Efficient Distributed Outlier Detection in Data Streams

Ioannis Kalliantzis¹, Apostolos N. Papadopoulos¹, Anastasios Gounaris¹, and Konstantinos Tsichlas¹
Aristotle University of Thessaloniki, Thessaloniki, Greece
{ioankall,papadopo,gounaria,tsichlas}@csd.auth.gr

Abstract. Anomaly detection is one of the major data mining tasks in modern applications. An element that shows significant deviation from the “usual” behavior is marked as an outlier. This means that this element either corresponds to noise or it requires more careful examination because it may be important. Also, many clustering algorithms are very sensitive to outliers. In any case, outliers must be identified and explored further, meaning that efficient outlier mining techniques are required. In this paper, we focus on distributed density-based outlier detection over multi-dimensional data streams. In particular, we focus on the approximation method for computing the Local Correlation Integral (LOCI) of multi-dimensional points. Each object \( p \) is assigned a score \( \text{score}(p) \) which represents the outlier score of \( p \). Thus, one can select the top-\( k \) elements from the dataset that have the highest outlier scores. Our proposal has been implemented in Apache Spark using Scala and experiments have been conducted in a physical cluster running Apache Hadoop 2.7 and Apache Spark 2.4.0. Performance evaluation results demonstrate that the proposed algorithm is efficient and scalable and therefore it can be used to mine outliers in large distributed datasets.

Keywords: Anomaly detection · Distributed algorithms · Data streams · Approximate LOCI · Apache Spark.

1 Introduction

An outlier is an object that deviates from the typical behavior. Although this is a very simple and intuitive concept, it turns out that detecting outliers is not at all simple, since the quantification of the deviation and also the definition of the typical behavior are difficult to handle and largely depend on the application domain. Outlier mining is also known as anomaly detection, since in many cases outliers correspond to abnormal situations. In general an outlier may be present due to a noisy dataset or because of an event that should be inspected carefully.

The topic of outlier discovery has been studied by many different perspectives and across numerous application fields. From a statistical point of view, based on [8], “an outlying observation, or outlier, is one that appears to deviate markedly from other members of the sample in which it occurs”. Also, based
“an outlier is an observation in a data set which appears to be inconsistent with the remainder of that set of data”. In the sequel, we assume that data objects are represented as multidimensional points in some vector space and the detection of outliers is based on the values of the attributes. There are many diverse applications that can benefit from meaningful outlier discovery. Some of them are listed below:

- **Novelty Detection.** Given a set of images or text documents the challenge is to detect novel information.
- **Financial Fraud.** In this case, given a set of financial transactions we are looking for suspicious transactions that may be fraudulent.
- **Unexpected Records.** Many databases contain records that are not that “usual”, and detecting them is a significant task.
- **Intrusion Detection.** The detection of unauthorized access in a computer system is considered a high-priority task in cyber-security applications.

Outliers may be detected by using either global or local methods. An element \( p \) is marked as a global outlier if the attributes of \( p \) are significantly different than the attribute values of the majority of the data objects. On the other hand, \( p \) is a local outlier when the attributes of \( p \) are very different compared to those contained in the neighborhood of \( p \). In general, the discovery of global outliers is more efficient in comparison to local outlier detection which is far more challenging and requires more sophisticated algorithms.

Orthogonal to the global/local distinction, a second classification scheme for outliers involves the techniques applied to detect them. There are three major classes of algorithms, depending on the point of view: i) **statistical** or **model-based** techniques, which assume specific properties with respect to the data distribution to classify an object as an outlier, ii) **distance-based** techniques, which use the distance to the nearest-neighbors of an object as a deviation criterion, and iii) **density-based** techniques, which decide about the outlierness of an element based on density measures computed based on the local neighborhood.

Density-based methods have a huge advantage in contrast to distance-based methods, since they are not sensitive to data locality, an issue first studied in [2]. Therefore, although such methods are computationally expensive, they provide the best insights when there is significant heterogeneity in the local data distribution. In this work, we adopt the methodology of the **Local Correlation Integral (LOCI)** technique proposed in [13]. For each element \( p \), LOCI computes \( \text{score}(p) \) which quantifies the outlierness of \( p \). More specifically, we use the approximate version of LOCI (ALOCI) which is more efficient than LOCI without penalizing the accuracy of the result significantly.

**Motivation and Contributions.** Motivated by the fact that density-based outlier mining is in general computationally intensive, in this paper we design distributed algorithms to provide efficient computation by using multiple resources. Moreover, taking into account that many modern applications process the input data as a stream, it is essential to also support stream-based processing
to allow continuous updates of the outlier scores. This is extremely important in order to attack concept drift in the input data. Parallel processing together with stream-based anomaly detection constitutes a powerful combination to support anomaly detection in Big Data. The major contributions of our work are summarized as follows:

- Initially, we provide a baseline solution to discover outliers in a stream of elements, using distributed techniques. The baseline algorithm is termed Basic Distributed ALOCI (BDALOCI) and it is used as an initial proof-of-concept.
- Next, we offer a set of optimizations towards a more efficient distributed computation of the outlierness score. These optimizations lead to the Fast Distributed ALOCI algorithm (FDALOCI), which gives orders of magnitude better runtimes with respect to the baseline.
- We adopt the sliding-window streaming model, which allows insertions and deletions of elements, whereas the number of active elements is defined by the size of the window.
- Performance evaluation results are offered based on experiments performed on a cluster of 32 physical machines running Spark on top of YARN and HDFS.

We note that this is the first work studying the efficient distributed computation of density-based outliers over data streams using Big Data technologies.

Roadmap. The rest of the article is organized as follows. In the next section, we describe briefly related work in the area of outlier mining. Next, in Section 3 we present some fundamental concepts related to our research, in order to make the paper self-contained. Section 4 explains in detail our methodology for mining density-based outliers in arbitrary multi-dimensional data streams by using the approximate LOCI technique in a distributed environment. Next, Section 5 contains representative experimental results based on real-world as well as synthetic datasets, conducted on a physical cluster. Finally, Section 6 concludes the paper and describes briefly future work in the area.

2 Related Work

The literature is rich in algorithmic techniques for outlier mining [1]. The topic has been studied by taking a supervised [7], a semi-supervised [4] or an unsupervised point of view. The supervised version treats anomaly detection as a difficult special case of a classification problem. In this work, we take the unsupervised direction, aiming at the distributed detection of density-based outliers over data streams.

The two most prevailing techniques in outlier detection assume either a distance-based [10] or a density-based [2] point of view. The discovery of distance-based outliers in data streams has been studied by [11, 6, 3] whereas [17] performs an exhaustive performance evaluation of the most representative techniques.
Moreover, algorithms for density-based streaming outliers have been proposed as well [14,15,12].

Recently, distributed algorithms for outlier mining were proposed, in order to enhance scalability when running over massive datasets. An algorithm for distributed distance-based outlier detection over data streams was proposed in [16], which is implemented in the Apache Flink engine.

A distributed implementation of the LOF algorithm (DLOF) in a shared-nothing architecture is reported in [18]. The algorithm was tested on a Hadoop cluster and shows significant scalability. In order for this methodology to work efficiently, the authors leverage the so called invariant observation: In the LOF computation process of a point \( p \), although each step requires different types of intermediate values, these intermediate values are only related to the direct \( k \)-NNs of \( p \). Therefore, in the step-by-step LOF computation pipeline, \( p \) only needs to directly access the intermediate values of its direct \( k \)-NNs in each step. Thus, the authors propose a support-aware assignment strategy leveraging the invariant observation to solve the intermediate data management problem.

The LOF algorithm has been also studied with respect to its ability to handle continuous data streams. For example, the DILOF technique [12] supports the update of the LOF values subject to changes performed in the input stream. The target of DILOF is to reduce the number of elements that must be updated when new elements are inserted.

Our proposal differs from the previous works in several aspects. First of all, we use the LOCI approach, which, in contrast to LOF, is more easily applied. Recall that LOF depends heavily on input parameters which are not tuned easily. Secondly, our research involves both the design of scalable distributed algorithms and incremental computation of the result due to the input data stream. Last but not least, our techniques are implemented in Apache Spark which constitutes the state-of-the-art with respect to knowledge discovery from Big Data.

3 Fundamental Concepts

In this work, we concentrate on enabling the Approximate LOCI algorithm to both work in a distributed way and to incrementally process batches of the input stream. Before we get into more details of our proposed solution we present some important background related to the LOCI algorithm.

3.1 The Exact LOCI Algorithm

The LOCI algorithm was proposed in [13] as an alternative to the computationally intensive LOF [2] algorithm. Firstly, it does not require a predefined \( k \)-parameter (the size of an element’s neighborhood), but instead, it runs using multiple values of radius at different levels of locality. In contrast to LOF, instead of providing just an outlierness score for each data object, LOCI uses the information collected at multiple locality levels for each point, to construct its LOCI plot. This plot summarizes a wealth of information about the data in the
LOCI relies on a metric called Multi-granularity Deviation Factor (MDEF) which is the relative deviation of an object’s local neighborhood’s density from the densities of the neighboring objects’ local neighborhoods. MDEF has the ability to cope with local density variations in the feature space and detect both isolated outliers as well as outlying clusters.

Definition 1 (MDEF). For any element \( p_i \), a radius \( r \) and a parameter \( a \) (a parameter that defines how much larger is the sampling region from the counting neighborhood) we define the multi-granularity deviation factor (MDEF) at radius (or scale) \( r \) as:

\[
MDEF(p_i, r, a) = \frac{n(p_i, r, a) - \hat{n}(p_i, ar)}{\hat{n}(p_i, r, a)} = 1 - \frac{n(p_i, ar)}{\hat{n}(p_i, a, r)}
\]

Additionally, the standard deviation of \( n(p_i, ar) \) is defined as \( \sigma_n(p_i, r, a) \) and based on this the normalized standard deviation is defined as:

\[
\sigma_{MDEF} = \frac{\sigma_n(p_i, r, a)}{\hat{n}(p_i, r, a)}
\]

In order to achieve faster computation of MDEF the algorithm estimates both \( n(p_i, ar) \) and \( \hat{n}(p_i, r, a) \). The following definition explains:

Definition 2 (Counting and sampling neighborhood). The counting neighborhood (or \( ar \)-neighborhood) is the neighborhood of radius \( ar \), over which each \( n(p, ar) \) is estimated. The sampling neighborhood (or \( r \)-neighborhood) is the neighborhood of radius \( r \), over which we collect samples of \( n(p, ar) \) in order to estimate \( \hat{n}(p_i, r, a) \).

LOCI supports multiple levels of locality by taking into consideration a set of radii while processing each data elements. The maximum value of a radius is \( r_{\text{max}} \approx a^{-1}R_p \), while the minimum value \( r_{\text{min}} \) is defined in such a way that a minimum number of objects exists in the neighborhood of any other object.

3.2 The Approximate LOCI Algorithm

Approximate LOCI was also proposed in the same paper. Leveraging the fact that the MDEF-based approach is well-suited to fast approximations that avoid costly iterations over each object in the (sampling) neighborhood of each \( p_i \), Approximate LOCI is able to overcome the multi-granularity problem that other approximate density-based outlier detection algorithms could not. This is because this approach essentially requires only counts at various scales.

To quickly estimate the average number of \( ar \)-neighbors over all points in an \( r \)-neighborhood of an object \( p_i \in P \), the following procedure is applied. Consider a grid of cells with side \( 2ar \) over the set \( P \) of points. Perform a box count of the grid: For each cell \( C_j \) in the grid, compute the count \( c_j \) of the number of objects in the cell. Each object in \( C_j \) has \( c_j \) neighbors in the cell (counting itself), so the total number of neighbors over all objects in \( C_j \) is \( c_j^2 \). By \( C(p_i, r, a) \) we refer to...
the set of all cells in the grid that are entirely within distance $r$ from $p_i$, which is actually an approximation of the $r$-neighborhood of $p_i$. Summing over the entire $r$-neighborhood, the result is $S^2(p_i, r, a)$, where $S^2(p_i, r, a) = \sum_{C_j \in C(p_i, r, a)} c_j^2$.

The total number of objects is simply the sum of all box counts, i.e., $S^1(p_i, r, a)$.

**Definition 3 (Approximate average neighbor count).** Let $a = 2^{-l}$ for some positive integer $l$. The average neighbor count over $p_i$’s sampling neighborhood is approximately:

$$\hat{n}(p_i, r, a) = \frac{S_2(p_i, r, a)}{S_1(p_i, r, a)}$$

ALOCI monitors information at several granularities by storing cell counts in a $k$-dimensional QuadTree: the first grid consists of a single cell, namely the bounding box of the address space. Each cell of side $2ar$ is subdivided recursively into $2^k$ sub-cells, each with radius $ar$, until it reaches the desirable scale. The subdivision is configurable by the parameter $n$, which defines the maximum allowed number of objects per cell and it is used by the algorithm as an approximation factor.

To achieve more accurate results, ALOCI supports multiple trees randomly shifted. To compute the outlierness score of an element, the algorithm uses the QuadTree that has the minimum distance between the data element and the center of the leaf node that the element belongs to. This is efficient due to the fact that only storing the $c_j$ values (one number per non empty cell), and not the objects themselves, is sufficient.

In addition to the approximate $\hat{n}$ value defined above, ALOCI requires an estimation of $\sigma_{\hat{n}}$, as shown below:

**Definition 4 (Approximate std. deviation of neighbor count).** Let $a = 2^{-l}$ for some positive integer $l$. The standard deviation of the neighbor count is approximately:

$$\sigma_{\hat{n}}(p_i, r, a) = \sqrt{\frac{S_3(p_i, r, a)}{S_1(p_i, r, a)} - \left(\frac{S_2(p_i, r, a)}{S_1(p_i, r, a)}\right)^2}$$

Combining all the previous concepts we reach to the conclusion that box counting within QuadTrees can be used to quickly estimate the MDEF values and $\sigma_{MDEF}$ values needed for the approximate LOCI approach.

An example is given below demonstrating the computation of the outlierness score in ALOCI. The dataset is shown in Figure 1. Using the formulas described previously, we will compute the scores of the two points located in region $A$. Regions $A$, $B$ and $C$ are used as the sampling neighborhood, in order to work on three different levels of granularity. Using the box counting numbers we compute that at the $A$-level $MDEF = 1$ and the $\sigma_{MDEF} = 0$, at the $B$-level $MDEF = 3.27$ and $\sigma_{MDEF} = 0.17$ and at the $C$-level $MDEF = 2.06$ and $\sigma_{MDEF} = 0.5$. This leads to the conclusion that the two objects are mostly considered outliers in $B$-level since their $MDEF$ value is 19.23 times bigger than the $\sigma_{MDEF}$. On a broader locality level (i.e., $C$-level) $MDEF$ is only four times larger than $\sigma_{MDEF}$ which barely marks the objects as outliers. The strong feature of LOCI though is that working on multiple levels, enables spotting outliers that would be missed by other outlier mining algorithms like LOF.
Fig. 1. Example of a dataset and the associated QuadTree for computing outlier scores.

4 Stream-based Distributed Approximate LOCI

This section presents our solution in detail. First, we present the way the fundamental data structures are adapted to work in a distributed setting, and then we describe the operations that must be supported in order for this scheme to work over data streams.

4.1 Basic Facilities

As described in Section 3, ALOCI is supported by a QuadTree data structure. Adapting this structure in a distributed environment is the first step towards the design of a distributed version of ALOCI. Although this step involves some technical concepts, it is essential for the algorithm to work and we present it briefly. Normally, trees are built using memory pointers. Each node of a tree is an object that apart from the application-related information, it also contains a number of variables that point to other objects, linked to this node in a parent-child relation. On a distributed setting however, there is no straightforward way to apply pointers when data are lying on different machines in a cluster with a shared-nothing architecture.

For our implementation, we leveraged the Apache Spark framework and its primary data abstraction, the Resilient Distributed Dataset (RDD) [19], which is a partitioned collection of objects distributed over the machines of the cluster. Considering a QuadTree as a collection of nodes, we have used an RDD of nodes to keep them in a distributed collection. The only loose end in this approach was the replication of the pointers mechanism. To solve this issue, we propose the following naming convention strategy for assigning a unique identifier to each node, which was used by the Linear QuadTree data structure [5].
Definition 5 (Morton codes). Every node is annotated with a unique identifier, which is generated by the id of its parent node appended by an identifier of node’s position at the level of the QuadTree it belongs, in order to be recognized by other nodes and to also contain all the required information regarding the parent-children relations in which it is involved.

In the 2-d case, the root node is assigned the id “0”, the ids of the four children are “00”, “01”, “02” and “03”, the ids of “03”’s children are “030”, “031”, “032” and “033”, and so on. This rule allows to easily detect the level of each node (distance from the root) and to quickly identify all descendants and ascendants of the node.

Note that, although the original ALOCI algorithm requires only the counts of the nodes, the need to support data streams with a sliding window, forces us to also keep the data elements as well. However, this small difference has a large impact in performance.

Score Computation. The first step to calculate the ALOCI score for a data element is to detect the node it belongs to and to identify all the ancestors of this node which is an easy task considering the Morton codes. The next step is to create all possible tuples containing a consecutive nodes from these nodes and to calculate the $MDEF$ value for each such tuple. Finally, the maximum $MDEF$ value is being used to calculate the ALOCI score of the object.

For example, the ALOCI score of an element belonging to node “01232” for $\alpha = 2$ is computed using the maximum of the $MDEF$ values of tuples: (“01232”, “0123”), (“0123”, “012”), (“012”, “01”) and (“01”, “0”). This process should be applied to any element in order to compute its score.

Insertions and Deletions. To enable the use of the algorithm for arbitrary data streams (e.g., turnstile streams, sliding-window streams) there is a need to support insertions of new data elements and deletions of the expiring ones.

For the insertion part, the first step is to determine the leaf node $L$ that should accommodate the new data object being inserted ($p$). We focus on leaf nodes, because they represent the whole space, in the highest available level of detail. The second step is to insert $p$ into $L$ and to increment the counts of all its ancestors. Then, the next step is to find all leaf nodes that contain more than $n$ data objects, where $n$ is the parameter we described earlier, and recursively break these nodes to children. Finally, we need to update the ALOCI score for every object in memory following the process described above.

The second part of fully supporting data streams, is an efficient way to delete elements that are considered expired from the QuadTree. The first step in doing so, is to filter the nodes of the QuadTree keeping only the nodes that are leaves and then to iterate over them getting the lists of the data objects they contain. Then, iterating over these lists let us spot and delete expired objects. Additionally, decreasing the counts of elements at all the ancestors nodes is obviously required. The next required step is to trim the tree where necessary.
Particularly, we need to filter the RDD with the nodes of the tree, deleting all descendants of nodes that are not leaves but contain less than \( n \) elements.

### 4.2 Algorithms

Having explained how insertions and deletions are performed over the distributed QuadTree, we proceed in designing a baseline algorithm, **Baseline Distributed ALOCI** (BDALOCI), which corresponds to a straightforward adaptation of ALOCI to the distributed setting.

Before any of the data objects arrive in the system and while the QuadTree has just been initialized, the RDD representing the tree contains only one node, which is the root. For each incoming batch, we insert the new data objects to the main data structure, the QuadTree. Then, we delete the expiring elements and we recompute the new outlierness scores based on the new counts per node, that represent the density in each available discrete locality level.

Comparing this approach to the brute-force algorithm that runs the ALOCI algorithm from scratch every time a new batch arrives, leads us to the intuition that the incremental approach is much faster, mainly because it avoids the massive cost of rebuilding the complete QuadTree each time.

Having a complete working solution, we proceed by enhancing the baseline solution with effective optimizations, towards improving the performance. The outline of **Fast Distributed ALOCI** (FDALOCI) is shown in Algorithm 1. Bellow, the four major enhancements that lead to more than two orders of magnitude better runtimes are described.

1. **Separating data objects from the QuadTree.** Our initial approach, as described above, required the data objects to be stored in leaf nodes. This approach, especially when ALOCI uses multiple QuadTrees for better accuracy, would result in a very costly replication of all the data objects to all the QuadTrees. Having this observation as a motivation, we have designed the following solution which improved the algorithm’s performance more than what we have initially expected, since it also gave us a great speedup during the process of adding new elements, handling expired elements and collecting outlying elements.

   The idea is based on the inverted index structure. Instead of having all the elements kept in a list inside the QuadTree’s leaves, we used a separate RDD that contains all the data objects, while each object also has an array that contains the IDs of the nodes it object belongs to, for each QuadTree. The first and most obvious advantage of this approach is that we only need to keep one instance of each data object instead of replicating it multiple times. This saves both memory and time since the algorithm avoids the costly I/O to the HDFS. Additionally, the insertion of new elements became faster as well, since instead of accessing all nodes of the QuadTree and appending multiple times the incoming elements to several lists, the new elements are now just unioned with the RDD that keeps all the elements of the sliding window.

   The greatest speedup though, due to this enhancement, was noticed when deleting expiring elements and on collecting the outlying elements. For both
Algorithm 1: Fast Distributed ALOCI (S,N)

**Input:** Stream of batches S, n-parameter N
1. initialize QT ← QuadTree(N) based on an RDD[Node] structure
2. allElements ← newRDD[Element]
3. for batch ∈ S do
   // Inserting new objects in system
   for record ∈ batch do
     record.nodeId = QT.findMatchingNodeFor(record)
   allElements = allElements.union(batch)
   Map < nodeId, count > counts = collectAsMap(batch)
   QT.updateCountsInSinglePass(counts)
   // Deleting expired objects from the QuadTree
   expiredElements = allElements.filter(checkExpired == true)
   allElements = allElements.filter(checkExpired == false)
   Map < nodeId, count > counts = collectAsMap(expiredElements)
   QT.updateCountsInSinglePass(counts)
   // Trimming the QuadTree
   List nodesShouldNotHaveChildren = new List()
   for node ∈ QT.getNodes() do
     if node.isLeaf == false AND node.getCount() < N then
       nodesShouldNotHaveChildren.append(node)
   for unsuitableParent ∈ nodesShouldNotHaveChildren do
     for node ∈ QT.getNodes() do
       if node.id.startsWith(unsuitableParent.id) AND
       node:id:length > unsuitableParent:id:length then
         QuadTree.removeNode(node)
   // Splitting overcrowded nodes
   for node ∈ QT.getNodes() do
     if node.isLeaf() == true AND
     node.getCount() ≥ unsuitableParent.id:length then
       QT.breakToChildrenRecursively(node)
   // Computing ALOCI scores
   for node ∈ QT.getNodes() do
     if node.isLeaf == true then
       node.updateALOCIscore()
   for element ∈ allElements do
     element.ALOCIscore = QT.getScore(element.nodeId)
tasks, we may avoid accessing all nodes of the QuadTree multiple times by filtering directly the new RDD keeping all objects that were either inserted to system after a specific timestamp or had a large ourlierness score.

2. Calculating outlierness score per neighborhood. The second enhancement we implemented, was to calculate the outlierness score for each leaf node, instead of doing so for each object. The ALOCI algorithm groups items in squares and calculates an outlierness score for each such group which is actually the outlierness score of the center of the square. Then all the elements that are located inside this square are given the same outlierness score.

Therefore, instead of calculating the ALOCI score for each element, we could actually find the ALOCI score for each leaf node. Then we could give to each element the score of the node it belongs to. This change lead to a significant speedup which can be elaborated from the fact that the number of nodes are significantly lower than the total number of elements in the sliding window.

3. Replicating ancestors' counts to descendants. Another major change in our design that gave a huge boost in the algorithm’s performance was to keep in each node all the counts of its ancestor nodes. Leveraging the ID assignment strategy for nodes’ ids, we implemented this by initializing an array of size equal to the length of each node id. The cost of populating this array is minimal. When each node needs to break, it passes its own array of counts to its children, so, no extra counting and iterating over the RDD nodes is required. Additionally, the cost of updating these arrays instead of just one counter is increased only slightly. The only actual cost is the replication of the counts. Instead of keeping the count of elements of root, in root, we actually replicate it on each of its descendants. Although, there is a small memory price to pay, the gains that we have from this change are huge. Each leaf node has all the necessary information to calculate its outlierness score without the need of iterating over the whole tree to get these numbers.

In a rough estimation, without this enhancement, for a tree of 20 thousand nodes we would need to iterate the tree more than 10 thousand times to get the outlierness score for each node. Instead of this, having all the information replicated in each node allows the algorithm to calculate the outlierness score for each node in a single pass.

4. Updating counts in one pass. Instead of accessing all the nodes of the QuadTree for each data object in the incoming batch, we propose the following strategy: We take the RDD with the objects being inserted and we update each object with the node that it belongs to. Then, we group the elements by the node they belong to and we get the count of them. We transform the result in a Map structure with keys the node ids and values the total count of data objects that should be placed in this node. In case of having multiple trees, we have an outer Map with keys the trees’ ids and values of the Map type described above. For deletions, we collect the count of the expiring elements grouped by node id.

The contents of the Map structure are collected and aggregated to the driver node of the cluster and then the complete Map is broadcasted to all worker nodes. Then, the counts of all nodes are adjusted using this Map in just a single
pass over the RDD, which reduces the possible millions of iterations over the RDD to just a single iteration. Being able to insert and delete new data objects in such an efficient way, enables the application to handle large batches of the input data stream.

5 Performance Evaluation

In this section, we present representative experimental results based on experiments conducted on a cluster of 32 physical machines running Spark with YARN and HDFS. The cluster is composed of one master with 32GB of main memory and 31 workers with 16GB of main memory each. The application has been coded in the Scala programming language and it is publicly available at [https://github.com/ioankall/Stream-Based-Distributed-ALOCI](https://github.com/ioankall/Stream-Based-Distributed-ALOCI). We have used two different datasets for evaluating the performance of the studied algorithms shown in Figure 2 and described briefly below:

- **CITIES.** This dataset contains lat-lon coordinates of cities all over the world and it is available at [https://github.com/petewarden/dstkdata/blob/master/worldcitiespop.csv](https://github.com/petewarden/dstkdata/blob/master/worldcitiespop.csv). The number of unique locations is $2 \cdot 10^6$.

- **MOG.** This dataset is synthetically generated by a mixture of Gaussian distributions. We generated different datasets containing $10^6$, $10^7$ and $10^8$ number of elements.

**FDALOCI vs. BDALOCI.** First, we compare the optimized algorithm against the baseline. Both algorithms are executed over the same hardware, on an append-only way, receiving the exact same input files in equal-sized batches consisting of 10,000 data elements. Even from the first batch, we reach to the conclusion that the optimized version is significantly more efficient than the baseline. In particular, the baseline required around 12 minutes to handle the first batch, while the optimized version requires about 15 seconds. When the

![Scatter plots of CITIES and MOG datasets.](image)
second batch is inserted, the optimized algorithm requires 16 seconds whereas the baseline requires 15 minutes. Based on these preliminary results which are shown in Figure 3, for the rest of the experimental evaluation we focus on the optimized algorithm only.

**Approximation Factor.** One of the most significant features of FDALOCI is that it provides an approximate way of calculating the LOCI scores and that the approximation factor is configurable. By managing the maximum allowed objects per node in the QuadTree, we can work on the trade-off between the accuracy of the results and the required runtime per batch. We ran an experiment, by running the Fast Distributed ALOCI over the same data set and the same hardware (a cluster with 32 executors), one time with $n = 20$ and one with $n = 40$.

![Efficient Distributed Outlier Detection in Data Streams](image)

**Fig. 3.** Comparing Baseline and Fast Distributed ALOCI algorithms.

Doubling the parameter $n$ leads to almost halving the runtime as shown in Figure 4, which is a significant boost especially for very large datasets. The

![Efficient Distributed Outlier Detection in Data Streams](image)

**Fig. 4.** Comparing runtime between different approximation factors.
question we need to answer is how much are the results affected by this change. Using the results returned by the algorithm with \( n = 20 \) as the ground truth, and defining outliers as the data objects that have a LOCI score greater than 3, we compared the outliers between the two executions of the algorithm counting the correct guesses along with the false positives and negatives. This results in an average precision of 96.4% and an average recall of 99.8% which were combined in an average F1-score of 98%. Additionally, we compared the LOCI scores of the same objects across the two different executions of the algorithm, which gave us an average root-mean-square error (RMSE) on the LOCI scores of 0.21. Whether this is a small price to pay for halving the required time or not, it depends on the application. The thing that really matters though, is that the approximation feature of the algorithm is actually configurable.

**Scalability.** To verify that FDALOCI scales well as the number of executors increases, we ran a series of tests. Each batch contains 200K elements, whereas the sliding window size is set to \( 10^7 \). Figure 5 depicts the runtime per batch for a different number of executors (8, 16, 32). It is evident that by increasing the level of parallelism there is a significant impact on performance. Notice that after the sliding window is full (this happens in the 50-th batch) the runtime per batch is constant. However, the performance gap is very large as the number of executors increases. The only case where the algorithm does not provide satisfactory speedup is when relatively small datasets are being used. This is illustrated in Figure 6(a), which shows that with 32 executors there is no significant difference in runtime in comparison to 16 executors. On the other hand, for large dataset increasing the number of executors provides significant performance improvement, as shown in Figure 6(b).

![Fig. 5. Scalability of FDALOCI for different input size and number of executors.](image)
Fig. 6. Average runtime per batch vs. number of executors.

6 Conclusions

Outlier detection is an important data mining task with many applications in diverse fields, which is a difficult problem on its own. Combined with the need to facilitate distributed computation over a continuous data stream the problem becomes significantly harder. In this paper, we have focused on the design of an efficient distributed outlier detection algorithm for data streams (FDALOCI) which is based on the concept of Local Correlation Integral. This algorithm is the first proposed in the literature attacking at the same time the distribution of the computation and the incremental processing of the input data stream with insertions and deletions. Based on the experimental results obtained, it turns out that the optimized version (FDALOCI) is efficient and also it scales well by increasing the number of executors. Future research in the area may involve:

- the use of alternative flexible indexing schemes to replace the QuadTree, in order to be more efficient in high dimensionalities with less memory footprint,
- the adaptation of the algorithm to support subspace outlier detection, which is based on the set of user-selected subset of dimensions.

References