

Mining Outliers with Adaptive Cutoff Update and Space Utilization (RACAS)

Chi-Cheong Szeto¹ and Edward Hung²

Abstract. Recently the efficiency of an outlier detection algorithm ORCA was improved by RCS (Randomization with faster Cutoff update and Space utilization after pruning), which changes the frequencies of updating the cutoff value and reclaiming the memory space at some pre-specified time. How and when to change the frequencies were only determined empirically. However, the optimal setting may vary for different data sets and computers with different CPU and disk I/O performance. In this paper, we theoretically formulate two methods to further reduce the execution time of RCS by dynamically adapting the frequencies at each step to different data sets and computers with different CPU and disk I/O performance. We conducted experiments on a KDD CUP real data set from a network intrusion detection problem under different conditions. The results show that our substantial time-saving from optimized ORCA is up to five times that of RCS and increases with the relative disk I/O cost, the percentage of outliers to find and the data set size.

1 INTRODUCTION

Outliers are data objects that are dissimilar to the majority of objects in a data set. Outlier detection is important in various applications. For example, in network intrusion detection problems, data objects are features of TCP data and outliers indicate where intrusions occur. Sometimes, outliers represent noises, which we would like to remove to increase the accuracy of further analyses.

This paper focuses on a distance-based approach proposed by Knorr and Ng [5] and uses an object's distances to its nearest neighbors as a measure of dissimilarity. The advantage of this approach is that we do not take any assumption on the distribution of the data set. However, the disadvantage is its long processing time. Finding nearest neighbors of every object of the whole data set requires $O(N^2)$ distance computations, where N is the data set size. In many application domains such as banking systems, government census systems and geoinformatics systems, N is usually too large for an algorithm with $O(N^2)$ complexity to be practical. Hence, it is important to develop fast outlier detection algorithms.

This paper focuses on the outlier definition proposed by Ramaswamy et al. [7] and its variation proposed by Angiulli et al. [2]. The former defines the score of an object as the distance to its k th nearest neighbor while the later defines the score of an object as the sum of distances to its k nearest neighbors. Given n (the number of outliers to find) and k (the number of nearest neighbors to find to calculate the scores), outliers are the top n objects with the highest scores.

A fast algorithm called ORCA [3] was developed. It employs two techniques to reduce the processing time. The first technique is a simple pruning rule. ORCA keeps the n th highest score found so far. We call it the cutoff value, which is monotonically increasing with new better outliers found. If an object's temporary score calculated from its nearest neighbors found so far is already smaller than the cutoff value, the object can be classified as a non-outlier because the temporary score is monotonically decreasing with new nearest neighbors found. Its distance computations with the remaining objects in the data set are no longer necessary and can be skipped. The second technique randomizes the order of objects in the data set. The order that objects are scanned affects the convergence rate of objects' temporary scores to their final values and the order of finding the objects' scores affects the convergence rate of the cutoff value to its final value. Both of them affect the effectiveness of the pruning rule. Randomizing the order ensures that the worst case seldom happens. ORCA was shown to give near linear time performance on real, large and high-dimensional data sets.

However, there are two areas of inefficiency of ORCA. First, the whole process uses one single frequency of updating the cutoff value. At the beginning of the process, a low cutoff value initially set to zero is used to process objects. During this time, the cutoff value increases rapidly but the frequency of updating the cutoff value is not high enough. Objects are processed with a far-too-low cutoff value leading to a larger number of unnecessary distance computations. Second, the memory space is seriously under-utilized leading to an increase in the disk I/O cost.

Randomization with faster Cutoff update and Space utilization after pruning (RCS) [8, 9] employs two techniques to improve these two areas of inefficiency. The first one named Randomization with faster Cutoff update (RC) updates the cutoff value more frequently to reduce the number of distance computations in the earlier stage when the cutoff value increases rapidly. The second one named Randomization with Space utilization after pruning (RS) reclaims the memory space more frequently to reduce the disk I/O time in a later stage when the cutoff value increases slowly. RCS was tested on four real, large and high-dimensional data sets. Its speed is as fast as 1.4 to 2.3 times that of ORCA and the improvement is relatively insensitive to the increase in the data size.

However, there are two problems with RCS. First, the following important parameters could only be experimentally determined: (i) the frequency of updating the cutoff value, (ii) the frequency of reclaiming the memory space, and (iii) the time to change the above frequencies. These values vary with the data set, the CPU and disk I/O performance of the computer. Second, both RC and RS roughly divide the whole process into two stages and assign a single frequency value to each stage. However, to minimize the processing

¹ Department of Computing, The Hong Kong Polytechnic University, Hong Kong, email: cscszeto@comp.polyu.edu.hk

² Department of Computing, The Hong Kong Polytechnic University, Hong Kong, email: csehung@comp.polyu.edu.hk

```

1  Shuffle the objects in the data set D; initialize the cutoff value to be zero.
2  While there are objects in D unprocessed do
3      Load a batch of unprocessed objects of a fixed size to the memory
        buffer B.
4      For each object  $d$  in D
5          For each object  $b$  in B
6              Compute the distance between  $b$  and  $d$ . If  $d$  is closer
                to  $b$  than the objects kept previously as the  $k$  nearest
                neighbors of  $b$ , update the nearest neighbors of  $b$  with
                 $d$  and compute a new score of  $b$ . If the score is lower
                than the cutoff value, prune  $b$ .
7          End for
8      End for
9      At the end of a data scan, remove the objects remaining in B.
        Rank them together with the top  $n$  outliers previously kept. Keep
        the new top  $n$  outliers and update the cutoff value.
10 End while

```

Figure 1. Algorithm of ORCA

time, this value may not be optimal throughout the whole stage.

In this paper, we propose methods to reduce processing time by dynamically determining the frequency values of updating the cutoff value and reclaiming the memory space at each step. These methods take into account the characteristics of the data set (the number of computations required to calculate the distance between two objects and the storage size of an object), the CPU and disk I/O performance of the computer. Thus, they can automatically adapt the frequency values to different data sets and computers with different CPU and disk I/O performance.

The remainder of this paper is organized as follows. Sec. 2 and 3 review how ORCA and RCS work respectively. Sec. 4 formulates our proposed methods. Sec. 5 tests our proposed methods on a network intrusion detection problem and analyzes the time saving from optimized ORCA (with optimized batch size). Sec. 6 reviews the related work. Sec. 7 offers our conclusion.

2 ORCA

ORCA aims to find top n outliers with highest scores given n (the number of outliers to find) and k (the number of nearest neighbors to calculate the scores).

ORCA works as follows. The first step shown in Fig. 1 randomizes the order of objects in the data set. The rest of the steps is a 3-level nested loop. In each iteration, the outermost loop loads part of the data set into the memory to process. It terminates when all objects in the data set have been loaded and processed. At the end of each iteration, objects with scores higher than the current cutoff value may be found. If so, the cutoff value will be updated. The middle loop makes a data scan. It loads one object from the data set in each iteration. It terminates when all the objects in the data set have been loaded or all objects in the memory have been processed. The innermost loop computes the distances between the object loaded in the outer loop and all the objects in the memory. It also updates the temporary scores of the objects if nearer neighbors are found. If the temporary score of an object is smaller than the cutoff value, this object is confirmed as a non-outlier and removed from the memory.

3 RCS

RCS employs two techniques to speed up ORCA.

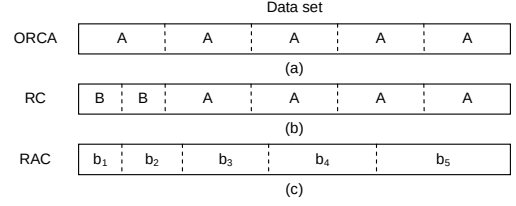


Figure 2. A data set is divided into a number of batches to process.

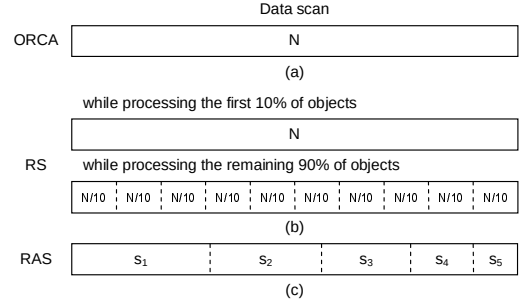


Figure 3. A data scan is divided into a number of partition scans.

3.1 RC

One named RC changes the frequency of updating the cutoff value by modifying the batch size. The cutoff value is updated after a batch of objects has finished processing. A smaller batch size implies a higher frequency of updating the cutoff value, which implies that objects are processed with a higher cutoff value and fewer distance computations. Hence, CPU time decreases. However, processing a batch of objects requires one data scan at the worst case. A higher frequency of updating the cutoff value also implies a larger number of data scans. Hence, disk I/O time increases. To reduce the overall running time, RC uses a higher frequency of updating the cutoff value when the cutoff value increases rapidly (that is, at the beginning of the process) so the CPU time saving is larger than the increase in the disk I/O time. RC changes to a lower frequency of updating the cutoff value when the cutoff value no longer increases rapidly, which is detected by the condition that the ratio of new to old cutoff values is smaller than a threshold. Figures 2(a) and (b) depict the difference between ORCA and RC. ORCA divides the data set into a number of batches of an equal size A while RC first uses a smaller batch size B and later uses a larger one A . In Sec. 4.1, we will propose a method to dynamically determine the size for each batch (Fig. 2(c)).

3.2 RS

The other named RS changes the frequency of reclaiming the memory space by dividing a data scan into a number of equal partitions, each of which is called a partition scan. After each of the partition scans, RS reclaims the memory space by pruning confirmed non-outliers. A smaller partition size implies a higher frequency of reclaiming the memory space. Objects are loaded earlier and so more objects are processed in one data scan. The total number of data scans decreases. Hence, disk I/O time decreases. However, objects loaded earlier are processed with a lower cutoff value. Hence, CPU

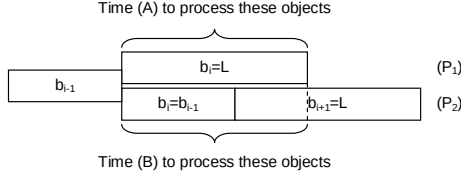


Figure 4. *RAC* chooses P_1 for b_i if time (A) < time (B). Otherwise, it chooses P_2 .

time increases. To reduce the overall running time, RS uses a higher frequency of reclaiming the memory space (10 times per data scan) when the cutoff value increases slowly (after 10% of a data set is processed) so the disk I/O time saving is larger than the increase in the CPU time. Figures 3(a) and (b) depict the difference between ORCA and RS. The partition size in ORCA equals the data scan size N while RS first uses a partition size that equals N to process the first 10% of objects and later uses a partition size that equals one-tenth of N to process the rest of objects. This paper proposes a method described in Sec. 4.2 to dynamically determine the size for each partition scan (Fig. 3(c)).

4 OUR PROPOSED METHODS

4.1 Randomization with Adaptive Cutoff Update (*RAC*)

Our first method named Randomization with Adaptive Cutoff Update (*RAC*) determines batch sizes automatically by predicting the processing time of the next L objects where L is some number larger than the last batch size b_{i-1} . Let the current batch be the i th batch. *RAC* considers two possible timing of using a larger batch size L , where $L > b_{i-1}$. One named P_1 uses the larger batch size for the current batch, that is, $b_i = L$. The other named P_2 keeps the current batch size as the last one and uses the larger batch size L in the next batch, that is, $b_i = b_{i-1}$ and $b_{i+1} = L$. The criterion to choose between P_1 and P_2 is based on their estimated time to process the next L objects. Figure 4 depicts the comparison. If the processing time using P_1 is shorter, *RAC* sets $b_i = L$. Otherwise, it sets $b_i = b_{i-1}$.

The following presents the derivation of the mathematical formulation of the criterion. Let c_i be the cutoff value after processing i batches of objects. By using c_i , let DC_i be the average number of distance computations required to process one object and $OR_i(b)$ be the number of objects read from the disk in a data scan to process a batch of b objects. Also, let t_{DC} be the time to calculate the distance between two objects and t_{OR} be the time to read an object from the disk. Using P_1 and P_2 , the time to process the next L objects is expressed as (1) and (2) respectively.

$$\underbrace{DC_i L t_{DC}}_{\text{CPU time}} + \underbrace{OR_i(L) t_{OR}}_{\text{disk I/O time}}. \quad (1)$$

$$\begin{aligned} & \underbrace{DC_i b_{i-1} t_{DC}}_{\text{first } b_{i-1} \text{ objects}} + \underbrace{DC_{i+1} (L - b_{i-1}) t_{DC}}_{\text{next } (L - b_{i-1}) \text{ objects}} + \\ & \underbrace{OR_i(b_{i-1}) t_{OR}}_{\text{first } b_{i-1} \text{ objects}} + \underbrace{OR_{i+1}(L) \left(\frac{L - b_{i-1}}{L} \right) t_{OR}}_{\text{next } (L - b_{i-1}) \text{ objects}}. \end{aligned} \quad (2)$$

At the time of determining b_i , we do not have the measured values of DC_i and DC_{i+1} , which are required by (1) and (2). We predict them by assuming that DC changes at a constant rate with respect to the number of objects processed. Moreover, the minimum value of DC is k , where k is the number of nearest neighbours to calculate the score of an object. Hence, DC_i and DC_{i+1} are predicted by the following extrapolations.

$$DC_i = \max \left(DC_{i-1} + \left(\frac{b_{i-1}}{b_{i-2}} \right) (DC_{i-1} - DC_{i-2}), k \right). \quad (3)$$

$$DC_{i+1} = \max \left(DC_{i-1} + 2 \left(\frac{b_{i-1}}{b_{i-2}} \right) (DC_{i-1} - DC_{i-2}), k \right). \quad (4)$$

We also do not know the measured values of $OR_i(L)$, $OR_i(b_{i-1})$ and $OR_{i+1}(L)$, which are required by (1) and (2). We predict them by assuming that a complete scan of the data set is required for each batch disregarding the batch size and the cutoff value being used. Hence, they are predicted as follows.

$$OR_i(L) = OR_i(b_{i-1}) = OR_{i+1}(L) = N. \quad (5)$$

where N is the data set size.

By subtracting (1) from (2) and substituting the predicted values of (3), (4) and (5), we have the amount of time saved using P_1 as follows.

$$f(L) = \begin{cases} \beta \left(1 - \frac{b_{i-1}}{L} \right) & \text{if } \alpha_i = 0 \\ -\frac{\alpha_i}{L} (L - b_{i-1}) \left(L - \frac{\beta}{\alpha_i} \right) & \text{if } \alpha_i \neq 0 \end{cases}. \quad (6)$$

where $\alpha_i = \max(DC_i - DC_{i+1}, 0)$, $t_{DC} \geq 0$ and $\beta = N t_{OR} > 0$. The expected value of $(DC_i - DC_{i+1})$ is always non-negative because the corresponding cutoff values $c_i \leq c_{i+1}$. However, the measured value of it may be negative due to variations of samples because the sample size of the measurement may not be large enough. For simplicity, we restrict α_i to be non-negative. Then, the valid range of α_i is $[0, (N - k)t_{DC}]$ because DC_i is upper bounded by N and DC_{i+1} is lower bounded by k . We arbitrarily set the initial value α_0 to be its maximum value $(N - k)t_{DC}$.

When $\alpha_i = 0$, (6) becomes positive, so *RAC* chooses P_1 with the value of L that maximizes the time saving per object, $\frac{f(L)}{L}$. The optimized value of L is $2b_{i-1}$.

When $\alpha_i > 0$, if (6) gives a negative number, *RAC* chooses P_2 (i.e., $b_i = b_{i-1}$); otherwise, it chooses P_1 with the value of L that maximizes the time saving per object, $\frac{f(L)}{L}$. The optimized value of

L is $\frac{2\beta}{\alpha_i} \frac{b_{i-1}}{b_{i-1} + b_{i-1}}$. To sum up, *RAC* adapts the batch size as follows.

$$b_i = \begin{cases} 2b_{i-1} & \text{if } b_{i-1} < \frac{M}{2}, \alpha_i = 0 \\ M & \text{if } \frac{M}{2} \leq b_{i-1}, \alpha_i = 0 \\ b_{i-1} & \text{if } \frac{\beta}{\alpha_i} \leq b_{i-1}, \alpha_i \neq 0 \\ \frac{2\frac{\beta}{\alpha_i}b_{i-1}}{\frac{\beta}{\alpha_i} + b_{i-1}} & \text{if } b_{i-1} < \frac{\beta}{\alpha_i} < \frac{Mb_{i-1}}{2b_{i-1}-M}, \alpha_i \neq 0 \\ M & \text{if } \frac{Mb_{i-1}}{2b_{i-1}-M} \leq \frac{\beta}{\alpha_i}, \alpha_i \neq 0 \end{cases} \quad (7)$$

where M is the maximum buffer size. This method requires one initial value b_0 and it initializes b_0 with the value of $\max(4n, 100)$ where n is the number of outliers to find.

4.2 Randomization with Adaptive Space Utilization (RAS)

Our second method named Randomization with Adaptive Space Utilization (RAS) determines the current partition size automatically based on the processing time of its previous partition. Let the current partition size be s_i . RAS considers two possibilities. One named P_1 keeps the partition size as the previous one, that is, $s_i = s_{i-1}$. The other named P_2 uses a partition size that is half of the previous one, that is, $s_i = \frac{s_{i-1}}{2}$. RAS checks whether the processing time could have been reduced if the size of the last partition scan p had been halved and p had become two shorter equal scans p_1 and p_2 . If the processing time could have been reduced, RAS chooses P_2 . Otherwise, it chooses P_1 .

The following presents the derivation of the mathematical formulation of the criterion. RAS compares the processing time of two cases to choose between P_1 and P_2 . Case I did actually happen in the last partition scan p of the size s_{i-1} . Case II did not happen and assumes the size of p was halved and p became two shorter scans p_1 and p_2 of the size $\frac{s_{i-1}}{2}$. For Case I, new objects X_p are loaded into the empty space of the buffer (available due to pruning) after p . For Case II, new objects $X_{p1} \subseteq X_p$ are loaded after p_1 and another new objects $X_{p2} \supseteq (X_p - X_{p1})$ are loaded after p_2 .

The first difference between Cases I and II is: in Case II, objects X_{p1} are loaded earlier (after p_1) and processed using a lower cutoff value c_a ; in Case I, objects X_{p1} are loaded later (after p) and processed using a higher cutoff value $c_b \geq c_a$. Hence, Case II by halving the partition size would require longer CPU time to process X_{p1} than Case I. The difference is

$$|X_{p1}|(DC_a - DC_b)t_{DC} \quad (8)$$

where $|X_{p1}|$ is the number of objects of X_{p1} . DC_a and DC_b are the average numbers of distance computations required per object using the cutoff values c_a and c_b respectively. No new objects are loaded into the buffer a complete data scan before the end of p_1 ³ so no objects in the buffer can be confirmed as outliers at the end of p_1 . There is no cutoff update at the end of p_1 and the same cutoff value continues to be used during p_2 . Thus, we measure the value of DC_a in the middle of p . However, we do not have the measured value of DC_b . We assume there is a cutoff update right after p and estimate it by assuming that DC changes at a constant rate with respect to the number of objects processed. Moreover, the minimum value of DC_b is k . Thus, the estimated value of DC_b is linearly extrapolated as: $DC_b = \max\left(DC_j + \left(\frac{DC_j - DC_{j-1}}{n_j - 1}\right)n_j, k\right)$ where DC_j is the

³ Previous partition sizes are either p or multiples of p . Since loading of new objects could only be done after a partition scan, there could not be a complete data scan for objects (still in the buffer) at the end of p_1 (which is half of p).

average distance computations required per object using the j th updated cutoff value c_j . n_j is the number of objects processed using c_j and j is the most recent index.

The second difference between Cases I and II is that the processing time of Case II includes the disk I/O cost of objects X_{p1} , which are loaded after p_1 and processed during p_2 (a partition scan of $\frac{s_{i-1}}{2}$ objects). However, the processing time of Case I does not include this cost because objects X_{p1} are loaded after p . The disk I/O cost of objects X_{p1} during p_2 equals the proportion of X_{p1} to all objects in the buffer (i.e., $\frac{|X_{p1}|}{M}$) times the disk I/O cost of p_2 (i.e., $\left(\frac{s_{i-1}}{2}\right)t_{OR}$), where M is the maximum buffer size. Hence, the disk I/O time saving by halving the partition size is

$$\left(\frac{|X_{p1}|}{M}\right)\left(\frac{s_{i-1}}{2}\right)t_{OR} \quad (9)$$

Therefore, the overall time saving of halving the partition size equals (9) minus (8). RAS uses this criterion to choose between P_1 and P_2 .

4.3 Randomization with Adaptive Cutoff Update and Space Utilization (RACAS)

Randomization with Adaptive Cutoff Update and Space Utilization (RACAS) combines methods RAC and RAS. RACAS initially uses RAC to determine the batch size in each iteration. After the batch size increases to the maximum memory size, RACAS ends RAC and starts to use RAS to determine the partition sizes.

5 EXPERIMENTAL STUDY

We evaluated our proposed methods by comparing them with existing methods, (i) ORCA with the optimal batch size and (ii) RCS. We tested them with increasingly larger data sets under different conditions. Data sets of different sizes were generated by truncating the original shuffled data set.

We tested our proposed methods on a network intrusion detection problem. The data set is “KDD Cup 1999”, which contains higher-level features of TCP data for a local-area network simulating a military network environment with a wide variety of intrusions. Among the 42 features, 34 are continuous and 8 are categorical. The data set size is 4898431. The aim is to distinguish intrusions (outliers) from normal network connections.

We measure the performance by running time, which is the sum of CPU and disk I/O time. The CPU time is measured by the number of distance computations times t_{DC} (the time to calculate a distance). The disk I/O time is measured by the number of objects read from the disk times t_{OR} (the time to read an object from the disk). The values of t_{DC} and t_{OR} are directly related to the CPU and disk I/O performance of computers. They are also related to the characteristics of data sets. t_{DC} depends on the number of computations required to calculate the distance between two objects and t_{OR} depends on the storage size of an object. We define a ratio $r = \frac{t_{OR}}{t_{DC}}$ to characterize the performance of a computer on a data set.

Our tests were performed on a computer with a 2GHz processor and a 5400rpm hard drive running on Windows Vista operating system. The measured values of t_{DC} and t_{OR} of the computer on the data set “KDD Cup 1999” were 2.7269×10^{-7} and 5.9186×10^{-6} seconds respectively. They were the average values of calculating 300,000,000 distances and reading 10,000,000 objects from the disk respectively. The performance ratio of the computer on the data set

was $r_{ours} = \frac{5.9186 \times 10^{-6}}{2.7269 \times 10^{-7}} = 21.70$. The maximum memory size M was set to 1000 data objects to simulate a situation that the ratio of the data set size to the memory size was high. If the batch size was smaller than M , the remaining memory space would act as a cache for data scans.

Table 1. The three problem and system settings used in our experiments.

Case	Number of outliers to find	r
A	0.01% of N	21.70
B	0.01% of N	217.0
C	0.1% of N	21.70

Table 1 shows the three problem and system settings used in our experiments. In Case A, tests were run on a computer with a performance ratio r of 21.70 and the number of outliers to find is 0.01% of the data set size N . In Case B, r was 10 times larger than in Case A. This simulated a situation that the relative disk I/O performance of a computer was ten times slower or the relative CPU performance was ten times faster, which is a reasonable projection to the future since the improvement on CPU speed is much faster than that on disk I/O speed. Finally, in Case C, we increased the number of outliers to find from 0.01% to 0.1% of the data set size N .

5.1 Optimal batch size of ORCA

This section reports the results of finding the batch size of ORCA that gives the shortest running time. We searched for the optimal batch size by varying the batch size from 0.05% to 2% of the data set size N . We repeated the search for different N (from 100,000 to 1,000,000) under different conditions (Cases A to C in Table 1).

The results show that the optimal batch size is a fixed percentage of N , given the same number of outliers to find and same performance ratio r of a computer. The optimal batch sizes for Cases A to C are 0.1%, 1% and 1% of N respectively. Moreover, the optimal batch size increases with N , the number of outliers to find and r .

5.2 Performance of RAC, RAS and RACAS

Figure 5 shows the percentage of time saved by our proposed methods by varying the data set size N , the number of outliers to find and the performance ratio of a computer. The reference of the comparisons is ORCA with the *optimal batch size* found in Sec. 5.1. If the optimal batch size is larger than the maximum memory size M , M will be selected as the batch size. The three plots in each row of the figure correspond to the problem and system settings Cases A to C shown in Table 1 respectively.

The parameters of RC, RS and RCS were chosen as in [8, 9]. For the RC part, (i) the initial batch size b_s equals $\max(4n, 100)$ where n is the number of outliers to find. (ii) The main batch size equals M . (iii) The criterion to change the initial batch size is that the ratio of new to old cutoff values is smaller than $1.1^{b_s/100}$. For the RS part, (i) the initial partition size equals N . (ii) The main partition size equals $N/10$. (iii) The time to change the initial partition size is when 10% of the data set has finished processing.

Figure 5 shows that RAC, RAS and RACAS perform better than their unadaptive counterparts in nearly all cases. RAC in Figs. 5(a), (b) and (c) usually saves a few percentages of time more than RC.

The percentages of time saved by RAS and RACAS are substantial when the data set size N , the performance ratio r of a computer or the percentage of outliers to find is large. The largest time savings attained by RAS in Figs. 5(d), (e) and (f) are 45%, 89% and 66% respectively. The time savings of RAS are 286%, 470% and 133% of those of RS. The largest time savings attained by RACAS in Figs. 5(g), (h) and (i) are 49%, 87% and 66% respectively. The time savings of RACAS are 230%, 507% and 135% of those of RCS.

6 RELATED WORK

One closely related work is Recursive Binning and Re-Projection (RBRP) [4] which aims at speeding up ORCA. It employs a pre-processing step of data clustering. Within each cluster, objects are further reorganized according to their order in the projection along the principal component. This pre-processing step saves the total running time by allowing fast determination of neighbors of non-outliers.

Another outlier definition proposed by Knorr and Ng [5] is closely related to the one used in this and other papers [2, 3, 4, 7, 8, 9]. An object is defined to be an outlier if there are fewer than p objects within distance d from the object. Knorr and Ng [5] presented two algorithms for mining outliers defined in this way. One is a Nested-Loop (NL) algorithm, which has a complexity of $O(lN^2)$, where l and N are the dimension and size of a data set respectively. The other is a cell-based algorithm, which partitions the data space into cells and uses cell-by-cell processing. It has a complexity that is linear with respect to N , but exponential with respect to l . It guarantees at most 3 passes over a data set. Later, Tao et al. [10] proposed an algorithm called scan with prioritized flushing (SNIF), which scans the data set at most twice. Angiulli et al. [1] proposed DOLPHIN, which has the disk I/O cost corresponding to scanning the data set twice and has linear-time performance in the data set size.

Apart from the above distanced-based approaches, there are many other approaches [6] such as (i) statistical tests, which find outliers with a low probability to be generated by the assumed underlying distribution, (ii) depth-based approaches, which find outliers at the border of the data space, (iii) deviation-based approaches, which find outliers whose removal from the data set minimizes the data set variance, (iv) density-based approaches, which find outliers where the densities around them are considerably different from the densities around their neighbors, and (v) approaches used in high-dimensional data sets.

7 CONCLUSION

In this paper, we propose RACAS to further reduce the execution time of RCS by dynamically adapting the frequency values of updating the cutoff value and reclaiming the memory space at each step. Our main contributions are as follows. (i) We empirically analyzed the relationship between the performance of ORCA and the setting of the problem, data set and system. (ii) We theoretically derived the formulation of RACAS so that it can adapt its parameters automatically while the existing counterpart RCS requires empirically determined parameters for its maximum performance. (iii) We conducted our empirical evaluation which shows that the time saving of RACAS is substantial (up to 5 times the time saving of RCS). We can attribute the results to the fact that RACAS dynamically adapts its parameters for each step while RCS uses empirically determined parameters for two chunks of steps.

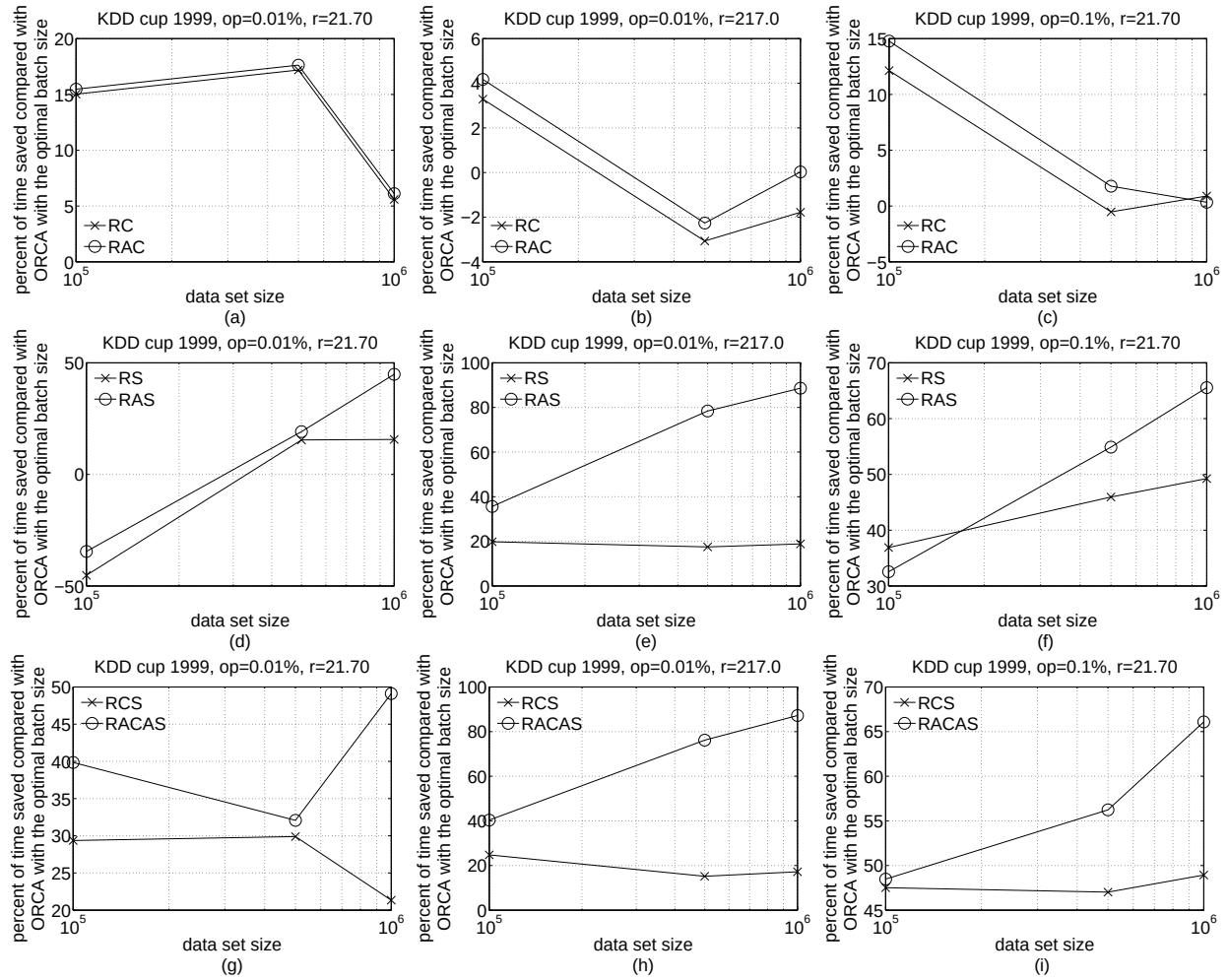


Figure 5. Percentage of time saved compared with ORCA with the optimal batch size by varying the data set size N , the outlier percentage op , and the performance ratio r .

Acknowledgements

The work described in this paper was partially supported by grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. PolyU 5191/09E, PolyU 5182/08E, PolyU 5181/06E, PolyU 5174/07E).

REFERENCES

- [1] Fabrizio Angiulli and Fabio Fasseti, 'Dolphin: An efficient algorithm for mining distance-based outliers in very large datasets', *ACM Trans. Knowl. Discov. Data*, **3**(1), 1–57, (2009).
- [2] Fabrizio Angiulli and Clara Pizzuti, 'Fast outlier detection in high dimensional spaces', in *PKDD '02: Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, pp. 15–26, London, UK, (2002). Springer-Verlag.
- [3] Stephen D. Bay and Mark Schwabacher, 'Mining distance-based outliers in near linear time with randomization and a simple pruning rule', in *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 29–38, ACM, (2003).
- [4] Amol Ghoting, Srinivasan Parthasarathy, and Matthew Eric Otey, 'Fast mining of distance-based outliers in high-dimensional datasets', *Data Min. Knowl. Discov.*, **16**(3), 349–364, (2008).
- [5] Edwin M. Knorr and Raymond T. Ng, 'Algorithms for mining distance-based outliers in large datasets', in *VLDB '98: Proceedings of the 24th International Conference on Very Large Data Bases*, pp. 392–403, Morgan Kaufmann Publishers Inc., (1998).
- [6] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Outlier detection techniques. Tutorial at the 13th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2009), Bangkok, Thailand, 2009.
- [7] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim, 'Efficient algorithms for mining outliers from large data sets', *SIGMOD Rec.*, **29**(2), 427–438, (2000).
- [8] Chi-Cheong Szeto and Edward Hung, 'Mining outliers with faster cut-off update and space utilization', in *PAKDD*, eds., Thanaruk Theeramunkong, Boonserm Kijsirikul, Nick Cercone, and Tu Bao Ho, volume 5476 of *Lecture Notes in Computer Science*, pp. 823–830. Springer, (2009).
- [9] Chi-Cheong Szeto and Edward Hung, 'Mining outliers with faster cut-off update and space utilization', *Pattern Recognition Letters*, **In Press, Corrected Proof**, –, (2010).
- [10] Yufei Tao, Xiaokui Xiao, and Shuigeng Zhou, 'Mining distance-based outliers from large databases in any metric space', in *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 394–403. ACM, (2006).