VHT: Vertical Hoeffding Tree

Nicolas Kourtellis
Telefonica I+D, Spain
nicolas.kourtellis@telefonica.com

Gianmarco De Francisci Morales
QCRI, Qatar
gdfm@acm.org

Albert Bifet
Telecom ParisTech, France
albert@albertbifet.com

Arinto Murdopo
LARC-SMU, Singapore
arintom@smu.edu.sg

***Abstract*—IoT big data requires new machine learning methods able to scale to large size of data arriving at high speed. Decision trees are popular machine learning models since they are very effective, yet easy to interpret and visualize. In the literature, we can find distributed algorithms for learning decision trees, and also streaming algorithms, but not algorithms that combine both features. In this paper we present the Vertical Hoeffding Tree (VHT), the first distributed streaming algorithm for learning decision trees. It features a novel way of distributing decision trees via vertical parallelism. The algorithm is implemented on top of Apache SAMOA, a platform for mining big data streams, and thus able to run on real-world clusters. Our experiments to study the accuracy and throughput of VHT, prove its ability to scale while attaining superior performance compared to sequential decision trees.**

***Keywords*-distributed streaming decision tree, vertical parallelism, hoeffding tree, SAMOA, big data**

I. Introduction

Nowadays, we produce data with many of our daily activities as we interact with software systems, and this data generation can be seen as a *stream*. Extracting knowledge from these massive streams of data to create models, and using them, e.g., to choose a suitable business strategy, or to improve healthcare services, can generate substantial competitive advantages. Many applications need to process incoming data and react on-the-fly by using comprehensible prediction mechanisms (e.g., card fraud detection).

Streaming data analytic systems need to process and manage data streams in a fast and efficient way, due to the stringent restrictions in terms of time and memory imposed by the streaming setting. The input to the system is an unbounded stream of data instances arriving at high speed. Therefore, we need to use simple models that scale gracefully with the amount of data. Additionally, we need to let the model take the right decision at any time, online.

In this work we focus on *decision trees* which typically perform well in such tasks, but also allow interpretability due to easy visualization. A decision tree is a classic decision support tool that uses a tree-like model. In machine learning, it can be used for both classification and regression [4]. At its core, a decision tree is a model where internal nodes are tests on attributes, branches are possible outcomes of these tests, and leafs are decisions, e.g., a class assignment.

Decision trees, and in general tree-based classifiers, are widely popular, for several reasons. First, the model is very easy to interpret. It is easy to understand how the model reaches a classification decision, and the relative importance of features. Trees are also easy to visualize, and to modify according to domain knowledge. Second, prediction is very fast. Once the model is trained, classifying a new instance requires just a logarithmic number of very fast checks (in the size of the model). For this reason, they are commonly used in one of the most time-sensitive domains nowadays – Web search [5, 15]. Third, trees are powerful classifiers that can model non-linear relationships. Indeed, their performance, especially when used in ensemble methods such as boosting, bagging, and random forests, is outstanding [9].

Learning the optimal decision tree for a given labeled dataset is NP-complete even for very simple settings [13]. Practical methods for building tree models usually employ a greedy heuristic that optimizes decisions locally at each node [4]. In a nutshell, the greedy heuristic starts with an empty node (the root) as the initial model, and works by recursively sorting the whole dataset through the current model. Each leaf of the tree collects statistics on the distribution of attribute-class co-occurrences in the part of dataset that reaches the leaf. When all the dataset has been analyzed, each leaf picks the best attribute according to a *splitting criterion* (e.g., entropy or information gain). Then, it becomes an internal node that branches on that attribute, splits the dataset into newly created children leaves, and calls the procedure recursively for these leaves. The procedure usually stops when the leaf is pure (i.e., only one class reaches the leaf), or when the number of instances reaching the leaf is small enough. This recursive greedy heuristic is inherently a batch process, as it needs to process the whole dataset before taking a split decision. However, streaming variants of tree learners also exist.

The *Hoeffding tree* [8] (a.k.a. VFDT) is a streaming decision tree learner with statistical guarantees. In particular, by leveraging the Chernoff-Hoeffding bound [12], it guarantees that the learned model is asymptotically close to the model learned by the batch greedy heuristic, under mild assumptions. The learning algorithm is very simple. Each leaf keeps track of the statistics for the portion of the stream it is reached by, and computes the best two attributes according to the splitting criterion. Let ΔG be the difference between the value of the functions that represent the splitting criterion of these two attributes. Let ϵ be a quantity that depends on a user-defined confidence parameter δ, and that decreases with the number of instances processed. When $\Delta G > \epsilon$, then the currently best attribute is selected to split the leaf. The Hoeffding bound guarantees that this choice is the correct one with probability larger than $1 - \delta$.

Streaming algorithms are only one of the two main ways to deal with massive datasets, the other being distributed algorithms [6]. However, even though streaming algorithms are very efficient, they are still bounded by the limits of a single machine. Nowadays, the data itself is usually already distributed, and the cost of moving it to a single machine is too high. Furthermore, cluster computing with commodity servers is economically more viable than using powerful single machines, as testified by innumerable web companies [2]. Finally, "the largest problem solvable by a single machine will always be constrained by the rate at which the hardware improves, which has been steadily dwarfed by the rate at which our data sizes have been increasing over the past decade" [1].

For all the aforementioned reasons, the goal of the current work is to propose a tree learning algorithm for the streaming setting that runs in a distributed environment. By combining the efficiency of streaming algorithms with the scalability of distributed processing we aim at providing a practical tool to tackle the complexities of "big data", namely its velocity and volume. Specifically, we develop our algorithm in the context of Apache SAMOA [7], an open-source platform for mining big data streams.[1]

We name our algorithm the Vertical Hoeffding Tree (VHT). The *vertical* part stands for the type of parallelism we employ, namely, vertical data parallelism. Similarly to the original Hoeffding tree, the VHT features anytime prediction and continuous learning. Naturally, the combination of streaming and distributed algorithms presents its own unique challenges. Other approaches have been proposed for parallel algorithms, which however do not take into account the characteristics of modern, shared-nothing cluster computing environments [3].

Concisely, we make the following contributions:

- we propose VHT, the first distributed streaming algorithms for learning decision trees (Section III);
- in doing so, we explore a novel way of parallelizing decision trees via vertical parallelism;
- we deploy our algorithm on top of SAMOA, and run it on a real-world Storm cluster to test scalability and accuracy (Section IV);
- we experiment with large datasets of tens of thousands of attributes and millions of examples, and obtain high accuracy (up to 80%) and high throughput (offering up to $20\times$ speedup over sequential streaming solutions).

II. Related Work

The literature abounds with streaming and distributed machine learning algorithms, though none of these features both characteristics simultaneously. Reviewing all these algorithms is out of the scope of this paper, so we focus our attention on decision trees. We also review the few attempts at creating distributed streaming learning algorithms proposed so far. Discussion on relevant frameworks (e.g., StormMOA, Jubatus) is omitted due to space constraints [14].

[1] http://samoa.incubator.apache.org

One of the pioneer works in decision tree induction for the streaming setting is the Very Fast Decision Tree algorithm (VFDT) [8]. This work focuses on alleviating the bottleneck of machine learning application in terms of time and memory, i.e. the conventional algorithm is not able to process it due to limited processing time and memory. Its main contribution is the usage of the Hoeffding Bound to decide the number of data required to achieve certain level of confidence. This work has been the basis for a large number of improvements, such as dealing with concept drift [11] and handling continuous numeric attributes [10].

Ben-Haim and Tom-Tov [3] present an algorithm for building decision trees in a streaming and parallel setting. The distribution of attributes from the master process proceeds in a horizontal fashion, and the tree is built while taking advantage of histograms of the data maintained at the working processes. However, the horizontal splitting can quickly increase the memory overhead of the replicated model and challenge its scalability; we demonstrate this point in our experiments (*sharding* algorithm version).

Ye et al. [19] show how to distribute and parallelize Gradient Boosted Decision Trees (GBDT). The authors first implement MapReduce-based GBDT that employs horizontal data partitioning. Converting GBDT to MapReduce model is fairly straightforward. However, due to high overhead from HDFS as communication medium when splitting nodes, the authors conclude that MapReduce is not suitable for this kind of algorithm. The authors then implement GBDT by using MPI using vertical data partitioning by splitting the data based on their attributes. This partitioning technique minimizes inter-machine communication cost. This is the data partitioning strategy we also choose for the VHT.

While technically not a tree, Vu et al. [18] propose the first distributed streaming rule-based regression algorithm. The algorithm is in spirit similar to the VHT, as it uses vertical parallelism and runs on top of distributed SPEs. However, it creates a different kind of model and deals with regression rather than classification.

III. VHT Algorithm

In this section, we explain the details of our proposed algorithm, the *Vertical Hoeffding Tree*, which is a data-parallel, distributed version of the Hoeffding tree. We implement this algorithm in SAMOA. Due to space constraints, details on SAMOA can be found in our extended report [14]). First, we describe the parallelization and the ideas behind our design choice. Then, we present the engineering details and optimizations we employed to obtain the best performance.

A. Vertical Parallelism

Data parallelism is a way of distributing work across different nodes in a parallel computing environment such as a cluster. In this setting, each node executes the same operation on different parts of the dataset. Contrast this definition with task parallelism (aka pipelined parallelism), where each node executes a different operator and the whole dataset flows through each node at different stages. When applicable, data parallelism is able to scale to much larger

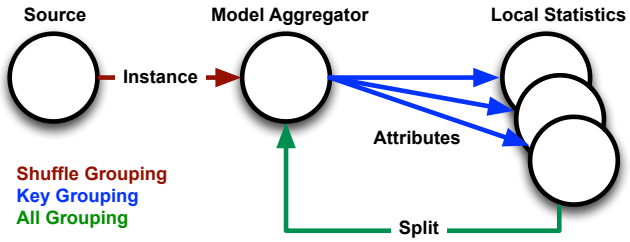

Figure 1: High level diagram of the VHT topology.

deployments, for two reasons: (i) data has usually much higher intrinsic parallelism that can be leveraged compared to tasks, and (ii) it is easier to balance the load of a data-parallel application compared to a task-parallel one. These attributes have led to the high popularity of the currently available DSPEs. For these reasons, we employ data parallelism in the design of VHT.

In machine learning, it is common to think about data in matrix form. A typical linear classification formulation requires to find a vector x such that $A \cdot x \approx b$, where A is the data matrix and b is a class label vector. The matrix A is $n \times m$-dimensional, with n being the number of data instances and m being the number of attributes of the dataset.

There are two ways to *slice* the data matrix to obtain data parallelism: by row or column. The former is called *horizontal parallelism*, the latter *vertical parallelism*. With horizontal parallelism, data instances are independently from each other, and can be processed in isolation while considering all available attributes. With vertical parallelism, instead, attributes are considered independent from each other.

The fundamental operation of the algorithm is to accumulate statistics n_{ijk} (i.e., counters) for triplets of attribute i, value j, and class k, for each leaf l of the tree. The counters for each leaf are independent, so let us consider the case for a single leaf. These counters, together with the learned tree structure, constitute the state of the VHT algorithm.

Different kinds of parallelism distribute the counters across computing nodes in different ways. With horizontal parallelism [3], the instances are distributed randomly, thus multiple instances of the same counter can exist on several nodes. On the other hand, when using vertical parallelism, the counters for one attribute are grouped on a single node.

This latter design has several advantages. First, by having a single copy of the counter, the memory requirements for the model are the same as in the sequential version. In contrast, with horizontal parallelism a single attribute may be tracked on every node, thus the memory requirements grow linearly with the parallelism level. Second, by having each attribute tracked independently, the computation of the split criterion can be performed in parallel by several nodes. Conversely, with horizontal partitioning the algorithm needs to (centrally) aggregate the partial counters before being able to compute the splitting criterion.

Of course, the vertically-parallel design has also its drawbacks. In particular, horizontal parallelism achieves a good load balance more easily, even though solutions for these problems have recently been proposed [16, 17]. In addition,

Algorithm 1 Model Aggregator: VerticalHoeffdingTreeInduction(E, VHT_tree)

Require: E is a training instance from source PI, wrapped in `instance` content event
Require: VHT_tree is the current state of the decision tree in model-aggregator PI
1: Use VHT_tree to sort E into a leaf l
2: Send `attribute` content events to local-statistic PIs
3: Increment the number of instances seen at l (which is n_l)
4: **if** $n_l \ mod \ n_{min} = 0$ **and** not all instances seen at l belong to the same class **then**
5: Add l into the list of splitting leaves
6: Send `compute` content event with the id of leaf l to all local-statistic PIs
7: **end if**

if the instance stream arrives in row-format, it needs to be transformed in column-format, and this transformation generates additional CPU overhead at the source. Indeed, each attribute that constitutes an instance needs to be sent independently, and needs to carry the class label of its instance. Therefore, both the number of messages and the size of the data transferred increase. Nevertheless, as shown in Section IV, the advantages of vertical parallelism outweigh its disadvantages for several real-world settings.

B. Algorithm Structure

We are now ready to explain the structure of the VHT algorithm. In general, there are two main parts to the Hoeffding tree algorithm: *sorting* the instances through the current model, and accumulating *statistics* of the stream at each leaf node. This separation offers a neat cut point to modularize the algorithm in two separate components. We call the first component *model aggregator*, and the second component *local statistics*. Figure 1 presents a visual depiction of the algorithm, specifically, of its components and of how the data flow among them.

The model aggregator holds the current model (the tree) produced so far. Its main duty is to receive the incoming instances and sort them to the correct leaf. If the instance is unlabeled, the model predicts the label at the leaf and sends it downstream (e.g., for evaluation). Otherwise, if the instance is labeled it is used as training data. The VHT decomposes the instance into its constituent attributes, attaches the class label to each, and sends them independently to the following stage, the *local statistics*. Algorithm 1 shows a pseudocode for the model aggregator.

The local statistics contain the sufficient statistics n_{ijk} for a set of attribute-value-class triplets. Conceptually, the local statistics can be viewed as a large distributed table, indexed by leaf id (row), and attribute id (column). The value of the cell represents a set of counters, one for each pair of attribute value and class. The local statistics accumulate statistics on the data sent by the model aggregator. Pseudocode for the update function is shown in Algorithm 2.

In SAMOA, we implement vertical parallelism by connecting the model to the statistics via key grouping. We use a composite key made by the leaf id and the attribute

Algorithm 2 Local Statistic: UpdateLocalStatistic($attribute$, $local\_statistic$)

Require: $attribute$ is an `attribute` content event
Require: $local\_statistic$ is the local statistic, could be implemented as $Table < leaf\_id, attribute\_id >$
1: Update $local\_statistic$ with data in $attribute$: attribute value, class value and instance weights

Algorithm 3 Local Statistic: ReceiveComputeMessage($compute$, $local\_statistic$)

Require: $compute$ is an `compute` content event
Require: $local\_statistic$ is the local statistic, could be implemented as $Table < leaf\_id, attribute\_id >$
1: Get leaf l ID from `compute` content event
2: For each attribute i that belongs to leaf l in local statistic, compute $\overline{G}_l(X_i)$
3: Find X_a^{local}, which is the attribute with highest $\overline{G}_l$ based on the local statistic
4: Find X_b^{local}, which is the attribute with second highest $\overline{G}_l$ based on the local statistic
5: Send X_a^{local} and X_b^{local} using `local-result` content event to model-aggregator PI via `computation-result` stream

id. Horizontal parallelism can similarly be implemented via shuffle grouping on the instances themselves.

Leaf splitting. Periodically, the model aggregator will try to see if the model needs to evolve by splitting a leaf. When a sufficient number of instances n_{min} have been sorted through a leaf, and not all instances that reached l belong to the same class (line 4, Algorithm 1), the aggregator will send a broadcast message to the statistics, asking to compute the split criterion for the given leaf id. The statistics will get the table corresponding to the leaf, and for each attribute compute the splitting criterion in parallel (an information-theoretic function such as information gain or entropy). Each local statistic will then send back to the model the top two attributes according to the chosen criterion, together with their scores ($\overline{G}_l(X_i^{local}), i = a, b$; Algorithm 3).

Subsequently, the model aggregator (Algorithm 4) simply needs to compute the overall top two attributes received so far from the available statistics, apply the Hoeffding bound (line 4), and see whether the leaf needs to be split (line 5). The algorithm also computes the criterion for the scenario where no split takes places ($X_\emptyset$). Domingos and Hulten [8] refer to this inclusion of a no-split scenario with the term *pre-pruning*. The decision to split or not is taken after a time has elapsed, as explained next.

By using the top two attributes, the model aggregator computes the difference of their splitting criterion values $\Delta\overline{G}_l = \overline{G}_l(X_a) - \overline{G}_l(X_b)$. To determine whether the leaf needs to be split, it compares the difference $\Delta\overline{G}_l$ to the Hoeffding bound ϵ for the current confidence parameter δ (where R is the range of possible values of the criterion). If the difference is larger than the bound ($\Delta\overline{G}_l > \epsilon$), then X_a is the best attribute with high confidence $1 - \delta$, and can therefore be used to split the leaf. If the best attribute is the no-split scenario ($X_\emptyset$), the algorithm does not perform any split. The algorithm also uses a tie-breaking τ mechanism

Algorithm 4 Model Aggregator: Receive($local\_result$, VHT_tree)

Require: $local\_result$ is an `local-result` content event
Require: VHT_tree is the current state of the decision tree in model-aggregator PI
1: Get correct leaf l from the list of splitting leaves
2: Update X_a and X_b in the splitting leaf l with X_a^{local} and X_b^{local} from $local\_result$
3: **if** $local\_results$ from all local-statistic PIs received or time out reached **then**
4: Compute Hoeffding bound $\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}}$
5: **if** $X_a \neq X_\emptyset$ **and** $(\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ **or** $\epsilon < \tau)$ **then**
6: Replace l with a split-node on X_a
7: **for all** branches of the split **do**
8: Add a new leaf with derived sufficient statistic from the split node
9: **end for**
10: Send `drop` content event with id of leaf l to all local-statistic PIs
11: **end if**
12: **end if**

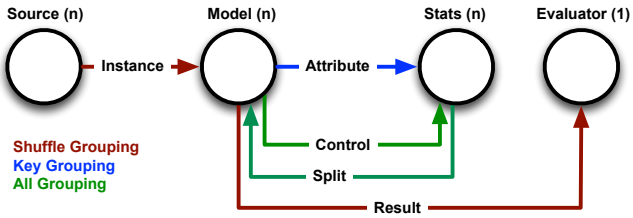

Figure 2: Deployment diagram for VHT.

to handle the case where the difference in splitting criterion between X_a and X_b is very small. If the Hoeffding bound becomes smaller than τ ($\Delta\overline{G}_l < \epsilon < \tau$), then the current best attribute is chosen regardless of the values of $\Delta\overline{G}_l$.

Two cases can arise: the leaf needs splitting, or it doesn't. In the latter case, the algorithm simply continues without taking any action. In the former case instead, the model modifies the tree by splitting the leaf l on the selected attribute, replacing l with an internal node (line 6), and generating a new leaf for each possible value of the branch (these leaves are initialized using the class distribution observed at the best attribute splitting at l (line 8)). Then, it broadcasts a `drop` message containing the former leaf id to the local statistics (line 10). This message is needed to release the resources held by the leaf and make space for the newly created leaves. Subsequently, the tree can resume sorting instances to the new leaves. The local statistics will create a new table for the new leaves lazily, whenever they first receive a previously unseen leaf id. In its simplest version, while the tree adjustment is performed, the algorithm drops the new incoming instances. We show in the next section an optimized version that buffers them to improve accuracy.

Messages. During the VHT execution, the type of events being sent and received from the different parts of the algorithm are summarized in Table I and Figure 2.

C. System Optimizations

Here we introduce some optimizations that improve the performance of the VHT: *optimistic split execution*, *instance*

Table I: Different type of content events used during the execution of the VHT algorithm.

Name	Parameters	From	To
`instance`	$<$ attribute 1, ..., attribute m, class $C>$	Source PI	Model-Aggregator PI
`attribute`	$<$ attribute id, attribute value, class $C>$	Model Aggregator	Local Statistic $id =<$ leaf id + attribute $id>$
`compute`	$<$ leaf $id>$	Model Aggregator	All Local Statistic s
`local-result`	$< \overline{G}_l(X_a^{local}), \overline{G}_l(X_b^{local}) >$	Local Statistic id	Model Aggregator
`drop`	$<$ leaf $id>$	Model Aggregator	All Local Statistic s

buffering, *timeout* and *model replication*. The first three deal with the problem computing the split criterion in a distributed environment and temporarily storing instances for improved instance learning, whereas the last one deals with the throughput and I/O capability of the algorithm, by removing its single bottleneck at the model aggregator. Due to space, we present details of the last opimization in [14].

Optimistic split execution. In the simplest version of the VHT algorithm, whenever the decision on splitting is being taken, labeled instances are simply thrown away. This behavior clearly wastes useful data, and is thus not desirable.

Note that there are two possible outcomes when a split decision is taken. If the algorithm decides to split the current leaf, all the statistics accumulated so far for the leaf are dropped. Otherwise, the leaf keeps accumulating statistics. In either case, the algorithm is better served by *using* the instances that arrive during the split. If the split is taken, the in-transit instances do not have any effect in any case. However, if the split is not taken, the instances can be correctly used to accumulate statistics.

Given these observations, we modify the VHT algorithm to keep sending instances that arrive during splits to the local statistics. We call this variant of the algorithm **wk(0)**.

Instance buffering. The feedback for a split from the local statistics to the model aggregator comes with a delay that can affect the performance of the model. While the model is waiting to receive this feedback from the local statistics to decide whether a split should be taken, the information from the instances that arrive can be lost if the node splits. To avoid this waste, we add a buffer to store instances in the model during a split decision. The algorithm can replay these instances if the model decides to split. That is, instances that arrive during a split decision are sent downstream and are accounted for in the current local statistics. If a split occurs, these statistics are dropped, and the instances are replayed from the buffer before resuming with normal operations. Conversely, if no split occurs, the buffer is simply dropped.

To avoid increasing the memory pressure of the algorithm, the buffer resides on disk. The access to the buffer is sequential both while writing and when reading, so it does not represent a bottleneck for the algorithm. We also limit the maximum size of the buffer, to avoid delaying newly arriving instances excessively. The optimal size of the buffer depends on the number of attributes of the instances, the arrival rate, the delay of the feedback from the local statistics, and the specific hardware configuration. Therefore, we let the user customize its size with a parameter z, and we refer to this version of the algorithm as **wk(z)**.

Timeout. Each model waits for a timeout to receive all responses back from the statistics, before computing the new splits. This timeout is primarily a system check to avoid the model waiting indefinitely. This timeout parameter may impact the performance of the tree: if it's too large, many instances will not be stored in the available buffer and therefore lost. Thus, the size of the buffer is closely related to this timeout parameter, allowing it to have enough instances to be replayed if the model has a leaf that decides to split.

IV. Experiments

In our experimental evaluation of the VHT method, we aim to study the following questions:

Q1: How does a centralized VHT compare to a centralized hoeffding tree (available in MOA) with respect to accuracy and throughput?

Q2: How does the vertical parallelism used by VHT compare to horizontal parallelism?

Q3: What is the effect of number and density of attributes?

Q4: How does discarding or buffering instances affect the performance of VHT?

A. Experimental setup

In order to study these questions, we experiment with five datasets (two synthetic generators and three real datasets), five different versions of the hoeffding tree algorithm, and up to four levels of computing parallelism. We measure classification accuracy during the execution and at the end, and throughput (number of classified instances per second). We execute each experimental configuration ten times, and report the average of these measures.

Synthetic datasets. We use synthetic data streams produced by two random generators for dense and sparse attributes.

Dense attributes are extracted from a random decision tree. We test different number of attributes, and include both categorical and numerical types. The label for each configuration is the number of categorical-numerical used (e.g, 100-100 means the configuration has 100 categorical and 100 numerical attributes). We produce 10 differently seeded streams with 1M instances for each tree, with one of two balanced classes in each instance, and take measurements every 100k instances.

Sparse attributes are extracted from a random tweet generator. We test different dimensionalities for the attribute space: 100, 1k, 10k. These attributes represent the appearance of words from a predefined bag-of-words. On average, the generator produces 15 words per tweet (size of a tweet is Gaussian), and uses a Zipf distribution with skew $z = 1.5$ to

select words from the bag. We produce 10 differently seeded streams with 1M tweets in each stream. Each tweet has a binary class chosen uniformly at random, which conditions the Zipf distribution used to generate the words.

Real datasets. We also test VHT on three real data streams to assess its performance on benchmark data.[2]

($elec$) Electricity. This dataset has 45312 instances, 8 numerical attributes and 2 classes.

(phy) Particle Physics. This dataset has 50000 instances for training, 78 numerical attributes and 2 classes.

($covtype$) CovertypeNorm. This dataset has 581012 instances, 54 numerical attributes and 7 classes.

Algorithms. We compare several versions of HT algorithm:

MOA: This is the standard Hoeffding tree in MOA.

local: This algorithm executes VHT in a local, sequential execution engine. All split decisions are made in a sequential manner in the same process, with no communication and feedback delays between statistics and model.

wok: This algorithm discards instances that arrive during a split decision. This version is the vanilla VHT.

wk(z): This algorithm sends instances that arrive during a split decision downstream. In also adds instances to a buffer of size z until full. If the split decision is taken, it replays the instances in the buffer through the new tree model. Otherwise, it discards the buffer, as the instances have already been incorporated in the statistics downstream.

sharding: Splits the incoming stream horizontally among an ensemble of Hoeffding trees. The final prediction is computed by majority voting. This method is an instance of horizontal parallelism applied to Hoeffding trees.

Experimental configuration. All experiments are performed on a Linux server with 24 cores (Intel Xeon X5650), clocked at 2.67GHz, L1d cache: 32kB, L1i cache: 32kB, L2 cache: 256kB, L3 cache: 12MB, and 65GB of main memory. On this server, we run a Storm cluster (v0.9.3), zookeeper (v3.4.6) and use SAMOA v0.4.0 (development version) and MOA v2016.04 available at the respective project websites.

We use several parallelism levels in the range of p=2, . . . , 16, depending on the experimental configuration. For dense instances, we stop at p=8 due to memory constraints, while for sparse instances we scale up to p=16. Results for p=4 and p=16 are shown in [14] due to space. We disable model replication (i.e., use a single model aggregator), as in our setup the model is not the bottleneck.

B. Accuracy and execution time of VHT local vs. MOA

In this first set of experiments, we test if VHT is performing as well as its counterpart hoeffding tree in MOA. This is mostly a sanity check to confirm that the algorithm used to build the VHT does not affect the performance of the tree when all instances are processed sequentially by the model. To verify this fact, we execute VHT local and MOA with both dense and sparse instances. Figure 3 shows that VHT local achieves the same accuracy as MOA, even besting it at times. However, VHT local always takes longer than MOA

[2]moa.cms.waikato.ac.nz/datasets/, osmot.cs.cornell.edu/kddcup/datasets.html

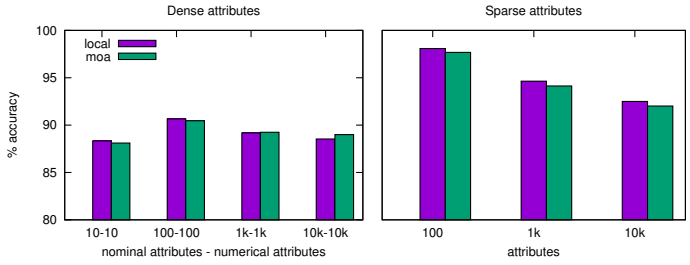

Figure 3: Accuracy of VHT executed in local mode on SAMOA compared to MOA, for dense and sparse datasets.

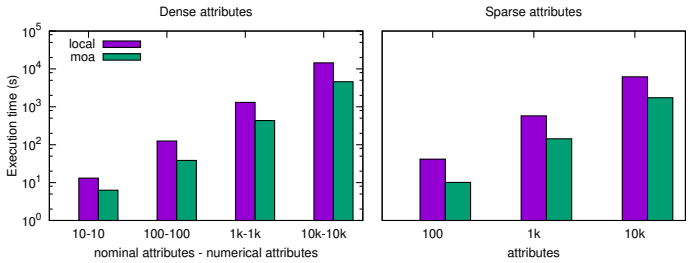

Figure 4: Execution time of VHT in local mode on SAMOA compared to MOA, for dense and sparse datasets.

to execute, as shown by Figure 4. Indeed, the local execution engine of SAMOA is optimized for simplicity rather than speed. Therefore, the additional overhead required to interface VHT to DSPEs is not amortized by scaling the algorithm out. Future optimized versions of VHT and the local execution engine should be able to close this gap.

C. Accuracy of VHT local vs. VHT distributed

Next, we compare the performance of VHT local with VHT built in a distributed fashion over multiple processors for scalability. We use up to p=8 parallel statistics, due to memory restrictions, as our setup runs on a single machine. In this set of experiments we compare the different versions of VHT, **wok** and **wk(z)**, to understand what is the impact of keeping instances for training after a model's split. Accuracy of the model might be affected, compared to the local execution, due to delays in the feedback loop between statistics and model. That is, instances arriving during a split will be classified using an older version of the model compared to the sequential execution. As our target is a distributed system where independent processes run without coordination, this delay is a characteristic of the algorithm as much as of the DSPE we employ.

We expect that buffering instances and replaying them when a split is decided would improve the accuracy of the model. In fact, this is the case for dense instances with a small number of attributes (i.e., around 200), as shown in Figure 5. However, when the number of available attributes increases significantly, the load imposed on the model seems to outweigh the benefits of keeping the instances for replaying. We conjecture that the increased load in computing the splitting criterion in the statistics further delays the feedback to compute the split. Therefore, a larger number of instances are classified with an older model, thus negatively affecting

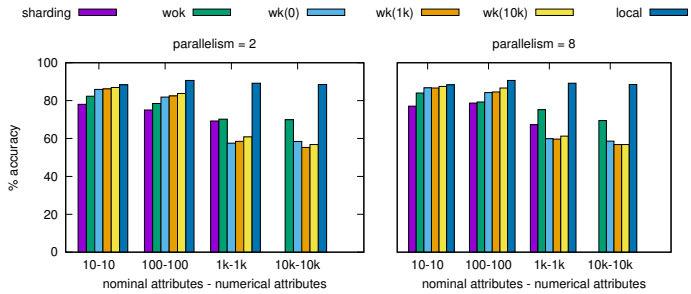

Figure 5: Accuracy of several versions of VHT (local, **wok**, **wk(z)**) and sharding, for dense datasets.

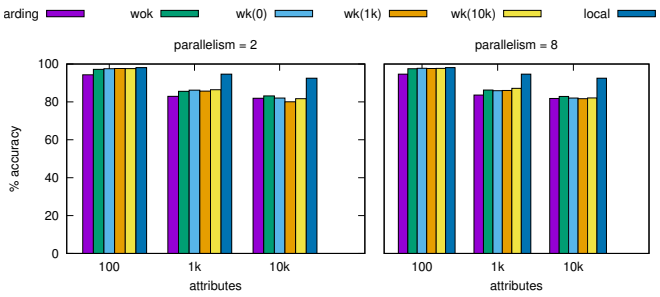

Figure 6: Accuracy of several versions of VHT (local, **wok**, **wk(z)**) and sharding, for sparse datasets.

the accuracy of the tree. In this case, the additional load imposed by replaying the buffer further delays the split decision. For this reason, the accuracy for VHT **wk(z)** drops by about 30% compared to VHT local. Conversely, the accuracy of VHT **wok** drops more gracefully, and is always within 18% of the local version.

VHT always performs approximatively 10% better than sharding. For dense instances with a large number of attributes (20k), sharding fails to complete due to its memory requirements exceeding the available memory. Indeed, sharding builds a full model for each shard, on a subset of the stream. Therefore, its memory requirements are p times higher than a standard hoeffding tree.

When using sparse instances, the number of attributes per instance is constant, while the dimensionality of the attribute space increases. In this scenario, increasing the number of attributes does not put additional load on the system. Indeed, Figure 6 shows that the accuracy of all versions is quite similar, and close to the local one. This observation is in line with our conjecture that the overload on the system is the cause for the drop in accuracy on dense instances.

We also study how the accuracy evolves over time. In general, the accuracy of all algorithms is rather stable, as shown in Figure 7 (similar results are found for dense attributes and are omitted for brevity). For instances with 10 to 100 attributes, all algorithms perform similarly. However, for a few thousand attributes per instance, the performance of the more elaborate VHT **wk(z)** drops, but VHT **wok** performs relatively well and better than sharding. This performance, coupled with good speedup over MOA (as

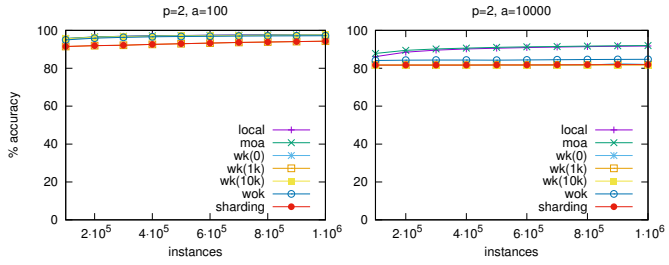

Figure 7: Evolution of accuracy with respect to instances arriving, for several versions of VHT (local, **wok**, **wk(z)**) and sharding, for sparse datasets.

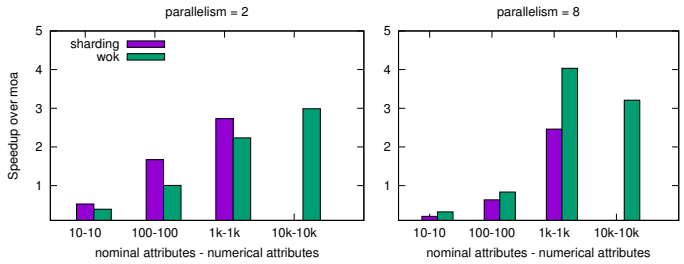

Figure 8: Speedup of VHT **wok** executed on SAMOA compared to MOA for dense datasets.

shown next) makes it a viable option for streams with a large number of attributes and a large number of instances.

D. Speedup of VHT distributed vs. MOA

Since the accuracy of VHT **wk(z)** is not satisfactory for both types of instances, next we focus on VHT **wok**. Figure 8 shows the speedup of VHT for dense instances. VHT **wok** is about 2-10 times faster than VHT local and up to 4 times faster than MOA. Clearly, the algorithm achieves a higher speedup when more attributes are present in each instance, as (i) there is more opportunity for parallelization, and (ii) the implicit load shedding caused by discarding instances during splits has a larger effect. Even though sharding performs well in speedup with respect to MOA on small number of attributes, it fails to build a model for large number of attributes due to running out of memory. In addition, even for a small number of attributes, VHT **wok** outperforms sharding with a parallelism of 8. Clearly, the vertical parallelism used by VHT offers better scaling behavior than the horizontal parallelism used by sharding.

When testing the algorithms on sparse instances, as shown in Figure 9, we notice that VHT **wok** can reach up to $60\times$ the throughput of VHT local and $20\times$ the one of MOA (for brevity we only show results with respect to MOA). Similarly to what observed for dense instances, a higher speedup is observed when a larger number of attributes are present for the model to process. This very large superlinear speedup ($20\times$ with p=2), is due to the aggressive load shedding implicit in VHT **wok**. The algorithm actually performs consistently less work than VHT local and MOA.

However, note that for sparse instances the algorithm processes a constant number of attributes, albeit from an increasingly larger space. Therefore, in this setup, **wok** has

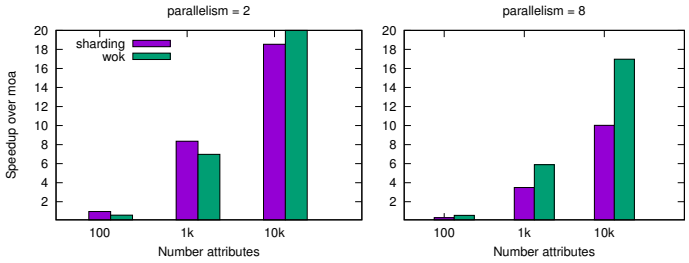

Figure 9: Speedup of VHT **wok** executed on SAMOA compared to MOA for sparse datasets.

a constant overhead for processing each sparse instance, differently from the dense case. VHT **wok** outperforms sharding in most scenarios and especially for larger numbers of attributes and larger parallelism.

Increased parallelism does not impact accuracy of the model (see Figure 5 and Figure 6), but its throughput is improved. Boosting the parallelism from 2 to 4 makes VHT **wok** up to 2 times faster. However, adding more processors does not improve speedup, and in some cases there is a slow-down due to additional communication overhead (for dense instances). Particularly for sparse instances, parallelism does not impact accuracy which enables handling large sparse data streams while achieving high speedup over MOA.

E. Performance on real-world datasets

Table II shows the performance of VHT, either running in a local mode or in a distributed fashion over a storm cluster of a few processors. We also test two different versions of VHT: **wok** and wk(0). In the same table we compare VHT's performance with MOA and sharding. The results from these real datasets demonstrate that with respect to accuracy, VHT can perform similarly to MOA and better than sharding, while at the same time process the instances faster (speedup results shown in [14] due to space).

Table II: Average accuracy (%) for different algorithms, with parallelism level (p), on the real-world datasets.

dataset	MOA	VHT					Sharding	
		local	**wok** p=2	**wok** p=4	wk(0) p=2	wk(0) p=4	p=2	p=4
elec	75.4	75.4	75.0	75.2	75.4	75.6	74.7	74.3
phy	63.3	63.8	62.6	62.7	63.8	63.7	62.4	61.4
covtype	67.9	68.4	68.0	68.8	67.5	68.0	67.9	60.0

V. Summary

In this paper we presented the Vertical Hoeffding Tree (VHT), the first distributed streaming algorithm for learning decision trees that can be used for performing classification tasks on large data streams arriving at high rates. VHT features a novel way of distributing decision trees via vertical parallelism. The algorithm is implemented on top of Apache SAMOA, a platform for mining big data streams, and is thus able to run on real-world clusters. Through exhaustive experimentation, and in comparison to a centralized sequential tree model, we showed that VHT can process dense and sparse instances with thousands of attributes up to $4\times$ and $20\times$ faster, respectively, and with small degradation in accuracy. In addition, VHT's ability to build the decision tree in a distributed fashion by using tens of processors allows it to scale and accommodate thousands of attributes and parse millions of instances. We also showed that competing distributed methods cannot handle the same data sizes due to memory and computational complexity.

VI. References

[1] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *arXiv preprint*, arXiv:1110, 2011.

[2] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[3] Y. Ben-Haim and E. Tom-Tov. A Streaming Parallel Decision Tree Algorithm. *JMLR*, Mar. 2010.

[4] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and regression trees*. CRC Press, 1984.

[5] O. Chapelle and Y. Chang. Yahoo! Learning to Rank Challenge Overview, 2011.

[6] G. De Francisci Morales. SAMOA: A Platform for Mining Big Data Streams. In *RAMSS @WWW*, 2013.

[7] G. De Francisci Morales and A. Bifet. SAMOA: Scalable Advanced Massive Online Analysis. *JMLR*, 2015.

[8] P. Domingos and G. Hulten. Mining high-speed data streams. In *SIGKDD*, pages 71–80, 2000.

[9] M. Fernández-Delgado and E. Cernadas. Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? *JMLR*, 2014.

[10] J. Gama, R. Rocha, and P. Medas. Accurate decision trees for mining high-speed data streams. In *SIGKDD*, 2003.

[11] J. Gama, I. Zliobaite, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, 2014.

[12] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13—-30, 1963.

[13] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

[14] N. Kourtellis, G. De Francisci Morales, A. Bifet, and A. Murdopo. VHT: Vertical Hoeffding Tree. *arXiv*, 1607.08325, 2016.

[15] S. Tyree, K. Q. Weinberger, K. Agrawal, and J. Paykin. Parallel boosted regression trees for web search ranking. In *WWW*, 2011.

[16] M. A. Uddin Nasir, G. De Francisci Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini. The Power of Both Choices: Practical Load Balancing for Distributed Stream Processing Engines. In *ICDE*, 2015.

[17] M. A. Uddin Nasir, G. De Francisci Morales, N. Kourtellis, and M. Serafini. When Two Choices Are not Enough: Balancing at Scale in Distributed Stream Processing. In *ICDE*, 2016.

[18] A. T. Vu, G. De Francisci Morales, J. Gama, and A. Bifet. Distributed adaptive model rules for mining big data streams. In *IEEE Big Data*, 2014.

[19] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng. Stochastic Gradient Boosted Distributed Decision Trees. In *CIKM*, 2009.