Multi-Tactic Distance-based Outlier Detection

Lei Cao*, Yizhou Yan†, Caitlin Kuhlman†, Qingyang Wang†, Elke A. Rundensteiner†, Mohamed Eltabakh†
*Massachusetts Institute of Technology Cambridge, MA 02139, USA
lcao@csail.mit.edu
†Worcester Polytechnic Institute Worcester, MA 01609, USA
(yyan2, cakuhlman, wangqy, rundenst, meltabakh)@cs.wpi.edu

***Abstract*—As datasets increase radically in size, highly scalable algorithms leveraging modern distributed infrastructures need to be developed for detecting outliers in massive datasets. In this work, we present the first distributed distance-based outlier detection approach using the MapReduce-based infrastructure, called *DOD*. *DOD* features a single-pass execution framework that minimizes communication overhead. Furthermore, DOD overturns two fundamental assumptions widely adopted in the distributed analytics literature, namely *cardinality-based load balancing* and *one algorithm for all data*. The multi-tactic strategy of DOD achieves a *truly balanced workload* by taking into account the data characteristics in data partitioning and assigns most appropriate algorithm for each partition based on our theoretical cost models established for distinct classes of detection algorithms. Thus, DOD effectively minimizes the end-to-end execution time. Our experimental study confirms the efficiency of DOD and its scalability to terabytes of data, beating the baseline solutions by a factor of 20x.**

I. Introduction

Motivation. Outlier detection is recognized as an important data mining method [1]. It concerns the discovery of abnormal phenomena that may exist in the data, namely data values that deviate significantly from the common trends in the data [2]. Outlier detection is critical in many applications ranging from credit fraud prevention, network intrusion detection, stock investment tactical planning, to disastrous weather forecasting. For such mission-critical applications, the anomalies (outliers) must be detected efficiently and in a timely manner. Even a short time delay may lead to losses of huge funds, investment opportunities, or even human lives.

Distance-based outlier detection [3], one of the most popular outlier detection techniques, has been widely adopted in many applications [1]. In the seminal distance-based outlier definition, also called distance-threshold outlier [3], a data point p is considered to be an outlier if it has very few neighbors within a certain distance range. Despite this simplicity, state-of-the-art centralized algorithms [3]–[5] that realize this popular outlier semantics can no longer satisfy the stringent response time requirements of emerging big data applications, especially now that the data itself is inherently becoming more distributed. Therefore, the development of highly distributed solutions for distance-based outlier detection is no longer an option, but a necessity.

Numerous Map-Reduce [6] style distributed computing platforms from Hadoop [7], Spark [8] to Hadapt [9] have become prevalent in recent years. This is due to their desirable features including scalability to thousands of machines, flexibility in the data model, efficient fault tolerant execution, and cost effectiveness as open-source technologies. Nevertheless, despite the importance of outlier detection and the popularity of these compute infrastructures, no work has been proposed to date to leverage their power in developing an efficient distance-based outlier detection solution.

State-of-the-Art. No MapReduce-based distributed algorithm has been designed to date for supporting this popular distance-based outlier semantics [3]. Some distributed techniques focus on distinct outlier detection semantics such as kNN-based outliers [10]. Furthermore, they utilize a message-passing distributed system, e.g., [11]–[13]. These specific purpose architectures suffer from heavy centralized and bottlenecked computations before the distribution across the slave nodes [13], and/or require strict synchronization and exchange of data among the slave nodes [11], [12] — which limits the effectiveness of the distribution strategies. As demonstrated by their experiments [11]–[13], these techniques cannot process datasets larger than 10G − and thus clearly are not being able to meet the rising requirements of this big data era.

Unlike these systems, the *modern shared-nothing infrastructures* neither assume centralized processing in any stage nor require synchronization between the worker nodes. Therefore, a MapReduce-based outlier detection solution would have the potential to achieve higher scalability and better efficiency than prior limited infrastructures. Plus, use of these widely adopted open-source compute paradigms to solve our proposed detection problem will assure broader adoption as well as the possible transfer of our proposed strategies to tackle other data analytics problems beyond outliers.

Moreover, a common limitation in all prior work [11]–[13] is that they apply *one single* outlier detection algorithm to all compute nodes. This simplistic "monolithic" detection approach is based on the implicit assumption that one central detection algorithm is superior to all other algorithms for all types of datasets. However, we observe that although a number of centralized algorithms have been proposed to speed up the outlier detection process, e.g., [3]–[5], none has shown consistent superiority in all circumstances. This is confirmed both by our theoretical analysis as well as by our empirical study in Sec. IV. Instead, their performance strongly varies depending on the characteristics of the dataset being processed. As we will demonstrate, such misguided assumption overlooks the well-known fact that real world datasets tend to be skewed [1]. Therefore instead distinct algorithms should be assigned to different data partitions to minimize the overall costs of the

This project is partially supported by NSF-CRI 1305258 grant.

distributed outlier detection process.

In this work, we target the design and development of a scalable MapReduce-based outlier detection technique that overcomes aforementioned limitations while minimizing the end-to-end execution time.

Challenges. The design of an efficient distributed outlier detection approach is challenging. First, we must develop an **effective partitioning strategy**. Intuitively, a random partitioning solution would spread the neighbors of one data point into possibly many compute nodes. Therefore, a point p would not be able to discover all its neighbors within its local partition. This inevitably would lead to a multi-job map-reduce solution, causing prohibitive costs involved in reading, writing, and re-distribution of the data over a series of separate jobs. In contrast, domain-based partitioning that takes the locality of the data into account may do a better job, grouping nearby data points together on one compute node. This would increase the chances for a given point p to find its neighbors locally on the machine where it resides. However, real-world datasets tend to be skewed over the domain space [1]. Therefore, domain-based partitioning struggles with the inherent problem of **load imbalance**, which may not only cause a significant slowdown in processing time, but also risks excessive job failures [14].

Second, to solve the problem caused by the traditional "monolithic" detection approach, we must **assign an appropriate detection algorithm** to each partition optimized to leverage its data characteristics. This requires a thorough understanding of the correlations between the characteristics of the data and the performance of the respective classes of algorithms. However, to date no such work appears in the literature. Furthermore, a mechanism must be designed that determines the most suitable algorithm for each partition *on the fly* when allocating data partitions to each compute node.

Third, even more challenging is that the partition generation problem (Challenge 1) and the algorithm-selection problem (Challenge 2) are **strongly interdependent**. On the one hand, the partitioning should consider the performance properties of different detection algorithms to orchestrate its partitions so that the selected algorithms can best utilize the characteristics of the dataset. On the other hand, the algorithm assignments must be determined based on the characteristics of the data subsets produced by the partitioning plan. This raises the proverbial chicken and egg question.

Proposed Approach. In this paper, we propose the first Distributed Outlier Detection approach using MapReduce, called *DOD*, that detects the distance-threshold outliers hidden in large distributed datasets with minimized execution time.

As foundation, we first design the distributed framework for *DOD*, that correctly discovers all outliers in *one single pass*. This is achieved by DOD leveraging the general notion of *support* [15], [16]. It ensures that each core point p will have all information within its local partition needed to classify it as either outlier or inlier by replicating so called support points into the partition . Therefore any centralized algorithm can be applied independently on each partition to detect outliers.

We further enhance *DOD* with our novel *multi-tactic* approach to minimize the end-to-end execution time by optimizing both the map-side partitioning and the reducer-side detection algorithm selection in an *integrated fashion*. This integrated solution leverages two key observations on *load balancing* and *on performance of detection algorithms* as described below.

The first observation concerns that the traditional cardinality-based load balancing assumption, namely that an equal input data size would lead to roughly equal workload, — although commonly adopted by distributed analytics work [14], [17]–[19] — does not hold for outlier detection. We observe that the computational costs of an outlier detection algorithm on a partition P do not only depend on the number of points in P, i.e., its cardinality, but also on its *data density* — the ratio of data cardinality to the domain area. Based on this observation, we establish the first theoretical cost models customized for distinct classes of central detection algorithms [3], [5]. This in turn provides the theoretical foundation for a true cost-based load balancing strategy.

Our second observation is based on our analysis of the cost models characterizing different classes of prominent distance-based outlier detection algorithms from the literature [5]. We observe that the *density* of a data partition P determines which algorithm performs better on P. In other words, we uncover that data partitions with similar density characteristics are best served by the same detection algorithm.

Leveraging these two observation, our density-aware *multi-tactic* approach *DMT* successfully solves the *highly interdependent partition-generation and algorithm-selection problems in one step* while ensuring a balanced work load across different compute nodes.

Generality of the Proposed Methodology. The key observations and corresponding optimization principles put forth in this work are not limited to the MapReduce framework. On the contrary, they are equally applicable to other shared-nothing distributed platforms as long as the architecture does not require synchronization among the compute nodes. Moreover, these principles are also of potential value to a much broader class of distributed data analytics tasks beyond outlier detection. For example, our analysis and cost models reveal that the widely adopted load balancing assumption does not necessarily hold. This opens interesting research questions in the context of other classes of techniques such as association rule mining [20] and reverse kNN [21]. Similarly, our *multi-tactic* approach is not specific to distance-based outliers, since the observation that no single algorithm is superior to all others for all datasets can be shown to hold also in other contexts such as the above mentioned techniques.

Contributions. The key contributions of our work include:

- We propose the first distributed approach that supports distance-based outlier detection using MapReduce.
- We design a general framework called *DOD* that using the *supporting area* technique detects all outliers in a single MapReduce job.
- For the first time, we theoretically analyze and contrast the costs of classes of outlier detection algorithms under a variety of data distributions. Based on this theoretical foundation, we prove that the traditional cardinality-based load balancing assumption does not hold in the outlier detection context.

• We furthermore propose a *multi-tactic* strategy that automatically selects the best outlier detection algorithm for a given data partition. Our proposed density-aware technique successfully solves the two interdependent problems of partition-generation and algorithm-selection in one step.

• We experimentally evaluate the *DOD* technique using terabyte sized real world datasets. The results demonstrate that our technique outperforms the baseline solutions by a factor of 20x and drives down the execution time from days to hours, making outlier detection for the first time practical in big data.

II. Preliminaries

Let $D = \{p_1, p_2, ..., p_n\}$ be a dataset composed of n d-dimensional points, where each point $p_i \in D$ is represented as $p_i = \langle v_{i_1}, v_{i_2}, ..., v_{i_d} \rangle$. Assume the function $dist(p_i, p_j)$ is a distance function that measures the proximity between two points p_i and p_j.

Definition 2.1: Two points p_i and p_j in D are said to be **neighbors** if $dist(p_i, p_j) \leq r$, where r is a distance threshold.

The neighbor set of p_i w.r.t a distance threshold r in D is denoted as $\mathbb{N}^r(p_i)$, with $|\mathbb{N}^r(p_i)|$ denoting its cardinality.

Definition 2.2: Given a distance threshold r and a neighbor-count threshold k, a data point p_i is said to be a **distance-based outlier** iff $|\mathbb{N}^r(p_i)| < k$.

Given the distance-based outlier parameters r, k and a dataset D distributed over HDFS in a Hadoop cluster, *our goal is to discover and report all outliers in D with minimal end-to-end execution time*.

III. Distributed Outlier Detection Framework

In this section, we introduce the basic *DOD* distributed framework that discovers distance-based outliers using a single MapReduce job. The key of this framework is the *supporting area* partitioning strategy that ensures the outliers in each partition can be detected in isolation from the other partitions. Therefore any centralized algorithm can be applied independently on each partition to detect outliers.

A. The DOD Framework

DOD involves three key steps, namely *grid cell partitioning*, *enhancement with supporting area*, and *parallelized outlier detection*.

Step 1: Grid Cell Partitioning. As a first step, *DOD* partitions the entire domain space $Domain(D)$ of a dataset D into m disjoint grid cells C_i such that $C_1 \cup C_2 \cup ... \cup C_m = Domain(D)$. A grid cell is formally defined below.

Definition 3.1: A **grid cell** C_i in a d-dimensional domain space is a hyper rectangle $C_i = \langle (low_{1i}, high_{1i}), (low_{2i}, high_{2i}), ..., (low_{di}, high_{di}) \rangle$, where $(low_{xi}, high_{xi})$ are the boundaries of C_i in the x^{th} dimension, where $1 \leq x \leq d$.

The points inside cell C_i, referred to as C_i's **core points**, are denoted as $C_i.core = \{p_j \mid p_j \in C_i\}$. The areas of the domain space covered by each grid cell need not be of equal size. In general, any partitioning strategy could be utilized to produce such grid cells. Fig. 1 depicts a two-dimensional space partitioned into grid cells using an *equi-width partitioning* method. It divides each dimension of the domain space into equal width segments. Then points are grouped based on their membership in a particular grid cell. Fig. 1 shows such a grouping by representing the points in each grid cell with the same shape.

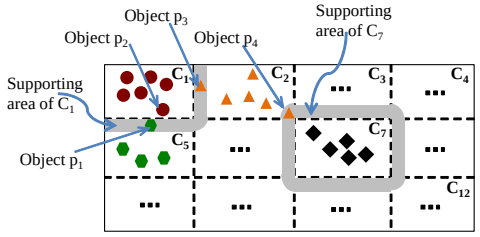


Fig. 1: DOD Framework.

Step 2: Enhancement with Supporting Areas. The data points inside each grid cell are not sufficient for detecting the outliers in each cell independent from the other cells. For example, data point p_2 in grid cell C_1 appears to be an outlier when considering only grid cell C_1. However, p_2 may have neighbor points in grid cell C_5, e.g., p_1, which may make p_2 an inlier. To break such dependency between the grid cells and thereby enable true parallelization for outlier detection among different grid cells, we introduce the notion of *supporting area*. As formally defined in Def. 3.2, the data points within the *supporting area* of cell C_i may affect the outlier decision of at least one core point of C_i.

Definition 3.2: The **supporting area** of a grid cell C_i, denoted as $C_i.suppArea$, is an extension of the boundaries of C_i in the dimensions of D such that $C_i.suppArea$ exactly contains all points p_j, also called support points, iff p_j satisfies the following two conditions: (1) $p_j \notin C_i.core$, and (2) there exists at least one point $p_k \in C_i.core$ such that dist(p_j, p_k) $\leq$ r, where r is the distance threshold parameter in Def. 2.2.

Fig. 1 highlights in grey the supporting areas of grid cells C_1 and C_7 respectively. Each grid cell C_i will now be augmented with its support points in addition to its core points. For example, C_1 will be extended to contain *support points* $\{p_1, p_3\}$ in C_1's supporting area, along with its circle-shaped *core points*.

Next we prove that enhanced by the supporting area, each grid cell C_i now would have enough information to classify the core points in C_i as inliers or outliers.

Lemma 3.1: For a given grid cell C_i, the support points in $C_i.suppArea$ are the **necessary and sufficient set of points** to determine the outlier status of the core points in C_i.

Proof. *"Necessity" Proof.* By Def. 3.2, any point $p_j \in C_i.suppArea$ is the neighbor of at least one point $p_i \in C_i$. If p_j is excluded from $C_i.suppArea$, then possibly p_i in C_i would have been falsely reported as an outlier if p_i happens to only acquire $k - 1$ neighbors.

"Sufficiency" Proof. Any data point $p_j \notin C_i.suppArea$ cannot become the neighbor of any point $p_i \in C_i$ because it is not within threshold r distance from any $p_i \in C_i$. Therefore p_j has no influence on the decision of whether or not p_i is an outlier by the outlier definition in Def. 2.2. ■

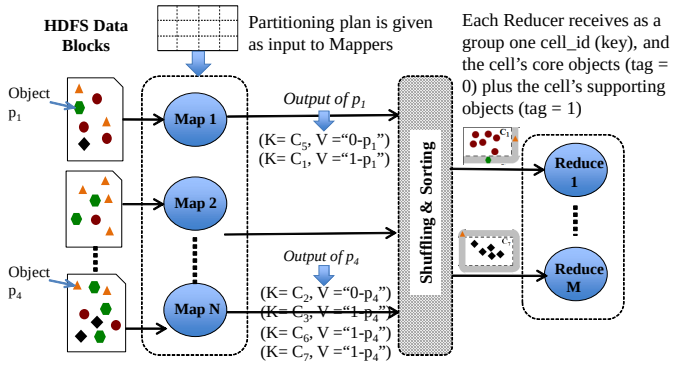


Fig. 2: MapReduce Implementation of DOD.

Step 3: Parallelized Outlier Detection. The final step is to apply a centralized outlier detection algorithm, e.g., the *Nested-Loop* algorithm [3], to each of the grid cells C_i to identify the outliers among the core points contained within that cell only. This step can be applied to each grid cell in total isolation from the other cells, i.e., distributed to different machines. Since each point of D is the core point of exactly one single cell, this correctly leads to DOD identifying all outliers.

B. MapReduce Implementation of DOD

We sketch one MapReduce strategy of the DOD framework in Fig. 2. This can be easily adapted to support other mining tasks that can take advantage of the supporting area partitioning strategy, such as density-based clustering [16] and LOCI outlier detection [22].

For the ease of implementation, instead of directly applying the supporting area definition in Def. 3.2, we utilize the simplified definition in Def. 3.3.

Definition 3.3: Given a d-dimensional grid cell C_i, the supporting area of C_i is an r-extension to the boundaries of C_i in each dimension. That is, $C_i.suppArea = \langle (low_{1i} - r, high_{1i} + r), (low_{2i} - r, high_{2i} + r), ..., (low_{di} - r, high_{di} + r) \rangle - C_i$, where $(low_{xi}, high_{xi})$ are the boundaries in the x^{th} dimension of C_i ($1 \leq x \leq d$).

Since the supporting area defined in Def. 3.3 is a superset of the supporting area in Def. 3.2, it is guaranteed to be sufficient to support each grid cell to be processed independently without relying on the points in any other cell.

The pseudocode of the map and reduce functions of DOD is presented in Figure 3. The input dataset, which resides in HDFS, has no prior partitioning properties, i.e., the data points are randomly distributed over the HDFS blocks. Each map function retrieves one data block and the space partitioning strategy (Fig. 2). Then for each data point p_i, the map function produces two types of output records, i.e., core- and supporting-related records.

A **core-related** record is one key-value pair record (K = C_i, V ="0-p_i"), where key is the ID of the grid cell for which p_i is a core point, i.e., $p_i \in C_i$. The prefixed flag "0" in the value component indicates that p_i is a core point for C_i (Lines 2-3 in the map function in Figure 3). For example, in Fig. 2, the mapper *Map 1* generates output record (K = C_5, V ="0-p_1") for point p_1.

Map (key k, value p) Function
Auxiliary Inputs
- S ➔ Space partitioning strategy
- r ➔ distance threshold *of* the outlier algorithm

1- **Begin**
2- C_i ← Compute the grid cell in which p is a core point;
3- output (C_i, "0-p");
4- **For** (each grid cell C_j in which p is a supporting point) **Loop**
5- output (C_j, "1-p");
6- **End Loop**
7- **End**

Reduce (key cell-Id, v-list [p_1, p_2, …, p_m]) Function
Auxiliary Inputs
- k ➔count threshold *of* the outlier algorithm
- r ➔ distance threshold *of* the outlier algorithm

1- **Begin**
2- core-list ← the set of points in v-list with prefix tag = "0"
3- support-list ←the set of points in v-list with prefix tag = "1"
4- outlier-list ← Execute OutlierDetection(v-list, r, k);
5- **For** (each outlier o in outlier-list) **Loop**
6- **If** (o exists in core-list) Then
7- output (null, o) // Report Outliers
8- **End Loop**
9- **End**

Fig. 3: MapReduce pseudocode of the DOD Framework.

Mappers also create zero or more **supporting-related** records for a point p_i in the form of (K = C_j, V ="1-p_i"), where key $p_i \in C_j$ is the ID of the grid cell for which p_i is a support point, i.e., $p_i \in C_j.suppArea$. The prefixed flag "1" in the value component indicates that p_i is a support point for C_j (Lines 4-6 in the map function in Fig. 3). For example, in Fig. 2, the mapper *Map N* generates three additional output records for point p_4 since it is a support point for C_3, C_6, and C_7.

After the internal shuffling and sorting phases based on the cell ID, each group of records received by a reducer will correspond to a specific grid cell, say C_i. It will consist of the union of the core and support points belonging to C_i (See Fig. 2). The reducer function categorizes the data points according to their attached flag encoded in the value (lines 2-3 in the reduce function in Fig. 3). Lastly, it executes an outlier detection algorithm to detect outliers among the $C_i.core$ set (lines 4-8) while using both $C_i.core$ and its $C_i.support$ as possible neighbors.

C. Multi-Tactic Outlier Detection Problem Formalization

Based on the one pass DOD framework(Sec. III-A), we now are ready to formally define our **multi-tactic outlier detection optimization problem**.

A ***partition plan*** of dataset D is a set of m pairwise *disjoint* partitions whose union *covers* the domain space of D, where m is the number of the partitions specified as an *input parameter*. We use $P(D)$ to denote a particular partition plan of D and $\mathfrak{P}(D)$ to denote the set of all possible partition plans of D.

Given a partition plan $P(D)$, an *algorithm plan* $AP(D, P)$ of $P(D)$ is a set of m detection algorithms, namely one for each of the m partitions in $P(D)$ selected from an algorithm candidate set $\mathfrak{A}$.

Definition 3.4: The **optimal algorithm plan** with respect to a partition plan $P(D)$, denoted as $AP^{opt}(D,P)$, is the algorithm plan $AP(D,P)$ in which the algorithm $A_i \in AP(D,P)$ assigned to the corresponding partition $P_i \in P(D)$ ($1 \leq i \leq m$) corresponds to the algorithm with minimal processing costs compared to any other algorithm $A_j \in \mathfrak{A}$, namely, $cost(A_i, P_i) = min(cost(A_j, P_i) | \forall A_j \in \mathfrak{A})$.

The cost of the partition plan $P(D)$ with respect to its optimal algorithm plan $AP^{opt}(D,P)$, denoted as $cost(P(D))$, is defined as $cost(P(D)) = max\{cost(A_i, P_i) \mid A_i \in AP^{opt}(D,P), P_i \in P(D), 1 \leq i \leq m\}$. Intuitively $cost(P(D))$ represents the processing costs of the most expensive partition, which in turn indicates the end-to-end execution time of the outlier detection process.

Multi-tactic optimization can now be defined as below.

Definition 3.5: Given all possible partition plans $\mathfrak{P}(D)$ of *D*, the **multi-tactic optimization problem** MT is to find a partition plan $P(D)_{opt}$ and the corresponding algorithm plan $AP^{opt}(D,P)$, such that $cost(P(D)_{opt})$ is minimal among $cost(P(D)_j)$ of all possible $P(D)_j$ in $\mathfrak{P}(D)$.

Complexity Analysis of Multi-tactic Optimization Problem. Given a dataset *D* with a continuous domain space $Domain(D)$, there are an *infinite* number of options to divide $Domain(D)$ into *m* partitions. Therefore the search space of the partitioning problem itself is infinite. Even if $Domain(D)$ were to be discretized, the search space of the multi-tactic optimization is still prohibitively large.

Let $C = |Domain(D)|$ denote the cardinality of $Domain(D)$. Then the cardinality of $\mathfrak{P}(D)$, namely, $|\mathfrak{P}(D)| = \frac{1}{m!}\sum_{j=1}^{m}(-1)^{m-j}\binom{n_i}{k}j^C$ is exponential in C. Clearly for large C, the search is prohibitively expensive. Furthermore, if the optimal algorithm plan has to be generated for each partition plan in $\mathfrak{P}(D)$, then the overall complexity of the problem search space would be $O(|\mathfrak{P}(D)|) * m * E)$. Here *E* denotes the cost of finding a best algorithm for one partition. Worst yet, to date no formal cost models have been established for the typical outlier detection algorithms found in the literature. In essence, the complexity of MT problem (Def. 3.5) originates from two key factors: (1) the combinatorial number of possible partition plans and (2) the strong interdependency of the partitioning and algorithm selection subproblems. Given this exponential complexity of the *MT* problem, it is thus imperative that efficient yet effective search heuristics are devised for tackling the MT problem defined in Def. 3.5.

IV. Key Observations

Next, we introduce the key observations upon which our multi-tactic approach is built, namely *load balancing* and *algorithm performance* in the context of outlier detection.

A. Load Balancing Observation

By Definitions 3.4 and 3.5, the objective of our multi-tactic problem is to minimize the end-to-end execution time which is determined by the processing costs of the most expensive partition. To achieve this, the whole workload has to be *equally* divided into a limited number of compute nodes – in other words into a balanced workload. In fact load balancing has been shown to be one of the most critical problems for distributed data processing. Load imbalance may not only result in significant slowdown, but also cause job failures [14].

In the literature most distributed analytical algorithms [11]–[13] ensure load balancing by adopting the traditional load balancing assumption that *an equal number of data points leads to a balanced workload*. In this work we overturn this fundamental assumption in the context of distance-based outlier detection supported by an empirical study and a theoretical analysis. Our cost model established for the typical class of outlier detection algorithms demonstrates that both the cardinality and the distribution characteristics (data density) determine the processing costs of a dataset. This model provides a theoretical foundation for the design of cost-based partitioning strategies that generate partitions of balanced workloads.

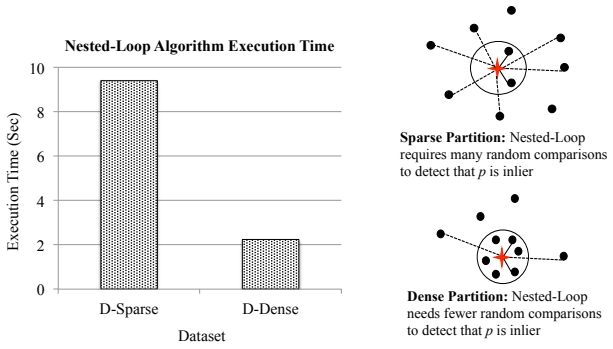
For this, we first design an experiment to study the effect of the data's density on the execution of central outlier detection algorithms (Figure 4). We use two datasets, each consisting of the same number of data points (100KB). However their densities are very different, where *D-Dense* is much "denser" than *D-Sparse*. Here density is defined as the *ratio of data cardinality to the domain area covered by the data*. The domain area covered by the *D-Dense* dataset is only $\frac{1}{4}$ of the domain area covered by the *D-Sparse* dataset. By the above measure, *D-Dense* is four times denser than *D-Sparse*.

We then apply the *Nested-Loop* algorithm [3] to both datasets with the *r* and *k* parameters set to 5 and 4 respectively. The *Nested-Loop* algorithm is a popular albeit simple algorithm for distance-based outlier detection. Its logic is based on the following idea. Given a data point p_i, the algorithm evaluates the distance between p_i and other points in the dataset *D* in random order until either *k* neighbors of p_i are found (p_i becomes inlier), or all data points in *D* have been examined (p_i becomes outlier). As depicted in Figure 4, although the input data size and the algorithm's input parameters are exactly identical, the execution performance is entirely different, namely, 4.5x slower in the case of *D-Sparse* as compared to *D-Dense*.

The intuition is that the points in *D-Dense* are closer to each other. Thus finding enough neighbors of a point p_i within a distance r to declare p_i as inlier is relatively faster in *D-Dense* than in *D-Sparse*. That is, the likelihood that a randomly picked point in *D-Dense* is a neighbor of p_i is higher than in *D-Sparse*. Since outliers tend to be rare and thus the vast majority of the points can be expected to be inliers, the algorithm applied to *D-Dense* will terminate early for most points. This explains the significantly lower overall costs for *D-Dense* compared to *D-Sparse*. This experiment confirms that the processing costs do not only depend on the dataset's cardinality, but also on its density over the domain space.

Next, we theoretically support this observation by establishing a formal cost model (Lemma 4.1) for the family of outlier detection algorithms that rely on random selection and comparison among the data points, such as Nested Loop [3].

Lemma 4.1: Given a uniformly-distributed dataset D of cardinality $|D|$ data points, and parameter settings k and r for the distance-based outlier algorithm, the cost of detecting



(a) Experimental Evaluation **(b) Sparse vs. Dense Partitions**

Fig. 4: Sensitivity of Nested-Loop's Performance to Dataset Densities.

distance-based outliers based on random selection and comparison is $Cost(D) = \frac{|D| \times A(D) \times k}{A(p_i)}$, where $A(D)$ and $A(p_i)$ represent the areas of the domain space covered by the dataset D and by the distance parameter r around p_i, respectively.

Proof. Since D follows a uniform distribution, we have:

$$Cost(D) = Cost(p_i) \times |D| \tag{7}$$

where $Cost(p_i)$ denotes the cost of determining whether or not a data point p_i is outlier, since p_i has on average $|D| \times \frac{A(p_i)}{A(D)}$ neighbors in D. Then, given any randomly picked point p_j, the probability that p_j is a neighbor of p_i denoted as μ equals to:

$$\mu = |D| \times \frac{A(p_i)}{A(D)} \; / \; |D| = \frac{A(p_i)}{A(D)} \tag{8}$$

The cost of processing p_i, denoted by $Cost(p_i)$, is determined by the number of trials N to acquire k neighbors. Considering the event that a randomly picked point is or is not a neighbor of p_i as a binary variable, then the probability of observing k occurrences of neighbors in a set of N samples (random trials) follows the Binomial distribution $Bin(k \mid N, \mu) = \binom{N}{k} \mu^k (1-\mu)^{N-k}$. The expected value of N is $E(N) = \frac{k}{\mu}$, which leads to:

$$Cost(p_i) = \frac{k}{\mu} = \frac{k}{\frac{A(p_i)}{A(D)}} = \frac{k \times A(D)}{A(p_i)} \tag{9}$$

By substitution in Eq. (7), we get $Cost(D) = \frac{|D| \times A(D) \times k}{A(p_i)}$, which proves the lemma. ■

Based on Lemma 4.1, the cost of detecting the distance-based outliers in D relies on both the number of the data points, i.e., $|D|$, and the domain space area covered by the dataset $A(D)$. Since the domain space covered by a sparse dataset *D-Sparse* is larger than the domain space covered by a dense dataset *D-Dense*, we thus can conclude that *Cost(D-Sparse)* $>$ *Cost(D-Dense)*.

B. Algorithm Performance Observation

To the best of our knowledge distributed analytics work on shared-nothing infrastructures typically applies *one single* analytics algorithm to all compute nodes. This "monolithic" approach is based on the implicit assumption that one central analytics algorithm is superior to the other known algorithms. However both our empirical study and theoretical analysis have shown that although a number of centralized algorithms have been proposed including *Nested-Loop* and *Cell-Based* [3], [5], no conclusive winner emerged that consistently outperformed all other algorithms on all datasets. Furthermore, our theoretic analysis has revealed that the *density* of each data partition is a key factor that determines the performance of these algorithms. This provides a solid foundation for translating the observation of selecting different algorithms for different data partitions into actionable multi-tactic optimization decisions.

Similar to *Nested-Loop* introduced in Section IV-A, the *Cell-Based* algorithm [3] is another popular detection algorithm. As an index-based solution, it relies on pruning strategies to avoid unnecessary checking points. First, it uniformly partitions the domain space into a set of d-dimensional non-overlapped grid cells, where the length of the cell in each dimension is $r/2$ and r is the distance threshold input parameter. Then it hashes each point to exactly one grid cell. Each cell maintains the number of points it contains so that the algorithm can quickly identify all grid cells that have no outlier or no inlier. Both types of cells can be excluded from any further processing. The points in the remaining grid cells have to be evaluated individually, in a fashion similar to *Nested-Loop*.

First, we evaluate the performance of *Nested-Loop* versus *Cell-Based* under different data densities. Again *density* is defined as *the ratio of data cardinality to the domain area covered by the data*. In this experiment, we vary the density of the datasets by varying the size of the domain area while keeping the number of data points constant as 10,000 (Sec. IV-A). The r and k parameters are set as 5 and 4, respectively.

The results (Fig.5) confirm our expectation that density matters, and no algorithm is superior in all cases. Better yet, we observe a general trend in Figure 5. Namely, the *Cell-Based* algorithm outperforms *Nested-Loop* when the data is either very sparse or very dense. In contrast, in the intermediate density cases the *Nested-Loop* algorithm is faster.

Intuitively *Cell-Based* performs well when handling very sparse or very dense datasets, because in both cases, many of the d-dimensional grid cells can be directly marked as outliers (in the very sparse case) or as inliers (in the very dense case). This then saves computations. In other cases, deciding on the outlier status of the data points requires more computations on top of the pre-processing phase. Here, *Cell-Based* suffers from the additional overhead of having to index the data points. It thus performs worse than *Nested-Loop*.

Next, we theoretically support this observation based on the cost model we establish for the *Cell-Based* algorithm (Lemma 4.2). Without loss of generality, we use a two-dimensional dataset as example.

Lemma 4.2: Given a uniformly distributed two-dimensional dataset D of cardinality $|D|$ and data points that cover a domain space of area $A(D)$, the cost of detecting distance-based outliers using the *Cell-Based* algorithm with parameters r and k is defined as follows:

(1) $Cost(D) = |D|$ If $\frac{9}{8} r^2 \times \frac{|D|}{A(D)} \geq k$;

(2) $Cost(D) = |D|$ If $\frac{49}{8} r^2 \times \frac{|D|}{A(D)} < k$;

(3) $Cost(D) = |D| + \frac{|D| \times A(D) \times k}{\pi \times r^2}$ Otherwise

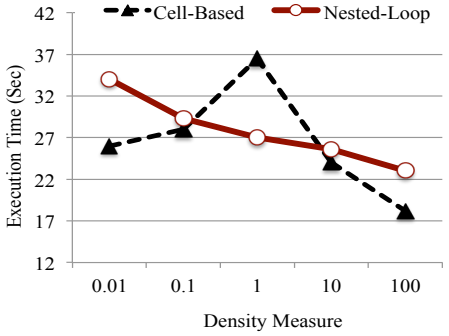


Fig. 5: Performance of Different Detection Algorithms w.r.t Data's Densities.

Proof. In the *Cell-Based* algorithm, the area covered by each cell C_i corresponds to $A(C_i) = \frac{1}{2}(\frac{r}{2})^2 = \frac{r^2}{8}$, where $\frac{r}{2}$ is the diameter of one rectangular cell. According to the algorithm, given a cell C_i, if there are more than k points in C_i and its direct adjacent cells (nine cells in total), then all data points in C_i are marked as inliers. In other words, if $\frac{9}{8}r^2 \times \frac{|D|}{A(D)} \geq k$, then the cost of processing the entire dataset is equivalent to scanning and indexing the data points. That is $Cost(D) = |D|$. This proves Equation (1) of Lemma 4.2.

On the other hand, if there are fewer than k points in the combined cells of C_i and the cells within $2r$ distance from it (in total 49 cells), then all points in C_i are guaranteed to be outliers without requiring any explicit comparisons. In other words, if $\frac{49}{8}r^2 \times \frac{|D|}{A(D)} < k$, then $Cost(D) = |D|$. This proves Equation (2) of Lemma 4.2.

If neither of the two aforementioned cases holds, then the unmarked cells need to execute a *Nested-Loop* algorithm, in addition to the indexing costs of the entire dataset. That is, the cost will be $Cost(D) = |D| + \frac{|D| \times A(D) \times k}{\pi \times r^2}$, where $\frac{|D| \times A(D) \times k}{\pi \times r^2}$ represents the cost of *Nested-Loop* as proven in Lemma 4.1. This proves Equation (3) of Lemma 4.2. ■

According to Lemma 4.2, in the extreme cases of very sparse and very dense, the cost of *Cell-Based* is linear w.r.t $|D|$. Thus it outperforms *Nested-Loop*. Whereas in the other cases, it is more expensive than *Nested-Loop* due to the overhead introduced by the indexing phase without gaining any added benefit from this extra step. Therefore given a dataset *D*, the algorithm $Alg(D)$ best serving *D* can be determined by applying the following corollary.

Corollary 4.3: Given a two-dimensional dataset D of cardinality $|D|$ and data points that cover a domain space of area $A(D)$, the algorithm that most efficiently detects the distance-based outliers with parameters r and k is:

(1) $Alg(D) = Cell-based$ If $\frac{9}{8}r^2 \times \frac{|D|}{A(D)} \geq k$;

(2) $Alg(D) = Cell-based$ If $\frac{49}{8}r^2 \times \frac{|D|}{A(D)} < k$;

(3) $Alg(D) = Nested-Loop$ Otherwise

V. Density-Aware Multi-Tactic Optimization

Leveraging the load balancing and the detection algorithm performance observations introduced in Sec. IV, we propose a simplistic yet effective *density-aware multi-tactic* approach (*DMT*) that generates a partition plan and algorithm plan pair.

By Lemma 4.2, since the density of a partition *P* determines which algorithm performs better on *P*, partitions with similar densities are expected to share the same appropriate detection algorithm. Thus the overall idea is the following, while the detailed algorithm described below.

Overview of DMT. Given a dataset *D*, its continuous domain space is first discretized into a large number of small regions called *mini buckets*. DMT clusters the mini buckets with similar densities into larger clusters (partitions). Next the best detection algorithm is selected for each cluster based on its density by Corollary 4.3. By this, the partitioning already caters to the subsequent algorithm assignment goal. Hence the algorithm plan (AP as defined in Sec. III-C) is naturally derived based on the data characteristics of each cluster. Therefore the interdependency deadlock of partitioning versus algorithm selection is broken. Lastly, a cost-aware allocation algorithm is introduced to allocate the clusters (partitions) to each reducer. This algorithm accurately estimates the detection costs of each partition using the cost model of the selected algorithm. The estimated costs are then utilized to balance the workload of each reducer. Therefore this approach abandons the traditional cardinality-based load balancing assumption. The *DOD* framework enhanced with the *DMT* optimization *effectively minimizes the end-to-end execution time*.

A. The Density-Aware Multi-tactic Approach

DMT executes a lightweight pre-processing job to generate a multi-tactic plan composed of a partition plan and a algorithm plan pair. This pre-processing phase is composed of two stages, namely *distribution estimation* and *multi-tactic plan generation*. Both stages can be performed using one MapReduce job.

In the first stage, *DMT* estimates the distribution of the data by drawing a sample from the input dataset. We opt for random sampling since it preserves the distribution of the underlying dataset [23]. Since we only need to roughly estimate the distribution, the sampling rate Υ as an input parameter by default is set to a small value, e.g., 0.5 %. Considering the size of big datasets, the sample is generated in a distributed fashion by drawing samples within the map phase of a MapReduce job. The map tasks assume the entire data space is discretized to *"mini buckets"* that form the unit of processing. The map task then will aggregate the individual sample points and produce the statistics at the *mini bucket* level. Then, the mappers' output is passed to a centralized node, i.e., a single reducer, for the plan generation stage.

The reduce function is more sophisticated. It involves three steps, namely, (1) density and spatial-aware hierarchical clustering (DSHC), (2) algorithm plan generation and (3) partition allocation.

DSHC algorithm. In Step 1, we group the mini buckets with similar densities into one partition such that each partition best conforms to one particular detection algorithm. This key step of DMT corresponds to a highly customized multi-objective clustering problem with constraints or in short *MOC* clustering.

(1) First, it is both *density* and *spatial-aware*. It takes the spatial properties of the clusters into account, i.e., only the spatially-adjacent clusters of similar densities are considered for possible expansion. Grouping the adjacent data points

together in one partition reduces the size of the supporting area, avoiding unnecessary data duplication.

(2) Only *rectangle-shaped* clusters are supported to keep the resulting plan simple, because partitioning the data based on an irregular shaped plan (map phase of DOD framework, Sec. III-B) is expensive − possibly even more expensive than the actual outlier detection process.

(3) It solves a *constrained clustering* problem. The cardinality of each cluster is upper bounded by a threshold corresponding to the maximum number of data points that a single reducer can handle in main-memory.

Inspired by the traditional BIRCH clustering algorithm [24], we thus design a *density and spatial-aware hierarchical clustering (DSHC)* algorithm that efficiently solves this problem with a single scan of the mini buckets.

The key realization of DSHC relies on a well-designed *Aggregate Features (AF)* data structure and a R-tree like index structure to hold the AF information in its node as well as indexing spatial information, called *AF tree*. Each leaf node of the AF tree corresponds to one cluster $\mathbb{C}$ represented by an AF structure. An AF maintains the aggregated summarization of a set of mini buckets that form cluster $\mathbb{C}$. AF contains sufficient information that given an incoming mini bucket MB_i, we can determine whether MB_i satisfies all requirements and constraints of our *MOC* clustering problem including density, spatial location, and cardinality such that it can be merged into cluster $\mathbb{C}$. The AF meta data is defined as below.

Definition 5.1: **Aggregate Feature.** Given a cluster $\mathbb{C}$ with M mini buckets MB_i, where $i = 1, 2, ..., M$, its Aggregate Feature (AF) of the cluster $\mathbb{C}$ is defined as a quadruple: $AF = (numPoints, \tilde{minB}, \tilde{maxB}, Density)$, where $numPoints = \sum_{i=1}^{M} MB_i.numPoints$ corresponds to the number of points in $\mathbb{C}$, $\tilde{minB}$ and $\tilde{maxB}$ are the vectors corresponding to the minimum and maximum coordinates of MBs at each dimension, and $Density = \frac{|numPoints|}{\prod_{i=1}^{d}(maxB(i)-minB(i))}$ represents the density of $\mathbb{C}$.

A non-leaf node is represented by a R-tree like data structure, that is, a pair ($Rect$, *child-pointer*). Here $Rect$ is an *d-dimensional* rectangle which is a bounding box that covers all rectangles in the lower nodes' entries. *Child-pointer* is the address of a lower node in the AF-tree.

DSHC starts with one single mini bucket as the only cluster in the AF tree. It then incrementally inserts mini buckets into the AF tree. When a new mini-bucket comes, it utilizes the search operation defined below to identify a list of merging candidate clusters. The new mini bucket can either merge with existing clusters to form a new cluster or serve as an independent cluster depending on the given density similarity and adjacency requirements.

Search Operation. Starting form the root, the searching operation descends the AF-tree in a manner similar to R-tree. However, in addition to search the overlapping rectangles, DSHC also queries nodes that are adjacent to the new mini-bucket. The search operation will generate a list of merging candidate clusters or in short *LMC*. If the *LMC* is empty, we maintain the parent node *pn* of the leaf node that can accommodate this new mini bucket with least enlargement.

Merge Operation. If the *LMC* list is not empty, then the new mini bucket potentially can be merged into an existing cluster. Then DSHC further filters the *LMC* list by the merging criteria defined below.

Definition 5.2: **Merging criteria.** Given the maximum density difference threshold T_{diff} and the maximum number of points threshold $T_{max\#}$, two clusters $\mathbb{C}_i$ and $\mathbb{C}_j$ can be merged if and only if the following three criteria are satisfied: (1)$|\mathbb{C}_i.Density - \mathbb{C}_j.Density| < T_{diff}$; (2) $\mathbb{C}_i$ and $\mathbb{C}_j$ form a rectangular shape; (3) $\mathbb{C}_i.numPoints + \mathbb{C}_j.numPoints < T_{max\#}$.

Next, we define how to determine whether given two clusters $\mathbb{C}_1$ and $\mathbb{C}_2$ can be merged into a rectangular shape.

Definition 5.3: **Rectangular Shape.** In a d-dimensional domain space, given two clusters $\mathbb{C}_1$ and $\mathbb{C}_2$ with coordinates $(\tilde{minB}_1, \tilde{maxB}_1)$ and $(\tilde{minB}_2, \tilde{maxB}_2)$, $\mathbb{C}_1$ and $\mathbb{C}_2$ can form a rectangular shape iff they satisfy the following two criteria: (1) For $d-1$ dimensions, $minB_1(i) = minB_2(i)$ and $maxB_1(i) = maxB_2(i)$ and (2) For the remaining one dimension, either $minB_1(i) = maxB_2(i)$ or $maxB_1(i) = minB_2(i)$.

After the *LMC* is filtered by the merging criteria, a final list of candidate clusters that are able to merge with the new mini-bucket will be generated. Then the mini bucket will be merged into the cluster that has the most similar density with the new bucket.

After the merge operation, a new AF will be computed for the augmented cluster as below.

Definition 5.4: Given two Aggregate Features $AF_1 = (numPoints_1, \tilde{minB}_1, \tilde{maxB}_1, Density_1)$, $AF_2 = (numPoints_2, \tilde{minB}_2, \tilde{maxB}_2, Density_2)$, then $AF_{new} = AF_1 + AF_2 = (numPoints_1 + numPoints_2, min(\tilde{minB}_1, \tilde{minB}_2), max(\tilde{maxB}_1, \tilde{maxB}_2), \frac{numPoints_1+numPoints_2}{\prod_{i=1}^{d}(maxB_{new}(i)-minB_{new}(i))})$.

Similar to R-tree, the merge action of the AF-tree will trigger the recursive merge and information update along the path from the cluster (leaf) to the root if possible. More specifically, *DSHC* will try to merge the new augmented cluster $\mathbb{C}_a$ with other clusters in the AF-tree by applying the same process described above. The recursive merge stops if the new cluster cannot be merged with any other cluster in the AF-tree.

Insert Operation. If the incoming mini bucket cannot be merged with any existing cluster (leaf), a new leaf will be created. If the *LMC* list is not empty, again a cluster $\mathbb{C}_i$ in *LMC* that has the most similar density with the new mini bucket will be selected. The new leaf will then be attached to the parent of $\mathbb{C}_i$ as its child. Otherwise, the new leaf will be created under the pn node found in the search operation.

Split Operation. The insertion of a new leaf might trigger the split operation. The standard split operation defined by R-tree can be equally applied here. Due to space constraint, the details are omitted.

Finally, after inserting all mini buckets into the AF tree, every leaf node in the tree represents one cluster (partition).

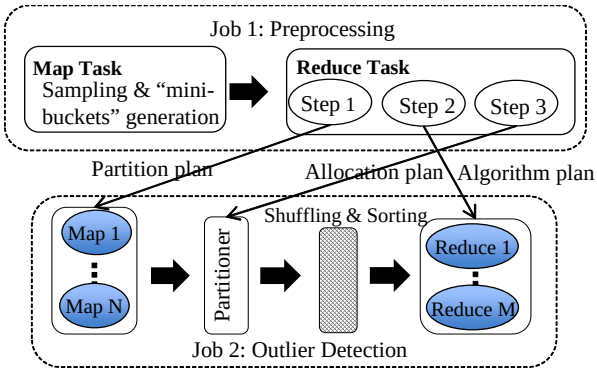


Fig. 6: Execution Workflow of the Multi-Tactic DOD Framework.

Therefore, the final partition plan can be directly obtained. Moreover, the AF tree can be leveraged as an index to accelerate the process of mapping data points into partitions.

The outcome of Step 1 is the *final partition plan* to be used in the *DOD* framework by the mappers.

Step 2 then *decides on the algorithm plan*, i.e., which outlier detection algorithm should be used for which partition. For each partition P and a specific algorithm A, the cost of A is estimated based its cost model (Sec. IV). The most efficient algorithm will be assigned to P.

The last step of this pre-processing phase is then to allocate the partitions to reducers and balance the workload of each reducer. The work load is measured as the sum of costs of the partitions allocated to a given reducer. This problem is equivalent to the problem of *multi-bin packing*, in which a set of N numbers needs to be divided into K subsets, such that the sums within each subset are as similar as possible. This problem is known to be NP-Complete. Several approximation algorithms have been proposed to solve it. In *DOD*, we adopt the polynomial-time algorithm proposed in [25].

Overall Workflow of DOD. In Figure 6, we summarize the execution workflow of our full-fledged DOD approach. The workflow consists of two MapReduce jobs: the pre-processing job on a small data sample (top) corresponding to *DMT* and the outlier-detection job on the actual data D (bottom) corresponding to the DOD framework. The pre-processing job produces three types of outputs passed to the DOD framework. More specifically, the output of Step 1 (the partition plan) is passed to its map phase. The output of Step 2 (the algorithm plan) is passed to the reduce phase. The output of Step 3 (the allocation plan) is passed to the partitioner functions to stipulate which partitions are assigned to which reducers.

VI. Experimental Evaluation

A. Experimental Setup & Methodologies

Experimental Infrastructure. All experiments are conducted on a shared-nothing cluster with one master node and 40 slave nodes. Each node consists of 16 core AMD 3.0GHz processors, 32GB RAM, 250GB disk, and nodes are interconnected with 1Gbps Ethernet. Each server runs Hadoop 2.4.1. Each node is configured to run up to 8 map and 8 reduce tasks concurrently. The replication factor is set to 3.

Datasets. We utilize two real datasets and one synthetic dataset to evaluate our proposed strategies and observations.

The first one is the TIGER dataset [26]. TIGER contains spatial extracts from the Census Bureau's MAF/TIGER database, containing features such as roads, railroads, rivers, as well as legal and statistical geographic areas. TIGER contains 70 million records with a total size of 60G.

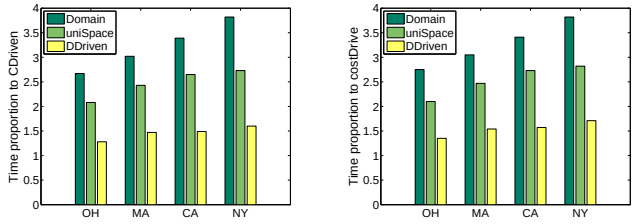
The second one is *OpenStreetMap* dataset [19]. *OpenStreetMap* (500 GBs) is one of the largest real datasets publicly available and has been used in other similar research work [19]. *OpenStreetMap* contains the geolocation information of buildings all over the world. Each row in this dataset represents a building. Four attributes are utilized in this experiment, namely *ID, timestamp, longitude*, and *latitude*.

To evaluate the robustness of our proposed methods for diverse data distributions, we pick four segments from the whole openStreetMap dataset corresponding to buildings in Massachusetts, Ohio, California, and New York respectively. The four segments are equally sized (≈30 million points). However, they vary significantly in their densities, i.e., New York and California are very dense, Ohio is relatively sparse, and Massachusetts is in the middle between them. In addition, we build hierarchical datasets with Massachusetts as the smallest unit, then New England, then the United States, up to the whole planet. The number of data points gradually grows from 30 million to 4 billion.

Lastly, to evaluate how *DOD* performs on terabyte level data we further generate a 2TB synthetic dataset based on the real *OpenStreetMap* dataset. More specifically, we developed a tool that creates a distortion of the original dataset D by replicating each point p in D three times to generate p', p'', p''', each with a random degree of alteration on each dimension.

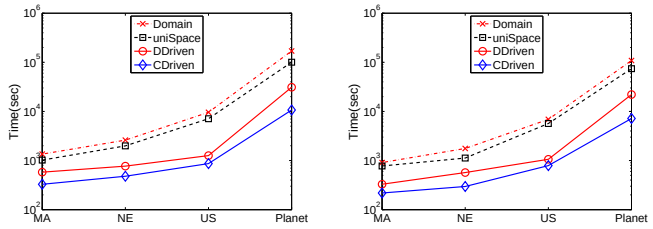
Metrics. We measure the **end-to-end execution time** common for the evaluation of distributed algorithms. We also measure the **breakdown of the execution time for the key stages** of the MapReduce workflow including preprocessing, partitioning (map), and processing (reduce) time.

Experimental Methodology. We evaluate two key components of our MapReduce outlier detection algorithms, namely the *partitioning* method at the mapper side and the *outlier detection* method at the reducer side. In particular for the partitioning method, we evaluate four alternative strategies, namely (1) the default domain-based partitioning without supporting area *Domain* that needs an additional MapReduce job to confirm the outlier status of a point p if p is at the edge of a partition and is classified as an outlier in the first MapReduce job, (2) the uniform domain space partitioning *uniSpace* (Sec. III-A), (3) the data-driven partitioning *DDriven* that divides the dataset into partitions with similar number of data points, and (4) the cost-driven partitioning *CDriven* that divide the dataset into partitions with similar workload. The workload of each partition is estimated utilizing our cost model (Sec. IV) with respect to the selected detection algorithm. Similar to our full-fledged multi-tactic approach DMT (Sec. V), *uniSpace*, *DDriven*, *CDriven* also feature a pre-processing job to generate the statistics and produce the partition plan. All three algorithms are based on our one pass DOD framework with supporting area. We utilize these algorithms to demonstrate the effectiveness of our optimization strategies. Due to space constraint the details of these algorithms are omitted in this manuscript.



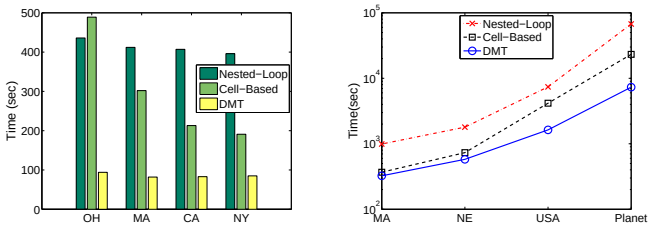
(a) Nested-Loop (b) Cell-Based

Fig. 7: Partitioning: Effectiveness Evaluation for Various Distributions.



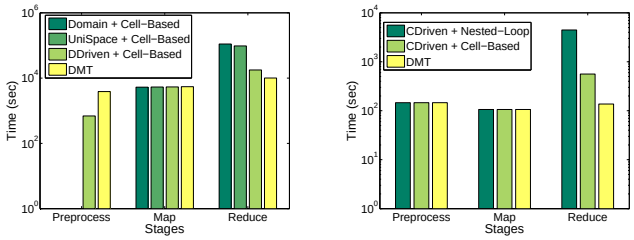
(a) Nested-Loop (log scale) (b) Cell-Based (log scale)

Fig. 8: Partitioning: Scalability Evaluation For Varying Data Sizes.



(a) Varying Distributions (b) Varying Data Sizes

Fig. 9: Detection Methods: Effectiveness Evaluation.



(a) 2TB synthetic dataset (b) TIGER dataset

Fig. 10: Overall Approach: Performance Breakdown.

Their performance is evaluated for varying sizes of datasets and diverse distributions. We use the domain-based partitioning *Domain* as the baseline approach to compare against the algorithms designed by us.

For outlier detection at the reducer side, we evaluate three alternatives, namely *Nested-Loop*, *Cell-Based*, and our *multi-tactic* approach *DMT* proposed in Sec. V. *Nested-Loop* is the basic distance-based outlier algorithm widely adopted in the literature, while *Cell-Based* is the most commonly used index-driven detection algorithm. To minimize the influence of the partitioning method on the result the same partitioning method is deployed for *Nested-Loop* and *Cell-Based*, namely the most advanced cost-driven partitioning *CDriven*.

We also evaluate the execution time breakdown of *DMT* and compare it against other alternatives.

B. Evaluation of Partitioning Methods

First we conduct experiments to evaluate the effectiveness of the partitioning methods.

Effectiveness Evaluation For Various Distributions With Real Datasets. In this set of experiments we evaluate the performance of our partition methods under diverse data distributions using Ohio, Massachusetts, California, and New York areas. We show the performance of the *Domain*, *uniSpace*, and *DDriven* strategies relative to our proposed *CDriven* partitioning strategy. To exclude the influence of detection algorithm, we fixed the detecting algorithm at the reducer side to be the *Nested-Loop* solution in Figure 7(a) and the *Cell-Based* algorithm in Figure 7(b).

Clearly, as depicted in both Figures 7(a) and 7(b), the *cost-driven* partitioning method significantly outperforms all other alternatives up to 5 fold, no matter how the data distribution changes. Our *uniSpace* partitioning strategy outperforms the default *Domain* partitioning method. This is due to the fact that *uniSpace* ensures that the detection task can be done in a single pass, therefore incurring much smaller communication costs. However *uniSpace* is not effective at load balancing, because the real datasets tend to be skewed. Therefore on average, the performance is 40% worse than that of *DDriven*. On the other hand, although *DDriven* ensures that each partition has a similar number of data points, the workload on each reducer is not effectively balanced, confirming our observation that an equal number of data points does not guarantee an equal workload (Sec. IV-A). Our final *CDriven* partitioning strategy instead achieves true load balancing. Therefore it outperforms *DDriven* by at least 50% and all other methods more significantly (up to five fold).

Scalability Evaluation For Varying Data Sizes. Next we evaluate the scalability of our partitioning method on increasing dataset sizes, from the Massachusetts dataset, New England dataset, United States dataset, to the entire *OpenMap-Street* dataset. The results in Fig. 8 show that the *cost-driven* partitioning method *CDriven* consistently wins in all cases. *Better yet, the larger the dataset, the more it wins*. In particular when the dataset is the largest (the planet dataset), *CDriven* is 6 times faster than the second best partitioning method *DDriven* and 17 times faster than the default *Domain* partitioning. This thus demonstrates that our partitioning method is scalable to real-world large datasets.

C. Evaluation of Detection Methods

In this section, we focus on the evaluation of the outlier detection algorithms applied at the reducer side. Three algorithms are considered in this set of experiments, namely the *Nested-Loop* and *Cell-Based* algorithms and the *DMT* approach introduced in Sec. V.

Effectiveness Evaluation For Varying Data Distributions. In this experiment, we utilize the Massachusetts, Ohio, California, and New York areas to evaluate the efficiency of the three detection algorithms with diverse data distributions. As shown in Figure 9(a), *Cell-Based* is at least two times faster than *Nested-Loop* when processing the California and New York datasets. The reason is that overall, California and New York are densely populated. As proven by Lemma 4.2, *Cell-Based* theoretically performs better than *Nested-Loop* on dense

datasets. *DMT* further outperforms *Cell-Based* by a factor of 2, because it takes the advantages of both *Cell-Based* and *Nested-Loop*. That is, it adapts to *Cell-Based* when handling dense or sparse data partitions, while automatically switching to *Nested-Loop* when the density of the data partition is in the middle.

As the dataset gets sparser and sparser, the dataset contains more outliers. In this case, the pruning ability of the *Cell-Based* method becomes less effective. Therefore its running time increases dramatically. On the other hand, the running time of *Nested-Loop* increases at a more steady pace. As shown in Figure 9(a) *Nested-Loop* outperforms *Cell-Based* when processing the Ohio data − the most sparse out of the four. This again confirms our algorithm performance observation in Sec. IV-B with respect to these two typical classes of detection algorithms. The running time of *DMT* remains stable as the data distribution changes. *DMT* has overall much better performance in all cases.

Scalability Evaluation For Varying Data Sizes. In this experiment we evaluate the scalability of different detection algorithms utilizing real datasets. Similar to the scalability test in Sec. VI-B we increase the size of the real dataset by utilizing first the Massachusetts only dataset, then New England, the United States, up to the entire *OpenStreetMap* dataset. As shown in Figure 9(b), *DMT* consistently outperforms *Nested-Loop* and *Cell-Based*. The larger the dataset, the more *DMT* wins. This is so because the larger datasets tend to be more skewed. In other words, large data usually contains not only many sparse partitions, but also many dense partitions. However as demonstrated in Sec. IV-B neither *Nested-Loop* nor *Cell-Based* performs well under all circumstances. *DMT* is able to dynamically adapt to *Nested-Loop* or *Cell-Based* for each partition based on its distribution. Therefore *DMT* scales well to large datasets.

D. Evaluation of Overall DOD Approach

Next we focus on the evaluation of the overall approach. We measure the breakdown of the execution time for the key stages of the MapReduce workflow.

2TB Synthetic Dataset. We measure the preprocessing time, the partitioning time, and the detection time separately. At the mapper side, four partitioning algorithms are considered, namely the baseline *Domain* partitioning and our *uniSpace*, *DDriven*, and our full-fledged DMT approach. For the other three partitioning methods we apply the *Cell-Based* detection algorithm. This is because *Cell-Based* is confirmed by our additional experiments to be the algorithm that on average fits this dataset better than *Nested-Loop*. In this experiment we use the 2TB dataset derived from the *OpenStreetMap* dataset (see Sec. VI-A). This also confirms the scalability of DOD on terabyte-scale data.

As shown in Fig. 10(a), the preprocessing time of *DMT* is longer than *DDriven*. This is expected because *DMT* utilizes a hierarchical like clustering approach to group data with similar densities together. This is expensive. *Domain* and *uniSpace* do not feature this preprocessing stage. Therefore no preprocessing costs are experienced. In the partitioning map stage, all four approaches take almost the same amount of time. For all, each datum can be mapped to its corresponding partition in near constant time. At the reduce stage, *DMT* is up to 10 times faster than other alternatives for the following two reasons. First, *DMT* achieves true workload balancing across the reducers by utilizing our cost models designated for each known detection algorithm. Second, given a particular partition, *DMT* automatically adapts to the algorithm best fitting the data characteristics of that partition. Although this "dense" dataset on average fits the *Cell-Based* algorithm better than the *Nested-Loop* algorithm, there are still many relatively sparse partitions for which *Nested-Loop* is more appropriate.

TIGER Dataset. We also evaluate the breakdown of the execution time on the TIGER real dataset. In this set of experiments, we compare DMT against *CDriven + Nested-Loop* and *CDriven + Cell-Based*. That is, the two alternative solutions apply the most sophisticated partitioning strategy at the mapper side while apply different detection algorithms at the reducer side. As shown in Fig. 10(b) DMT significantly outperforms the alternatives (up to 20 times faster). This again demonstrates the effectiveness of our multi-tactic optimization technique.

VII. Related Work

Centralized Outlier Detection. Distance-based outliers was first proposed in [3] along with two popular algorithms described in Sec. IV. [4] improved upon these prior results [3] by introducing the pivot-based index technique. However, this depends on building a global index. Thus it does not fit well the shared-nothing distributed architectures such as MapReduce because no single compute node can accommodate such a big global index for large data.

Distributed Outlier Detection. Hung and Cheung [27] presented a parallel version of the *Nested-loop* algorithm of the distance-based semantics from [3]. This technique requires strict synchronization between the worker nodes. Namely each node has to send messages and receive messages at the same time. It assumes all nodes can communicate with each other at will by message passing. Thus, it is not suitable for MapReduce-like infrastructures where mappers (and similarly reducers) work independently from each other.

Angiulli et al. [11] presented a distributed algorithm to support another major variant semantics of distance-based outliers, namely the kNN based definition [10]. First, it represents the original dataset using a compact solving set. Then given a point p, its status as an outlier can be approximated by comparing p to only the elements in the solving set. Therefore [11] provides an approximate result, whereas we instead focus on providing an exact solution for our targeted distance-based outlier semantics [3]. Furthermore, [11] requires the solving set to be broadcasted to each node. This is not scalable to big datasets which in turn indicate a large solving set.

Bhaduri et al. [13], also working with the kNN based outlier definition, developed a distributed algorithm on a ring overlay network architecture. Their algorithm passes data blocks around the ring allowing the computation of neighbors to proceed in parallel. Along the way, each point's neighbor information is updated and distributed across *all* nodes. For this, a central node is utilized to maintain and update the top-n points with largest kNN distances. Their strategies, such as checking the test blocks in a round robin fashion, which

requires m iterations, is clearly not practical for Map-Reduce. Furthermore, MapReduce also does not feature a central node.

Data Analytics in MapReduce. The efficiency of our *DOD* approach relies on and advances important strategies in distributed systems such as *load balancing* and *efficient partitioning*. These strategies have previously been discussed with respect to other analytics tasks. We examine these works now to look for relationships.

In [14] the authors studied the problem of performing **similarity joins** using MapReduce. The proposed *MR-MAPSS* algorithm partitions the input data into work sets with minimal redundancy. It achieves *load balancing* by repartitioning the densely clustered large work sets. However their *load balancing* method relies on the traditional load balancing assumption, namely an equal number of data points indicating equal work load. This assumption is proven to be faulty in our context of distance-based outlier detection (see Sec. IV-A).

Research work also has been done for **KNN-Joins** on MapReduce. The approximation algorithm in [19] first maps the multi-dimensional dataset into one dimension using space-filling curves (z-values), and then transforms the KNN join into a sequence of one-dimensional range searches. This way, the partitioning of a multi-dimensional dataset is reduced to an equal size one-dimensional partitioning problem. This technique is not applicable in our context. First, we focus on producing exact and complete results of distance-based outliers instead of an approximation. Second, this approximation method is shown to be not suitable for skewed data, while most real world datasets are known to be skewed [5].

[17] studies density-based **clustering** on MapReduce. Although density-based clustering is the clustering definition most closely related to distance-based outliers, this approach cannot be directly applied to solve our outlier detection problem. First, introducing outliers as by-products of clustering has already been shown not to be effective in capturing abnormal phenomena [5]. Further, density-based clustering has been demonstrated to be more expensive than distance-based outlier detection [1], because the cluster structure is more complex to detect and update than individual outlier points due to the inter-dependence among data points. Therefore applying the density-based clustering algorithm to attempt to detect outliers is neither effective nor efficient.

VIII. Conclusion

Using outlier detection algorithms to extract abnormal phenomena from huge volumes of data is an extremely important yet expensive task. Existing techniques lack proper scaling to terabyte of data. In this paper, we proposed the first distributed distance-based outlier detection approach called *DOD* using the MapReduce paradigm. Innovations of *DOD* includes a single-pass execution framework to minimize communication overhead and the multi-tactic optimization strategy leveraging the localized characteristics of each data partition to minimize the processing time. Our experiments show the efficiency and scalability of *DOD* on terabytes of data.

References

[1] C. C. Aggarwal, *Outlier Analysis*. Springer, 2013.

[2] D. M. Hawkins, *Identification of Outliers*. Springer, 1980.

[3] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," in *VLDB*, 1998, pp. 392–403.

[4] F. Angiulli and F. Fassetti, "Dolphin: An efficient algorithm for mining distance-based outliers in very large datasets," *TKDD*, vol. 3, no. 1, 2009.

[5] G. H. Orair, C. Teixeira, Y. Wang, W. M. Jr., and S. Parthasarathy, "Distance-based outlier detection: Consolidation and renewed bearing," *PVLDB*, vol. 3, no. 2, pp. 1469–1480, 2010.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, 2010, pp. 1–10.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.

[9] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson, "Efficient processing of data warehousing queries in a split execution environment," in *SIGMOD*, 2011, pp. 1165–1176.

[10] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," in *SIGMOD Conference*, 2000, pp. 427–438.

[11] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "Distributed strategies for mining outliers in large data sets," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 7, pp. 1520–1532, 2013.

[12] X. Wang, D. Shen, M. Bai, T. Nie, Y. Kou, and G. Yu, "An efficient algorithm for distributed outlier detection in large multi-dimensional datasets," *J. Comput. Sci. Technol.*, vol. 30, no. 6, pp. 1233–1248, 2015.

[13] K. Bhaduri, B. L. Matthews, and C. Giannella, "Algorithms for speeding up distance-based outlier detection," in *KDD*, 2011, pp. 859–867.

[14] Y. Wang, A. Metwally, and S. Parthasarathy, "Scalable all-pairs similarity search in metric spaces," in *KDD*, 2013, pp. 829–837.

[15] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz, "Hadoop-gis: A high performance spatial data warehousing system over mapreduce," *PVLDB*, vol. 6, no. 11, pp. 1009–1020, 2013.

[16] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan, "MR-DBSCAN: an efficient parallel density-based clustering algorithm using mapreduce," in *ICPADS*, 2011, pp. 473–480.

[17] R. L. F. Cordeiro, C. T. Jr., A. J. M. Traina, J. López, U. Kang, and C. Faloutsos, "Clustering very large multi-dimensional datasets with mapreduce," in *KDD*, 2011, pp. 690–698.

[18] Y. Kim and K. Shim, "Parallel top-k similarity join algorithms using mapreduce," in *ICDE*, 2012, pp. 510–521.

[19] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *EDBT*, 2012, pp. 38–49.

[20] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD*, 1993, pp. 207–216.

[21] Y. Tao, D. Papadias, and X. Lian, "Reverse knn search in arbitrary dimensionality," in *VLDB*, 2004, pp. 744–755.

[22] S. Papadimitriou, H. Kitagawa, P. B. Gibbons, and C. Faloutsos, "Loci: Fast outlier detection using the local correlation integral," in *ICDE*, 2003, pp. 315–326.

[23] F. Olken, D. Rotem, and P. Xu, "Random sampling from hash files," in *SIGMOD*, 1990, pp. 375–386.

[24] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: An efficient data clustering method for very large databases," in *SIGMOD*, H. V. Jagadish and I. S. Mumick, Eds., 1996, pp. 103–114.

[25] P. Lemaire, G. Finke, and N. Brauner, "Models and complexity of multibin packing problems," *J. Math. Model. Algorithms*, vol. 5, no. 3, pp. 353–370, 2006.

[26] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in *ICDE*, 2015, pp. 1352–1363.

[27] E. Hung and D. W. Cheung, "Parallel mining of outliers in large database," *Distributed and Parallel Databases*, vol. 12, no. 1, pp. 5–26, 2002.