	868.1 (4.4x)	977.0 (3.6x)	2070.5 (4.4x)	2006.4 (2.6x)	5125.6 (4.1x)	4943.9 (2.7x)

64, 128, 256 and 400 leaves. The AdsCTR is the dataset used in Bing Ads for ad click prediction with 870 features, which was sampled from real traffic. The query-ad pairs for training and testing are 810,516 and 263,509 respectively. We trained GBDT models on this dataset with 8, 16, 32, 64, 128, 256 and 400 leaves. Specifically, the model with 400 leaves is used in the production.

6.1.2 Baselines. We have implemented two versions of RapidScorer (RS): RS(SSE) using SSE4.2 instructions and RS(AVX) using AVX2 instructions. The scoring efficiency of RS is compared with four baselines: QuickScorer [10, 23] (QS), V-QuickScorer [24] (vQS), VPred [2], and XGBoost [7] (XB). As QS and vQS are not publicly available, we implemented them to our best, [9] which achieved similar scoring performance to that reported in [10, 23, 24] on the MSN dataset. It should be mentioned that vQS only supports trees with 32 and 64 leaves in the AVX-based implementation [24]. We adopt the implementation of VPred[10] and XB from Github. All models and the test set were transformed to be XGBoost-compatible for fair evaluation. Each test ran 3 times and the averaged per-sample scoring latency was used for evaluation.

All the implementations were compiled with GCC 5.4.0 with the highest optimization settings (-O3). The code was executed in a single core on a machine equipped with an Intel Core E5-1650 v4 clocked at 3.60Ghz. The machine has 32GiB RAM, running Windows 10. The CPU has three levels of cache. Level 1 and 2

[9]For both QS and RS, we do not adopt the strategies of *blocking* and *tree reversing*, as they need model-specific tuning (e.g., block size and the percentage of FALSE nodes in each tree), while their gains are limited even in the best case reported in [10, 23] (i.e., 1.55x when $\Lambda = 64$ and $|\mathcal{T}| = 20,000$), RS (AVX) achieves much better improvement of 4.9x under the same conditions in Table 4.

[10]https://github.com/lintool/OptTrees.git

caches are core-private with 32 KB and 256 KB, respectively, while level 3 is a shared cache with 15 MB.

6.2 Evaluation Result

As both RS and other baseline algorithms provide exactly the same scoring results (same scoring result as the naïve tree traversal), our experiments focus on comparing their runtime performance and analyzing the contributions of the proposed strategies in RS.

6.2.1 Compared with baselines. The average time (in μs) needed by the different algorithms to score each document of the two datasets MSN and AdsCTR are reported in Table 4. We list the results *w.r.t.* the algorithm name, the ensemble size, and the number of leaves (Λ). For each test the table also reports between parentheses the gain factor of RS over its competitors. From Table 4, we find that: (i) RS outperforms the state-of-the-art algorithms, with speed-up 1.3x-3.6x for $\Lambda \leq 64$, and 2.3x-9.0x for $\Lambda > 64$. (ii) RS works consistently well across the ensemble sizes and the number of leaves. We provide detailed discussions as below.

First, we compare RS with vQS and QS, which are all based on the *feature by feature* traversal of tree ensembles. We observe that RS surpasses vQS and QS significantly even when Λ is small. This owes to the compactness of `epitome` structure in SIMD register which maximizes the parallelism. (i) Compared with vQS(AVX), RS(AVX) executes 1.3x-1.9x faster when $\Lambda = 32$, in which vQS(AVX) was reported to perform the best in the literature [24]. When $\Lambda = 64$, RS(AVX) executes 1.7x to 3.5x faster than vQS(AVX). Specifically, RS(AVX) obtains 32-way parallelism independent of the number of leaves; however, vQS(AVX) only obtains 8-way parallelism when $\Lambda = 32$, 4-way parallelism when $\Lambda = 64$, and no parallelism when $\Lambda > 128$. (ii) Compared with QS, RS(AVX) achieves significant speed-up, i.e. 2.1x-25x. The speed-up is relative stable (2.1x to 4.9x) when $\Lambda \leq 64$, and it increases sharply (5.8x to 25x) when $\Lambda > 64$. The reason is that, to update `leafindex` $[T_h]$, multiple 64-bit `AND` operations per node are needed in QS when $\Lambda > 64$.

Then, we compare RS with VPred and XG. Different with RS, VPred and XG adopt *tree by tree* traversal of tree ensembles, evaluating each tree from root to leaf. As QS is not good at dealing with large Λ, VPred and XG surpass QS when $\Lambda \geq 256$. However, RS(AVX) still achieves huge speed-up: 2.3x-18.3x faster than VPred, and 2.6x-42.5x faster than XG. This is due to the combination of `epitome` structure and the vectorization with SIMD instructions. Note that, the speed-up of RS(AVX) over VPred and XG decreases slightly when Λ grows, which shows the RS(AVX)'s advantage on cache locality is partially counteracted by dealing with more nodes.

Finally, we compare RS(AVX) with RS(SSE), the speed-up is from 1.1x to 1.3x, less than 2x. The main reason is that, the benefit of longer register width is partially counteracted by larger padding cost of AVX register and pressure on memory access bandwidth.

6.2.2 Analysis on strategy contributions. The superior performance of RS relies on a combination of several proposed strategies. Figure 4 shows the improvement of each strategy, in which QS is considered as the baseline and each strategy is added at a time. (i) "+Epitome" (EP) means replacing the `nodemask` $[T_h]$ by `epitome` encoding and bitwise `AND` operation by `epitome AND` operation (Section 4.1.2). (ii) "+NodesMerging" (NM) means merging the equivalent nodes to reduce the cost of detecting the `FALSE` nodes upon EP (Section 4.2). (iii) "+VecAND" (VA) means vectorizing the `FALSE` node detection of multiple samples and the `AND` operation upon NM (Step 1 and 2 in Section 5.2). (iv) "+VecExitLeafIndex" (VE) means vectorizing the computation to locate the exit leaf nodes upon VA (Step 3 in Section 5.2). Note that VE exactly equals to RS(SSE). From Figure 4, we have the following observations.

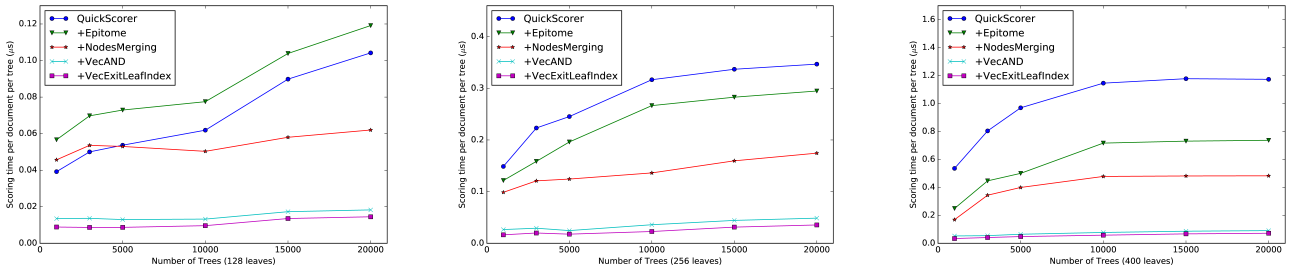
EP and NM optimizations aim at preserving only necessary computations. (i) EP changes the data structure of nodes to save computation and maximize parallelism. This speed-up effect is apparent in leaf sizes 256 and 400, but there is a little drop for EP v.s. QS at small leaf sizes. This is because in the update of `leafindex` $[T_h]$, `epitome AND` works at byte granularity, but QS works at coarse granularity which is limited by the machine word of modern CPUs, typically 64 bits. As the number of leaves increases, EP achieves a notable speed-up. The improvement is around 80% when $\Lambda = 400$ (Figure 4c), where QS needs 7 operations of 64 bits `AND` while EP only needs average 1.67 `AND` operations, shown in Table 2. In other words, computational complexity in RS is independent of Λ. (ii) For NM, the speed-up over QS and EP increases when the number of trees grows (X-axis). This validates the observation in Table 3 that the ratio of unique nodes drops when the number of trees grows. As NM merges all the nodes with the same feature and threshold as one unique node to perform one-time `FALSE` node detection, it significantly saves the average detection time.

The VA and VE optimizations together vectorize all operations in RS with SIMD, which achieves a speed-up from 5.5x to 6.5x over non-vectorized version. (i) VA achieves a significant speed-up, ranging from 42.8% to 603.7%. This is due to the perfect combination of `epitome` structure and `ByteTransposition` layout, which improves the parallelism in vectorization, i.e., 16 samples can be served using SSE4.2 instructions. (ii) VE further improves the scoring speed from 24.3% to 330.8%, as it vectorizes the process of finding exit leaf indexes such that the entire algorithm can be vectorized to avoid the sequential bottleneck.

7 CONCLUSIONS AND FUTURE WORK

We presented RapidScorer to speed up the scoring process of tree ensembles by systematically compacting CPU operations and making full use of SIMD register. Specifically, we defined the `epitome` structure to reduce the computational complexity of the update process for each `FALSE` node from $O(\Lambda)$ to $O(\sqrt{\Lambda})$; we designed the equivalent nodes merging strategy to reduce the cost of `FALSE` node detection by grouping nodes with the same feature and threshold; we proposed the RapidScorer algorithm that can maximize the compactness of data units and fully utilize SIMD data-level parallelization. Experimental evaluations show that RapidScorer significantly outperforms the state-of-the-art algorithms.

For the future work, for much larger Λ, we would like to explore a hybrid approach that combines RapidScorer with the conventional root-to-leaf evaluation, to leverage the fact that there are exponentially less nodes in top layers than those in bottom layers in the decision trees; we would also like to extend RapidScorer to work on other tree ensemble models such as Random Forest [3].



(a) 128 leaves (b) 256 leaves (c) 400 leaves

Figure 4: Per-tree per-document scoring time in μs of each optimization strategy from QuickScorer to RapidScorer (SSE) in GBDT models with 128, 256, and 400 leaves in MSN dataset.

8 ACKNOWLEDGEMENTS

The authors would like to thank Yang Qi (Bing Ads), Xiaoliang Ling (Bing Ads), Weiwei Deng (Bing Ads), Yunzhi Zhou (Bing Ads), Chengchao Yu (Azure Data), and Yangyang Lu (OneNote) for their support and discussions that benefited the development of RapidScorer.

REFERENCES

[1] 2011. Intel® 64 and IA-32 Architectures Software Developer Manual. (2011).
[2] Nima Asadi, Jimmy Lin, and Arjen P De Vries. 2014. Runtime optimizations for tree-based machine learning models. *IEEE Transactions on Knowledge and Data Engineering* 26, 9 (2014), 2281–2292.
[3] L. Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[4] C.J.C. Burges. 2010. *Fr. RankNet to LambdaRank to LambdaMART: An Overview.*
[5] B Barla Cambazoglu, Hugo Zaragoza, Olivier Chapelle, Jiang Chen, Ciya Liao, Zhaohui Zheng, and Jon Degenhardt. 2010. Early exit optimizations for additive machine learned ranking systems. In *Proceedings of the third ACM international conference on Web search and data mining*. ACM, 411–420.
[6] Olivier Chapelle and Yi Chang. 2011. Yahoo! Learning to Rank Challenge Overview. In *Proceedings of the Yahoo! Learning to Rank Challenge*. 1–24.
[7] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. ACM, 785–794.
[8] T. Chen, L. Tang, Q. Liu, D. Yang, S. Xie, X. Cao, C. Wu, E. Yao, Z. Liu, Z. Jiang, et al. 2012. Combining factorization model and additive forest for collaborative followee recommendation. *KDD CUP* (2012).
[9] Gianni Conte, Stefano Tommesani, and Francesco Zanichelli. 2000. The long and winding road to high-performance image processing with MMX/SSE. In *Computer Architectures for Machine Perception, 2000. Proceedings. Fifth IEEE International Workshop on*. IEEE, 302–310.
[10] Domenico Dato, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2016. Fast ranking with additive ensembles of oblivious and non-oblivious regression trees. *ACM Transactions on Information Systems (TOIS)* 35, 2 (2016), 15.
[11] Jeffrey Dean. 2014. Large Scale Deep Learning. (2014). https://research.google.com/people/jeff/CIKM-keynote-Nov2014.pdf
[12] W. Dong, J. Li, R. Yao, C. Li, T. Yuan, and L. Wang. 2016. Characterizing Driving Styles with Deep Learning. *arXiv:1607.03611* (2016).
[13] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. 2008. Intel AVX: New frontiers in performance improvements and energy efficiency. *White paper* (2008).
[14] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.
[15] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. 2014. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*. ACM, 1–9.
[16] Peter Hilton and Jean Pedersen. 1991. Catalan numbers, their generalization, and their uses. *The Mathematical Intelligencer* 13, 2 (1991), 64–75.
[17] Xin Jin, Tao Yang, and Xun Tang. 2016. A comparison of cache blocking methods for fast execution of ensemble-based score computation. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 629–638.
[18] Rie Johnson and Tong Zhang. 2014. Learning nonlinear functions using regularized greedy forest. *IEEE transactions on pattern analysis and machine intelligence* 36, 5 (2014), 942–954.
[19] Pat Langley and Stephanie Sage. 1994. Oblivious decision trees and abstract cases. In *Working notes of the AAAI-94 workshop on case-based reasoning*. Seattle, WA, 113–117.
[20] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. 2017. Model Ensemble for Click Prediction in Bing Search Ads. In *Proceedings of the 26th International Conference on World Wide Web Companion*. 689–698.
[21] Yin Lou and Mikhail Obukhov. 2017. BDT: Gradient Boosted Decision Tables for High Accuracy and Scoring Efficiency. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1893–1901.
[22] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Fabrizio Silvestri, and Salvatore Trani. 2016. Post-learning optimization of tree ensembles for efficient ranking. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 949–952.
[23] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2015. Quickscorer: A fast algorithm to rank documents with additive ensembles of regression trees. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 73–82.
[24] Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2016. Exploiting CPU SIMD extensions to speed-up document scoring with tree ensembles. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 833–836.
[25] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. 2009. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. In *VLDB*.
[26] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2017. CatBoost: unbiased boosting with categorical features. *arXiv preprint arXiv:1706.09516* (2017).
[27] Tao Qin and Tie-Yan Liu. 2013. Introducing LETOR 4.0 Datasets. *CoRR* abs/1306.2597 (2013). http://arxiv.org/abs/1306.2597
[28] AH Robinson and Colin Cherry. 1967. Results of a prototype television bandwidth compression scheme. *Proc. IEEE* 55, 3 (1967), 356–364.
[29] Dan Romik. 2000. Stirling's approximation for n!: The ultimate short proof? *The American Mathematical Monthly* 107, 6 (2000), 556.
[30] Schigehiko Schamoni. 2012. Ranking with Boosted Decision Trees. (2012). http://www.ccs.neu.edu/home/vip/teach/MLcourse/4_boosting/materials/Schamoni_boosteddecisiontrees.pdf
[31] Xun Tang, Xin Jin, and Tao Yang. 2014. Cache-conscious runtime optimization for ranking ensembles. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. ACM, 1123–1126.
[32] S Thakkur and T Huff. 1999. Internet streaming SIMD ext. *Computer* (1999).
[33] Stephen Tyree, Kilian Q. Weinberger, Kunal Agrawal, and Jennifer Paykin. 2011. Parallel Boosted Regression Trees for Web Search Ranking *(WWW)*. 387–396.
[34] F. Yan, Y. He, O. Ruwase, and E. Smirni. 2016. SERF: Efficient Scheduling for Fast Deep Neural Network Serving via Judicious Parallelism. *Supercomputing* (2016).
[35] Jie Zhu, Ying Shan, JC Mao, Dong Yu, Holakou Rahmanian, and Yi Zhang. 2017. Deep embedding forest: Forest-based serving with deep embedding features. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1703–1711.