A

Fast Ranking with Additive Ensembles of Oblivious and Non-Oblivious Regression Trees

Domenico Dato, Tiscali Italia S.p.A.
Claudio Lucchese, ISTI–CNR
Franco Maria Nardini, ISTI–CNR
Salvatore Orlando, Ca' Foscari University of Venice
Raffaele Perego, ISTI–CNR
Nicola Tonellotto, ISTI–CNR
Rossano Venturini, University of Pisa

Learning-to-Rank models based on additive ensembles of regression trees have been proven to be very effective for scoring query results returned by large-scale Web search engines. Unfortunately, the computational cost of scoring thousands of candidate documents by traversing large ensembles of trees is high. Thus, several works have investigated solutions aimed at improving the efficiency of document scoring by exploiting advanced features of modern CPUs and memory hierarchies. In this paper, we present QUICKSCORER, a new algorithm that adopts a novel cache-efficient representation of a given tree ensemble, it performs an interleaved traversal by means of fast bitwise operations, and also supports ensembles of oblivious trees. An extensive and detailed test assessment is conducted on two standard Learning-to-Rank datasets and on a novel very-large dataset we made publicly available for conducting significant efficiency tests. The experiments show unprecedented speedups over the best state-of-the-art baselines ranging from 1.9x to 6.6x. The analysis of low-level profiling traces shows that QUICKSCORER efficiency is due to its cache-aware approach both in terms of data layout and access patterns, and to a control flow that entails very low branch mis-prediction rates.

CCS Concepts: **•Information systems → Learning to rank; Retrieval efficiency;**

Additional Key Words and Phrases: Learning to Rank, Additive Ensembles of Regression Trees, Document Scoring, Efficiency, Cache-awareness

ACM Reference Format:
Domenico Dato, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, Rossano Venturini. 2015. Fast Ranking with Additive Ensembles of Oblivious and Non-Oblivious Regression Trees *ACM Trans. Inf. Syst.* V, N, Article A (January YYYY), 32 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

Author's addresses:
D. Dato, Tiscali Italia S.p.A. – loc. Sa Illetta SS 195, Km 2,300 - 09123 Cagliari (Italy). E-mail: ddato@tiscali.com.
C. Lucchese, F. M. Nardini, R. Perego, N. Tonellotto, High Performance Computing Laboratory, ISTI–CNR – Via G. Moruzzi, 1 - 56124, Pisa (Italy). E-mail: {c.lucchese, f.nardini, r.perego, n.tonellotto}@isti.cnr.it.
S. Orlando, Dept. of Env. Science, Informatics, and Statistics – Ca' Foscari University of Venice – Via Torino, 155 – 30172, Venezia Mestre (Italy). E-mail: orlando@unive.it.
R. Venturini, Dept. of Computer Science, University of Pisa – Largo B. Pontecorvo, 3 - 56127 Pisa (Italy). e-mail: rossano.venturini@unipi.it.
This paper is an extension of [Lucchese et al. 2015]; it adds an additional scoring algorithm for ensembles of obvious trees, a blockwise version of the scoring algorithm, a new large-scale learning to rank dataset as well as results from experiments on this new dataset.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© YYYY ACM. 1046-8188/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Learning-to-Rank (LtR) has emerged in the last years as the most effective solution to the fundamental problem of ranking the query results returned by an Information Retrieval (IR) system [Liu 2009; Burges 2010]. A LtR algorithm learns a scoring function from a *ground-truth* set of training examples, basically a collection of queries $\mathcal{Q}$, where each query $q \in \mathcal{Q}$ is associated with a set of assessed documents $\mathcal{D} = \{d_0, d_1, \ldots\}$. Each query-document pair (q, d_i) is in turn represented by a set of numerical features, i.e., a set of heterogeneous, possibly highly discriminative, ranking signals. Moreover, each pair is labeled by a *relevance judgment* y_i, usually a positive integer in a fixed range, stating the degree of relevance of the document for the query. These labels induce a partial ordering over the assessed documents, thus defining their *ideal ranking* [Järvelin and Kekäläinen 2002]. The learned scoring function aims to approximate such *ideal ranking*.

The ranking process is particularly challenging for large-scale Web retrieval systems. Besides the demanding requirements for high quality results in response to user queries, Web retrieval systems have also to deal with strict efficiency constraints, which are not so common in other ranking-based applications. Indeed, two of the most effective LtR-based rankers are based on additive ensembles of regression trees, namely GRADIENT-BOOSTED REGRESSION TREES (GBRT) [Friedman 2001] and LAMBDA-MART (λ-MART) [Wu et al. 2010]. In the case of large-scale Web retrieval systems, where huge training sets are available and hundreds of features are used to represent query-document pairs, the best ranking quality is achieved with ensembles of (tens of) thousands regression trees. All the trees in these ensembles have to be traversed at scoring time for each candidate document, thus impacting on the response time and throughput of query processing. In order to limit this impact, LtR-based scorers are embedded in complex two-stage ranking architectures [Cambazoglu et al. 2010; Wang et al. 2011], thus preventing such expensive scorers from being applied to all the candidate documents possibly matching a user query. The first stage retrieves from the inverted index a relatively large set of possibly relevant documents matching the user query. This phase is aimed at optimizing the recall and is usually carried out by using a simple and fast ranking function, e.g., BM25 combined with some document-level scores [Robertson and Zaragoza 2009]. The expensive LtR-based scorers, optimized for high precision, are exploited in the second stage to *re-rank* the relatively smaller set of candidate documents coming from the first stage. In this two-stage architecture, the time budget available to re-rank the candidate documents is limited, due to the incoming rate of queries and the users' expectations in terms of response time. Therefore, devising techniques and strategies to speed up document ranking without losing in quality is definitely an urgent research topic in Web search [Viola and Jones 2004; Cambazoglu et al. 2010; Segalovich 2010; Ganjisaffar et al. 2011; Xu et al. 2012]. Moreover, also the ranking quality can benefit from speeding up the scoring process since, within the same time budget, more candidates selected by the first stage can be evaluated (i.e., higher recall) or more complex and ranking models exploited (i.e., higher precision).

Strongly motivated by similar considerations, the IR community has started to investigate low-level optimizations to reduce the scoring time of the most effective LtR rankers based on ensembles of regression trees, by dealing with advanced features of modern CPUs and memory hierarchies [Asadi et al. 2014; Tang et al. 2014]. Within this research line, we recently proposed QUICKSCORER (QS), a new algorithm to score documents with *ensembles of regression trees* [Lucchese et al. 2015]. The QS algorithm remarkably outperforms previous proposals thanks to a novel representation of the regression trees, allowing a fast interleaved traversal of the ensemble by using efficient

logical bitwise operations. The performance benefits of QS are unprecedented, due to a cache-aware approach, both in terms of data layout and access patterns, and to a program control flow that entails very low branch mis-prediction rates. In this paper we extend our previous work with the following novel and unpublished contributions:

— a variant of QUICKSCORER explicitly designed for ensembles of *oblivious trees*, i.e., balanced trees where, at each level, all the branching nodes test the same feature-threshold pair [Langley and Sage 1994]. Oblivious trees significantly change the scoring process, thus requiring deep adaptation of the QS algorithm. Nevertheless, ensemble of oblivious trees are a very effective and efficient ranking solution adopted, for example, by the largest Russian Web search engine Yandex [Segalovich 2010; Gulin et al. 2011];
— an optimized block-wise version of QS for scoring large ensembles named BLOCKWISE-QS (BWQS). BWQS splits the set of candidate documents and the set of tree ensemble in disjoint groups that entirely fit in cache memory. A new optimization technique is introduced that allows BWQS to restructure some blocks of trees in order to further reduce the number of operations performed at scoring time.
— a novel, very large, publicly available LtR dataset, that overcomes the limitations of previously available datasets, thus allowing researchers to conduct significant efficiency tests on both learning and scoring phases of LtR. The dataset contains 33,018 queries and 10,454,629 query-document pairs, while each query-document pair is represented by 220 numerical features and is labeled with relevance judgments ranging from 0 (irrelevant) to 4 (perfectly relevant). The dataset is made available by TISCALI ITALIA S.p.A, an Italian company running the web search engine ISTELLA (`http://www.istella.it`).
— an extensive experimental assessment conducted on the above and other publicly available LtR datasets, with several tree-based ranking models, based on both oblivious and non-oblivious trees. The tested models differ in the size of the ensembles and the maximum number of tree leaves.

The results of the new extensive experimental assessment conducted show that QS and BWQS achieve significant speedups over the best state-of-the-art competitors, ranging from 1.9x to 6.6x. In order to explain the unprecedented performance of QS, we evaluate and discuss in depth the results of low-level profiling that uses CPU counters to measure important performance events, such as number of instructions executed, cache-misses suffered, or branches mis-predicted.

The rest of the paper is structured as follows. Section 2 provides background information and discusses the related work, while Section 3 details the QS algorithm and its features. Section 4 discusses the implementation of a variant of the QS algorithm supporting ensembles of oblivious regression trees. Then, Section 5 details the experimental settings and reports on the results of our comprehensive evaluation. Section 6 proposes a cache-friedly block-wise variant of QS. Finally, we conclude our investigation in Section 7 by reporting some conclusions and suggestions for future research.

2. BACKGROUND AND RELATED WORK

GRADIENT-BOOSTED REGRESSION TREES (GBRT) [Friedman 2001] and LAMBDA-MART (λ-MART) [Wu et al. 2010] are two of the most effective LtR algorithms. The GBRT algorithm builds a model by approximating the root mean squared error on a given training set. This loss function makes GBRT a point-wise LtR algorithm: query-document pairs are exploited independently at learning time, and GBRT is trained to guess the relevance label associated with each of these pairs. The λ-MART algorithm improves over GBRT by optimizing list-wise IR measures like NDCG [Järvelin and Kekäläinen 2002], involving the whole list of documents associated with each query.

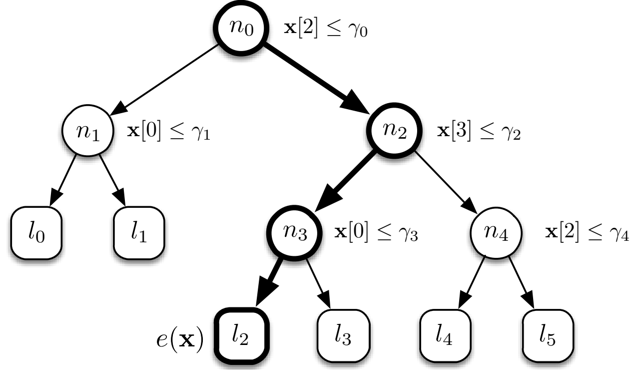

Fig. 1: A decision tree.

Thus, λ-MART aims at finding a scoring function that generates an ordering of documents as close as possible to the ideal ranking, even if the final guessed scores may differ from the relevance labels of the query-document pairs in the ground truth. At scoring times, there is no difference between λ-MART and GBRT, since they both generate a large ensemble of weighted regression trees. For both algorithms it is possible to impose some constraints on the structure of the weak tree learners to include in the ensemble. The most common one is to limit the maximum number of leaves of each tree. We also investigate a specific restriction on the structure of such trees, according to which these trees are called *oblivious* [Langley and Sage 1994], where all the nodes at the same level of each (balanced) tree must perform an identical Boolean test.

In this paper, we discuss algorithms and optimizations for scoring efficiently documents by means of ensembles of regression trees, both non-oblivious and oblivious one, such as those produced by λ-MART and GBRT. Indeed, the findings of this work apply beyond LtR, and in any application where large ensembles of decision trees are used for classification or regression tasks. In the following we introduce the adopted notation and discuss related works.

Ensembles of non-oblivious regression trees for document scoring

Each query-document pair (q, d_i) is represented by a real-valued vector $\mathbf{x}$ of *features*, namely $\mathbf{x} \in \mathbb{R}^{|\mathcal{F}|}$ where $\mathcal{F} = \{f_0, f_1, \ldots\}$ is the set of features characterising the document d_i and the user query q, and $\mathbf{x}[i]$ stores feature f_i. Let $\mathcal{T}$ be an ensemble of trees representing the ranking model. Each tree $T = (N, L)$ in $\mathcal{T}$ is a decision tree composed of a set of internal or branching nodes $N = \{n_0, n_1, \ldots\}$, and a set of leaves $L = \{l_0, l_1, \ldots\}$. Each $n \in N$ is associated with a *Boolean test* over a specific feature with id ϕ, i.e., $f_\phi \in \mathcal{F}$, and a constant threshold $\gamma \in \mathbb{R}$. This test is in the form $\mathbf{x}[\phi] \leq \gamma$. Each leaf $l \in L$ stores the *prediction* $l.\texttt{val} \in \mathbb{R}$, representing the potential contribution of tree T to the final score of the document.

For a given document $\mathbf{x}$, all the nodes whose Boolean conditions evaluate to FALSE are called *false nodes*, and *true nodes* otherwise. The scoring of a document requires the traversing of all the trees in the ensemble, starting from their root nodes. If a visited node is a false one, then the *right* branch is taken, and the *left* branch otherwise. The visit continues recursively until a leaf node is reached, where the value of the *prediction* is returned. Such leaf node is named *exit leaf* and denoted by $e(\mathbf{x}) \in L$. Therefore, given a vector $\mathbf{x}$, the contribution to the overall prediction by a tree $T \in \mathcal{T}$

is obtaining by tracing the unique path from the root of T to the exit leaf. We omit x when it is clear from the context.

Hereinafter, we assume that nodes of T are numbered in breadth-first order and leaves from left to right, and let ϕ_i and γ_i be the feature id and threshold associated with the i-th internal node, respectively. It is worth noting that the same feature can be involved in multiple nodes of the same tree. For example, in the tree shown in Figure 1, the features f_0 and f_2 are used twice. Assuming that x is such that $\mathbf{x}[2] > \gamma_0$, $\mathbf{x}[3] \leq \gamma_2$, and $\mathbf{x}[0] \leq \gamma_3$, the exit leaf $e(\mathbf{x})$ of the tree in Figure 1 is l_2.

The tree traversal process is repeated for all the trees of the ensemble $\mathcal{T}$, denoted by $\mathcal{T} = \{T_0, T_1, \ldots\}$. The score $s(\mathbf{x})$ of the whole ensemble is finally computed as a *weighted sum* over the contributions of each tree $T_h = (N_h, L_h)$ in $\mathcal{T}$ as:

$$s(\mathbf{x}) = \sum_{h=0}^{|\mathcal{T}|-1} w_h \cdot e_h(\mathbf{x}).\mathtt{val}$$

where $e_h(\mathbf{x}).\mathtt{val}$ is the predicted value of tree T_h, having weight $w_h \in \mathbb{R}$.

Efficient traversal of ensembles of non-oblivious regression trees.

A naïve implementation of the tree traversal may exploit a node data structure that stores the feature id, the threshold and the pointers to the left and right children nodes data structures. This method is enhanced with an optimized data layout by Asadi et al. [2014]. The resulting algorithm is named STRUCT+. This simple approach entails a number of issues. First, the next node to be processed is known only after the test is evaluated. As the next instruction to be executed is not known, this induces frequent *control hazards*, i.e., instruction dependencies introduced by conditional branches. As a consequence, the efficiency of a code strongly depends on the *branch mis-prediction rate* [Patterson and Hennessy 2014]. Finally, due to the unpredictability of the tree nodes visited for each scored document, the traversal has low temporal and spatial locality, generating low *cache hit ratio*. This is apparent when processing a large number of documents with a large ensemble of trees, since neither the documents nor the trees may fit in cache.

Another basic, but well performing approach is IF-THEN-ELSE. Each decision tree is translated into a complex structure of nested if-then-else blocks (e.g., in C++). The resulting code is compiled to generate an efficient document scorer. IF-THEN-ELSE aims at taking advantage of compiler optimization strategies, which can potentially re-arrange the tree ensemble traversal into a more efficient procedure. IF-THEN-ELSE was proven to be efficient with small feature sets [Asadi et al. 2014]. However, the algorithms suffers from *control hazards*, and the large size of the generated code may lead to poor performance of the instruction cache.

Asadi et al. [2014] proposed to rearrange the computation to transform *control hazards* into *data hazards*, i.e., data dependencies introduced when one instruction requires the result of another. To this end, node n_i of a tree stores, in addition to a feature id ϕ_i and a threshold γ_i, an array `idx` of two positions holding the addresses of the left and right children nodes data structures. Then, the output of the test $\mathbf{x}[\phi_i] > \gamma_i$ is directly used as an index of such array in order to retrieve the next node to be processed. The visit of a tree of depth d is then statically *un-rolled* in d operations, starting

from the root node n_0, as follows:

$$d \text{ steps} \begin{cases} i \leftarrow n_0.\texttt{idx}\left[\mathbf{x}[\phi_0] > \gamma_0\right] \\ i \leftarrow n_i.\texttt{idx}\left[\mathbf{x}[\phi_i] > \gamma_i\right] \\ \vdots \qquad\qquad \vdots \\ i \leftarrow n_i.\texttt{idx}\left[\mathbf{x}[\phi_i] > \gamma_i\right] \end{cases}$$

Leaf nodes are encoded so that the indexes in idx generate self loops, with dummy ϕ_i and γ_i. At the end of the visit, the exit leaf is identified by variable i, and a look-up table is used to retrieve the prediction of the tree.

This approach, named PRED, removes control hazards as the next instruction to be executed is always known. On the other hand, data dependencies are not solved as the output of one instruction is required to execute the subsequent. Memory access patterns are not improved either, as they depend on the path along the tree traversed by a document. Finally, PRED introduces a new source of overhead: for a tree of depth d, even if a document reaches a leaf early, the above d steps are executed anyway.

To reduce *data hazards* the same authors proposed a *vectorized* version of the scoring algorithm, named VPRED, by interleaving the evaluation of a small set of documents (16 was the best setting). VPRED was shown to be 25% to 70% faster than PRED on synthetic data, and to outperform other approaches. The same approach of PRED was also adopted in some previous works exploiting GPUs [Sharp 2008], and a more recent survey evaluates the trade-off among multi-core CPUs, GPUs and FPGA [Van Essen et al. 2012].

In this work we compare against VPRED which can be considered the best performing algorithm at the state of the art. In the experimental section, we show that the proposed QS algorithm has reduced *control hazards*, smaller *branch mis-prediction* rates and better *memory access patterns* than VPRED.

Memory latency issues of scoring algorithms are tackled and analyzed by Tang et al. [2014] and by Jin et al. [2016]. In most cases, the cache memory may be insufficient to store the candidate documents to be scored or the set of regression trees. The authors propose a cache-conscious optimization by splitting documents and regression trees in *blocks*, such that one block of documents and one block of trees can both be stored in cache at the same time. The scoring process thus requires to evaluate all the tree blocks against all the document blocks. The authors applied this computational scheme on top of both IF-THEN-ELSE and PRED, with an average improvement of about 28% and 24% respectively. The blocking technique is indeed very general and can be used by all algorithms. The same computational schema is applied in this work to QS in order to improve the *cache hit ratio* when large ensembles are used.

Oblivious regression trees

In order to avoid overfitting, some constraints can be enforced on the structure of the learned regression trees. For example, boosting approaches limit the depth of each tree in the ensemble, or the number of tree leaves [Chen and Guestrin 2016]. Other methods enforce the read-once property [Kohavi 1994], according to which each feature can be tested only once along each path from the root to a leaf. Another popular constraint enforces the learned trees of an ensemble to be *oblivious* [Langley and Sage 1994], i.e., the trees must be balanced and all branching nodes at the same level of each tree have to perform the same test.

In this paper, besides the ensembles of non-oblivious decision trees, we also investigate ensembles of oblivious trees. Ranking models based on such trees are used by Yandex within their MATRIXNET LtR tool [Segalovich 2010], which was successfully

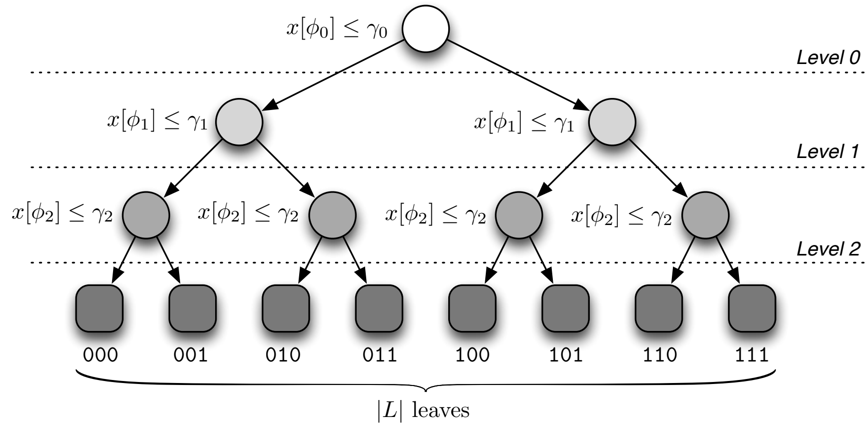

Fig. 2: An oblivious decision tree.

exploited during the Yahoo LtR Challenge [Gulin et al. 2011]. Formally, let $T = (N, L)$ be an oblivious tree of ensemble $\mathcal{T}$, where T is balanced and has depth $d = \log_2 |L|$. Given a generic level i of the tree, let $\mathrm{x}[\phi_i] \leq \gamma_i$ be the test performed by all the internal nodes at level i, where ϕ_i is the index of the feature and γ_i the corresponding threshold. Therefore, the number of distinct tests occurring in the branching nodes N in T are exactly d, and the tree can be traversed on the basis of the outcomes of these d tests only.

Figure 2 illustrates an example of oblivious tree of depth 3 ($d = \log_2 |L| = 3$, and thus $|L| = 2^d = 8$). where the leaf identifiers are binary encoded from left to right, in the range [000, 111].

We are interested in oblivious trees because the ensemble models based on these trees can be evaluated very efficiently at scoring time. Indeed, we implemented a very efficient algorithm as baseline for traversing such ensembles, called OBLIVIOUS. Despite the efficiency of OBLIVIOUS, however, we show that a variant of QS, namely QS_Ω, specifically tailored for these ensembles, outperforms OBLIVIOUS.

OBLIVIOUS takes advantage of the property that all the branching nodes at the same depth of an oblivious tree perform the same test. Even if at scoring time we can traverse an oblivious tree as a non-oblivious ones, i.e., by following for each input feature vector x a unique path from the root to the exit leaf, OBLIVIOUS does not materialize and traverse the oblivious trees, thus avoiding conditional branches. Specifically, the outcomes of the d tests are used to directly determine the binary encoding of the identifier of the exit leaf.

For details on OBLIVIOUS and QS_Ω, the reader is referred to Section 4.

Other approaches and optimizations

Unlike our QS and QS_Ω algorithms that aim to devise an efficient strategy for evaluating all the ensemble of trees, other approaches try to approximate the final ranking to reduce the scoring time.

Indeed, Cambazoglu et al. [2010] proposed to early terminate the tree-based scoring of documents that are unlikely to be ranked within the top-k results. Their work, which applies to an ensemble of additive trees, saves scoring time by reducing the number of tree traversals, and thus trades better efficiency for little loss in raking quality. Although QS is thought for globally optimizing the traversal of thousands of trees, the

idea of early termination can be applied as well along with our method, by evaluating some proper exit strategy after the evaluation of some subsets of the regression trees.

Wang *et al.* [Wang et al. 2010; Wang et al. 2010; Wang et al. 2011] deeply investigated different efficiency aspects of the ranking pipeline. In particular, Wang et al. [2011] proposes a novel cascade ranking model, which unlike previous approaches, can simultaneously improve both top-k ranked effectiveness and retrieval efficiency. Their work is mainly related to the tuning of a two-stage ranking pipeline.

A different approach is to modify the learning process to build more efficient trees by observing that hallow and balanced trees are likely to be evaluated faster [Asadi and Lin 2013]. Asadi and Lin [2013] modify the training phase with different split criterion and pruning strategy which provide up to a 40% reduction of the scoring cost. Post-learning strategies aimed at simplifying a given tree ensemble were proposed by Lucchese et al. [2016]. They show that it is possible to achieve a speed-up of a factor 2 at scoring time by removing some of the trees and tuning the weights of the remaining trees. We note that all these approaches could be used in conjunction with QS and QS_Ω.

3. QUICKSCORER: EFFICIENT TRAVERSAL OF ENSEMBLES OF NON-OBLIVIOUS DECISION TREES

The QUICKSCORER (QS) algorithm exploits a totally novel approach for the traversal of a given tree ensemble. The bitvector-based representation (see Subsection 3.1) and the data layout adopted (see Subsection 3.2) permit an efficient exploitation of memory hierarchies and a reduced branch mis-prediction rate. Given a document and a tree, our traversal method evaluates the branching nodes of the tree, and produces a bitvector which encodes the exit leaf. In isolation this traversal is not advantageous, since in principle it requires to evaluate all the nodes of a tree. However, it has the nice property of being insensitive to the order in which the nodes are processed. This makes it possible to interleave the evaluation of the various trees in the ensemble in a *cache-aware* fashion, and to save the computation of several test conditions. Rather than traversing the ensemble by taking a tree at the time, QS performs a global visit of the ensemble by traversing portions of all the trees together, feature by feature. For each feature, we store all the associated thresholds occurring anywhere in the ensemble in a sorted array, to easily compute the result of all the test conditions involved. A bitvector for each tree is updated after each test, in such a way to encode, at the end of the process, the exit leaves of each tree for the given document. These bitvectors are eventually used to retrieve the contribution of each tree to the final score of the document.

3.1. Tree traversal using bitvectors

We first present a simple version of the proposed tree traversal, and then we introduce two crucial refinements for the performance of this algorithm when used in the interleaved evaluation of the full ensemble as described in Subsection 3.2.

Given an input feature vector x and a tree $T_h = (N_h, L_h)$, the proposed tree traversal algorithm processes the internal nodes of T_h with the goal of identifying a set of *candidate exit leaves*, denoted by $\mathcal{C}_h$, $\mathcal{C}_h \subseteq L_h$. Initially $\mathcal{C}_h$ contains all the leaves in L_h, i.e., $\mathcal{C}_h = L_h$. Then, the algorithm evaluates one after the other, in arbitrary order, the test conditions of all the internal nodes of T_h. When an internal node $n \in N_h$ is considered, the algorithm removes from $\mathcal{C}_h$ those leaves that cannot be anymore reached during the evaluation of x according to the outcome of the node's test. Indeed, if n is a *false node* (i.e., its test condition is false), the leaves in the left subtree of n cannot be the exit leaf and they can be safely removed from $\mathcal{C}_h$. Similarly, if n is a *true node*, the

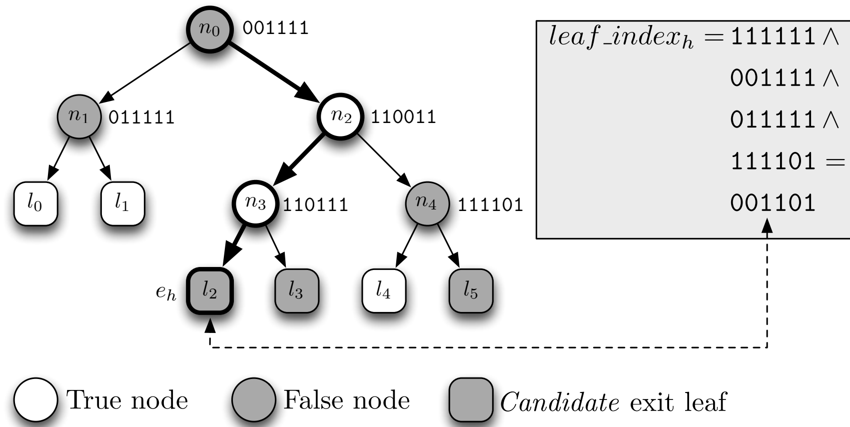

Fig. 3: Tree traversal example.

leaves in the right subtree of n can be removed from $\mathcal{C}_h$. Once all the nodes have been processed, the only leaf left in $\mathcal{C}_h$ is the exit leaf e_h.

The first important refinement turns the above algorithm into a lazy one. This lazy algorithm processes only the tree's *false nodes*, which we assume are provided by an oracle called `FindFalse`. Thus, the algorithm removes from $\mathcal{C}_h$ the leaves in the left subtrees of all the false nodes returned by the oracle. In the following, we focus on the set $\mathcal{C}_h$ resulting from the processing of all the false nodes of a tree and we defer the materialization of the above oracle to Subsection 3.2, where the interleaved evaluation of all the trees makes its implementation feasible.

Observe that the set $\mathcal{C}_h$ may in principle contain several leaves. As an extreme example, in absence of false nodes in T_h, $\mathcal{C}_h$ will contain all the leaves in L_h. Interestingly, we can prove (see Theorem 3.1 below) that the *exit leaf* e_h is always the one associated with the smallest identifier in $\mathcal{C}_h$, i.e., the leftmost leaf in the tree. A running example is reported in Figure 3 which shows the actual traversal (bold arrows) for a vector $\mathbf{x}$, and the true and false nodes. The figure shows also the set $\mathcal{C}_h$ after the removal of the leaves of the left subtrees of false nodes: $\mathcal{C}_h$ is $\{l_2, l_3, l_5\}$ and, indeed, the exit leaf is the leftmost leaf in $\mathcal{C}_h$, i.e., $e_h = l_2$.

The second refinement implements the operations on $\mathcal{C}_h$ with fast bit-wise operations. The idea is to represent $\mathcal{C}_h$ with a bitvector $leaf_index_h$, where each bit corresponds to a distinct leaf in L_h, i.e., $leaf_index_h$ is the characteristic vector of $\mathcal{C}_h$. Moreover, every internal node n is associated with a *bit mask* of the same length encoding (with 0's) the set of leaves to be removed from $\mathcal{C}_h$ whenever n turns to be a false node. In this way, the bitwise logical AND between $leaf_index_h$ and the bit mask of a false node n corresponds to the removal of the leaves in the left subtree of n from $\mathcal{C}_h$. We finally observe that, once identified all the false nodes in a tree and performed the associated AND operations over $leaf_index_h$, the exit leaf of the tree corresponds to the leftmost bit set to 1 in $leaf_index_h$. Figure 3 shows how the initial bitvector $leaf_index_h$ is updated by using bitwise logical AND operations.

The full approach is described in Algorithm 1. Given a binary tree $T_h = (N_h, L_h)$ and an input feature vector $\mathbf{x}$, let $n.\texttt{mask}$ be the precomputed bit mask associated with a generic $n \in N_h$. First the result bitvector $leaf_index_h$ is initialized with all bits set to 1. Then, $\texttt{FindFalse}(\mathbf{x}, T_h)$ returns all the false nodes in N_h. For each of such nodes,

$leaf_index_h$ is masked with the corresponding node mask. Finally, the position of the leftmost bit of $leaf_index_h$ identifies the exit leaf e_h, whose output value is returned. The correctness of this approach is stated by the following theorem.

THEOREM 3.1. *Algorithm 1 identifies the correct exit leaf* e_h *for every binary decision tree* T_h *and input feature vector* x.

PROOF. First, we prove that the bit corresponding to the exit leaf e_h in $leaf_index_h$ is always set to 1. Consider the internal nodes along the path from the root to e_h, and observe that only the bit masks applied for those nodes may change the e_h's bit to 0. Since e_h is the exit leaf, it belongs to the left subtree of any true node and to the right subtree of any false node in this path. Thus, since the masks are used to set to 0 leaves in the left subtrees of false nodes, the bit corresponding to e_h remains unmodified, and, thus, will be set to 1 at the end of Algorithm 1.

Second, we prove that the leftmost bit equal to 1 in $\mathbf{v}_h$ corresponds to the exit leaf e_h. Let $l_\leftarrow$ be the leaf corresponding to the leftmost bit set to 1 in $leaf_index_h$. Assume by contradiction that e_h is not the leftmost bit set to 1 in $leaf_index_h$, namely, $l_\leftarrow \neq e_h$. Let u be their lowest common ancestor node in the tree. Since $l_\leftarrow$ is smaller than e_h, the leaf $l_\leftarrow$ belongs to u's left subtree while the leaf e_h belongs to u's right subtree. This leads to a contradiction. Indeed, on one hand, the node u should be a true node otherwise its bit mask would have been applied setting $l_\leftarrow$'s bit to 0. On the other hand, the node u should be a false node since e_h is in its right subtree. Thus, we conclude that $l_\leftarrow = e_h$ proving the correctness of Algorithm 1. □

Algorithm 1 represents a general technique to compute the output value of a single binary decision tree stored as a set of precomputed bit masks. Given an additive ensemble of binary decision trees, to score a document x we have to loop over all the trees $T_h \in \mathcal{T}$ by repeatedly applying Algorithm 1. Unfortunately, this naïve algorithm is too expensive unless $\texttt{FindFalse}(\mathrm{x}, T_h)$ can be implemented efficiently. In the following section we present QS, which overcomes this issue and is able to identify efficiently the false nodes in the tree ensemble by exploiting an *interleaved* evaluation of all its trees.

ALGORITHM 1: Scoring a feature vector x using a binary decision tree T_h

Input :
— x: input feature vector
— $T_h = (N_h, L_h)$: binary decision tree, with
- $N_h = \{n_0, n_1, \ldots\}$: internal nodes of T_h
- $L_h = \{l_0, l_1, \ldots\}$: leaves of T_h
- n.mask: node bit mask associated with $n \in N_h$
- l_j.val: score contribution associated with leaf $l_j \in L_h$

Output:
— tree traversal output value

```
Score(x,T_h):
1   leaf_index_h ← 11...11
2   U ← FindFalse(x, T_h)
3   foreach node n ∈ U do
4       leaf_index_h ← leaf_index_h ∧ n.mask
5   j ← index of leftmost bit set to 1 of leaf_index_h
6   return l_j.val
```

3.2. The QS Algorithm

Our QS algorithm scores a feature vector x with an interleaved execution of several tree traversals. The algorithm does not loop over all the trees in $\mathcal{T}$ one at a time, as one would expect, but loops instead over all the features in $\mathcal{F}$, hence incrementally discovering for each $f_k \in \mathcal{F}$ the false nodes involving f_k in any tree of the ensemble. This is a very convenient strategy for two reasons: i) we are able to simply identify all the false nodes for all the trees, thus effectively implementing the oracle introduced in the previous section; ii) we are able to operate in a cache-aware fashion with a small number of Boolean comparisons and branch mis-predictions.

During its execution, QS has to maintain the bitvectors $leaf_index_h$'s, encoding the set $\mathcal{C}_h$'s for all the tree T_h in the ensemble. The content of $leaf_index_h$ of a certain tree is updated each time that a false node for that tree is identified. Once the algorithm has processed all the features in $\mathcal{F}$, each of these $leaf_index_h$ is guaranteed to encode the exit leaf of the corresponding tree. Finally, the algorithm computes the overall score of x by summing up (and, possibly, weighting) the scores associated with all these exit leaves.

Let us focus on the processing of a feature f_k and describe the portion of the data structure of interest for this feature. The overall algorithm simply iterates this process over all features in $\mathcal{F}$. Each node involving f_k in any tree $T_h \in \mathcal{T}$ is represented by a *triple* containing: (i) the feature threshold involved in the Boolean test; (ii) the id of the tree that contains the node, where the id is used to identify the bitvector $leaf_index_h$ to update; (iii) the node mask used to possibly update $leaf_index_h$. We sort these triples in *ascending order* of the feature thresholds.

This sorting is crucial for obtaining a fast implementation of the $\texttt{FindFalse}(\mathbf{x}, T_h)$ oracle. Recall that all the test conditions occurring at the internal nodes of the trees are all of the form $\mathbf{x}[k] \leq \gamma_s^h$. Hence, given the sorted list of all the thresholds involving $f_k \in \mathcal{F}$, the feature value $\mathbf{x}[k]$ splits the list in two, possibly empty, sublists. The first sublist contains all the thresholds γ_s^h for which the test condition $\mathbf{x}[k] \leq \gamma_s^h$ evaluates to FALSE, while the second sublists contains all thresholds for which the test condition evaluates to TRUE. Thus, if we sequentially scan the sorted list of the thresholds associated with f_k, all the values in the first sublist will cause negative tests. Associated with these thresholds entailing false tests, we have false nodes belonging to the trees in $\mathcal{T}$. Therefore, for all these false nodes we can take in sequence the corresponding bit masks, and perform a bitwise logical AND with the appropriate bitvector $leaf_index_h$. This large sequence of tests that evaluates to FALSE corresponds to the repeated execution of conditional branch instructions, whose behavior is indeed very predictable. This is confirmed by our experimental results, showing that our code incurs in very few branch mis-predictions.

We now present the layout in memory of the required data structure since it is crucial for the efficiency of our algorithm. The triples of each feature are stored in three separate arrays, one for each component: `thresholds`, `treeids`, and `nodemasks`. The use of three distinct arrays solves some data alignment issues arising when tuples of heterogeneous data types are stored contiguously in memory. The arrays of the different features are then juxtaposed one after the other as illustrated in Figure 4. Since arrays of different features may have different lengths, we use an auxiliary array `offsets` which marks the starting position of each array in the global array. We also juxtapose the bitvectors $leaf_index_h$ into a global array `leafindexes`. Finally, we use an array `leafvalues` which stores the output values of the leaves of each tree (ordered from left to right) grouped by their tree id.

Algorithm 2 reports the steps of QS as informally described above. After the initialization of the result bitvectors of each tree (loop starting at line 1), the algorithm iter-

ates over all features (loop starting at line 3), and inspects the sorted lists of thresholds to identify the false nodes in the ensemble (line 6) and to update the result bitvectors (line 8). The second step of the algorithm (loop starting at line 13) consists in inspecting all the result bitvectors. For each of them QS identifies the position of the leftmost bit set to 1, and uses this position to retrieve the score contribution associated with the corresponding leaf stored in array `leafvalues`. This value is finally used to update the final score.

An example of an ensemble of regression trees, with the associated data structures as used by the QS algorithm, are shown and discussed at the end of this section.

Implementation details. In the following we discuss some details about our data structures, their size and access modes.

A few important remarks concern the bitvectors stored in `leafindexes` and `nodemasks`. The learning algorithm controls the accuracy of each single tree with a parameter Λ, which determines the maximal number of leaves for each $T_h = (N_h, L_h)$ in $\mathcal{T}$, namely $|L_h| \leq \Lambda$. Usually, the value of Λ is kept small (≤ 64). Thus, the length of bitvectors, which have to encode tree leaves, is equal to (or less than) a typical machine word of modern CPUs (64 bits). As a consequence, the bitwise operations performed by

ALGORITHM 2: The QUICKSCORER Algorithm

Input :
- x: input feature vector
- $\mathcal{T}$: ensemble of binary decision trees, with
  - $w_0, \ldots, w_{|\mathcal{T}|-1}$: weights, one per tree
  - `thresholds`: sorted sublists of thresholds, one sublist per feature
  - `treeids`: tree's ids, one per node/threshold
  - `nodemasks`: node bit masks, one per node/threshold
  - `offsets`: offsets of the blocks of triples
  - `leafindexes`: result bitvectors, one per each tree
  - `leafvalues`: score contributions, one per each tree leaf

Output:
- Final score of x

```
QUICKSCORER(x,T):
1   foreach h ∈ 0, 1, ..., |T| − 1 do
2       leafindexes[h] ← 11...11
3   foreach k ∈ 0, 1, ..., |F| − 1 do                              // Step ①
4       i ← offsets[k]
5       end ← offsets[k + 1]
6       while x[k] > thresholds[i] do
7           h ← treeids[i]
8           leafindexes[h] ← leafindexes[h] ∧ nodemasks[i]
9           i ← i + 1
10          if i ≥ end then
11              break
12  score ← 0
13  foreach h ∈ 0, 1, ..., |T| − 1 do                              // Step ②
14      j ← index of leftmost bit set to 1 of leafindexes[h]
15      l ← h · |L_h| + j
16      score ← score + w_h · leafvalues[l]
17  return score
```

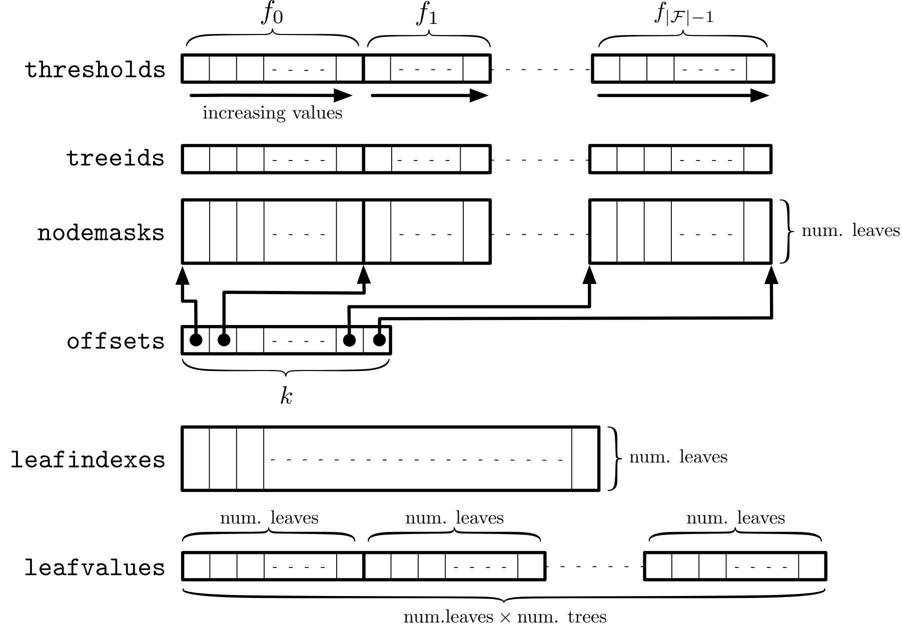

Fig. 4: QUICKSCORER data structure layout.

Algorithm 2 on them can be realized very efficiently, because they involve machine words (or halfwords, etc).

We avoid any possible performance overhead due to shifting operations to align the operands of bitwise logical ANDs by forcing the bitvectors to have uniform length of B bytes. To this end, we pad each bitvector on its right side with a string of 0 bits, if necessary. We always select the minimum number of bytes $B \in \{1, 2, 4, 8\}$ fitting Λ.

Let us now consider Table I, which shows an upper bound for the size of each linear array used by our algorithm. The array `offsets` has $|\mathcal{F}|$ entries, one entry for each distinct feature. The array `leafindexes`, instead, has an entry for each tree in $\mathcal{T}$, thus, $|\mathcal{T}|$ entries overall. The sizes of the other data structures depends on the number of total internal nodes or leaves in the ensemble $\mathcal{T}$, besides the datatype sizes. Any internal node of some tree of $\mathcal{T}$ contributes with an entry in each array `thresholds`, `nodemasks` and `treeids`. Therefore the total number of entries of each of these arrays, i.e., $\sum_0^{|\mathcal{T}|-1} |N_h|$, can be upper bounded by $|\mathcal{T}| \cdot \Lambda$, because for every tree T_h we have $|N_h| < |N_h| + 1 = |L_h| \leq \Lambda$. Finally, the array `leafvalues` has an entry for each leaf in a tree of $\mathcal{T}$, hence, no more than $|\mathcal{T}| \cdot \Lambda$ in total.

Table I: QUICKSCORER data structures size and access mode.

Data structure	*Maximum Size (bytes)*	*Access pattern*

`thresholds`
`treeids`
`nodemasks`
`offsets`
	$\lvert\mathcal{T}\rvert \cdot \Lambda \cdot \texttt{sizeof(float)}$
	$\lvert\mathcal{T}\rvert \cdot \Lambda \cdot \texttt{sizeof(uint)}$
	$\lvert\mathcal{T}\rvert \cdot \Lambda \cdot B$
	$\lvert\mathcal{F}\rvert \cdot \texttt{sizeof(uint)}$
		1. Sequential (R)

`leafindexes`	$\lvert\mathcal{T}\rvert \cdot B$	*1. Random (R/W)* *2. Sequential (R)*
`leafvalues`	$\lvert\mathcal{T}\rvert \cdot \Lambda \cdot \texttt{sizeof(double)}$	*2. Seq. Sparse (R)*

The last column of Table I reports the *access patterns* to the arrays, where the leading number, either 1 or 2, corresponds to the step of the algorithm during which the data structures are read/written. We first note that `leafindexes` is the only array used in both phases. During the first step `leafindexes` is accessed randomly in reading/writing to update bitvectors $leaf_index_h$. During the second step the same array is accessed sequentially in reading mode to identify the exit leafs l_h of each tree T_h, and then to access the array `leafvalues` to read the contribution of tree T_h to the document score. Even if the trees and their leaves are accessed sequentially during the second step of QS, the reading access to array `leafvalues` is sequential, but very sparse: only one leaf of each block of $|L_h|$ elements is actually read.

Finally, note that the arrays storing the triples, i.e., `thresholds`, `treeids`, and `nodemasks`, are all sequentially read during the first step, though not completely, since for each feature we stop their inspection at the first test condition that evaluates to TRUE. The cache usage can greatly benefit from the layout and access modes of our data structures, thanks to the increased references locality.

We finally describe an optimization which aims at reducing the number of comparisons performed at line 6 of Algorithm 2. This inner loop iterates over the list of threshold values associated with a certain feature $f_k \in \mathcal{F}$ until the first index j such that $\mathrm{x}[k] \leq \mathtt{thresholds}[j]$ is found. Instead of testing each threshold, our optimized implementation tests only one every Δ thresholds, where Δ is a parameter. Since the `thresholds` vector is sorted in ascending order, if the ith thresholds corresponds to a false node, the same holds for any $k < i$. Therefore, we can save $\Delta - 1$ comparisons and directly updating the Δ bitvectors $leaf_index_h$. Instead, if the test fails, the preceding $\Delta - 1$ thresholds are processed as usual and the algorithm completes. We found empirically that $\Delta = 4$ provides the best results. We remark that even if it is possible to find the first true node via binary search, we have experienced that this is less efficient because the array `thresholds` is not sufficiently large to counterbalance the poor cache locality.

Scoring Example. An example of a tree ensemble $\mathcal{T}$, with the associated data structures as used by the QS algorithm, are shown and discussed in the following. The ensemble $\mathcal{T}$, which is illustrated in Figure 5, only includes the two trees T_0 and T_1. We assume that the ranking model of $\mathcal{T}$ was learned from a training dataset where each query-document pair is represented by a feature vector x with only three features, namely f_0, f_1, and f_2.

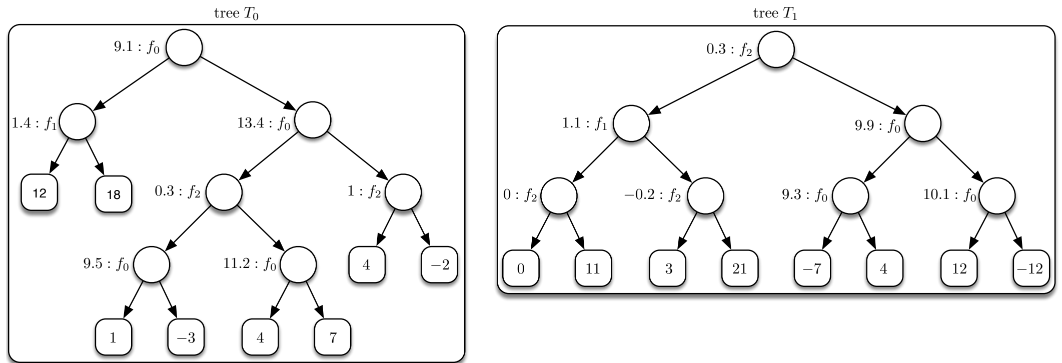

Fig. 5: An example of an ensemble of regression trees.

All the internal nodes of the two regression trees are labeled (see Figure 5) with a pair (f_ϕ, γ), with $f_\phi \in \{f_0, f_1, f_2\}$ and $\gamma \in \mathbb{R}$, specifying the pair of parameters of the *Boolean test* $\mathrm{x}[\phi] \leq \gamma$. The leaves of the two trees store a value representing the potential contribution of the tree to the final score of the document.

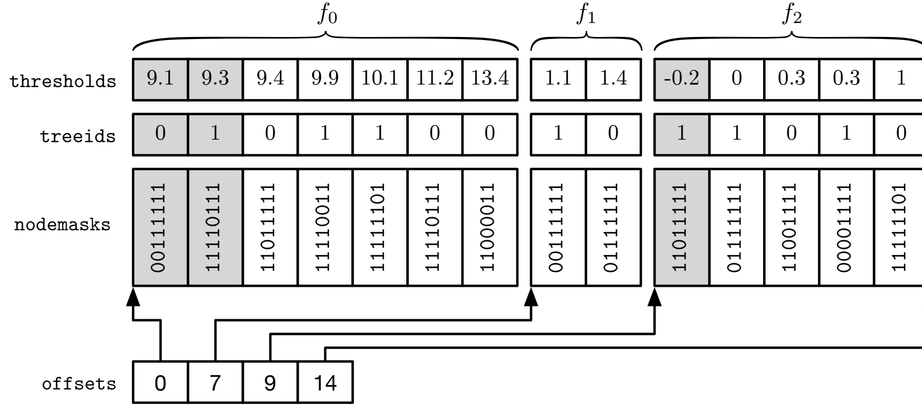

Fig. 6: QS representation of the toy ranking model.

Given this simple ranking model, QS compactly represents the ensamble $\mathcal{T}$ with the array data structures shown in Figure 6. We highlight that:

— array `thresholds` has 14 elements storing the values of 7, 2, and 5 thresholds γ associated, respectively, with the occurrences of the features f_0, f_1, and f_2 in the internal nodes of $\mathcal{T}$. We note that each block of thresholds is sorted in increasing order. Moreover, the first position of the ordered sequence of thresholds associated with a given feature f_ϕ can be accessed directly by using the corresponding offset value stored in array `offsets[`ϕ`]`.

— array `treeids` is aligned to array `thresholds`. Specifically, given the ϕ_{th} block of each array corresponding to feature f_ϕ, let `i` be an index used to identify the current element of the block. Thus, `i` ranges in the integer interval [`offsets[`ϕ`]`, `offsets[`ϕ`+1]-1`], and for each value of `i` the entry `treeids[i]` stores the ID of the tree, in turn containing a specific internal node with threshold `thresholds[i]`. For example, from Figure 6 we can see that a value 9.9 is stored in the 4-th position (i.e., element `thresholds[3]`) to indicate that this value is a threshold used for feature f_0 in the tree with ID `treeids[3]`$= 1$.

— the array `nodemasks` is also aligned to `thresholds` (and `treeids`). Specifically, it stores in each position a bitvector of size equal to the (maximum) number of leaves of the trees in $\mathcal{T}$ (8 in this case). The bits in these bitvectors are set to 0 in correspondence to the leaves of the tree that are not reachable if the associated test fails. For example, `nodemasks[3]` stores 11110011, stating that the 5-th and the 6-th leaves of tree T_1 (`treeids[3]`$= 1$) cannot be reached by documents for which the test $\mathrm{x}[0] \leq 9.9$ (`thresholds[3]`$= 9.9$) is FALSE.

Finally, Figure 7 shows how the bitvectors selected by the QS algorithm are used to devise the correct exit leaf of each tree. The Figure shows the feature vector x of a document to be scored. The bitvectors `leafindexes[0]` and `leafindexes[1]` are initialized with a string of 1's, whose length corresponds to the number of tree leaves (8 in this example). By visiting the ensemble $\mathcal{T}$ feature by feature, QS starts from

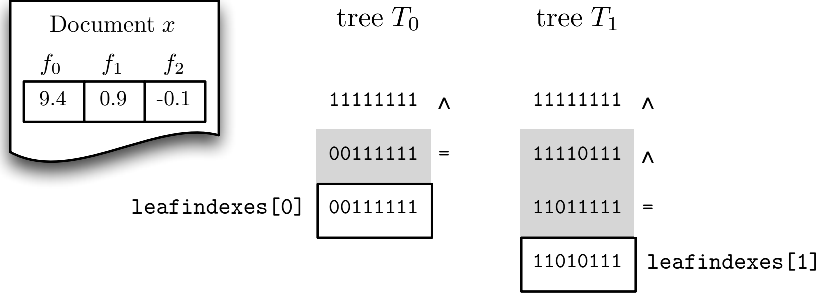

Fig. 7: Scoring of a document.

the first feature f_0, by inspecting $\mathbf{x}[0]$. The algorithm thus accesses the list of thresholds of the feature starting from `thresholds[offsets[0]]`, where `offsets[0]`$= 0$. QS first detects that the first two tests involving feature $\mathbf{x}[0] = 9.4$ fail, since $9.4 > 9.1$ (`thresholds[0]`$= 9.1$) and $9.4 > 9.3$ (`thresholds[1]`$= 9.3$) hold. Thus, the two bitvectors 00111111 and 11110111, associated with the trees having respectively IDs `treeids[0]`$=$ 0 and `treeids[1]`$= 1$, are retrieved. Then, a bitwise AND operation ($\wedge$) is performed between these bitvectors and the ones stored in `leafvalues[0]` and `leafvalues[1]`. Afterwards, since $9.4 \leq 9.4$ succeeds, feature f_0 is considered totally processed, and QS continues with the next feature f_1, by inspecting $\mathbf{x}[1] = 0.9$. The lists of thresholds for feature f_1 is accessed starting from `thresholds[offsets[1]]`, where `offsets[1]`$= 7$. Since $0.9 \leq 1.1$ (`thresholds[7]`$= 1.1$), the test succeeds, and thus the remaining elements of the threshold list associated with feature f_1 are skipped. Finally, the last feature f_2, namely $\mathbf{x}[2]$, is considered and compared with the first threshold stored in `thresholds[offsets[2]]`, where `offsets[2]`$= 9$. The first test involving $\mathbf{x}[2] = -0.1$, namely $-0.1 \leq -0.2$ (`thresholds[9]`$= -0.2$) fails. Since `treeids[9]`$= 1$, a bitwise AND operation is thus performed between `nodemasks[9]` and `leafindexes[1]`. At this point, the next test over $\mathbf{x}[2]$ succeeds, and thus QS finishes the ensemble traversal. The content of the bitvectors `leafindexes[0]` and `leafindexes[1]` is finally used to directly read from array `leafvalues` the contribution of trees T_0 and T_1 to the final score of the document.

4. FAST TRAVERSAL OF ENSEMBLES OF OBLIVIOUS TREES

In this section, we investigate the feasibility of QUICKSCORER for also scoring with ensembles of *oblivious decision trees*. These trees are balanced, and all the branching nodes at depth i of the tree implement the same Boolean test, which uses the same feature ϕ_i and the same threshold γ_i. Large ensembles of oblivious trees are used for ranking in the largest Russian Web search engine Yandex [Segalovich 2010; Gulin et al. 2011]. In addition, since ensembles of oblivious trees are less prone to overfitting, they perform very well in terms of effectiveness, sometimes outperforming GBRT and λ-MART [Capannini et al. 2016]. Moreover, they are very relevant from our point of view, as their regular structure makes it possible to implement a very efficient algorithm, named OBLIVIOUS, which traverses each tree without conditional branches. The OBLIVIOUS algorithm is used as baseline for our QS_Ω version of QUICKSCORER, specifically tailored for scoring with ensembles of oblivious trees.

Figure 8 illustrates an example of oblivious tree, namely a tree T_h in the ensemble $\mathcal{T}$, and the simple binary vector `leafID` (on the left side), used by OBLIVIOUS to identity

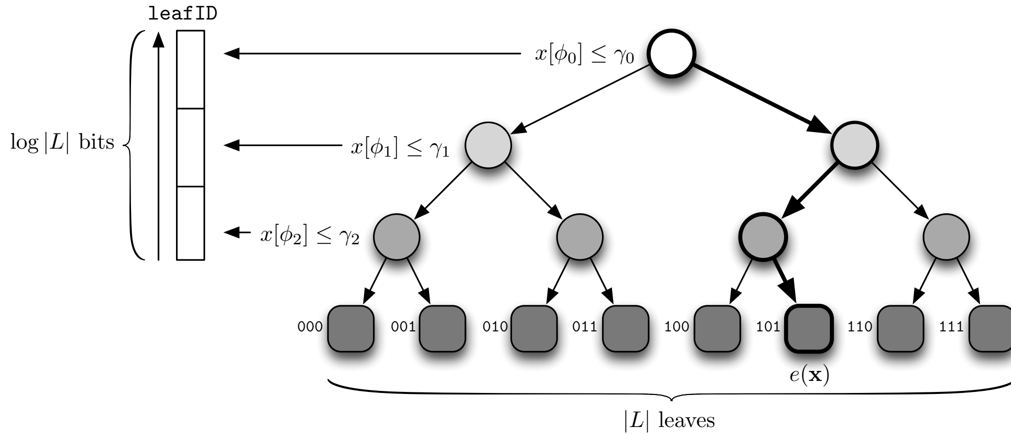

Fig. 8: Optimized traversal of an oblivious tree performed by OBLIVIOUS.

the exit leaf of the tree. Note the binary encoding of the leaves identifiers, which are numbered from left to right, corresponding to the binary indices from `000` to `111`. This encoding trivially reflects the path from the root to the exit leaf traversed during the evaluation of a document, where 0 stands for a *true node* and 1 for a *false node*. For example, supposing that the exit leaf for a given feature vector x, namely $e(\mathbf{x})$, is the one identified by `101`, and thus the traversal path is the one illustrated by bold arrows. This means that the two tests a levels 0 and 2 turn out to be false, while the test at level 1 is true. As a consequence of these test, the OBLIVIOUS algorithm sets/unsets the corresponding bits of `leafID`, which eventually is used to identify the exit leaf of the tree.

This traversal process can be easily generalized to any oblivious tree with $|L|$ leaves and depth $\log(|L|)$. Given a tree and a feature vector x to be scored, the sequence of $\log(|L|)$ steps, needed to compute the value of `leafID` that identifies the exit leaf, is illustrated in the following:

$$d = \log(|L|) \text{ steps} \begin{cases} \texttt{leafID} \leftarrow 0 \ \vee \ \neg\,(\mathbf{x}[\phi_0] \leq \gamma_0) \ll (d-1) \\ \texttt{leafID} \leftarrow \texttt{leafID} \ \vee \ \neg\,(\mathbf{x}[\phi_1] \leq \gamma_1) \ll (d-2) \\ \vdots \qquad\qquad \vdots \\ \texttt{leafID} \leftarrow \texttt{leafID} \ \vee \ \neg\,(\mathbf{x}[\phi_{d-1}] \leq \gamma_{d-1}) \end{cases} \tag{1}$$

where $\neg$, $\vee$ and $\ll$ are the bitwise logical *negation*, *or*, and *left shift* operations, and `leafID` eventually contains the index of the exit leaf for vector x.

The code of the OBLIVIOUS algorithms can be easily derived from the previous sequence of steps. To complete the traversal of a single tree, we use `leafID` to lookup a table of $|L|$ elements, storing the values of the tree leaves, and add the value of the selected exit leaf to an accumulator. In addition, the same code must be repeated for all the trees in the ensemble, by accumulating the partial scores derived from each oblivious tree.

It is worth noting that OBLIVIOUS is a very efficient algorithm in terms of time and space complexity, thus being a challenging competitor for our scorer QS_Ω, derived from QS and discussed in the following. Indeed, OBLIVIOUS does not need any branches for traversing a single tree, while it only uses a pair of arrays to store each tree. The first array is exploited to store $\log(|L|)$ pairs, namely feature ids and thresholds, used for

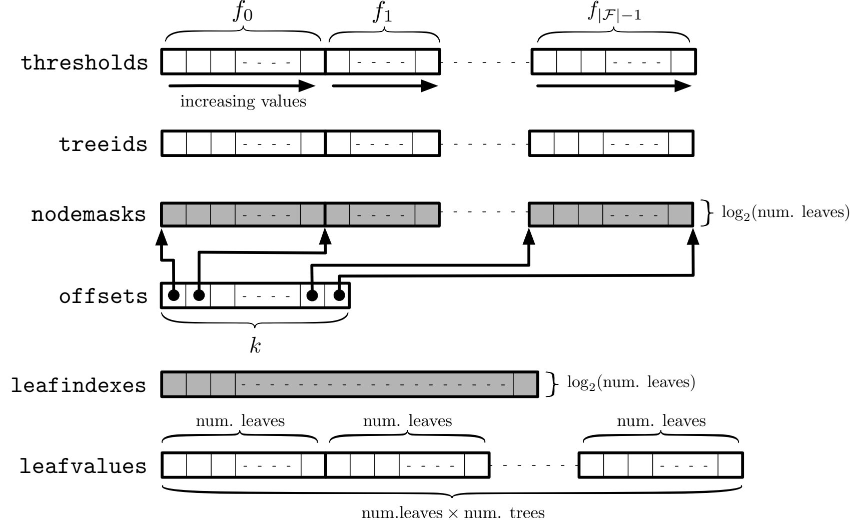

Fig. 9: QUICKSCORER-Ω data structure layout.

the tests at each level of the tree. The second array stores the values of the $|L|$ leaves of the tree. Therefore, the memory footprint of OBLIVIOUS is very small, since, unlike algorithms for non-oblivious trees, OBLIVIOUS does need to store information regarding the $|L| - 1$ branching nodes, but only the $d = \log(|L|)$ Boolean tests corresponding to the levels of the balanced tree. Such limited memory footprint allows OBLIVIOUS to make a more effective use of the first levels of the cache.

Note that in the sequence of operations (see Equation 1) performed by OBLIVIOUS, the true nodes do not contribute to the identification of the exit leaf, as they result in a logical *or* operation of `leafID` with a string of 0 bits. Therefore, in principle we can save many logical operations if we can devise an efficient method to identify the false nodes only in a given tree, thus saving many logical operations. To this end, we can resort to the technique used by QUICKSCORER to efficiently identify all the false nodes only, by using a data structure similar to that of the non-oblivious version of QS, as illustrated in Figure 9. Unlike QS, for oblivious trees we need a single bitvector mask for each level of each tree in the ensemble, rather than one for each node. Similarly, the exit leaf can be encoded with $\log(|L|)$ bits rather than $|L|$. This reduces the size of the `nodemasks` and `leafindexes` data structures by a logarithmic factor.

Figure 10 illustrates the same tree $T_h \in \mathcal{T}$ as Figure 8, for which QS_Ω stores three bitvector masks, namely `100`, `010`, and `001`, associated with levels 0, 1, and 2 of the tree, respectively. These masks are used by QS_Ω to update `leafindexes[`h`]`, which, at the end of the interleaved visit of the ensemble, will store the binary identifier of the exit leaf $e(\mathbf{x}) = l_5$. The figure shows that two logical *or* operations are enough to identify this exit leaf, once `leafindexes[`h`]` is initialized to `000`. For these two operations, we use the bitvector masks associates with levels 0 and 2, which include the only false nodes occurring the tree.

Algorithm 3 shows the modified QS algorithm, named QUICKSCORER-Ω (QS_Ω), that scores documents with an ensemble of oblivious trees. Beyond the smaller size of the data structures used, the main differences are in the initialization of `leafindexes[`h`]` (line 2) to all 0's, the logical *or* bitwise operation used to set a single bit of the result index `leafindexes[`h`]` (line 8), and the direct use of the value eventually stored in

ALGORITHM 3: The QUICKSCORER- Ω algorithm for oblivious trees

Input :
— $\mathbf{x}$: input feature vector
— $\mathcal{T}$: ensemble of binary decision trees, with
- $w_0, \ldots, w_{|\mathcal{T}|-1}$: weights, one per tree
- `thresholds`: sorted sublists of thresholds, one sublist per feature
- `treeids`: tree's ids, one per threshold/level
- `nodemasks`: node masks, one per threshold/level
- `offsets`: offsets of the blocks of triples
- `leafindexes`: result index, one per each tree
- `leafvalues`: output values, one per each tree leaf

Output:
— Final score of $\mathbf{x}$

```
QuickScorer-Ω(x,T):
1   foreach h ∈ 0,1,...,|T|−1 do
2       leafindexes[h] ← 00...00
3   foreach k ∈ 0,1,...,|F|−1 do                                  // Step ①
4       i ← offsets[k]
5       end ← offsets[k+1]
6       while x[k] > thresholds[i] do
7           h ← treeids[i]
8           leafindexes[h] ← leafindexes[h] ∨ nodemasks[i]
9           i ← i + 1
10          if i ≥ end then
11              break
12  score ← 0
13  foreach h ∈ 0,1,...,|T|−1 do                                  // Step ②
14      score ← score + w_h · leavalues[leafindexes[h]]
15  return score
```

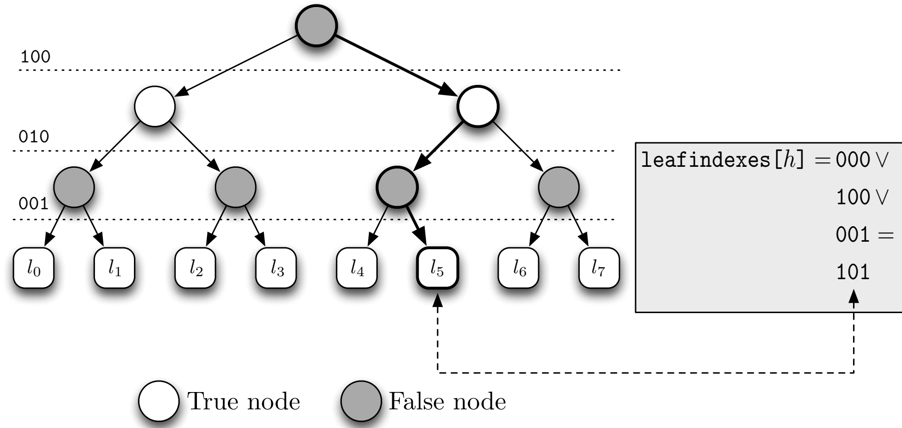

Fig. 10: Optimized traversal of an oblivious tree performed by QS_{Ω}.

`leafindexes`[h] as an index to lookup the array `leafvalues` to obtain the contribution of each tree to the final score (line 14).

5. EXPERIMENTS

In this section we discuss the results of the extensive experiments conducted to assess the performance of QS and QS_Ω with respect to competitors.

5.1. Datasets and experimental settings

For the experiments, we use three, publicly available, datasets. The first two datasets are the MSN and the Yahoo LETOR challenge datasets, commonly used in the scientific community for LtR experiments. The third one, which we make publicly available along with the publication of this paper, is a very large LtR dataset provided by TISCALI ITALIA S.p.A., an Italian company running the web search engine ISTELLA (http://www.istella.it). The characteristics of the three datasets are listed in Table II. For all of them the vectors associated with query-document pairs are labeled with relevance judgments ranging from 0 (irrelevant) to 4 (perfectly relevant).

— The MSN dataset is available at http://research.microsoft.com/en-us/projects/mslr/. It comes splitted into five folds. In this work, we used only the first MSN fold, named MSN-1.
— The Yahoo dataset is available at http://learningtorankchallenge.yahoo.com. It consists of two distinct datasets (Y!S1 and Y!S2). In this paper we used the Y!S1 dataset.
— The Istella (Full) dataset is available at http://blog.istella.it/istella-learning-to-rank-dataset/. To the best of our knowledge, this dataset is the largest publicly available LtR dataset, particularly useful for large-scale experiments on the efficiency and scalability of LtR solutions. Moreover, it is the first public dataset being representative of a real-world ranking pipeline, with long lists of results including large numbers of irrelevant documents for each query, as also discussed by Yin et al. [2016].

Table II. Main properties of the three dataset used: i) # of features, ii) # of queries in train/validation/test sets, iii) total # of documents in train/test sets, and iv) average # of document per query in test set.

Property	Dataset		
	MSN-1	Y!S1	Istella
# features	136	700	220
# queries in training	6,000	19,944	23,319
# queries in validation	2,000	2,994	-
# queries in test	2,000	6,983	9,799
Total # documents in train	723,412	473,134	7,325,625
Total # documents in test	241,521	165,660	3,129,004
Average # documents per query in test	120.7	23.72	319.31

The experimental methodology adopted is the following. We use training data from MSN-1, Y!S1, and Istella to train λ-MART [Wu et al. 2010], and OBLIVIOUS-λ-MART models (thus, both optimizing NDCG@10). The various generated models are ensembles of trees, whose maximum number of leaves is equal to 8, 16, 32, or 64. To train these models we use QuickRank[1], an open-source LtR C++11 framework providing

[1] http://quickrank.isti.cnr.it

efficient implementations of LtR algorithms [Capannini et al. 2016; Capannini et al. 2015]. It is worth noting that the results reported in the paper, concerning the efficiency at testing time of tree-based scorers, are however independent of the specific LtR algorithm used to train the specific ensembles of trees.

Effectiveness of the learned models. Although studying the ranking effectiveness of tree ensembles is out of the scope of this paper, in the following we assess the benefits deriving from using more complex ensembles, such as those used in our efficiency tests, obtained by increasing the number of additive trees.

Figure 11 shows the variations of NDCG@10 measured on the test sets of MSN-1, Y!S1, and Istella datasets as a function of the number of trees of a λ-MART ensemble, whose trees have 16-leaves. The baseline is a simple λ-MART model with only $1,000$ trees. As we vary the number of trees in the range from $5,000$ to $20,000$, we observe that NDCG@10 measured on Y!S1 increases only slightly, while it even decreases with MSN-1. This behavior is due to overfitting that prevents the ranking models learned on small training datasets to take advantage of so large number of trees. This instead does not happen with the Istella dataset: due to its large size, the learned models exhibit an opposite behavior, with an absolute NDCG@10 improvement of about 4% over the baseline model with 1,000 trees when an ensemble of $20,000$ trees is exploited. As a consequence, the Istella dataset may lead to the creation of larger and more expensive ranking models. For this reason, we limit part of the analysis to the models trained on the ISTELLA dataset.

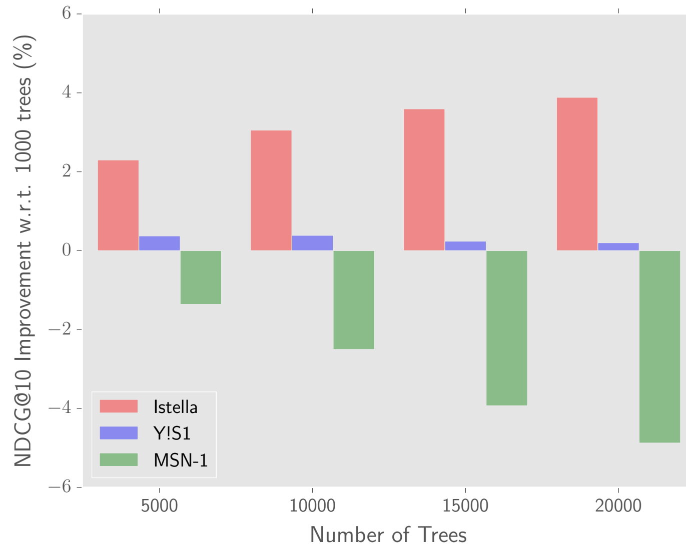

Fig. 11: Variations of NDCG@10 measured on the MSN-1, Y!S1, and Istella datasets as a function of the number of trees in a 16-leaves λ-MART ensemble, with respect to a baseline ranking model of $1,000$ trees.

Experimental setting for efficiency tests. The goal of all the tests, whose results are discussed in the following, is to measure the scoring efficiency of QS and QS_Ω, and

to compare their performance with other algorithms. The competitors of QS are the following:

— IF-THEN, the baseline that translates each tree of the forest into a nested structure of if-then-else blocks;
— CONDOP, the baseline exploiting C conditional operators (`?:`). It works by translating each tree of the forest into a nested block of conditional operators;
— VPRED and STRUCT+ [Asadi et al. 2014] kindly made available by the authors[2].

IF-THEN and CONDOP are analogous, since in both case each tree of the ensemble is compiled as a nested structure. The latter, however, produces very long expressions of nested conditional operators from which, according to our experiments, the compiler is able to generate a more efficient code than from the former.

The competitors of QS_Ω are all the previous algorithms, which are not aware, however, of the characteristics of the oblivious trees, and thus deal with them as non-oblivious ones. In addition, we compare the execution time of QS_Ω with the following very optimized competitor:

— OBLIVIOUS, a challenging baseline for scoring with ensembles of oblivious trees, as discussed in Section 4. OBLIVIOUS has a very small memory footprint due to the succinct data structures, and does not need to execute branches to traverse each tree.

All the algorithms are compiled with GCC 5.1.0 with the highest optimization settings. The tests are performed by using a single core on a machine equipped with an Intel Core i7-4770K clocked at 3.50Ghz, with 32GB RAM, running Ubuntu Linux with kernel 3.13.0-65-generic. The Intel Core i7-4770K CPU has three levels of cache. Each of the four cores has a dedicated L1 cache of 32 KB, and a dedicated L2 cache of 256 KB, while the L3 cache is shared and has a size of 8 MB.

To measure the efficiency of each of the above methods, we run 10 times the scoring code on the test sets of the MSN-1, Y!S1, and Istella datasets. We then compute the average per-document scoring cost. Moreover, to profile the behavior of each method above we employ `perf`[3], a performance analysis tool available under Ubuntu Linux distributions. We analyse each method by monitoring several CPU counters that measure the total number of instructions executed, number of branches, number of branch mis-predictions, cache references, and cache misses.

5.2. Scoring time analysis

The average time (in μs) needed by the different algorithms to score a document of MSN-1, Y!S1, and Istella is reported in Table III. In particular, the table reports the per-document scoring time by varying the number of trees and the maximum number of leaves of each tree ensemble, in turn obtained by training λ-MART [Wu et al. 2010] over MSN-1, Y!S1, and Istella. For each test, the table also reports (between parentheses) the speedup of QS over its competitors. At a first glance, the speedups are impressive, in many cases above one order of magnitude. Depending on the number of trees and of leaves, QS outperforms the state-of-the-art solution VPRED, of factors ranging from 1.9x up to 6.6x. For example, the average time required by QS and VPRED to score a document in the MSN-1 test set with a model composed of $1,000$ trees and 64 leaves, are 9.4 and 62.2 μs, respectively. The comparison between QS and IF-THEN is even more one-sided, with improvements of up to 24.4x for the model with $10,000$ trees

[2] `http://nasadi.github.io/OptTrees/`
[3] `https://perf.wiki.kernel.org`

and 32 leaves trained on the MSN-1 dataset. In this case the QS average per-document scoring time is 59.6 μs with respect to the $1,453.8$ μs of IF-THEN. CONDOP behaves slightly better than IF-THEN in all the tests, thus showing that the compiler is able to produce more optimized code for CONDOP than for IF-THEN. However, the compiler used (GCC 5.1.0) crashes when CONDOP models larger than a given size are compiled. We reported these crash conditions in Tables III and IV as "Not Available" (NA) result. The last baseline reported, i.e., STRUCT+, behaves the worst in all the tests conducted. Its performance is very low when compared not only to QS (up to 40.7x times faster), but even to the other four algorithms. There are multiple reasons that cause the superior performance of QS over competitors. We analyse the most relevant of them in the next section.

Table III. Per-document scoring time in μs of QS, VPRED, CONDOP, IF-THEN, and STRUCT+ on MSN-1, Y!S1, and Istella datasets with λ-MART models. Speedups are reported in parentheses.

Method	Λ	Number of trees/dataset					
		1,000			5,000		
		MSN-1	Y!S1	Istella	MSN-1	Y!S1	Istella
QS	8	**2.2** (–)	**4.4** (–)	**2.8** (–)	**10.1** (–)	**14.5** (–)	**12.0** (–)
VPRED		7.8 (3.5x)	8.4 (1.9x)	7.6 (2.7x)	39.8 (3.9x)	41.1 (2.8x)	39.3 (3.3x)
CONDOP		6.7 (3.0x)	10.3 (2.3x)	8.0 (2.9x)	67.8 (6.7x)	75.9 (5.2x)	77.0 (6.4x)
IF-THEN		7.3 (3.3x)	10.1 (2.3x)	9.0 (3.2x)	78.2 (7.7x)	84.6 (5.8x)	84.1 (7.0x)
STRUCT+		21.0 (9.5x)	23.1 (5.3x)	24.9 (8.9x)	98.7 (9.8x)	119.5 (8.2x)	117.9 (9.8x)
QS	16	**3.1** (–)	**6.4** (–)	**4.5** (–)	**15.9** (–)	**21.6** (–)	**17.9** (–)
VPRED		16.0 (5.2x)	16.4 (2.6x)	14.9 (3.3x)	82.0 (5.2x)	82.4 (3.8x)	79.3 (4.4x)
CONDOP		14.1 (4.5x)	17.2 (2.7x)	16.1 (3.6x)	100.0 (6.3x)	110.0 (5.0x)	155.0 (8.7x)
IF-THEN		18.1 (5.8x)	20.7 (3.2x)	19.6 (4.4x)	128.0 (8.0x)	128.8 (6.0x)	135.5 (7.6x)
STRUCT+		40.9 (13.2x)	41.6 (6.5x)	44.4 (9.9x)	411.2 (25.9x)	418.6 (19.4x)	407.8 (22.8x)
QS	32	**5.2** (–)	**9.7** (–)	**6.8** (–)	**26.8** (–)	**34.5** (–)	**26.9** (–)
VPRED		31.8 (6.1x)	31.5 (3.2x)	28.1 (4.1x)	164.5 (6.1x)	16 1.6 (4.7x)	157.7 (5.9x)
CONDOP		27.0 (5.2x)	30.3 (3.1x)	30.4 (4.5x)	NA (x)	NA (x)	NA (x)
IF-THEN		32.2 (6.2x)	34.0 (3.5x)	33.3 (4.9x)	270.5 (10.1x)	256.6 (7.4x)	240.6 (8.9x)
STRUCT+		69.4 (13.3x)	66.5 (6.9x)	67.8 (10.0x)	861.0 (32.1x)	833.2 (24.2x)	807.9 (x)
QS	64	**9.4** (–)	**15.1** (–)	**11.2** (–)	**57.6** (–)	**70.2** (–)	**57.8** (–)
VPRED		62.2 (6.6x)	57.3 (3.8x)	54.3 (4.8x)	347.2 (6.0x)	333.6 (4.8x)	326.8 (5.7x)
CONDOP		48.6 (5.2x)	48.4 (3.2x)	51.2 (4.6x)	NA (x)	NA (x)	NA (x)
IF-THEN		54.0 (5.8x)	53.2 (3.5x)	55.0 (4.9x)	901.1 (15.6x)	801.9 (11.4x)	911.2 (15.8x)
STRUCT+		132.3 (14.1x)	109.5 (7.3x)	112.6 (10.5x)	1485.5 (25.8x)	1498.2 (21.3x)	1487.3 (25.7x)

Method	Λ	Number of trees/dataset					
		10,000			20,000		
		MSN-1	Y!S1	Istella	MSN-1	Y!S1	Istella
QS	8	**20.2** (–)	**26.6** (–)	**33.2** (–)	**40.85** (–)	**47.4** (–)	**41.7** (–)
VPRED		79.8 (4.0x)	81.9 (3.0x)	78.9 (2.4x)	159.8 (3.9x)	163.1 (3.4x)	158.3 (3.8x)
CONDOP		161.0 (8.0x)	173.0 (6.5x)	181.1 (5.5x)	NA (x)	NA (x)	NA (x)
IF-THEN		182.6 (9.0x)	186.8 (7.0x)	196.1 (5.9x)	590.0 (14.4x)	622.1 (13.1x)	585.7 (14.0x)
STRUCT+		369.3 (18.3x)	398.8 (15.0x)	393.9 (11.9x)	1082.8 (26.5x)	1140.4 (24.0x)	1122.3 (26.9x)
QS	16	**32.7** (–)	**40.5** (–)	**33.8** (–)	**67.3** (–)	**81.1** (–)	**69.3** (–)
VPRED		164.9 (5.0x)	164.5 (4.0x)	160.5 (4.7x)	334.8 (5.0x)	332.6 (4.1x)	326.9 (4.7x)
CONDOP		NA (x)	NA (x)	NA (x)	NA (x)	NA (x)	NA (x)
IF-THEN		334.1 (10.2x)	353.8 (8.7x)	401.5 (11.9x)	1815.4 (27.0x)	1695.9 (20.9x)	1696.2 (24.5x)
STRUCT+		1037.1 (31.7x)	1063.1 (26.2x)	1025.4 (30.3x)	2359.2 (35.0x)	2377.1 (29.3x)	2276.5 (32.8x)
QS	32	**59.6** (–)	**70.3** (–)	**56.0** (–)	**155.0** (–)	**158.9** (–)	**132.7** (–)
VPRED		341.8 (5.7x)	335.3 (4.8x)	325.7 (5.8x)	708.1 (4.6x)	693.1 (4.4x)	678.2 (5.1x)
CONDOP		NA (x)	NA (x)	NA (x)	NA (x)	NA (x)	NA (x)
IF-THEN		1453.0 (24.4x)	1245.8 (17.8x)	1357.1 (24.3x)	3018.8 (19.5x)	2984.5 (18.8x)	2702.8 (20.4x)
STRUCT+		2424.3 (40.7x)	2346.5 (33.4x)	2279.1 (40.7x)	4002.2 (25.8x)	3824.4 (24.1x)	3648.3 (27.5x)
QS	64	**158.2** (–)	**156.3** (–)	**146.7** (–)	**428.1** (–)	**335.0** (–)	**289.6** (–)
VPRED		733.2 (4.6x)	704.7 (4.5x)	696.3 (4.7x)	1307.6 (3.0x)	1412.9 (4.2x)	1413.1 (4.9x)
CONDOP		NA (x)	NA (x)	NA (x)	NA (x)	NA (x)	NA (x)
IF-THEN		2364.3 (14.9x)	2350.5 (15.0x)	2334.8 (15.9x)	4397.1 (10.3x)	4647.2 (13.9x)	4678.8 (16.2x)
STRUCT+		3014.8 (19.0x)	2894.4 (18.5x)	2942.8 (20.1x)	6794.5 (15.9x)	6923.9 (20.7x)	7586.4 (26.2x)

Table IV. Per-document scoring time in μs of QS_Ω, OBLIVIOUS, VPRED, CONDOP, IF-THEN, and STRUCT+ on MSN-1, Y!S1, and Istella datasets with OBLIVIOUS-λ-MART models. Speedups are reported in parentheses.

Method	Λ	Number of trees/dataset					
		1,000			5,000		
		MSN-1	Y!S1	Istella	MSN-1	Y!S1	Istella
QS_Ω	8	**2.1** (–)	3.9 (1.4x)	**2.2** (–)	**9.3** (–)	**12.9** (–)	**10.4** (–)
OBLIVIOUS		2.5 (1.2x)	**2.7** (–)	2.6 (1.2x)	12.8 (1.4x)	**12.9** (–)	12.9 (1.2x)
VPRED		3.7 (1.8x)	4.0 (1.5x)	3.9 (1.8x)	18.6 (2.0x)	19.3 (1.5x)	18.8 (1.8x)
CONDOP		5.5 (2.6x)	5.6 (2.0x)	8.1 (3.7x)	54.2 (5.8x)	54.7 (4.2x)	71.9 (6.9x)
IF-THEN		5.4 (2.6x)	6.2 (2.3x)	8.2 (3.7x)	65.6 (7.0x)	64.4 (5.0x)	74.5 (7.2x)
STRUCT+		13.6 (6.5x)	13.7 (5.0x)	16.8 (7.6x)	66.3 (7.1x)	67.1 (5.2x)	78.5 (7.5x)
QS_Ω	16	**2.5** (–)	4.2 (1.1x)	**2.5** (–)	**12.2** (–)	**14.3** (–)	**12.7** (–)
OBLIVIOUS		3.0 (1.2x)	**3.7** (–)	3.8 (1.5x)	18.2 (1.5x)	17.9 (1.3x)	18.7 (1.5x)
VPRED		4.5 (1.8x)	4.7 (1.3x)	4.7 (1.9x)	22.8 (1.9x)	22.0 (1.5x)	23.4 (1.8x)
CONDOP		7.4 (3.0x)	7.1 (1.9x)	12.2 (4.9x)	75.9 (6.2x)	67.9 (4.7x	180.0 (14.2x)
IF-THEN		10.4 (4.2x)	8.7 (2.4x)	13.9 (5.6x)	91.7 (7.5x)	80.4 (5.6x)	101.9 (8.0x)
STRUCT+		19.6 (7.8x)	19.0 (5.1x)	21.8 (8.7x)	96.9 (7.9x)	91.1 (6.4x)	110.3 (8.7x)
QS_Ω	32	**2.9** (–)	4.3 (1.1x)	**3.0** (–)	**12.8** (–)	**14.7** (–)	**13.1** (–)
OBLIVIOUS		4.1 (1.4x)	**3.9** (–)	4.7 (1.6x)	21.7 (1.7x)	21.1 (1.4x)	22.5 (1.7x)
VPRED		4.9 (1.7x)	4.9 (1.3x)	5.5 (1.8x)	24.7 (1.9x)	22.9 (1.6x)	25.9 (2.0x)
CONDOP		9.1 (3.1x)	8.0 (2.0x)	16.6 (5.5x)	88.9 (6.9x)	75.6 (5.1x)	125.0 (9.5x)
IF-THEN		11.2 (3.9x)	9.0 (2.3x)	17.6 (5.9x)	102.7 (8.0x)	85.2 (5.8x)	118.5 (9.0x)
STRUCT+		21.9 (7.6x)	20.6 (5.3x)	27.4 (9.1x)	115.0 (9.0x)	98.8 (6.7x)	137.6 (10.5x)
QS_Ω	64	**3.0** (–)	5 (1.1x)	**3.2** (–)	**11.5** (–)	**13.8** (–)	**12.3** (–)
OBLIVIOUS		4.4 (1.5x)	**4.5** (–)	4.9 (1.5x)	22.0 (1.9x)	21.0 (1.5x)	28.2 (2.3x)
VPRED		5.1 (1.7x)	4.9 (1.1x)	5.9 (1.8x)	25.0 (2.2x)	23.2 (1.7x)	27.2 (2.2x)
CONDOP		10.0 (3.3x)	8.2 (1.8x)	20.0 (6.3x)	91.7 (8.0x)	75.6 (5.4x)	135.0 (11.0x)
IF-THEN		12.6 (4.2x)	9.7 (2.2x)	19.8 (6.2x)	105.2 (9.1x)	87.6 (6.3x)	132.0 (10.7x)
STRUCT+		24.1 (8.0x)	20.9 (4.6x)	33.1 (10.3x)	123.7 (10.8x)	101.9 (7.4x)	147.8 (12.0x)

Method	Λ	Number of trees/dataset					
		10,000			20,000		
		MSN-1	Y!S1	Istella	MSN-1	Y!S1	Istella
QS_Ω	8	**19.1** (–)	**20.6** (–)	**19.8** (–)	**37.2** (–)	**40.3** (–)	**36.2** (–)
OBLIVIOUS		26.0 (1.4x)	25.7 (1.2x)	25.2 (1.3x)	52.4 (1.4x)	51.8 (1.4x)	46.0 (1.3x)
VPRED		37.3 (2.0x)	38.5 (1.9x)	35.6 (1.8x)	74.8 (2.0x)	77.2 (1.9x)	62.4 (1.7x)
CONDOP		132.0 (6.9x)	134.0 (4.7x)	140.0 (7.1x)	NA (x)	NA (x)	234.0 (6.5x)
IF-THEN		144.8 (7.6x)	143.2 (7.0x)	142.2 (7.2x)	345.8 (9.3x)	338.3 (8.4x)	207.9 (5.7x)
STRUCT+		132.4 (6.9x)	134.3 (6.5x)	144.2 (7.3x)	432.7 (11.6x)	442.2 (11.0x)	332.0 (9.2x)
QS_Ω	16	**22.7** (–)	**25.0** (–)	**23.9** (–)	**44.4** (–)	**46.4** (–)	**45.8** (–)
OBLIVIOUS		37.2 (1.6x)	35.9 (1.4x)	37.6 (1.6x)	74.6 (1.7x)	71.9 (1.5x)	90.3 (2.0x)
VPRED		45.9 (2.0x)	44.5 (1.8x)	46.8 (2.0x)	93.9 (2.1x)	90.7 (2.0x)	95.3 (2.0x)
CONDOP		188.0 (8.3x)	168.0 (6.7x)	NA (x)	NA (x)	NA (x)	NA (x)
IF-THEN		206.1 (9.1x)	203.4 (8.1x)	235.0 (9.8x)	978.4 (22.0x)	778.5 (16.8x)	1117.7 (24.4x)
STRUCT+		198.9 (8.8x)	186.6 (7.5x)	229.3 (9.6x)	686.4 (15.5x)	627.8 (13.5x)	821.4 (17.9x)
QS_Ω	32	**23.8** (–)	**25.9** (–)	**25.4** (–)	**46.7** (–)	**48.4** (–)	**57.6** (–)
OBLIVIOUS		55.8 (2.3x)	41.6 (1.6x)	48.0 (1.9x)	95.6 (2.0x)	87.4 (1.8x)	205.0 (3.6x)
VPRED		49.5 (2.1x)	45.9 (1.8x)	51.9 (2.0x)	110.3 (2.4x)	96.2 (2.0x)	117.3 (2.0x)
CONDOP		NA (x)	NA (x)	NA (x)	NA (x)	NA (x)	NA (x)
IF-THEN		233.0 (9.8x)	196.0 (7.6x)	333.3 (13.2x)	1396.2 (29.9x)	929.2 (19.2x)	1567.7 (27.2x)
STRUCT+		246.7 (10.4x)	203.9 (7.9x)	284.7 (11.2x)	744.6 (15.9x)	609.2 (12.6x)	885.4 (15.4x)
QS_Ω	64	**22.0** (–)	**23.7** (–)	**22.2** (–)	**66.4** (–)	**74.0** (–)	**67.9** (–)
OBLIVIOUS		71.7 (3.2x)	44.7 (1.9x)	69.2 (3.1x)	195.0 (2.9x)	169.0 (2.3x)	206.0 (3.0x)
VPRED		50.7 (2.3x)	46.4 (2.0x)	42.7 (1.9)	118.1 (1.8x)	99.0 (1.3x)	69.3 (1.0x)
CONDOP		NA (x)	NA (x)	NA (x)	NA (x)	NA (x)	NA (x)
IF-THEN		251.7 (11.4x)	200.9 (8.5x)	187.7 (8.5x)	1561.4 (23.5x)	1018.5 (13.8x)	254.6 (3.7x)
STRUCT+		264.1 (12.0x)	211.6 (8.9x)	211.4 (9.5x)	805.1 (12.1x)	645.0 (8.7x)	380.3 (5.6x)

The same experiments have been conducted with oblivious models generated by OBLIVIOUS-λ-MART. The results obtained are reported in Table IV. As a general consideration we observe that the running time of all the scoring algorithms is significantly reduced. For example, for the same number of trees and leaves, STRUCT+ results to be up to 10 time faster on oblivious models than on non-oblivious ones. This is mainly due to the imposed balancing of the oblivious trees, which upper bounds the depth of the visit to $\log(|\Lambda|)$. Even in this case QS_Ω, the oblivious version of QS, is

faster than the state-of-the-art algorithm VPRED, but with a smaller gap, providing improvements in the range 1.1x up to 2.4x. The CONDOP, IF-THEN, and STRUCT+ algorithms exhibit a similar behavior as in previous experiments. A different behavior can instead be observed for OBLIVIOUS. The experiments conducted show in fact that OBLIVIOUS slightly outperforms QS_{Ω} in the tests involving Y!S1 and ensembles of $1,000$ trees. The relatively lower performance of QS_{Ω} in these specific cases is motivated by the very small memory footprint of OBLIVIOUS with respect to QS_{Ω} and all the other competitors. For example, QS_{Ω} needs to materialize the array `nodemasks`. Therefore, even if OBLIVIOUS performs a relatively larger number of tests than QS_{Ω}, since QS_{Ω} only needs to detect the false nodes of each tree, for small number of trees OBLIVIOUS can better exploit the first levels of the cache. On larger ensembles this does not hold, and all the other experiments conducted show a superior performance of QS_{Ω}, with speedups reaching a factor 3.6x over OBLIVIOUS.

5.3. Instruction level analysis

We used the `perf` tool to measure the total number of instructions, number of branches, number of branch mis-predictions, L3 cache references, and L3 cache misses for the different scorers. For these tests we limit ourselves to non-oblivious tree ensembles, and compare QS, VPRED, IF-THEN, and STRUCT+ on a λ-MART model trained on the largest and most challenging ISTELLA dataset. Experiments on the other datasets are not reported here as they exhibit a similar behavior. Table V reports the results obtained by varying the number of trees of the λ-MART model, for a fixed number of 64 leaves. The performance measures of CONDOP are not reported because, as discussed in the previous section, we experienced segmentation faults of the GCC compiler when compiling so complex models with 64 leaves.

As a clarification, L3 cache references accounts for those references which are not found in any of the previous levels of cache, while L3 cache misses are the ones among them which miss in L3 as well. Table V also reports the number of visited nodes. All measurements are normalized per-document and per-tree.

We first observe that number of instructions executed by VPRED is the largest one. This is because VPRED always runs d steps, if d is the depth of a tree, even if a document might reach an exit leaf earlier. IF-THEN executes much less instructions, as trees are traversed from the root to the exit leaf in a traditional way. STRUCT+ introduces some data structures overhead with respect to IF-THEN. QS executes the smallest number instructions. This is due to the different traversal strategy of the ensemble, as QS needs to process the *false nodes* only. Indeed, QS always visits an average percentage of branching nodes per tree between 20% and 25%. This is much less that IF-THEN, whose average percentage of visited nodes is between 51% and 81%, and the same trivially holds for STRUCT+. VPRED visits the largest number of nodes, namely an average percentage between 72% and 93%. This means that the interleaved traversal strategy of QS needs to process less nodes than in a traditional root-to-leaf visit. This mostly explains the results achieved by QS.

In terms of number of branches, we note that, not surprisingly, QS and VPRED are much more efficient than IF-THEN, CONDOP, and STRUCT+. QS has a larger total number of branches than VPRED, which uses scoring functions that are branch-free. However, those branches are highly predictable, so that the mis-prediction rate is very low, thus, confirming our claims in Section 3.

Observing again the timings in Table III, we notice that, by fixing the number of leaves, we have a super-linear growth of QS's timings when increasing the number of trees. For example, considering the ISTELLA dataset and a tree ensemble with $\Lambda = 64$ and $1,000$ trees, we have that QS scores a document in 11.2 μs. Therefore, when the ensemble size increases to $20,000$ trees, one would expect to score a document 20 times

slower, i.e., 224 μs, since the complexity is linear in the number of trees. However, the reported timing of QS in this setting is larger, i.e., 289.6 μs, which is roughly 26 times slower than the time obtained with $1,000$ trees. This super-linear effect is observable only for large number of leaves ($\Lambda = \{32, 64\}$) and large number of trees (greater than $5,000$). A similar behavior is reported also in Table IV. Table V relates this super-linear growth to the number of L3 cache misses.

Considering the sizes of the arrays as reported in Table I in Section 3, we can estimate the minimum number of trees that let the size of the QS's data structure to exceed the cache capacity, and, thus, the algorithm starts to have more cache misses. This number is estimated in $6,000$ trees when the number of leaves is 64. Thus, we expect that the number of L3 cache miss starts increasing around this number of trees. Possibly, this number is slightly larger, because portions of the data structure may be infrequently accessed at scoring time, due the small fraction of false nodes and associated bitvector masks accessed by QS.

These considerations are further confirmed by Figure 12, which shows the average per-tree per-document scoring time (μs) and the percentage of cache misses of QS, when scoring the Istella with a λ-MART model with $\Lambda = 64$ by varying the number of trees. First, there exists a strong correlation between QS's timings and its number of L3 cache misses. Second, the number of L3 cache misses starts increasing when dealing with $8,000$ trees and it becomes significant beyond $12,000$.

6. BWQS: A BLOCK-WISE VARIANT OF QS

The results of the experiments discussed in the previous section suggest that improving the cache efficiency of QS may result in significant benefits. To this purpose, we can modify QS in order to score δ documents simultaneously as done by VPRED. Additionally, we can exploit a 2D blocking strategy such the one proposed by Tang et al. [2014], and split the tree ensemble in disjoint blocks of τ trees to be evaluated separately for each bunch of δ documents. Besides partitioning the ensemble in blocks of trees, in the following we also discuss a novel optimization that reduces the overall number of instructions executed by QS for traversing blocks of trees. Although analogous block-wise strategies can also be adopted for QS_Ω, we do not report and discuss experimental results for ensembles of oblivious trees.

By tuning parameters τ and δ it is possible to let the corresponding QS's data structures fit into the faster levels of the memory hierarchy, still inheriting the efficiency of QS. Indeed, the size of the arrays required to score the documents over a block of trees depends on the number τ of trees per block instead of $|\mathcal{T}|$ (see Table I in Section 3). This strategy can enhance spatial locality by fitting into the cache, one at a time, sub-blocks of large models that in the previous experiments exceeded the cache capacity. The temporal locality can be instead improved by tuning the parameter δ thus allowing the algorithm to score blocks of documents together over the same block of trees before moving to the next block of documents. To allow QS to score a block of δ documents in a single run we have however to use δ different arrays `leafindexes` to store the partial scores accumulated so far by each document. This increases the space occupancy and therefore, we need to find the best balance between the number of documents δ and the number of trees τ to process in the body of a nested loop that first runs over the blocks of trees (outer loop) and then over the blocks of documents to score (inner loop).

Tree reversing to reduce the number of false nodes. The processing of the given ensemble in independent blocks of trees allows us to devise a novel optimization of QS, made possible by the specific QS's traversal of the tree ensemble. We have already seen that the cost of QS primarily depends on the number of (false) tree nodes actually vis-

Table V. Per-tree per-document low-level statistics on Istella with 64-leaves λ-MART models.

Method	Number of Trees				
	1,000	5,000	10,000	15,000	20,000
	Instruction Count				
QS	**73**	**76**	**87**	**90**	**80**
VPRED	495	584	619	636	637
IF-THEN	110	144	162	173	177
STRUCT+	273	350	391	418	428
	Num. branch mis-predictions (above) Num. branches (below)				
QS	0.19 7.90	**0.04** 7.41	**0.02** 8.34	**0.01** 8.61	**0.01** 7.63
VPRED	**0.02** **0.20**	**0.04** **0.20**	0.03 **0.20**	0.03 **0.20**	0.03 **0.20**
IF-THEN	1.96 34.08	2.56 43.35	2.01 48.11	1.75 51.0	1.53 35.71
STRUCT+	5.31 72.69	5.05 94.44	5.40 107.11	7.04 114.87	7.37 116.26
	L3 cache misses (above) L3 cache references (below)				
QS	0.005 **2.02**	**0.001** **1.44**	**0.016** **1.49**	**0.119** **1.73**	0.235 **1.89**
VPRED	0.005 14.87	0.093 12.37	0.160 9.37	0.158 7.37	**0.162** 6.06
IF-THEN	**0.001** 25.41	15.098 37.19	28.006 38.83	28.527 37.44	28.008 35.71
STRUCT+	0.242 11.80	13.879 17.39	12.935 17.659	13.845 21.88	20.768 26.14
	Num. Visited Nodes (above) Visited Nodes/Total Nodes (below)				
QS	**12.52** **20%**	**13.76** **22%**	**15.88** **25%**	**16.52** **26%**	**14.60** **23%**
VPRED	45.60 72%	53.84 85%	57.17 91%	58.77 93%	58.87 93%
STRUCT+ IF-THEN	32.18 51%	41.41 66%	46.14 73%	49.35 78%	50.77 81%

ited. Let us consider a limit case, that is a tree for which every node contains a test condition evaluating to *false* for a given document x. This situation would be the worst one for our solution, since all the associated tests should be executed. The cost of such a tree could be however zeroed out by exploiting a simple transformation of the tree. Our transformation that reduces the number of visited nodes consists in *reversing* the test condition in each node of the block, from $\mathrm{x}[\phi] \leq \gamma$ to $\mathrm{x}[\phi] > \gamma$, and in swapping the left and right subtrees of the node. In this way we would obtain an equivalent tree for which every node's test condition evaluates to *true* for the same document x. Such *reversed tree* only requires a dual data structure for storing the feature thresholds, which must be sorted in *descending* order.

Tree reversing should be profitable for all the trees that contain more *false* than *true* nodes on average over a collection of training documents. Let us call these trees *false trees*. We can easily estimate the probability of a tree being a *false tree* on the validation set during the learning phase. If the number of false trees in the ensemble is significant, we can reverse all of them as described before. The tree block-wise strategy

is then applied so as to achieve homogeneous blocks containing either *only* false trees or without false trees at all.

This resulting algorithm is called BLOCKWISE-QS (BWQS) and its efficiency is discussed in the remaining part of this section.

Table VI. Per-document scoring time in μs of BWQS, QS and VPRED algorithms on ISTELLA.

Λ	Method	ISTELLA		
		Block size		Time
		δ	τ	
8	BWQS	8	10,000	**31.1** (–)
	QS	1	20,000	41.7 (1.3x)
	VPRED	16	20,000	158.3 (5.1x)
16	BWQS	8	5,000	**52.3** (–)
	QS	1	20,000	69.3 (1.3x)
	VPRED	16	20,000	326.9 (6.3x)
32	BWQS	2	5,000	**105.1** (–)
	QS	1	20,000	132.7 (1.3x)
	VPRED	16	20,000	678.2 (6.5x)
64	BWQS	1	3,000	**216.8** (–)
	QS	1	20,000	289.6 (1.3x)
	VPRED	16	20,000	1413.1 (6.5x)

Evaluating the performance of BWQS. Table VI reports the average per-document scoring time in μs of algorithms QS, VPRED, and BWQS. The experiments were conducted on the largest Istella dataset by fixing the number of trees to $20,000$ and varying Λ. Experiments on the other datasets are not reported here, but they exhibit similar behavior. It is worth noting that our QS algorithm can be thought as a limit case of BWQS, where the blocks are trivially composed of 1 document and the whole ensemble of trees. VPRED instead vectorizes the process and scores 16 documents at the time over the entire ensemble. With BWQS the sizes of document and tree blocks can be instead flexibly optimized according to the cache parameters. Table VI reports the best execution times, along with the values of $\delta \in \{1, 2, 4, 8, 16\}$ and $\tau \in \{1,000, 2,000, 3,000, 4,000, 5,000, 10,000, 20,000\}$ for which BWQS obtained such results.

The speedup of BWQS with respect to QS is 1.3x on every experiment. The comparison against VPRED shows now a larger improvement in the speedup, up to 6.5x. As expected, we see that smaller blocks should be used with a large number of leaves, due to the larger footprint of the BWQS data structure.

The reason of the speedups highlighted in the table are clearly visible from the plot in Figure 12, where the per-document per-tree average scoring time of BWQS and its cache misses ratio is reported. For these tests, the best setting reported in Table VI was used, i.e., $\delta = 1$ and $\tau = 3,000$. As already mentioned, the scoring time of QS is strongly correlated to its cache misses. The more the cache misses are, the more time the algorithm takes to score documents. On the other hand, the curve of the BWQS cache misses shows that the block-wise implementation incurs in a negligible number of cache misses. This cache-friendliness is directly reflected in the per-document per-tree scoring time, which is only slightly influenced by the number of trees of the ensemble. A back-of-the-envelope analysis of QS based on Table I shows that its memory footprint is approximatively $1,096 \cdot |\mathcal{T}|$ bytes for models with 64 leaves, versus $2,040 \cdot |\mathcal{T}|$ needed by VPRED. Recall that the hardware used in the experiments includes a 8 MB L3 cache, which means that QS may fit between $7,000$ and $8,000$ trees in cache memory. Indeed, Figure 12 shows that the number of cache misses for QS

starts increasing at $8,000$ trees. Regarding VPRED, Table V clearly reports that the number of cache misses is already significant with $5,000$ trees.

The impact of the reversal strategy is less significant for BWQS than the cache optimizations, because the number of false trees in the ensemble is between 5% and 10% in our experiments. On the other hand, the reversal strategy makes BWQS less sensitive to the data distribution. It guarantees that the number of visited nodes is at most half the number of nodes in the ensemble for any given dataset.

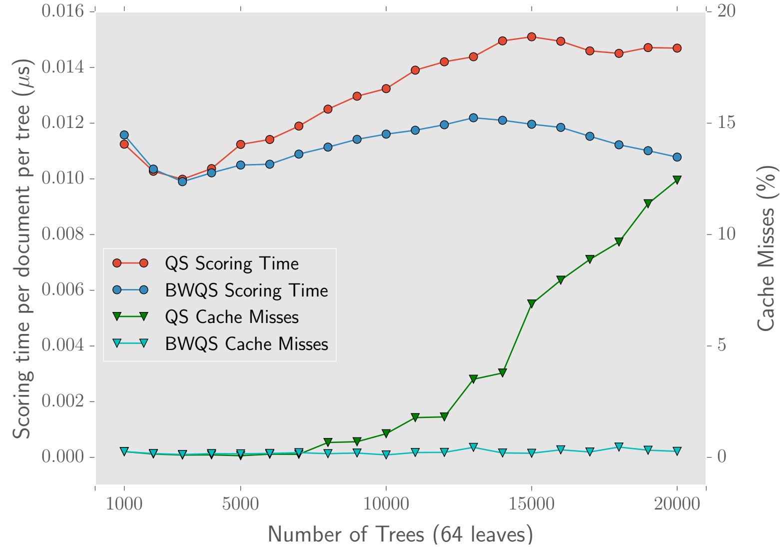

Fig. 12: Per-tree per-document scoring time in μs and percentage of cache misses of QS and BWQS on Istella with 64-leaves λ-MART models.

7. CONCLUSION

We presented a novel algorithmic framework aimed to efficiently score query results returned by large-scale Web search engines. In particular, the framework scores documents by exploiting machine-learned ranking functions modeled by state-of-the-art additive ensemble of oblivious and non-oblivious regression trees. While non-oblivious trees are binary decision tress without any constraints on the tests performed by each branching node, oblivious ones are balanced decision trees that performs the same test at each level of a tree. The last characteristic enables a more efficient tree traversal and is aimed at reducing overfitting. Specifically, we proposed two algorithms, QS and QS_Ω, for scoring with large ensemble of non-oblivious and oblivious trees, respectively.

The main contribution of QS and QS_Ω is a new representation of the tree ensemble based on bitvectors, where the tree traversal, aimed to detect the leaves that contribute to the final scoring of a document, is performed through efficient logical bitwise operations. In addition, the traversal is not performed one tree after another, as one would expect, but it is interleaved, feature by feature, over the whole tree ensemble.

Our tests conducted on publicly available LtR datasets confirm unprecedented speedups of QS (up to 6.6x) over the best state-of-the-art competitor. The motivations

of the very good performance figures of our QS algorithm are diverse. First, linear arrays are used to store the tree ensemble, while the algorithm exploits cache-friendly access patterns (mainly sequential patterns) to these data structures. Second, the interleaved tree traversal counts on an effective oracle that, with a few branch mis-predictions, is able to detect and return only the branching node in a tree whose conditions evaluate to FALSE. Third, the number of internal nodes visited by QS is in most cases consistently lower than in traditional methods, which recursively visits the small and unbalanced trees of the ensemble from the root to the exit leaf. All these remarks are confirmed by the deep performance assessment conducted by also analyzing low-level CPU hardware counters. This analysis shows that QS exhibits very low cache misses and branch mis-prediction rates, while the instruction count is consistently smaller than the counterparts. When the size of the data structures implementing the tree ensemble becomes larger than the last level of the cache (L3 in our experimental setting), we observed a slight degradation of performance.

Concerning oblivious trees, we also compared QS_{Ω} with a very efficient algorithm, OBLIVIOUS, specifically designed to take advantage of the features of oblivious trees. The last algorithm traverses trees without performing any branches, shares with QS_{Ω} the exploitation of fast logical bitwise operations, and has a very small memory footprint that advantages OBLIVIOUS when small tree ensembles are employed to score documents. On larger ensembles, QS_{Ω} outperform also OBLIVIOUS besides the other competitors, with speedups up to 3.6x.

To show that QS can be made scalable, we also presented BWQS, a block-wise version of QS that splits the sets of feature vectors and trees in disjoint blocks that entirely fit in the cache and can be processed separately. A novel optimization technique was introduced in BWQS that allows to further reduce the number of operations performed at scoring time by reversing the test condition in each node of the trees that contain more false than true nodes on average over a collection of training documents. Our experiments show that BWQS performs up to 1.3 times better than the original QS on large tree ensembles.

As future work, we plan to apply the same devised algorithm to other contexts, when a tree-based machine learned model must be applied to big data for classification/prediction purposes. Moreover, we aim at investigating whether we can introduce further optimizations in the algorithms, considering that the same tree-based model is applied to a multitude of feature vectors, and thus we could have the chance of partially reusing some work. Finally, we plan to investigate the parallelization of our method, which can involve various dimensions, i.e., the parallelization of the scoring task of each single feature vector, or the parallelization of the simultaneous scoring of many feature vectors. Some interesting results were achieved in [Lucchese et al. 2016] by exploiting the SIMD capabilities of modern microprocessors, while opportunities provided by general-purpose computing on graphics processing units (GPGPU) are still unexplored.

QUICKSCORER is currently patent pending [Dato et al. 2015] and its source code is available for research purposes under Non Disclosure Agreement with TISCALI ITALIA S.p.A.

REFERENCES

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.

Nima Asadi and Jimmy J. Lin. 2013. Training Efficient Tree-Based Models for Document Ranking. In *Proceedings of the 35th European Conference on Information Retrieval (ECIR)*. Springer, 146–157.

Christopher J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview*. Technical Report MSR-TR-2010-82.

Berkant 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 3rd International Conference on Web Search and Data Mining (WSDM)*. ACM, 411–420.

Gabriele Capannini, Domenico Dato, Claudio Lucchese, Monica Mori, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, and Nicola Tonellotto. 2015. QuickRank: a C++ Suite of Learning to Rank Algorithms. In *Proceedings of the 6th Italian Information Retrieval Workshop (IIR)*.

Gabriele Capannini, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, and Nicola Tonellotto. 2016. Quality versus efficiency in document scoring with learning-to-rank models. *Information Processing & Management* (2016). In press.

Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*. ACM. In press.

Domenico Dato, Claudio Lucchese, Franco Maria Nardini, Salvatore Orlando, Raffaele Perego, Nicola Tonellotto, and Rossano Venturini. 2015. A method to rank documents by a computer, using additive ensembles of regression trees and cache optimization, and search engine using such a method. *Tiscali S.p.A.* PCT29914, (pending) (2015).

Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of Statistics* (2001), 1189–1232.

Yasser Ganjisaffar, Rich Caruana, and Cristina Videira Lopes. 2011. Bagging Gradient-boosted Trees for High Precision, Low Variance Ranking Models. In *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 85–94.

Andrey Gulin, Igor Kuralenok, and Dmitry Pavlov. 2011. Winning the transfer learning track of Yahoo!'s learning to rank challenge with yetirank. In *Workshop and Conference Proceedings, JMLR*. 63–76.

Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated Gain-based Evaluation of IR Techniques. *ACM Transactions on Information Systems* 20, 4 (2002), 422–446.

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 (SIGIR)*. ACM, 629–638.

Ron Kohavi. 1994. Bottom-Up Induction of Oblivious Read-Once Decision Graphs: Strengths and Limitations. In *Proceedings of the 12th National Conference on Artificial Intelligence, (AAAI)*. AAAI Press, 613–618.

Pat Langley and Stephanie Sage. 1994. Oblivious decision trees and abstract cases. In *Working Notes of the AAAI-94 Workshop on Case-Based Reasoning*. AAAI Press, 113–117.

Tie-Yan Liu. 2009. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.

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 (SIGIR)*. ACM, 949–952.

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 (SIGIR)*. ACM, 73–82.

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 (SIGIR)*. ACM, 833–836.

David Patterson and John Hennessy. 2014. *Computer Organization and Design (5th ed.)*. Morgan Kaufmann.

Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Foundations and Trends in Information Retrieval* 3, 4 (2009), 333–389.

Ilya Segalovich. 2010. Machine learning in search quality at Yandex. Presentation at the industry track of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR). `http://download.yandex.ru/company/presentation/yandex-sigir.ppt`. (2010).

Toby Sharp. 2008. Implementing decision trees and forests on a GPU. In *Proc. Computer Vision 2008*. Springer, 595–608.

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 and Development in Information Retrieval (SIGIR)*. ACM, 1123–1126.

Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. 2012. Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?. In *Proceedings of the 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 232–239.

Paul Viola and Michael J. Jones. 2004. Robust Real-Time Face Detection. *International Journal of Computer Vision* 57, 2 (2004), 137–154.

Lidan Wang, Jimmy J. Lin, and Donald Metzler. 2010. Learning to efficiently rank. In *Proceeding of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 138–145.

Lidan Wang, Jimmy J. Lin, and Donald Metzler. 2011. A cascade ranking model for efficient ranked retrieval. In *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM, 105–114.

Lidan Wang, Donald Metzler, and Jimmy J. Lin. 2010. Ranking under temporal constraints. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management (CIKM)*. ACM, 79–88.

Qiang Wu, Christopher J.C. Burges, Krysta M. Svore, and Jianfeng Gao. 2010. Adapting boosting for information retrieval measures. *Information Retrieval* (2010).

Zhixiang Xu, Kilian Weinberger, and Olivier Chapelle. 2012. The Greedy Miser: Learning under Test-time Budgets. In *Proceedings of the 29th International Conference on Machine Learning (ICML)*. ACM, 1175–1182.

Dawei Yin, Yuening Hu, Jiliang Tang, Tim Daly Jr., Mianwei Zhou, Hua Ouyang, Jianhui Chen, Changsung Kang, Hongbo Deng, Chikashi Nobata, Jean-Marc Langlois, and Yi Chang. 2016. Ranking Relevance in Yahoo Search. In *Proceedings of the 22nd ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD)*. ACM. In press.

Received -; revised -; accepted -