Knowl Inf Syst (2011) 29:249–272
DOI 10.1007/s10115-010-0342-8

REGULAR PAPER

The ClusTree: indexing micro-clusters for anytime stream mining

Philipp Kranen · Ira Assent · Corinna Baldauf · Thomas Seidl

Received: 5 March 2010 / Revised: 5 July 2010 / Accepted: 4 September 2010 /
Published online: 22 September 2010

Abstract Clustering streaming data requires algorithms that are capable of updating clustering results for the incoming data. As data is constantly arriving, time for processing is limited. Clustering has to be performed in a single pass over the incoming data and within the possibly varying inter-arrival times of the stream. Likewise, memory is limited, making it impossible to store all data. For clustering, we are faced with the challenge of maintaining a current result that can be presented to the user at any given time. In this work, we propose a parameter-free algorithm that automatically adapts to the speed of the data stream. It makes best use of the time available under the current constraints to provide a clustering of the objects seen up to that point. Our approach incorporates the age of the objects to reflect the greater importance of more recent data. For efficient and effective handling, we introduce the ClusTree, a compact and self-adaptive index structure for maintaining stream summaries. Additionally we present solutions to handle very fast streams through aggregation mechanisms and propose novel descent strategies that improve the clustering result on slower streams as long as time permits. Our experiments show that our approach is capable of handling a multitude of different stream characteristics for accurate and scalable anytime stream clustering.

Keywords Data mining · Clustering · Anytime · Stream mining

P. Kranen · C. Baldauf · T. Seidl
RWTH Aachen University, Aachen, Germany
e-mail: kranen@cs.rwth-aachen.de

C. Baldauf
e-mail: baldauf@cs.rwth-aachen.de

T. Seidl
e-mail: seidl@cs.rwth-aachen.de

I. Assent (✉)
Aarhus University, Aarhus, Denmark
e-mail: ira@cs.au.dk

1 Introduction

Analysis of streaming data is gaining importance as sensors or other data gathering devices are widely deployed. Streams constitute data values or tuples that need to be processed as they arrive. With the wide applicability of streaming data, clustering of streaming data has recently received much attention in data mining research. The goal is to cluster the objects within the stream continuously, such that there is always an up-to-date clustering of all objects seen so far. As opposed to clustering of a fixed data set that is available entirely prior to the data mining analysis, clustering of streaming data poses additional challenges.

Single pass clustering. In streaming environments, data arrives continuously. This means that clustering streams has to be performed in a single pass over the data in an online fashion.

Limited memory. Since data streams are assumed to be endless, storing each arriving object is simply not feasible. Any streaming clustering model has to adhere to memory constraints.

Limited time. The algorithm has to be able to keep up with the speed of the data stream. Clustering of the data cannot take longer than the average time between any two objects in the stream. Clustering has to keep up with the stream to always maintain a current clustering model.

Varying time allowances. Many streams do not show a constant flow of data, but constitute bursty streams. This means that the time available to process any item in the stream may vary greatly. Examples include peak times for customer transactions or seasonal changes in consumer behavior. Existing stream clustering algorithms are not capable of handling such varying time allowances, unless they were to resort to the minimal time allowance in the stream. Clearly, this means downgrading to the worst case assumption.

Evolving data. It is important to take into account that the model underlying the data in the stream may change over time. For example, consumption patterns during holidays may differ from those that are seen the rest of the year. To capture such phenomena, stream clustering should be capable of clearly identifying such changes. Denoted as concept drift, changes in clusters should be reported separately. Likewise, newly created clusters, so-called novelty, and outliers should be detected as such.

Flexible number and size of clusters. Many clustering algorithms, e.g. from the family of partitioning algorithms, require parametrization of the number of clusters to be detected. While setting such a parameter is also difficult in traditional clustering, streams undergo changes that may cause clusters to emerge, disappear, merge, or split. As such, setting a fixed number of clusters for the stream would distort the model. Existing stream clustering algorithms have to fix the size of their model in advance, e.g. through a maximum number of micro clusters [2], even though such knowledge is usually not available apriori.

In brief, clustering streams cannot be done using traditional algorithms for batch processing of data sets. Intuitively, it is not possible to stop the stream to perform analysis, or to indefinitely postpone display of results (e.g. analysis results on a dynamic website) in order to have the time to perform offline clustering. So, the naive solutions would be (i) to try and buffer the stream for later processing, which is not possible for endless streams of data, (ii) to drop random data in order to keep up with the stream speed, which means that valuable data is lost forever, or (iii) to settle for the worst case (the fastest imaginable stream speed), which means that only poor clustering results are obtainable and that the algorithm is idle most of the time. These naive solutions clearly do not make best use of the time available and of the information that the stream contains.

We propose a parameter-free stream clustering algorithm *ClusTree* that is capable of processing the stream in a single pass, with limited memory usage. It always maintains an up-to-date cluster model and reports concept drift, novelty, and outliers. Moreover, our approach makes no apriori assumption on the size of the clustering model, but dynamically *self-adapts*. For handling of varying time allowances, we propose an *anytime* clustering approach. Anytime algorithms are capable of delivering a result at any given point in time, and of using more time if it is available to refine the result. For clustering, this means that our algorithm is capable of processing even very fast streams, but also of using greater time allowances to refine the clustering model. Thus, our anytime clustering ClusTree overcomes the issues with traditional batch processing. We show that our algorithm can be combined with existing techniques for aging objects in the stream using decay functions, reporting cluster snapshots at different points in time, and comparing views at different points in time [2,3,25]. To the best of our knowledge, our approach is the first anytime clustering algorithm for streams.

2 Related work

There is a rich body of literature on stream data mining. Approaches for frequent itemset mining [10,11,21] and classification [1,15] have been proposed. Recently, clustering of data streams has been studied under different clustering paradigms. Convex stream clustering approaches are based on a k-center clustering. [23] processes the data stream in chunks and clusters each chunk into k clusters using either k-means or LSEARCH, a k-median variant. The final clustering is then generated by clustering the stored results from the chunks. If the available space is exceeded, the individual results from each chunk are merged via clustering to allow reusing the space for new chunks. [26] uses k-means clustering and additionally maintains a list of objects that do not fit the current clustering w.r.t. a "global boundary" [26]. A reclustering is started once that list becomes too large. [2] maintains a fixed number of micro-clusters and assigns a new object to the closest micro-cluster if it falls within its "maximum boundary" [2]. If no match can be found, either the most outdated cluster is deleted or the closest two clusters are merged. The final clustering is computed in an offline component using a k-center clustering on the micro-clusters. [3] does not explicitly fix the number of maintained clusters, but uses a fixed total number of dimensions, i.e., the clusters are maintained only in their most significant dimensions. As in [2], a new object is checked against each maintained cluster and eventually new clusters are created and the least recently updated cluster is deleted.

Micro-clusters (also called clustering features) have been introduced earlier for non-streaming contexts for speeding up clustering in large databases. In [34], a hierarchical index is maintained for faster access. For multidimensional streams, mapping to a frequent itemset representation is proposed in [5]. These approaches do not study streams nor do they provide anytime capabilities.

Detecting clusters of arbitrary shape in streaming data has been proposed using kernels [18], graphs [22], fractal dimensions [6] and density-based clustering [8,9]. [8] maintains a number of micro-clusters in an online component and applies a common density- based clustering on these micro-clusters in an offline component. [9] and [6] both follow a grid-based approach and store the density for populated grid cells in an online component. [9] finds clusters of arbitrary shape by combining neighboring cells with a density-based approach, while [6] uses the fractal dimension of a cluster to determine the cells that belong to it. In Sect. 3.5, we discuss how our technique can be flexibly combined with these approaches.

None of the above approaches allows for anytime clustering nor for adapting the clustering model to the stream speed in an online fashion. Anytime algorithms denote approaches that are capable of delivering a result at any given point in time, and of using more time if it is available to refine the result. Anytime data mining algorithms such as top k processing [4], anytime learning [27,32] and anytime classification [12,19,20,24,29,33] are a very active field of research. An anytime clustering approach for time series has been proposed in [31], which is not directly targeted at streaming data. Our approach thus constitutes the first anytime stream clustering algorithm.

Finally, different approaches have been proposed to present an up-to-date view on the clustering result or to determine and visualize concept drift and novelty in stream clustering. [30] focuses on detecting a change in the underlying stream by means of the minimal description length. The pyramidal time frame proposed in [2] enables the view on clusterings of arbitrary time horizons defined by the user. [25] categorizes cluster transitions into internal and external transitions and describes how to detect these. [3,18,28] employ an exponential decay function to weigh down the influence of older data, thus focusing on keeping an up-to-date view of the data distribution. Tracking of clusters in data streams is discussed in [35]. We show the compatibility of our approach to the methods from [2,3,25] in Sect. 3.5.

3 Self-adaptive anytime stream clustering

We propose self-adaptive anytime stream clustering that relies on an index structure for storing and maintaining a compact view of the current clustering. The size of our clustering model automatically adapts to the stream speed, which cannot be achieved by any buffering outside the storage structure. Moreover, we do not want to drop any data object, but want to preserve a complete model. Hence, any object from the stream is inserted into the index and possibly merged with aggregates of previously inserted objects. We describe strategies for dealing with varying time constraints for anytime clustering, i.e., the possibility of interrupting the process at any given point in time.

The structure of the main part is as follows. In Sect. 3.1, we define the data structure of the ClusTree and describe how we realize the anytime insertion. In Sect. 3.2, we explain how aging of older objects is incorporated into our anytime algorithm and provide a proof for the ClusTree invariant. Dealing with exceptionally fast streams and making best use of additional time on slower streams is described in Sections 3.3 and 3.4, respectively. Section 3.5 details how different cluster shapes and cluster transitions, e.g. novelty or drift, can be handled using the ClusTree. Finally, we summarize the solutions and benefits as well as an overview of the ClusTree algorithm in Sect. 3.6.

3.1 The ClusTree—Micro-clusters and anytime insert

Our approach is based on micro-clusters as compact representations of the data distribution. By maintaining measures for incremental computation of mean and variance of micro-clusters, the infeasible access to all past stream objects is no longer necessary.

Micro-clusters are a popular technique in stream clustering or scaling clustering to large data sets [2,3,34] to create and maintain compact representations of the current clustering. Instead of storing all incoming objects, a cluster feature tuple $CF = (n, LS, SS)$ of the number n of represented objects, their linear sum LS, and their squared sum SS is maintained. This tuple suffices for computing mean and variance and can be incrementally updated. Any cluster feature (CF) then represents a micro-cluster, i.e., a set of objects

and the main characteristics of its distribution. With this, objects can be easily assigned to the most similar micro-cluster incrementally. Existing micro-cluster approaches lack support for varying stream inter-arrival times. It is therefore crucial to provide the means for anytime clustering and self-adaptation to stream speed. We propose maintaining cluster features (CFs) by extending index structures from the R-tree family [7,16,24]. Such hierarchical indexing structures provide the means for efficiently locating the right place to insert any object from the stream into a micro-cluster. The idea is to build a hierarchy of micro-clusters at different levels of granularity. Given enough time, the algorithm descends the hierarchy in the index to reach the leaf entry that contains the micro-cluster that is most similar to the current object. If this micro-cluster is similar enough, it is updated incrementally by this object's values. Otherwise, a new micro-cluster may be formed.

The important observation for anytime clustering of streaming data, however, is that there might not always be enough time to reach leaf level to insert the object. We therefore provide novel strategies for anytime inserts.

There are several possibilities for handling object arrivals before the current object insert reaches leaf level. The straightforward solution keeps a **global queue**. This approach is very simple, but it may require an infinite buffer. And we may never have the time to empty the queue, resulting in outdated clustering results. To reduce memory consumption, one could maintain a **global aggregate**, i.e., instead of the queue a single cluster feature. However, aggregating arbitrary objects loses too much information as they might be diverse. To maintain the necessary information for clustering, and to ensure that any newly arriving object can be inserted at once, we propose interrupting the insertion process. The object has to be temporarily stored in a **local aggregate** from which we can continue at a later time. This yields foreseeable space demands like with a global aggregate, albeit slightly larger ones. For the invested space, we obtain a greater accuracy. The great advantage of local aggregates over local queues is that we can easily use the time for regular inserts to take a buffered local aggregate along as a "hitchhiker". Moreover, they can be naturally integrated into the tree structure. We will discuss this in more detail shortly, after describing the structure of our ClusTree hierarchical index for maintaining the micro-cluster information.

Our ClusTree approach consists of a hierarchy of entries that describe the cluster feature properties of their respective subtrees. The structure of an inner entry and a leaf entry is illustrated in the left part of Fig. 1: Each entry contains a cluster feature of the number n of objects that were aggregated, their dimension-wise linear sum LS, and squared sum SS, as well as a pointer to the respective subtree. We propose integrating local aggregates into the tree structure as temporary entries, so, additionally, an inner entry provides a buffer b for temporary insertions of local aggregates (CFs). Leaf nodes' entries do not contain a buffer, since inserts at leaf level are final.

Definition 3.1 (*ClusTree*). A ClusTree with fanout parameters m, M and leaf node capacity parameters l, L is a balanced multi-dimensional indexing structure with the following properties:

- an inner node $node_s$ contains between m and M entries. Leaf nodes contain between l and L entries. The root has at least one entry.
- an entry in an inner node of a ClusTree stores:
  - a cluster feature of the objects it summarizes.
  - a cluster feature of the objects in the buffer. (May be empty.)
  - a pointer to its child node.

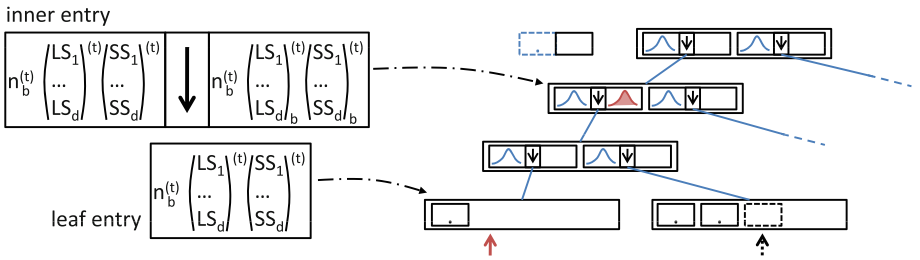

Fig. 1 Inner node and leaf node structure (*left*). Insertion object, hitchhiker and buffer (*right*)

– an entry in a leaf of a ClusTree stores a cluster feature of the object(s) it represents.
– a path from the root to any leaf node has always the same length (balanced).

The tree is created and updated like any multidimensional index such as R-tree, R*-tree, etc. [16,7,24]. Unlike the minimum bounding rectangles that they maintain in addition to the objects, we store only CFs in the ClusTree. For insertion, we descent into the subtree with the closest mean with respect to Euclidean distance. Splitting is based on pairwise distances between the entries, where entries are combined into two groups such that the sum of the intra-group distances is minimal. We will show in Sect. 4 that $M = 3$ is a good choice; hence, there are maximally six pairwise distances per node yielding a fast split operation.

The important property that reflects anytime capability of the ClusTree is its buffer in each entry. It serves as a temporary storage place of aggregates or objects that do not reach leaf level during insertion. Whenever insertion is interrupted, the current CF is simply stored in the buffer of the entry that corresponds to the subtree into which to descend next. At any future time when this subtree is next accessed, the temporary entry in the buffer is taken along as a "hitchhiker". This makes sure that future descent down the same subtree is used for continuing the insertion process. Whenever the descent destinations of the current insertion CF and the hitchhiker differ, the latter is placed in the corresponding buffer again to wait for the next ride down the tree.

The right part of Fig. 1 illustrates this process. Assume that the insertion object (drawn blue in the dashed box to the left of the root) belongs to the leaf that is marked by the dashed arrow (at the second leaf). Assume also, that the leftmost entry on the second level has a filled buffer (second distribution symbol in the entry), which belongs to a different leaf than the insertion object (indicated by the red solid arrow at the first leaf). The insertion object first descends to the second level and next descends into the left entry. It picks up the left entry's buffer in its buffer CF for hitchhikers (depicted as the solid box at the right of the insertion object). The insertion object descends to level three, taking the hitchhiker along. Because the hitchhiker and the insertion object belong to different subtrees, the hitchhiker is stored in the buffer of the left entry on the third level (to be taken along further down in the future) and the insertion object descends into the right entry alone to become (part of) a leaf entry.

Our buffer concept and the algorithmic idea of taking hitchhikers along are key to our anytime clustering algorithm. It allows the algorithm to be interrupted at any point in time and making best use of future descents down the tree. Moreover, unlike global aggregates, objects are kept separate as long as time permits.

When a leaf node is reached and the insertion would cause a split, the algorithm checks whether there is still time left. If there is no time for a split, the closest two entries are merged. For tracking of concept drift, novelty, etc. in the output clustering, leaf node entries contain

a unique id. When they are created, they are assigned a unique number in increasing order. When entries are merged, this is recorded as a pair of ids in a merging list.

The ClusTree can be initialized to improve the starting structure of the tree. Given an initial set of objects, each is transformed to a new CF. The leafs and internal nodes may be ordered for best structural properties through recursive top-down partitioning along the dimension with the largest variance and such that each partition contains equally many objects. Or any clustering algorithm, e.g. expectation maximization (EM) [13] or k-means, can be used to group the objects in a top-down or bottom-up fashion to initialize the tree. Please note that our focus is not on optimizing the initialization phase, and our experiments are performed without it.

3.2 Maintaining an up-to-date clustering

In order to maintain an up-to-date view, we would like new objects to be more important than older objects. A common solution is to weigh objects with an exponential time-dependent decay function $\omega(\Delta t) = \beta^{-\lambda \Delta t}$. The decay rate λ controls how much more one favors new objects compared to old ones. The higher λ is, the faster the algorithm "forgets" old data. We chose to set $\beta = 2$. For this basis, the half life of objects is $\frac{1}{\lambda}$.

To incorporate decay, temporal information has to be added to the ClusTree nodes. We ensure that the inner entries of the ClusTree still summarize their subtrees accurately by making elements of a cluster feature vector dependent on the current time t:

$$n^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i), \quad LS^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i) \cdot x_i,$$
$$SS^{(t)} = \sum_{i=1}^{n} \omega(t - ts_i) \cdot x_i^2$$

n denotes the (unweighted) number of contributing objects and ts_i is the timestamp at which object x_i was added to the CF.

We know that additive properties of cluster features are preserved, and also temporal multiplicity [3]: If no object is added to a $CF^{(t)}$ during the time interval $[t, t + \Delta t]$, then

$$CF^{(t+\Delta t)} = \omega(\Delta t) \cdot CF^{(t)}.$$

Details on this property and the corresponding proof can be found in [3].

Each insertion object x carries the timestamp ts_x of its arrival time. Furthermore, each entry e_s has a timestamp $e_s.ts$ specifying its last update. We use it to compute the time that passed between the last update of an object and t_x, which is the input of the decay function. Upon descending into a node, we update all entries e_s in the node to t_x by position-wise multiplication with the decay function and resetting the timestamp: $e_s.CF \leftarrow \omega(t_x - e_s.\text{ts}) \cdot e_s.CF$, $e_s.\text{buffer} \leftarrow \omega(t_x - e_s.\text{ts}) \cdot e_s.\text{buffer}$, $e_s.\text{ts} \leftarrow t_x$. Please note that entries in the same node always have the same timestamp, as we update all entries in the node we descend into.

We now show that inner entries summarize their subtrees correctly. We derive an invariant that incorporates the time aspect. The cluster feature of a parent entry e_s that was last updated at $t + \Delta t$ equals the sum of the CFs of the entries in its child node updated from time t of their last update to the parent's time plus the parent's buffer.

Lemma 3.1 (ClusTree Invariant) *For each inner entry e_s with timestamp $t + \Delta t$ and decay function $\omega(\Delta t) = 2^{-\lambda \Delta t}$ it holds*

$$e_s.CF^{(t+\Delta t)} = \left(\omega(\Delta t) \cdot \sum_{i=1}^{\nu_s} e_{soi}.CF^{(t)} \right) + e_s.buffer^{(t+\Delta t)}$$

Proof Each inner entry e_s is created first due to a split. Its summary is calculated directly as the sum of the cluster features in its child node entries e_{soi}. The child node entries are all on the same time, because we update all entries in a node. The timestamp of the children is the insertion time t of the object x that caused the split. There can only be a change in one of the e_{soi}, if there was first a change in e_s, because we always start from the root and descend downward.

Take the case of updating parent entry e_s (with filled buffer) to the new time $t + \Delta t + \Gamma t$, and addition of object y, where y descends into $node_s$ and gives the buffer a lift. Upon descending into $node_s$, all its entries e_{soi} are updated and y is added to the CF of exactly one of the $e_{soi}.e_s$ has a buffer, which y takes along. The buffer is also added to exactly one of the child entries' cluster features.

Following the above reasoning, we know that after updating e_s it holds that:

$$e_s.CF^{(t+\Delta t+\Gamma t)} = \left(\omega(\Delta t + \Gamma t) \cdot \sum_{i=1}^{\nu_s} e_{soi}.CF^{(t)} \right) + e_s.\text{buffer}^{(t+\Delta t+\Gamma t)}.$$

Because y descends into $node_s$, we update the child entries:

$$= \sum_{i=1}^{\nu_s} e_{soi}.CF^{(t+\Delta t+\Gamma t)} + e_s.\text{buffer}^{(t+\Delta t+\Gamma t)}.$$

Now we give $e_s.\text{buffer}^{(t+\Delta t+\Gamma t)}$ a lift. Afterward $e_s.\text{buffer}^{(t+\Delta t+\Gamma t)}$ contains zeros, and the values that it held before are added to the cluster feature of one of the child entries. Also adding y on both sides of the equation, once to $e_s.CF$ and once to the CF of one of the child node entries, leaves the invariant unchanged. This is also true for "hitchhiking" objects temporarily in a buffer o (replace $y.CF^{(t+\Delta t+\Gamma t)}$ with $y.CF^{(t+\Delta t+\Gamma t)} + o.CF^{(t+\Delta t+\Gamma t)}$), and if $node_s$ is a leaf node.

The last case in which we need to check violate the invariant is a split. Let us consider the split of a leaf node $node_{soi}$. Then two summaries in $node_s$ are computed from scratch. One overwrites the existing entry e_{soi} that pointed to the split node. The other one is the start of a new entry. The two new summaries naturally fulfill the invariant. The invariant also holds true for e_s, the entry pointing to $node_s$, because only the distribution of the summaries changed on the levels below e_s, not the total of the values. □

Thus, maintaining a single additional field in each node with a timestamp value of its last update, and weighing according to the above scheme, ensures that decay with time is correctly captured. Note that weighing does not require additional memory; the weighted CFs simply replace the non-weighted cluster features in Definition 3.1.

Weighing with time provides us with an interesting way of avoiding splits, to save valuable time. If a node is about to be split, our algorithm checks whether the least significant entry can be discarded, because it no longer contributes significantly to the clustering. Assuming

that a snapshot of the ClusTree is taken regularly after t_{snap} time, the significance is tested by checking whether the entry $\hat{e}$ with the smallest $n_{\hat{e}}^{(t)}$ satisfies

$$n_{\hat{e}}^{(t)} < \beta^{-\lambda \cdot t_{\text{snap}}} \tag{1}$$

If this is the case, $\hat{e}$ is discarded, making room for the entry to be inserted, and avoiding a split. The summary statistics of $\hat{e}$ are subtracted from the corresponding path up to the root. Note that according to Eq. 1, no entry is discarded if a new object has been added to it after the last snapshot has been taken. Moreover, Eq. 1 guarantees that each entry is stored in at least one snapshot.

We discuss in Sect. 3.5 how the ClusTree results can be used to detect clusters of arbitrary shape and time horizon. Moreover, applicability of recent approaches to concept drift detection is shown.

3.3 Dealing with very fast streams—Speed-up through aggregation

What happens to our index structure when it faces an exceptionally fast data stream? If insertion is interrupted at the top levels most of the time, the root and upper levels of the tree aggregate a lot of objects in their buffers that have little chance of getting a lift down to a leaf. Worse yet, dissimilar objects which belong to different subtrees and leaves become inseparable in a buffer. The quality of our results is bound to deteriorate if we are constantly interrupted on higher levels.

We propose a speed-up through aggregation before insertion: If we do not insert each object individually, there is more time to descend deeper with an aggregate of objects. Naively, one could add up a certain number m of incoming objects, insert the aggregate, sum up the next m objects, and so on. This is essentially a global aggregate with the problem of merging arbitrary objects, even very dissimilar ones, to the same aggregate.

Clearly we need to exercise some control over which objects should—literally—"go together". Ideally, we want to aggregate objects that would end up in the same leaf if we could descend the tree with them. Most probably, the arriving objects are not all similar to each other, but we expect subgroups with inter-similarity—representatives of the clusters we also find in the tree.

Our solution is to create several aggregates for dissimilar objects. This makes sure that the objects summarized in the same aggregate are similar. To this end, we set a $\max_{\text{radius}}$ for the maximum distance of objects in the aggregate. $\max_{\text{radius}}$ does not need to be set by the user. We propose determining its value from the leaf level, as the average variance of the leaves. This way, the aggregates for fast streams do not deteriorate the quality of the tree disproportionally.

For very fast streams, we store interrupted objects in their closest aggregate with respect to the distance to the mean, if this distance is below $\max_{\text{radius}}$. If $\max_{\text{radius}}$ is exceeded, we open up a new aggregate. We insert aggregates, just as we insert single objects. Whenever the insertion thread is idle, it simply picks the next aggregate (ordered first by the number of objects in the aggregates and then by their age). The number of aggregates is limited by the stream speed, i.e., it cannot exceed the number of distance computations that can be done between two arriving items. In the case of a varying data stream, the maximum number of aggregates has to be set by the user, constituting the only parameter of our approach. The aggregation is done by a different thread, i.e., the insertion of aggregates works in parallel and is not affected in terms of processing time. If no aggregate violates the max-radius constraint, the fullest aggregate is inserted. If several aggregates are equally full, the oldest of these is inserted.

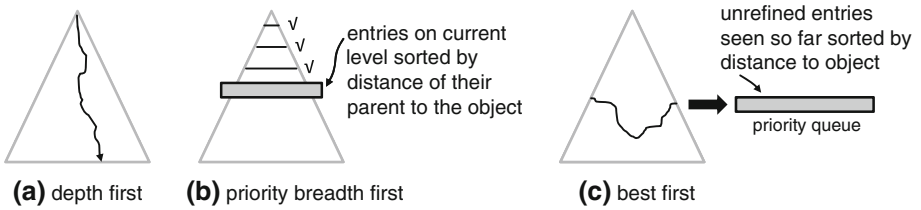

Fig. 2 Descent strategies depth first, priority breadth first and best first

3.4 Working on slower streams—making better use of time through alternative descent strategies

In this section, we suggest alternative strategies for inserting objects into micro-clusters. Assuming the insertion process of an object is not interrupted, so far we have continued down a single path. This path corresponds to always picking the child with the respective smallest distance between the object and the children reachable from the current node. This descent strategy down the tree can be considered a single-try *depth first* approach. This means that we continue down a path that has been chosen, and do not reconsider, i.e., we do not explore continuing a path that branches further up the tree. The advantage of this approach is that we spend the (unknown) time available to the anytime insertion process on trying to reach a level as far down the tree as possible (cf. Fig. 2). The further down the tree we insert the object, the more fine-grained the resolution of the micro-clusters. When the insertion process reaches the leaf level and additional time is available, the leaf is split and hence the model size is automatically adapted to the stream speed.

For very fast streams, we prevent frequent insertions on higher levels through our aggregation mechanisms (cf. Sect. 3.3). For slower data streams, however, we are very likely to often reach the leaf level and hence the model, i.e., the tree, continues to grow. As stated in the introduction, stream clustering algorithms naturally have to cope with limited memory, i.e., there is a maximal model size, either dictated through the available memory or given by the user or the context program. Continuous growth of the model is thus not ideal if more time is available.

Employing the proposed depth first strategy in this case would leave the algorithm idle once the leaf level is reached. And such idle times actually occur. One of the reasons is that our anytime processing is fast, so it reaches the leaf level after few computation steps. In the ClusTree algorithm, the number of distance computations is only logarithmic in the number of maintained micro clusters, so time for further model improvement is often available even for larger model sizes. As we will see in the experimental Sect. 4.3, maintaining 400,000 micro clusters at a stream speed of 50,000 points per second already yields idle times, which can be used for further computations and improvements of the clustering result.

In the following, we therefore explore alternative ways of choosing paths down the tree, as well as ways on how to spend any time that might still be available after reaching leaf level due to small model size and variation in object inter-arrival times.

3.4.1 Priority breadth first traversal

Our first alternative descent strategy is a priority breadth first descent. The single path depth first descent does not perform any kind of backtracking and hence cannot correct any misguided choice that might occur due to overlapping entries on higher levels of the tree. In

other words, even if we chose the closest entry on level k, we cannot be sure that the closest entry on level $k+1$ is one of its children. To assure finding the closest entry on each level, we therefore propose to evaluate all entries per level. While doing this, we sort the entries by the distance of their corresponding parent to quickly find the closest option. Figure 2b illustrates the priority breadth first descent: instead of checking a single node on each level as in depth first descent, each level is evaluated to find the closest entry by going through a list of entries sorted by the distance between the parent node and the current insertion object.

While at first the breadth first traversal sounds like a lot of additional computations compared to linearly checking against all maintained micro clusters as in e.g. [2,8], a closer look reveals advantages of this strategy. In the case of a binary tree, the number of non-leaf entries is about equal to the number of leaf entries. However, for a higher fanout, the number of inner entries is relatively smaller. Taking the fact into account that we sort the entries according to the distance of their corresponding parent entry, we are likely to find the closest leaf level entry, i.e., the closest maintained micro cluster, within the first entries of the priority queue on the leaf level. Moreover, we are still able to perform anytime clustering, since we always use the buffering strategies described above (cf. Sect. 3.1). The only change is that the entries on the final path are updated at the time of interruption and not as we go down the path. That means that we add the number, linear and quadratic sum when the object is inserted. Since maintaining 50,000 micro clusters with a fanout of 3 yields a tree height of 10, the number of operations (additions) is negligible.

3.4.2 Best first traversal

As mentioned before, the underlying idea for our alternative descent strategies is based on the observation that by descending the tree depth first, we basically use a greedy approach that is not able to revise the decision for any given subtree. However, it is possible that the aggregate information on upper levels in the tree is misleading. Misleading in the sense that we might find at lower levels that an entry that had a short distance to the current object actually separates into micro-clusters at lower levels that have a comparatively high distance to the object.

In such a situation, it might be beneficial for the structure of the obtained micro-clusters to not continue on this path, but instead evaluate the situation at the children of the next-best choice. This means that we need to keep track of which options existed on the current path. This allows us to decide whether one of the branches further up on the current path has a distance to the object that is smaller than what we see for the current entry. If this is the case, we can go back and follow a different path. In query processing on indexing structures, this approach corresponds to a *best first* strategy [7,16,24]. To implement it, we need to maintain a priority queue while making the descent down the tree.

The priority queue contains the entries seen so far that have not been refined yet and their corresponding distance to the insertion object. Given the time to make the next step in descending down the tree, the best first approach always takes the first element from the queue, i.e., the entry which has the smallest distance to the object. The distances from the object to the entries in the corresponding child node are computed and inserted into the priority queue (refinement). This process continues until insertion is interrupted or until all nodes have been visited. As in the priority breadth first descent we keep the property of anytime clustering and update the path with cluster features (CFs) when we buffer or insert the object on interruption.

The best first strategy means that the decision which node to refine is now based on all the information that the algorithm has at the time of the decision making. The next descent

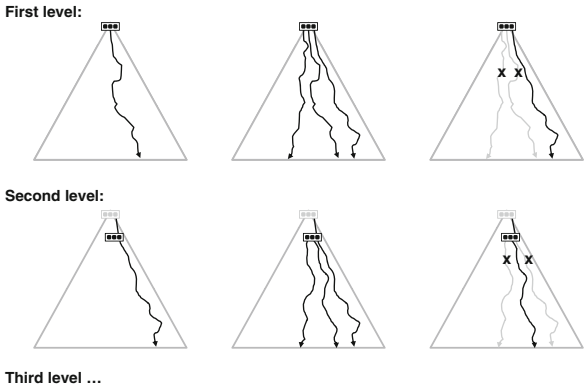

Fig. 3 Iterative depth first descent. When the algorithm is interrupted the best leaf seen so far is chosen for insertion

step is always to the entry that has the smallest distance to the object, regardless of whether this means that we continue on the deepest path or not. In this sense, best first descent is a global strategy that takes all nodes into account, whereas depth first descent is local in the sense that a choice is only made among the children of the currently visited node. Figure 2c illustrates this strategy. As we can see, this algorithm maintains a priority queue of the lowest entries on all paths started so far, i.e., unrefined entries sorted by their distance to the object. The path is always continued on the path corresponding to the first entry in the queue.

3.4.3 Iterative depth first descent

In terms of anytime clustering, best first descent tries to optimize the selection of insertion nodes. A possible drawback of this strategy is that depending on how often the algorithm has to go back and continue from upper nodes and on how soon it is interrupted, the algorithm might remain at the upper levels and buffer the object there. Similar drawbacks can be expected from the priority breadth first strategy. In contrast, depth first processing is most often able to reach leaf level.

Based on this analysis, we suggest an alternative descent strategy that tries to reach leaf level, and if more time is available uses this time to validate the decisions that were taken. This can be considered a compromise between depth first, priority breadth first and best first strategies. We denote this approach as the *iterative depth first* descent strategy. The idea here is to start with the original depth first descent. Upon reaching leaf level, we iteratively evaluate the alternatives for decisions taken at the nodes on the depth first path as long as time permits.

Figure 3 illustrates the strategy. The algorithm starts by descending down the tree as in the depth first approach (top left). Assuming that it is not interrupted, it then goes back to the root level and descends into the siblings of the entry chosen during the first iteration down the tree. Following down these alternatives to the leaf level, we eventually obtain two more candidate leaves among which we can choose to insert (top center of Fig. 3). Among these three options, we now pick the best one as shown at the top right of Fig. 3.

If there is still time, we repeat this process on the path leading to our current optimum (bottom left of Fig. 3). We descend down the paths corresponding to the siblings of the node one level below the root on our current optimum path (bottom center of Fig. 3). This yields once again three options to choose from (bottom right of Fig. 3). This process is continued until the algorithm is interrupted or until no more unchecked siblings on the path remain.

On interrupt we buffer/insert and update as in the other strategies described above. All in all, using this strategy we will have at most $log^2(n)$ comparisons, e.g. for 50,000 micro-clusters and a fanout of 3 about 100 comparisons, which is in stark contrast to 50,000 comparisons as in e.g. [2,8].

With these alternative descent strategies, we complete the concept of anytime clustering in the sense that we use any possible idle time to improve the insertion process. Whereas the depth first descent strategy stops once the maximal model size is obtained and a leaf is reached, priority breadth first, best first and iterative depth first descent make use of additional time to check alternative insertion options. In this manner, our anytime clustering accounts for very short time spans per object through aggregation and for very long time spans through further optimization of inserts.

3.5 Cluster shapes and cluster transitions

The clustering resulting from the ClusTree is the set of CFs stored at leaf level, i.e., the finest representation maintainable w.r.t. the speed of the data stream. This can be seen as our online component and it allows for using various offline clustering approaches. Taking the means of the CFs as representatives, we can apply a k-center clustering as in [23] or density based clustering as proposed in [8] to detect clusters of arbitrary shape. One main advantage of our approach is that we can maintain a way larger number of micro-clusters compared to other approaches [2,3,23,8] and hence the offline clustering, e.g. density based, has finer input granularity.

Regarding cluster transitions, e.g. concept drift or novelty, many approaches proposed in the literature can directly be applied to the output of our ClusTree. Using the unique ids to every new leaf entry, we are able to track micro-clusters. Pyramidal time frames [2] allow the user to view clusterings of arbitrary time horizons. Furthermore, the ids allow us also to apply the transition detection and distinction techniques described in [25], including outlier, novelty, and concept drift detection.

3.6 Summary of the ClusTree algorithm

The proposed technical solutions and their benefits can be summarized as follows:

- Hierarchical data structure—yields a logarithmic insertion complexity and a fine grained clustering, e.g. as input for an offline component (cf. Sect. 3.1)
- Buffer & hitchhiker concept—enables anytime clustering and a self-adaptive model size (cf. Sect. 3.1)
- Exponential decay—allows for aging of older items and reusing of insignificant entries (cf. Sect. 3.2)
- Aggregation—improves clustering quality on very fast streams (cf. Sect. 3.3)
- Alternative descent strategies—exploit additional time in case of slower streams and optimize the insertion (cf. Sect. 3.4)
- Cluster features and IDs—make the ClusTree compatible with existing work for drift, novelty, arbitrary shapes, etc. (cf. Sect. 3.5)

Figure 4 summarizes the complete ClusTree algorithm (using depth first descent) in a flow chart. We evaluate the performance of our approach and of the alternative descent strategies in the following empirical study.

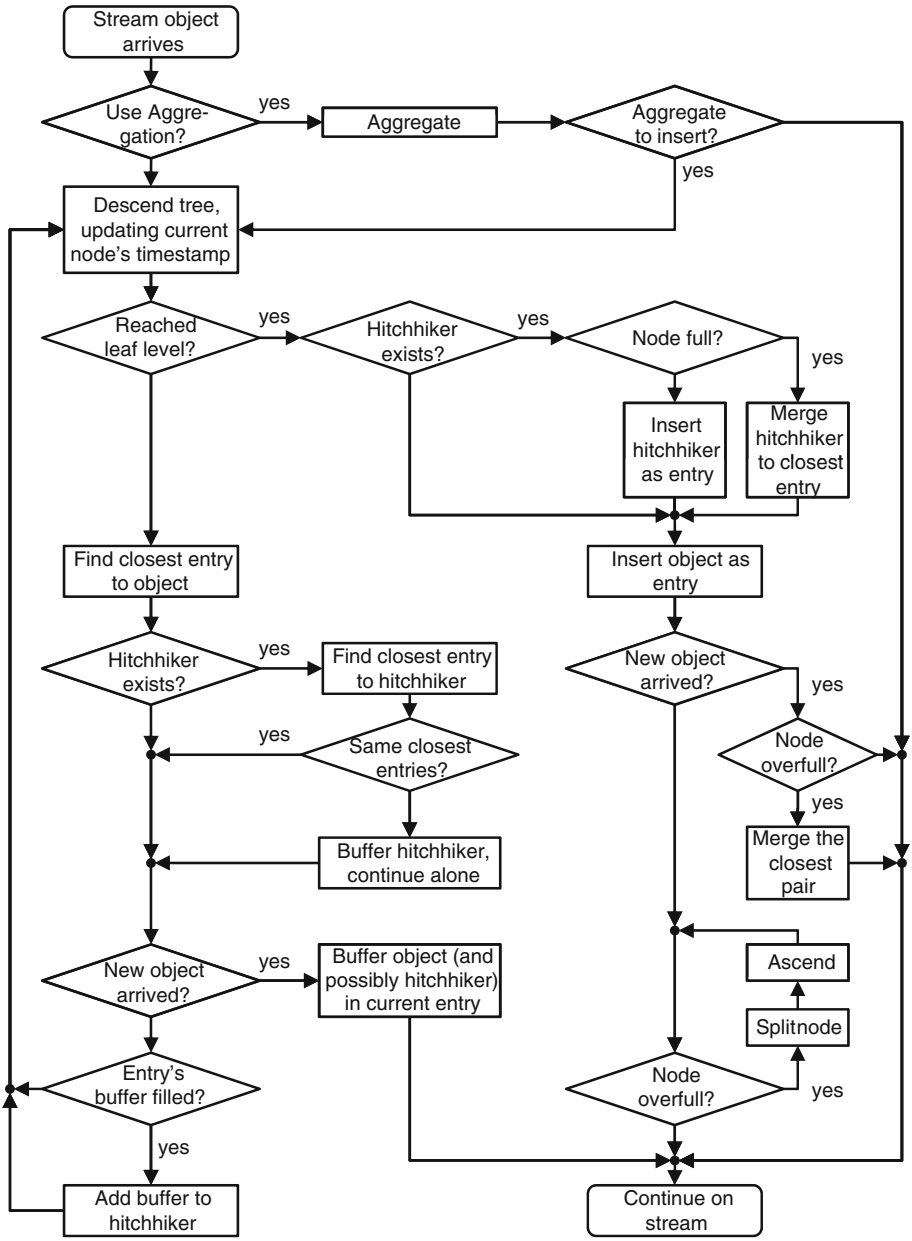

Fig. 4 Flow chart of the ClusTree algorithm

4 Analysis and experiments

We assess the performance of the ClusTree in the following. First, we examine the time and space complexity of building and maintaining a ClusTree in Sect. 4.1. In Sect. 4.2, we evaluate the anytime clustering property of the ClusTree and show the benefits of our

speed-up through aggregation. Finally, we demonstrate the adaptive clustering performance in Sect. 4.3 by comparing our results against CluStream [2] and DenStream [8]. The algorithms were implemented in C, all experiments were run on Windows machines with 3 GHz. The employed real and synthetic data sets are described in Sect. 4.2 before their first usage.

4.1 Time and space complexity

Our goal for efficient and effective clustering is a high granularity with low processing costs. Therefore, we investigate the effect of the fanout and of the number of distance computations required to insert an object from the stream on the granularity, i.e., the number of cluster features (CFs) at leaf level. Figure 5(a) shows the results for fanouts from 2 to 12. Depending on the speed of the stream, 12–72 distance computations are possible before interruption (we chose multiples of 12 on the x-axis because it is the smallest multiple of all tested fanouts). As can clearly be seen in all groups of bars, a fanout of three yields the highest granularity independent of the number of distance computations, i.e., of stream speed.

Next we evaluate the space demands with respect to the dimensionality and granularity in Fig. 5b. It shows the results for a fanout of 3 (assuming 4 Bytes per value). The space demands for the ClusTree are moderate even for high granularities and high dimensionality. For 128.000 CF at leaf level and 20 dimensions the ClusTree only needs 68 MB space, while the number of distance computations to reach the leaf level is only 33. Fanout 3, dimensionality 20 and one million CF at the finest level consume roughly 500 MB, i.e., still main memory, and the number of distance computations is still less than 40. This is opposed to any stream clustering algorithm that maintains one million micro-clusters and checking a new item against each of these. CluStream [2] for example stores q micro-clusters and hence has to calculate q distances (plus possible delete $O(q)$ and merge $O(q^2)$ checks). We only need $O(log(q))$ many distance calculations and only store $O(q)$ CF (cf. Fig. 5).

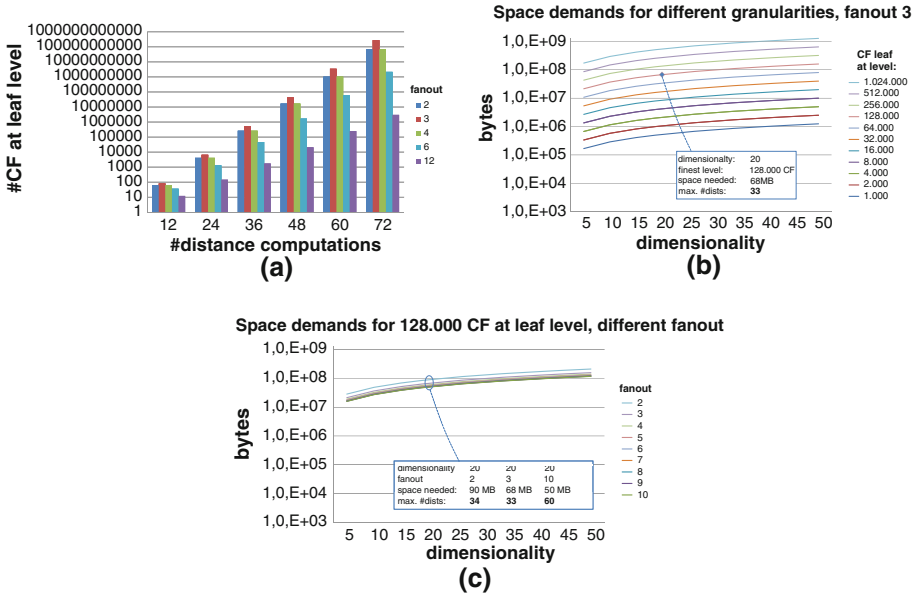

Fig. 5 **a** Granularity (number of CF at leaf level) w.r.t. fanout and number of distance computations. **b**, **c** Space consumption w.r.t granularity, fanout and dimensionality

Figure 5c shows the space demand of the ClusTree for 128.000 CF at leaf level, different dimensionality and different fanout values. While a higher fanout yields less space demands, the number of distance computations that are necessary to reach the same granularity is significantly higher. Combining the results from Fig. 5, we conclude that a fanout of 3 is the best choice in terms of time and space complexity.

Given the fanout of 3, the costs for a single split are low: 4 entries are present during split; hence, 6 distances are calculated. A new node and one new entry for the parent node are created, and the old node and the old entry pointing to it are updated. In the worst case, the number of splits is equal to the height of the tree. Moreover, once the tree size is adapted to the stream speed and decay invalidates old entries, the number of splits is practically zero.

4.2 Anytime clustering and aggregation

To evaluate the clustering quality of the ClusTree, we evaluate the average purity of the clusters on the different levels of the tree. To determine the purity, we use synthetic as well as real world data that contains objects labeled with one of several classes. For a set K of CFs, the purity is then calculated as the weighted average purity of all CFs in K : $\sum_{k=1}^{|K|} \frac{n_k}{n} \cdot \frac{\max_c(n_{ck})}{n_k} = \frac{1}{n} \sum_{k=1}^{|K|} \max_c(n_{ck})$, where n_k is the number of objects in the CF k, n_{ck} those belonging to class c and $n = \sum_{k \in K} n_k$. The real world data set Forest Covertype is available from [17] and contains roughly 580.000 objects from 7 classes and 10 continuous attributes. To investigate the scalability of the ClusTree in terms of dimensionality and the number of clusters we use synthetic data sets containing 550.000 objects each (including 5% noise) and a varying number of attributes and classes (see individual experiments for explicit numbers). The clusters are generated as a hierarchy of Gaussians with three levels, where centers lie at a uniformly distributed angle and distance from their parents in the unit cube. To simulate a varying stream we generated the arrival intervals according to a Poisson process, a stochastic model that is often used to model random arrivals [14]. For the anytime experiments, we generated a stream with an expected number of 90,000 points per second, i.e., $\lambda = 1/90000$.

Figure 6a shows the results for Forest Covertype (bottom) and the synthetic data set containing four classes and four dimensions (top). The results shown are the purity values after the complete data set has been processed. The top most bar (orange) represents the purity value at the root level and each following bar corresponds to the next deeper level. (Please note that for synthetic data the root level bar is not visible as the axis has been formatted to show the difference on the lower levels.) The resulting ClusTree had ten levels for the synthetic data and 9 levels for the Forest Covertype data set. The most interesting purity value is that of the leaf level representing the finest micro-clustering granularity. It is above 99% for the synthetic data and still 88% for Forest Covertype. The purity values on the higher levels of the tree give an indication for the clustering quality for higher stream speeds. We show further results on varying stream speed in Sect. 4.3. Except for the leaf level, the purity values on the synthetic data set are above 95% on all levels, showing that the noise objects have been separated very well. The purity decreases more significantly for the Forest Covertype data, but is still above 70% even three levels underneath the root.

Figure 6b, c show the results regarding scalability using the same anytime stream as before. We varied the number of classes from 2 to 8 at four dimensions (left) and the number of dimensions from 2 to 8 using 4 classes (right). For 2 and 4 classes the quality is consistently high on all levels, just the root level purity drops at 4 classes, and further (below the shown area) for 8 classes. Increasing the number of classes to 8 shows a higher impact on the root level and also one level below the root. Although the purity decreases also on the other

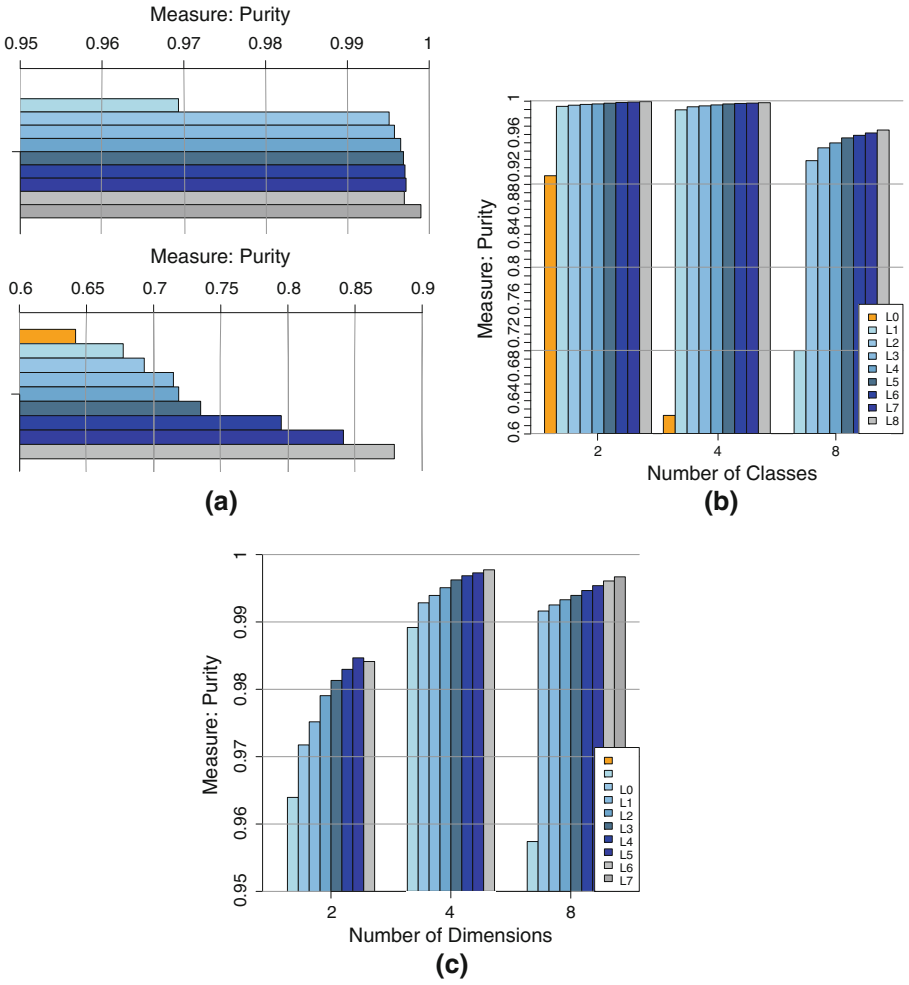

Fig. 6 **a** Clustering purity on synthetic data (*top*) and on Forest Covertype (*bottom*). **b** Scalability w.r.t. the number of clusters; **c** Scalability w.r.t. the number of dimensions

levels it is still above 95% on six levels, indicating a good separation of classes and even of noise objects. Comparing the results on different dimensionalities shows that the quality is similar for 4–8 dimensions, but lower in the 2-dimension case. This is due the fact that the overlapping of the classes is higher if the dimensionality decreases. However, once again the majority of the levels has a purity above 95%.

Finally, we evaluated different stream speeds, i.e., we varied the expected number of points per second (*pps*) from 60,000 to 150,000. Figure 7 shows the resulting purity values for the leaf level and the middle level of the ClusTree for Forest Covertype. For the slowest stream, the purity on the leaf level reaches 93%. While the purity is still very good (87%) at 120,000 pps it drops below 70% for even faster streams with 150,000 pps. For our proposed speed-up through aggregation (cf. Sect. 3), the results for 150,000 pps are shown in the left part of Fig. 7. Thanks to the aggregation the purity on the leaf level is significantly improved.

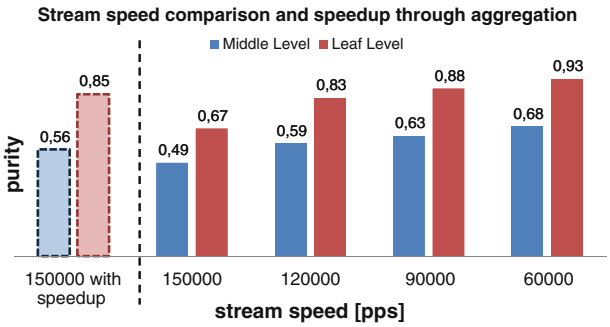

Fig. 7 Purity with and without aggregation w.r.t. stream speed for Forest Covertype

4.3 Adaptive clustering

To evaluate the adaptive clustering behavior of the ClusTree, we simulated constant data streams with different numbers of points per second using the Forest Covertype data set. We compare our performance against CluStream [2] and DenStream [8]. For all approaches, we report the results of the online component, i.e., we analyze the properties of the resulting micro clusters and do not employ an additional offline component afterward.

First of all, we investigate the number of micro clusters that can be maintained by the individual approaches for different stream speeds. The results are shown in Fig. 8, exact numbers are listed in Table 1. As indicated, the ClusTree can maintain roughly 430,000 micro clusters at 49,000 pps. With a stream speed of 140,000 pps, the ClusTree can still maintain 435 micro clusters. The competing approaches on the other hand can only process less than 10,000 pps when maintaining 500 micro clusters. This drastic difference is due to the hierarchical structure of the ClusTree which yields only a logarithmic amount of distance computations. In other words, the number of micro clusters we can maintain is exponential compared to CluStream or DenStream. This large number is beneficial, since the output of the online component is given to the offline component to compute the final clustering (using a clustering method of choice). A more detailed input to the final clustering enables more accurate results and detection of possible outliers. Moreover, the major advantage here is that the ClusTree automatically self-adapts to the stream speed without parametrization.

The question is at which price comes this benefit? Does the quality of the individual micro clusters deteriorate, because new points might not be added to the optimal micro cluster? To answer this question, we evaluated the radius and the purity of the resulting micro clusters from all three approaches. Figure 9 shows the results for the ClusTree, results for CluStream and DenStream are listed in Table 1.

For the radius, we report the maximum as well as the 75 percentile and the median in Fig. 9. Since the actual numbers are skewed, we plot a moving average value. Naturally, with increasing stream speed, and hence decreasing number of micro clusters, the radii generally become larger. However, while we see a constant increase in the maximum value, the median and even the 75 percentile stays very low even for 100,000 to 150,000 pps. While DenStream produces larger micro clusters, CluStream shows a similar performance for the same amount of micro clusters. However, to maintain this amount of micro clusters CluStream can again only process slow streams where it is outperformed by our approach.

The purity values for CluStream, DenStream and our novel ClusTree approach underline the above findings (cf. Table 1). DenStream does not exceed an average purity of 70%.

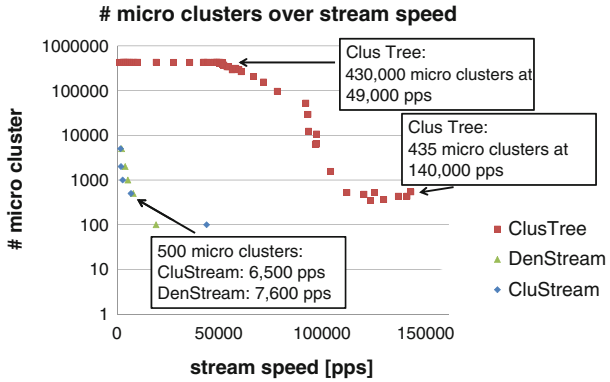

Fig. 8 Number of micro clusters that can be maintained w.r.t. stream speed

Table 1 Overall results on Forest Covertype

#MC	pps	Radius (median)	Radius (max.)	Purity
DenStream				
5,000	2,000	0.21	151.8	0.53
2,000	3,700	0.24	195.1	0.55
1,000	5,000	3.35	160.9	0.66
500	7,600	14.01	83.7	0.53
CluStream				
5,000	1,500	0.02	37.4	0.70
2,000	1,700	0.03	224.4	0.87
1,000	2,500	0.33	238.8	0.90
500	6,500	0.58	177.6	0.62
ClusTree				
5,000	80,000	0.44	13.8	0.72
2,000	94,000	0.51	18.9	0.71
1,000	105,000	0.55	21.8	0.70
500	120,000	2.25	29.7	0.67

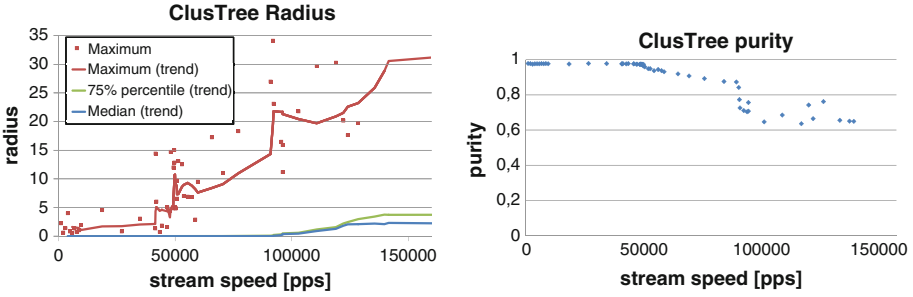

Fig. 9 Radius (*left*) and purity (*right*) for ClusTree micro clusters w.r.t stream speed

Clustream shows a higher purity than the ClusTree for 1,000 micro clusters (90% for CluStream vs. 78% for the ClusTree), but again these numbers are not comparable due to the huge difference in terms of points per second. In conclusion, it can be said that the

ClusTree can maintain an equal amount of micro clusters on streams that are faster by orders of magnitude and that it can maintain an exponential amount of micro clusters at equal stream speed while providing good results in terms of cluster size (radius) and quality (purity).

5 Evaluation of descent strategies

Due to the logarithmic number of distance computations that are necessary to maintain a certain number of micro-clusters in the ClusTree, in a given time frame we can maintain an exponential number of micro-clusters compared to linear approaches. This fact is inherent to the hierarchical ClusTree approach and was confirmed by the results in the previous section. If the model size, i.e., the tree size, is limited either through limited memory or user constraints, the ClusTree algorithm will be idle on slower streams once the maximal model size has been reached.

This can also be seen in some of our previous experimental results. For example, in Fig. 8 at the top left the algorithm maintains approximately the same number of micro-clusters for a large range of stream speeds. It reaches the maximal number of micro-clusters at approximately 50,000 pps. This means that for streams below this, the algorithm is idle once it has reached leaf level.

To exploit these possible idle times for improving the clustering result, we introduced different descent strategies in Sect. 3.4 that continue searching for better insertion options as long as time permits. Together with the speed-up strategies for very fast streams, the alternative insertion strategies complete our anytime stream clustering approach. In this section, we evaluate the workings and benefits of the proposed descent strategies.

We evaluate the four strategies *depth first*, *breadth first*, *best first* and *iterative depth first* on the previously mentioned Forest Covertype data set using different tree heights and varying stream speeds. The tested tree heights 7, 9 and 11 correspond to roughly 2000, 20,000 and 170,000 possible micro-clusters at leaf level. The stream speed was varied from 600 pps to 60,000 pps; we report the time per object on the x-axis (in μs/object) such that there is more time per insertion from left to right. As in the previous experiments we measure both the average purity values and the median of the resulting radii. Figure 10 summarizes the results.

Throughout the results in Fig. 10, the primary descent strategy depth first and the novel iterative depth first descent show the same results on the fastest stream speed setting (17 μs/object, corresponding to roughly 60,000 pps). This is due to the fact that both strategies start with the same initial solution. While the depth first approach stops once the leaf level is reached, the iterative depth first uses additional time and might find a better micro-cluster to insert the current object. For all tested tree heights the iterative depth first slightly improves in both measures for slower streams, i.e., higher purity values and smaller radii are achieved. Since the number of distance computations is in $O(\log^2(n))$, the iterative depth first strategy cannot profit from even more time per object. This means that for even slower streams, the corresponding graphs show a stagnating behavior early on.

The *best first* and the *priority breadth first* strategies show nearly the same performance. Since neither of these strategies favors an initial descent to reach the leaf level, they process all (the many) entries on the upper levels of the tree before continuing on the next level. As a consequence, these approaches do not reach the maximal tree size on faster streams. As can be seen in the left part of Fig. 10, the full tree height is only reached at 600 μs/object for tree height 7 and 1200 μs/object for tree height 9. With this time allowance, the strategies evaluate all possible option and hence no further improvement is reached on even slower

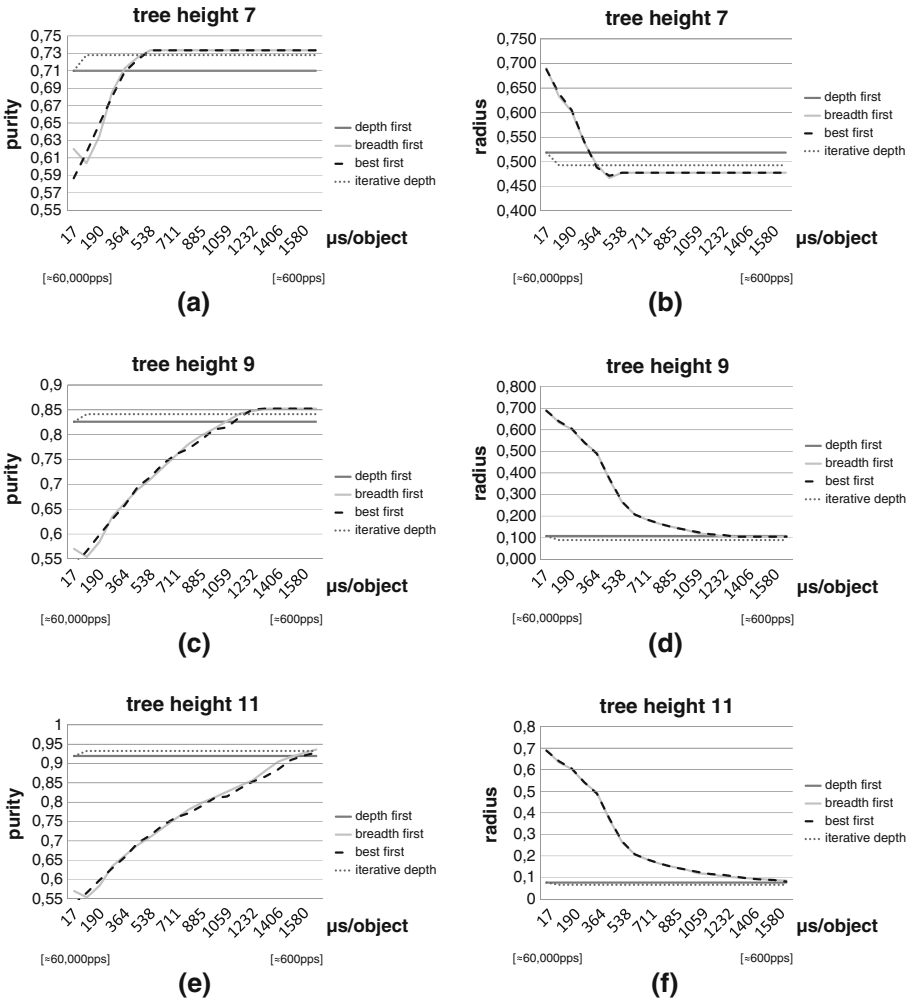

Fig. 10 Purity values for different tree heights and varying stream speeds (*left*). The corresponding radii (median) (*right*)

streams for the respective maximal model size. However, for all tested tree heights, both the *best first* and the *priority breadth first* strategy outperform the two depth first approaches in terms of the average purity. Regarding the achieved radii, the depth first approaches are outperformed on the smaller tree and their performance is met on the larger trees for slower streams (cf. Fig. 10 right). The maximum radius, however, as e.g. Fig. 9 illustrates, was the lowest throughout the experiments using best first or priority breadth first.

Summarizing the results for the four proposed descent strategies, we note that the simple *depth first* descent yields already good results, especially on larger tree/model sizes. The *best first* approach and the *priority breadth first* approach improved the results consistently on slower streams, which can be attributed to their strategy of testing all possible options (if time permits). The *iterative depth first* descent strategy constitutes an excellent alternative insertion strategy, since it starts with the same high performance as the depth first strat-

egy, has very low runtime ($O(\log^2(n))$), and yet improves the initial solutions on all tested settings. Moreover, it finally reaches comparable high-quality results compared to all other approaches.

6 Conclusion

Clustering streaming data is of increasing importance in many applications. In this work, we proposed a parameter free index-based approach that self-adapts to varying stream speed and is capable of anytime clustering. Our ClusTree maintains the values necessary for computing mean and variance of micro-clusters. By incorporating local aggregates, i.e., temporary buffers for "hitchhikers", we provide a novel solution for easy interruption of the insertion process that can be simply resumed at any later point in time. For very fast streams, aggregates of similar objects allow insertion of groups instead of single objects for even faster processing. For slower stream settings, we proposed alternative insertion strategies that exploit possible idle times of the algorithm to improve the quality of the resulting clustering. By comparison with recent approaches, we have shown that the ClusTree can maintain the same amount of micro clusters at stream speeds that are faster by orders of magnitude and that for equal stream speeds our granularity is exponential w.r.t. competing approaches. Moreover, we discussed compatibility of our approach to finding clusters of arbitrary shape and to modeling cluster transitions and data evolution using recent approaches.

Acknowledgments This work has been supported by the UMIC Research Centre, RWTH Aachen University, Germany.

References

1. Aggarwal C (2009) On classification and segmentation of massive audio data streams. Knowl Inf Syst 20(2):137–156
2. Aggarwal CC, Han J, Wang J, Yu PS (2003) A framework for clustering evolving data streams. VLDB, Berlin, pp 81–92
3. Aggarwal CC, Han J, Wang J, Yu PS (2004) A framework for projected clustering of high dimensional data streams. VLDB, Toronto, pp 852–863
4. Arai B, Das G, Gunopulos D, Koudas N (2007) Anytime measures for top-k algorithms. VLDB, Vienna, pp 914–925
5. Assent I, Krieger R, Glavic B, Seidl T (2008) Clustering multidimensional sequences in spatial and temporal databases. Knowl Inf Syst 16(1):29–51
6. Barbará D, Chen P (2000) Using the fractal dimension to cluster datasets, KDD, pp 260–264
7. Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. SIGMOD, Atlantic City, pp 322–331
8. Cao F, Ester M, Qian W, Zhou A (2006) Density-based clustering over an evolving data stream with noise, SDM
9. Chen Y, Tu L (2007) Density-based clustering for real-time stream data, KDD, pp 133–142
10. Cheng J, Ke Y, Ng W (2008) A survey on algorithms for mining frequent itemsets over data streams. Knowl Inf Syst 16(1):1–27
11. Dang X, Ng W, Ong K (2008) Online mining of frequent sets in data streams with error guarantee. Knowl Inf Syst 16(2):245–258
12. DeCoste D (2002) Anytime interval-valued outputs for kernel machines: fast support vector machine classification via distance geometry, ICML, pp 99–106
13. Dempster AP, Laird NM, Rubin DB (1977) Maximum likelihood from incomplete data via the em algorithm. J Royal Stat Soc B 39(1):1–38
14. Duda R, Hart P, Stork D (2000) Pattern classification, 2nd edn. Wiley, London

15. Gaber M, Zaslavsky A, Krishnaswamy S (2007) A survey of classification methods in data streams. Springer, Berlin, pp 39–59
16. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. SIGMOD, Boston, pp 47–57
17. Hettich S, Bay S (1999) The UCI KDD archive. http://kdd.ics.uci.edu
18. Jain A, Zhang Z, Chang EY (2006) Adaptive non-linear clustering in data streams, CIKM, pp 122–131
19. Kranen P, Krieger R, Denker S, Seidl T (2010) Bulk loading hierarchical mixture models for efficient stream classification. In: LNAI, Proceedings of 14th PAKDD
20. Kranen P, Seidl T (2009) Harnessing the strengths of anytime algorithms for constant data streams. DMKD Journal, Special Issue on Selected Papers from ECML PKDD 19(2):245–260
21. Li H, Shan M, Lee S (2008) DSM-FI: an efficient algorithm for mining frequent itemsets in data streams. Knowl Inf Syst 17(1):79–97
22. Lühr S, Lazarescu M (2009) Incremental clustering of dynamic data streams using connectivity based representative points. Data Knowl Eng 68(1):1–27
23. O'Callaghan L, Meyerson A, Motwani R, Mishra N, Guha S (2002) Streaming-data algorithms for high-quality clustering. ICDE
24. Seidl T, Assent I, Kranen P, Krieger R, Herrmann J (2009). Indexing density models for incremental learning and anytime classification on data streams, EDBT
25. Spiliopoulou M, Ntoutsi I, Theodoridis Y, Schult R (2006) Monic: modeling and monitoring cluster transitions, KDD, pp 706–711
26. Spinosa EJ, Ponce de Leon Ferreira de Carvalho AC, Gama J (2007) Olindda: a cluster-based approach for detecting novelty and concept drift in data streams, SAC, pp 448–452
27. Street WN, Kim Y (2001) A streaming ensemble algorithm (sea) for large-scale classification, KDD, pp 377–382
28. Udommanetanakit K, Rakthanmanon T, Waiyamai K (2007) E-stream: evolutionbased technique for stream clustering, ADMA, pp 605–615
29. Ueno K, Xi X, Keogh EJ, Lee D-Y (2006) Anytime classification using the nearest neighbor algorithm with applications to stream mining, ICDM, pp 623–632
30. van Leeuwen M, Siebes A (2008) Streamkrimp: detecting change in data streams, ECML/PKDD, pp 672–687
31. Vlachos M, Lin J, Keogh E, Gunopulos D(2003) A wavelet-based anytime algorithm for k-means clustering of time series. WS Clust. High Dim. Data & App. (at ICDM)
32. Wang H, Fan W, Yu PS, Han J (2003) Mining concept-drifting data streams using ensemble classifiers, KDD, pp 226–235
33. Yang Y, Webb GI, Korb KB, Ting KM (2007) Classifying under computational resource constraints: anytime classification using probabilistic estimators. Mach Learn 69(1):35–53
34. Zhang T, Ramakrishnan R, Livny M (1996) BIRCH: an efficient data clustering method for very large databases. SIGMOD, NY, USA
35. Zhou A, Cao F, Qian W, Jin C (2008) Tracking clusters in evolving data streams over sliding windows. Knowl Inf Syst 15(2):181–214

Author Biographies

Philipp Kranen received his Master degree in Computer Science from RWTH Aachen University in Germany in December 2006. After that he worked for half a year at Siemens Corporate Research in Princeton, NJ, USA and currently he is a PhD student at RWTH Aachen University. His research interests include data exploration techniques, such as efficient similarity search, and stream data mining, in particular modeling of data streams and classification on data streams.

Ira Assent received her Master's degree and her PhD in computer science from RWTH Aachen University, Germany, in 2003 and 2008, respectively. She is currently an assistant professor in the department of computer science at Aarhus University, Denmark. Prior to joining Aarhus University, she was an assistant professor at Aalborg University, Denmark. Her research interests are in databases with a special focus on efficient algorithms for similarity search and data mining.

Corinna Baldauf received her Master degree in Computer Science from RWTH Aachen University in Germany in December 2008. She wrote her diploma (master) thesis on flexible probabilistic clustering for concept drift detection in streaming data. Her thesis included a study on adaptive stream clustering as well as anytime approaches to outlier detection on data streams.

Thomas Seidl is a professor for computer science and head of the data management and data exploration group at RWTH Aachen University, Germany. His research interests include data mining and database technology for multimedia and spatio-temporal databases in engineering, communication and life science applications. Prof. Seidl received his Diplom (MSc) in 1992 from TU Muenchen and his PhD (1997) and venia legendi (2001) from LMU Muenchen.