Mining Distance-based Outliers from Large Databases in Any Metric Space

Yufei Tao Xiaokui Xiao
Dept. of Computer Science and Engineering
Chinese University of Hong Kong
Sha Tin, New Territories, Hong Kong
{taoyf, xkxiao}@cse.cuhk.edu.hk

Shuigeng Zhou
Dept. of Computer Science and Engineering
Fudan University
Handan Road, Shanghai, China
sgzhou@fudan.edu.cn

ABSTRACT

Let R be a set of objects. An object $o \in R$ is an *outlier*, if there exist less than k objects in R whose distances to o are at most r. The values of k, r, and the distance metric are provided by a user at the run time. The objective is to return all outliers with the smallest I/O cost.

This paper considers a generic version of the problem, where no information is available for outlier computation, except for objects' mutual distances. We prove an upper bound for the memory consumption which permits the discovery of all outliers by scanning the dataset 3 times. The upper bound turns out to be extremely low in practice, e.g., less than 1% of R. Since the actual memory capacity of a realistic DBMS is typically larger, we develop a novel algorithm, which integrates our theoretical findings with carefully-designed heuristics that leverage the additional memory to improve I/O efficiency. Our technique reports all outliers by scanning the dataset at most twice (in some cases, even once), and significantly outperforms the existing solutions by a factor up to an order of magnitude.

Categories and Subject Descriptors: H3.3 [Information Storage and Retrieval]: Information Search and Retrieval

General Terms: Algorithms, Experimentation

Keywords: Mining, Outlier, Metric Data

1. INTRODUCTION

Data mining aims at discovering interesting characteristics of a dataset, mainly in the forms of correlation (particularly, association rules) and clusters, in order to assist advanced decision making (e.g., classification of new objects, prediction of events). All these mining operations draw conclusions from a majority of the dataset, as in association rule mining, where a rule is useful only if it is supported by a sufficiently large subset of the database. In this case, "outliers", i.e., objects differing in behavior with the majority, are harmful (and hence, must be ignored), since they may reduce the accuracy of the mined results.

Outliers, however, have their own merits, as recognized by Knorr and Ng in their pioneering paper [8]. The merits arise from the fact that outliers typically indicate irregular patterns that deserve special attention. Such patterns are especially important in security systems, where, contrast to traditional mining tasks, the goal is *not* to understand the patterns of the "majority", but rather, to capture abnormal "minorities". In fact, in these systems, the majority patterns may even have been obtained previously, and are taken as an input for assisting outlier mining.

Using a popular example in the outlier-analysis literature, consider a system that detects frauds in creditcard transactions. Over 99.9% of the transactions are ordinary, and their behavior conforms to certain patterns which have long been understood by experts. Based on such understanding, the experts hope to identify the remaining (less than 0.1%) transactions demonstrating suspicious deviation from normal behavior, in order to alert the company about possible investigatory actions.

1.1 Problem Formulation

Let R be a relation with n objects o_1, o_2, ..., o_n. A user specifies a distance threshold r, an integer k by far smaller than n, and a distance function $d(o_i, o_j)$ $(i, j \in [1, n])$. Function $d(.)$ should be a metric, i.e., it satisfies the triangle inequality.

DEFINITION 1. *Given any o_i, o_j in R, if $d(o_i, o_j) \le r$, we say that o_i is a* **neighbor** *of o_j (likewise, o_j is also a neighbor of o_i). Specially, each object is a neighbor of itself.*

DEFINITION 2. *An* **outlier** *is an object that has less than k neighbors.*

The above "distance-based" outlier definition is proposed by Knorr and Ng [8], who provide solid justification about its usefulness and importance in practice[1] [8, 9, 10].

We are interested in generic solutions that utilize only objects' mutual distances, and do not place any constraint on data and the function $d(.)$, apart from the fact that $d(.)$ should be a metric. Such a solution, therefore, is applicable in all outlier-detection applications, regardless of the data types (pictures, movies, time series, ...) and $d(.)$ (Euclidean distance, road network distance, edit distance, ...). This requirement excludes the existing solutions (surveyed in Section 5) which constrain the objects and/or $d(.)$ to be in Euclidean space[2], or assume the possibility of creating an index on R.

[1]As an interesting property, the definition is compatible with the notion of "rare events" in statistics. For instance, if the underlying objects are known to obey a Gaussian distribution, an object is a rare event if its value deviates from the mean of the Gaussian by more than 10 times the standard deviation. Such objects can be captured as outliers, by setting r and k appropriately (see a formula in [8])

[2]For example, objects are multi-dimensional points, and $d(.)$ captures their L_2-distance.

Our discussion focuses on large datasets that do not fit in memory, rendering minimization of I/O cost to be a major concern in algorithm design. We do not demand any index structure on R, but the objects should have been stored in a *random* order, as can be easily achieved by a simple randomization process.

1.2 Contributions and Paper Organization

Our objective is to discover all the outliers with I/O overhead linear to the database size. A similar attempt has been made by Bay and Schwabancher [3]. They show that the problem can be solved with linear CPU time. Unfortunately, their solution incurs quadratic I/O cost, as will be analyzed in the next section.

Motivated by this, we present a systematic study of the problem, and make two major contributions. First, we establish, through a probabilistic analysis based on random sampling, an upper bound for the amount of memory required to retrieve all outliers by scanning the dataset 3 times. The upper bound indicates an important fact: the memory usually needs to hold only 1% of a practical dataset to achieve the 3-times-scan performance!

Since the memory capacity of a modern DBMS may be larger, as the second contribution, we develop a new algorithm, SNIF (*scan with prioritized flushing*), which integrates our theoretical findings with several carefully-designed heuristics that leverage the additional memory to improve I/O efficiency considerably. Extensive experiments demonstrate that SNIF completes outlier mining by scanning the dataset at most twice, and sometimes, even once.

The rest of the paper is organized as follows. Section 2 discusses the drawbacks of an existing method that deploys nested loop. Section 3 lays down the theoretical foundation for the proposed technique, based on which Section 4 explains the details of SNIF. Section 5 reviews the previous work that is related to ours. Section 6 verifies the efficiency of our method with extensive experiments. Section 7 concludes the paper with directions for future work.

2. PITFALLS OF NESTED LOOP

A straightforward solution to our problem is nested loop (NL). That is, for each object $o \in R$, scan the database from the beginning, counting the number of objects within distance r from o. The scan is terminated as soon as the counter reaches k, i.e., o is not an outlier. A complete scan of the database is necessary only if o is an outlier.

Despite the clear $O(n^2)$ complexity of the algorithm, Bay and Schwabancher [3] present a surprising, yet reasonable, result: the actual CPU time of NL is often linear to the dataset size. This phenomenon is due to the observation that, for most non-outlier objects in R, scan of the dataset terminates very early so that only a fraction of the dataset is examined.

To illustrate this specifically, denote x as the number of neighbors of o (i.e., x is the number of objects in R with distances at most r to o). Remember that objects in R have been randomized, so that the next object scanned always has a probability x/n to be a neighbor of o. So, if o is not an outlier, in expectation, $k/\frac{x}{n}$ objects need to be checked before k neighbors are found.

Therefore, if we assume that there are y outliers in R, and $\bar{x}$ is the average of x for all non-outliers, the expected number of scanned objects (in the entire execution of NL) equals

$$k/\frac{\bar{x}}{n} \cdot O(n) + y \cdot n \tag{1}$$

where the term $O(n)$ corresponds to the fact that the number of non-outliers is bounded by n.

The value of y is extremely small with respect to n, so the second term is roughly linear on n. In the first term, note that $\bar{x}/n$ actually does not depend on n, but instead it is a constant related to the data distribution. To understand this, imagine that more objects are added to R following the same distribution; then, along with the increase of n, the value of $\bar{x}$ also increases, such that $\bar{x}/n$ is still equivalent to the probability that the distance between two objects is at most r. Hence, the first term in Formula 1 is also linear to n.

Unfortunately, the analysis of [3] fails to account for the fact that, in database environments, nested loop is performed in *blocks* — block nested loop (BNL). Assume that each disk page can accommodate b objects, and that the memory has m pages. Each scan of the database is performed with $m - 1$ pages of objects in memory, and thus, the scan may terminate only when *all* the $(m - 1) \cdot b$ memory-resident objects have been confirmed as non-outliers. Next, we will show that, for typical values of b and m, each scan must cover a significant portion of the database, rendering the overall I/O cost $O((n/b)^2)$ (note that this does not contradict the earlier analysis, which only shows that the *number of distance computations* is linear to n).

Let us use p to denote the average probability that a non-outlier object $o \in R$ can be verified *without* scanning 90% of the dataset. Then, in BNL, a scan does not need to examine 90% of R, only with a probability approximately $p^{(m-1)\cdot b}$. Consider a dataset of 2D points where 0.05% of the objects are outliers. The value of p is at most 99.95%. In practice, $(m - 1) \cdot b$ can easily reach 10000 (e.g., m = 101, and b = 100) such that $p^{(m-1)\cdot b}$ evaluates to less than 1%! In other words, almost every scan must access 90% of the pages occupied by R, leading to quadratic I/O overhead.

3. RATIONALE OF OUR TECHNIQUE

This section justifies the possibility of discovering all outliers with I/O cost linear to n/b, where n is the cardinality of R, and b the number of objects in a disk page. Specifically, we show that a very small amount of memory (around 1% of the dataset) is usually sufficient for retrieving all the outliers by scanning the dataset 3 times. This motivates an algorithm presented in Section 4, which further improves performance by scanning the dataset even fewer times.

3.1 Fundamental Theoretical Facts

Let us randomly sample s objects from R. To allow rigorous analysis, we follow the strategy of "sampling with replacement" [12]. Specifically, each random sample is taken *independently* from *all* the objects in R, i.e., it is possible that multiple samples happen to be the same object.

We use the sample set to build s *partitions* PA_1, PA_2, ..., PA_s of R. Each sampled object is the *centroid* of a partition; hence, we denote the s samples as $PA_1.o$, ..., $PA_s.o$ respectively, after the partitions they represent. Besides its centroid, a partition PA_i (for some $i \in [1, s]$) includes all the objects $o \in R$ that satisfy two conditions:

1. the distance from o to $PA_i.o$ is no more than $r/2$ (i.e., half the parameter r of outlier definition), and
2. o is nearer to $PA_i.o$ than to the centroid of any other partition (in case the object is equi-distant to two or more centroids, the partition to which the object belongs is a random one among the partitions represented by these centroids).

The union of the partitions may *not* be R, since an object is not in any partition if it is farther away from all centroids than $r/2$.

We compute a *density*, $PA_i.den$, for each partition PA_i ($1 \leq i \leq s$). Specifically, $PA_i.den$ is the number of objects (including

$PA_i.o$ itself) whose distances to $PA_i.o$ are at most $r/2$. Note that an object contributing to the density of a partition does not necessarily belong to that partition. In particular, a single object o may contribute to the densities of multiple partitions, but o only belongs to a single partition, i.e., the one whose centroid is the nearest to o.

All the partitions, including their centroids and densities, constitute a *data summary* of R. The next lemma shows that non-outliers may be verified directly from the data summary.

LEMMA 1. *If $PA_i.den \geq k$ (for any $i \in [1, s]$), none of the objects belonging to PA_i can be an outlier.*

PROOF. The lemma follows the fact that function $d(.)$ satisfies the triangle inequality. Let o be an object belonging to PA_i, and o' an object that contributes to $PA_i.den$. Thus, it holds that $d(o, PA_i.o) \leq r/2$ and $d(o', PA_i.o) \leq r/2$, leading to $d(o, o') \leq r$. $PA_i.den \geq k$ means that there are at least k such o'; hence, o is not an outlier. $\square$

Let us divide the s partitions into two disjoint sets. The first one S_{good} includes all the partitions whose densities are at least k. The second set S_{bad} involves all the remaining partitions whose objects, therefore, cannot be asserted as non-outliers from the data summary.

Assuming the memory has m pages, we have:

LEMMA 2. *We can find all outliers by scanning the dataset 3 times, if $m-1$ pages can accommodate the objects qualifying one of these conditions: the object (i) is a partition centroid, (ii) does not belong to any partition at all, or (iii) belongs to a partition in S_{bad}.*

PROOF. The data summary, which includes the centroids and densities of all partitions, can be constructed by scanning R once. In particular, since objects are stored in a random order, the centroids can be simply set to the first s objects encountered in the scan; thus, it is not necessary to perform a separate sampling process for obtaining the centroids.

After the first scan, we identify the set S_{good} of partitions with densities at least k. Then, we keep the partition centroids in memory (but throw away partition densities), and perform a second scan over R. In this scan, a fetched object is discarded immediately if it belongs to any partition in S_{good} (by Lemma 1). An un-discarded object is retained in memory. In this way, the amount of consumed memory gradually increases; but, given the condition in Lemma 2, this amount is expected to be less than $m-1$ pages at the end of the second scan. Keeping all the non-prunable objects in memory, we perform a third scan over R, using all the un-occupied memory pages as the input buffer (there is at least one such page), in order to determine whether each object is an outlier. $\square$

How large should m (i.e., the memory) be, in order to allow a 3-times-scan algorithm? To answer this queston, our first step is to quantify the number of objects satisfying condition (ii) of Lemma 2. For each object $o_i \in R$ ($1 \leq i \leq n$), we use $o_i.n_{\leq r/2}$ to represent the number of objects in R (including o_i itself) whose distances to o_i do not exceed $r/2$.

LEMMA 3. *The expected number of objects in R that do not belong to any partition equals*

$$\sum_{i=1}^{n} \left(1 - \frac{o_i.n_{\leq r/2}}{n}\right)^s. \tag{2}$$

PROOF. Since the centroids are obtained from R following the sampling-with-replacement scheme, there are totally n^s possible centroid sets, each of which is taken with an equal probability. Let us denote them as $CS_1, CS_2, ..., CS_{n^s}$, respectively.

We construct a two-dimensional array with n rows and n^s columns, where the i-th ($1 \leq i \leq n$) row concerns object o_i in R, and the j-th ($1 \leq j \leq n^s$) column corresponds to CS_j. In each cell c_{ij} at the i-th row and j-th column, we fill in '0' if o_i belongs to some partition when the set of sampled centroids is CS_j; otherwise (i.e., o_i is not "captured" by any partition), we fill in '1'.

If we add up the cell-values at the j-th column, the sum, represented as col_j, equals the number of "un-captured" objects according to the centroid set CS_j. Hence, the expected number of un-captured objects (given an arbitrary centroid set) is the average sum of all columns:

$$\frac{1}{n^s} \sum_{j=1}^{n^s} col_j \tag{3}$$

Note that $\sum_{j=1}^{n^s} col_j$ in the above formula is exactly the number of 1's in the array. Next, we count the 1's in an alternative "row-oriented" manner. Let row_i be the number of 1's at the i-th row. Clearly, row_i is the number of centroid sets that do *not* capture object o_i. We call such a centroid set a "non-capturing CS of o_i".

The distances between o_i and all the centroids in a non-capturing CS are larger than $r/2$. Since $n - o_i.n_{\leq r/2}$ objects in R are farther away from o than $r/2$, *every* centroid in a non-capturing CS of o_i must originate from those $n - o_i.n_{\leq r/2}$ objects. Hence, there are $(n - o_i.n_{\leq r/2})^s$ different non-capturing CS's of o_i.

Therefore, $\sum_{j=1}^{n^s} col_j$ (the number of 1's in the array) equals $\sum_{i=1}^{n} (n - o_i.n_{\leq r/2})^s$, which, when plugged into Formula 3, gives Formula 2 $\square$

Clearly, the chance that an object belongs to no partition decreases exponentially with s. Let us set s to 1000 (the centroids constitute a very small sample set of R). As a result, for any non-outlier object $o \in R$, as long as $\frac{o.n_{\leq r/2}}{n}$ is a non-trivial selectivity, $(1 - \frac{o.n_{\leq r/2}}{n})^s$ evaluates to a negligible value. For example, if $\frac{o.n_{\leq r/2}}{n} = 0.5\%$, $(1 - \frac{o.n_{\leq r/2}}{n})^s$ becomes less than 1%. Hence, the number of objects not captured by any partition is very small (we will demonstrate this in the next section).

Let us use n_{dense} to denote the number of objects $o \in R$ whose $o.n_{\leq n/2}$ is at least k. We arrive at a formal result regarding the memory size for fulfilling the condition in Lemma 2.

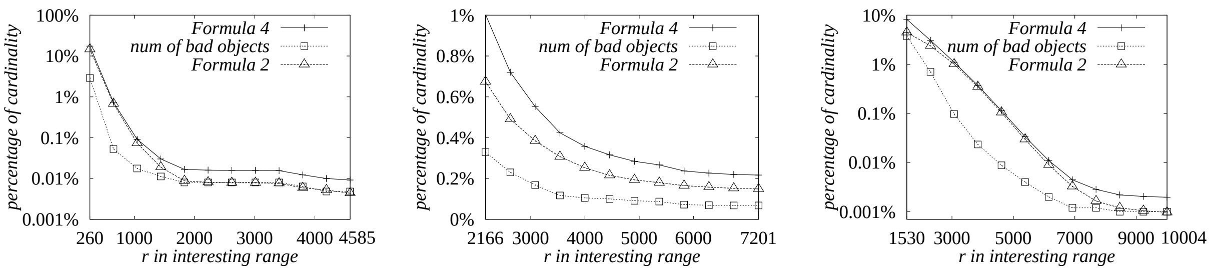
COROLLARY 1. *We expect to find all outliers by scanning the dataset 3 times, if*

$$s + \sum_{i=1}^{n} \left(1 - \frac{o_i.n_{\leq r/2}}{n}\right)^s + (k-1) \cdot s \cdot \left(1 - \frac{n_{dense}}{n}\right) \tag{4}$$

objects can be stored in $m-1$ pages.

PROOF. Lemma 2 says that, to achieve the designated query cost, $m-1$ pages should be sufficient for storing three types of objects (i), (ii), and (iii). The number of objects of type (i) is s, and the number for type (ii) has been given in Lemma 3. To prove the corollary, it remains to show that the number of type (iii) objects is at most $(k-1) \cdot s \cdot \left(1 - \frac{n_{dense}}{n}\right)$ in expectation.

Notice that a partition belongs to S_{bad} if and only if its centroid is one of the $n - n_{dense}$ objects o whose $o.n_{\leq r/2}$ is less than k. Since each centroid is randomly picked from R, it has $1 - \frac{n_{dense}}{n}$ probability to produce a "bad partition", or equivalently, the expected number of bad partitions is $s \cdot \left(1 - \frac{n_{dense}}{n}\right)$. Finally, each bad partition contains at most $k-1$ objects, thus completing the proof. $\square$



(a) *CA* (b) *Household* (c) *Server*

Figure 1: Minimum memory requirements for finding outliers by scanning a dataset 3 times ($s = 1000$)

3.2 Evidence from Real Data

The goal of this section is to identify the value of Formula 4 for practical data, since the formula indicates the minimum amount of memory for finding outliers by 3 scans of R. For this purpose, we examine the following datasets[3] popular in the literature:

- *CA*: a spatial dataset released by the TIGER project, containing 62k two-dimensional points representing addresses in California.
- *Household*: released by the US Census Bureau, containing 1 million three-dimension points, each of which represents the annual expenditure of an American family on electricity, gas, and water, respectively.
- *Server*: KDD Cup 1999 data containing the statistics of 500k network connections; we extract the following attributes[4] to create a five-dimensional point dataset: *count*, *srv-count*, *dest-host-count*, *dest-host-srv-count*, and *dest-host-same-srv-count*.

In all cases, the data space is normalized such that each axis has a domain of [0, 10000]. The distance function $d(.)$ corresponds to Euclidean distance.

Outlier formulation requires parameters r and k. The value of k is easier to set [8]: it ranges between 0.01% and 0.1% of the dataset cardinality n. In this subsection, we fix k to the median value $0.5\% \cdot n$. The setting of r is more difficult because an excessively small r leads to an unrealistically large number of outliers, whereas an overly-large r does not produce any outlier at all.

Therefore, for each dataset, we decide an *interesting range* $[r_{min}, r_{max}]$ of r as follows. Initially, r equals ∞, and obviously, no object qualifies as an outlier. Then, we gradually decrease r, until some object qualifies for the first time; the value of r at this point equals r_{max}. Next, we continuously reduce r (so that the number of outliers increases), and stop as soon as there are exactly $0.1\% \cdot n$ outliers. The current value of r equals r_{min}. As a result, by investigating $r \in [r_{min}, r_{max}]$, we simulate the practical environments [8, 10] where the number of outliers is below $0.1\% \cdot n$. The interesting ranges of r for *CA*, *Household*, and *Server* are $[260, 4585]$, $[2166, 7201]$, $[1530, 10004]$, respectively.

In Figure 1a, the curve labeled with '*Formula 4*' plots the value of the formula as a function of r (in its interesting range) for *CA*. The formula value is represented as a percentage of the dataset's cardinality (e.g., 1% means 6.2k objects). Formula 4 contains 3 terms; except the first term s, the other two terms vary with r. Hence, Figure 1a also demonstrates the values of the second and third terms as a function of r, with curves labeled with '*Formula 2*' and '*num of bad objects*', respectively (recall that the third term is the number of objects in partitions of S_{bad}). Again, all values are in percentages of the cardinality. Figures 1b and 1c illustrate the same information for *Household* and *Server*, respectively. In all figures, s equals 1000.

There are two important observations:

1. The value of Formula 4, which is the minimum memory size for achieving 3-times-scan performance, is less than 10% of a dataset for all r in the corresponding interesting range.

2. The value of Formula 4 decreases nearly exponentially as r increases! Specifically, for most r-values in an interesting range, the memory only needs to hold 1% of a dataset (in particular, this is true for the entire interesting range of dataset *Household*).

In fact, the above observations are general, and exist in a large number of real datasets. This is not surprising, because the value of r for meaningful outlier mining cannot be too small (see the values of r_{min} in Figure 1), resulting in a non-trivial $o.n_{\le r/2}$ for a vast majority of non-outlier objects $o \in R$. This means (i) $\frac{o.n_{\le r/2}}{n}$ may easily reach a selectivity above 0.5%, rendering Formula 2 to descend to a tiny value (as mentioned in Section 3.1), and (ii) n_{dense} is close to n (i.e., most objects o satisfy $o.n_{\le r/2} \ge k$), so that there are very few partitions in S_{bad}, leading to a small value for the third term of Formula 4. Finally, combining (i), (ii) and the fact that s accounts for a very small percentage of n, we have shown that outlier mining with linear I/O overhead is possible, usually with a small amount of memory.

4. THE SNIF TECHNIQUE

Motivated by the analysis in the last section, in the sequel, we propose a new algorithm, *scan with prioritized flushing* (SNIF), which finds all outliers by scanning R at most twice, as verified in our experiments.

The efficiency of SNIF owes to the fact that, the memory size is typically larger than the smallest size (often less than 1% of R as in Figure 1) necessary for a 3-times-scan solution. Thus, we can afford to retain more objects in memory during the first dataset scan, which allows us to claim a significant portion of R as non-outliers directly after the scan! As a result, the remaining objects that require further verification may fit in memory, so that another scan of R suffices to determine the exact outliers.

Based on this idea, SNIF deploys a novel *prioritized flushing* technique to minimize the chance of performing the third scan of R. Specifically, the technique associates each object with a "priority", and, whenever the memory becomes full, flushes the objects with the lowest priorities. The priorities are designed in such a way that, objects in memory are those deserving "more attention": (i) outliers, and (ii) non-outliers with relatively few neighbors. Since

[3]*CA* can be downloaded at *http://www.census.gov/geo/www/tiger*, *Server* at *http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html*, and *Household* at *https://www.ipums.org*.

[4]Semantics of these attributes are available here: *http://kdd.ics.uci.edu/databases/kddcup99/task.html*.

a memory-resident object o is checked with every subsequently scanned object, we obtain a highly accurate count of the neighbors of o, and may be able to use it for deciding whether o is an outlier.

Next, we explain the components of SNIF in detail.

4.1 Critical Moment

SNIF starts by reading R from the beginning, and retaining the retrieved objects in memory, until the memory becomes full for the first time — the *critical moment* of SNIF.

At this moment, we have obtained $b \cdot m$ objects, where b is the number of objects that can be accommodated in a page, and m the number of memory pages. For each of the these objects o, we obtain a *neighbor counter* $o.n_{nb}$, equal to the number of neighbors of o in the already-inspected part of the dataset.

Then, SNIF builds a data summary (as defined in the previous section) with respect to these $b \cdot m$ objects as follows. For each object o, the algorithm estimates the total number of its neighbors in the *entire* R as $o.n_{nb} \cdot \frac{n}{b \cdot m}$ (n is the cardinality of R), utilizing the property that these $b \cdot m$ objects constitute a random sample set of R. From the objects with estimates at least k, we randomly sample s (= 1000 in our implementation) *different* objects (i.e., sampling *without* replacement [12]) as the partition centroids $PA_1.o$, ..., $PA_s.o$. This way, none of the centroids is likely to be an outlier.

For each of the $b \cdot m$ memory-resident objects, SNIF decides the partition it belongs to, using the two conditions stated in Section 3.1 (some objects may not belong to any partition). Next, we set the density $PA_i.den$ of each partition ($1 \le i \le s$) to the number of objects (currently in memory) with no more than distance $r/2$ to $PA_i.o$.

For each partition PA_i, SNIF maintains a *radius* $PA_i.r$, which equals the maximum distance between the centroid $PA_i.o$ and any object belonging to PA_i. Obviously, due to the fact that an object is assigned to PA_i only if $d(o, PA_i) \le r/2$, the value of $PA_i.r$ never exceeds $r/2$.

After the above operations, SNIF performs the first flushing, after which only $b \cdot m/2$ objects (including all the partition centroids) remain in memory. We will elaborate the details of flushing later in Section 4.3. For now, it suffices to note several facts:

1. Among the objects removed from memory, those with $o.n_{nb} \ge k$ are directly discarded (i.e., they are definitely non-outliers), while the others are written to a *verification file*.
2. For each object appended to the file, we keep with it, in the file, the ID of the partition (in the data summary created earlier) it belongs to.
3. For each partition PA_i ($1 \le i \le s$), we record in memory the number $PA_i.n_{removed}$ of objects belonging to PA_i that have been removed (i.e., discarded or preserved in the verification file). We also record the number $n^{noPA}_{removed}$ of objects that have been removed, but do not belong to any partition.
4. During the first scan of the database, the neighbor counter $o.n_{nb}$ of any object o in memory is always a *lower bound* of the actual number of neighbors of o, in the part of the database already scanned.
5. At any time after the critical moment during the first scan, the number of objects that remain in memory is at least $b \cdot m/2$.

4.2 Processing Subsequent Objects

After the critical moment, SNIF continues to scan R. For each object o encountered, we compute the distances between o and the centroids of all partitions. For each $i \in [1, s]$, if $d(o, PA_i.o) \le r/2$, the density $PA_i.den$ is increased by 1. After this, the partition PA_j (for some $j \in [1, s]$) to which o belongs is also decided. If $d(o, PA_j.o)$ is larger than $PA_j.r$ (the radius of PA_j), then $PA_j.r$ is set to $d(o, PA_j.o)$. Hence, the data summary (particularly, the density and radius of each partition) is always *precise* with respect to the objects already scanned.

Next, we initiate the neighbor counter $o.n_{nb}$ of o as 0, and then compute its distance to every object o' being retained in memory[5]. If $d(o, o') \le r$, both $o.n_{nb}$ and $o'.n_{nb}$ are increased by 1. Hence, it is clear that the longer o' stays in memory, the more likely its neighbor counter can reach k, increasing the chance that it can be confirmed as a non-outlier before being removed from memory.

Now that we have calculated $o.n_{nb}$ using the memory-resident objects, we attempt to further increase it by incorporating the objects that have been removed from the memory (i.e., either discarded or flushed to the verification file), using the data summary. There are two independent ways to achieve this purpose, leading to values v_1, v_2, both of which satisfy the lower-bound property of $o.n_{nb}$ (Fact 4 in Section 4.1). Naturally:

$$o.n_{nb} = \max\{v_1, v_2\} \tag{5}$$

Next we clarify the computation of v_1 and v_2, respectively.

Deriving $\mathbf{v_1}$. Let v be the value of $o.n_{nb}$ so far (obtained with respect to only the memory-resident objects). We first set v_1 to v, and then inspect each partition PA_i ($1 \le i \le s$) in turn. In case

$$d(o, PA_i.o) + PA_i.r \le r, \tag{6}$$

we add $PA_i.n_{removed}$ (the number of objects in PA_i removed from memory) to v_1. Recall that $PA_i.r$ is the largest distance between $PA_i.o$ and any object o' assigned to PA_i. Hence, the validation of Inequality 6 indicates, by the triangle inequality, that $d(o, o') \le r$, implying that *all* the removed objects in PA_i are neighbors of r.

Deriving $\mathbf{v_2}$. The formulation of v_2 is simpler. Specifically, if o belongs to a partition PA_i (for some $i \in [1, s]$), then v_2 equals $PA_i.den$, i.e., the number of objects o' within distance $r/2$ from $PA_i.o$ among the objects already encountered (o' must have distance at most r to o). Note that $PA_i.den$ already includes both the objects of PA_i in memory and those removed; hence, unlike v_1, v_2 does not need to take into account v.

As the number of memory-resident objects increases, the memory eventually becomes full again. When this happens, SNIF invokes another flushing, before resuming the scan of R. In the next section, we clarify the procedures of flushing.

4.3 Prioritized Flushing

Let n_{seen} be the number of objects in R that have been scanned; these objects form a random sample set of R. From the current density $PA_i.den$ of each partition PA_i, we estimate its final density (after scanning the entire R) as $PA_i.den \cdot n/n_{seen}$. Based on the estimates, we classify the partitions into two disjoint sets. The first one S_{good} includes the *potentially prunable* partitions whose predicted final densities are larger than k (by Lemma 1, no object in such a partition can be an outlier). The second one S_{bad} involves the remaining, potentially un-prunable, partitions.

Next, we divide the memory-resident objects (other than the partition centroids, which must stay in memory) into five disjoint types.

1. Objects whose current neighbor counters are $\ge k$;

[5]As an optimization, we perform the distance calculation only if the neighbor counter of either o or o' is smaller than k.

2. Objects that are not of the previous type, and belong to a partition in S_{good};
3. Objects that are not of the previous types, and belong to a partition in S_{bad};
4. Objects that are not of the previous types, do not belong to any partition, and were scanned *after* the critical moment.
5. Objects that are not of the previous types, do not belong to any partition, and were scanned *before* the critical moment.

We compute a *priority* for each object o as

$$(\text{type-id of } o) + \frac{\min\{n, x_{est}\}}{n+1} \tag{7}$$

where x_{est} is the estimated number of neighbors of o in the entire R. The second term of the above formula is a value in [0, 1), meaning that if an object has a smaller type-id, it has a lower priority, thus a *higher* chance to be eliminated from memory. Notice that the second term is only for distinguishing objects of the same type.

The priorities decided this way reflect the likelihood that objects are outliers: the smaller priority is, the less likely. To explain this, let us use o_{T1}, o_{T2}, ..., o_{T5} to represent an object of type-1, -2, ..., -5, respectively.

Clearly, o_{T1} is definitely a non-outlier, and o_{T2} most probably can be verified as a non-outlier using Lemma 1, at the end of the first database scan (when the data summary about the entire R is ready). o_{T3} may not be verified by Lemma 1, and thus, should have a greater priority than o_{T1} and o_{T2}. Nevertheless, compared to objects of types-4 and -5, o_{T3} has a better chance of being in a cluster, since it belongs to a partition. As a result, o_{T3} should possess a lower priority than o_{T4} and o_{T5}. Finally, the difference between o_{T4} and o_{T5} is that, the neighbor counter of o_{T5} is *precise* (the distances between o_{T5} and all scanned objects have been calculated) but that of o_{T4} is not. Therefore, o_{T5} is given a higher priority to stay in memory, so that, at the end of the first scan, we can claim it to be an outlier, if its neighbor counter is still lower than k.

Given two objects o, o' of the same type, why should the one, say o, with a larger x_{est} have a higher chance to stay in memory? This is justified by two reasons, both related to the fact that every un-scanned object has a larger probability to be a neighbor of o than of o'. The first reason is that if both o and o' were kept in memory, the neighbor counter of o would increase faster, and hence, would have a better chance of being confirmed as a non-outlier. Second, o would be more likely to increase the neighbor counters of the subsequently scanned objects, thus increasing the probability that these objects are validated as non-outliers, too.

After calculating the priorities of all memory-resident objects, SNIF sorts them in ascending order of the priorities, and removes the first $b \cdot m/2$ objects in the sorted list. Specifically, "removing" an object means discarding it if it is of type-1, or otherwise, appending it (in blocks) to the verification file together with its partition ID. Whenever an object in partition PA_i (for some $i \in [1, s]$) is appended, $PA_i.n_{remove}$ is increased by 1. If the object belongs to no partition, we add 1 to $n^{noPA}_{removed}$. In fact, as will be demonstrated experimentally later, *the verification file is usually empty, i.e., only type-1 objects are discarded at each flushing.*

Now we clarify the computation of x_{est}, the expected number of neighbors of o. For this purpose, we only need to maintain two additional values for o. The first one λ_1 equals the number of objects whose distances to o have been computed. The second value λ_2 is the number of these objects whose distances to o are at most r. Then, x_{est} is calculated as $\lambda_2 \cdot n/\lambda_1$. Note that λ_1 is at least $b \cdot m/2$, i.e., the estimation of x_{est} is based on a large sample set[6]. Specifically, if o was scanned before the critical moment, then λ_1 is at least $b \cdot m$ (the number of objects in memory at that moment). Otherwise, when o is read from the file, there must be at least $b \cdot m/2$ objects in memory (see Fact 5 in Section 4.1), whose distances to o are computed.

Finally, we point out that the contents of S_{good} and S_{bad}, as well as the priorities of objects, may vary at different flushings, since they depend on the densities of the partitions at the time of the corresponding flushing.

4.4 After the First Scan

After the file of R is exhausted, we have: (i) a complete data summary, (ii) a set of objects in memory, and (iii) a verification file containing objects whose qualification (being an outlier or not) could not be decided at their flushing time.

At this point, SNIF obtains sets S_{good}, S_{bad}, and classifies the memory-resident objects into the 5 types stated in Section 4.3. Type-1 and -2 objects are discarded as non-outliers, and type-5 objects are directly reported as outliers.

For each type-4 object o (which is usually an outlier), we attempt to verify it as follows. First, we count the number x_{mem} of neighbors of o among the objects currently in memory. Next, we collect the set S of partitions whose centroids have distances at most $\frac{3}{2}r$ to o (different o leads to different S). No object belonging to a partition outside S can have distance $\leq r$ to o, since the radius of each partition is no more than $r/2$. Then, o is an outlier if

$$x_{mem} + n^{noPA}_{removed} + \sum_{PA_i \in S, \forall i \in [1,s]} PA_i.n_{removed} < k \tag{8}$$

where $PA_i.n_{removed}$ (or $n^{noPA}_{removed}$) is the number of objects that were removed from memory, and belong to PA_i (or do not belong to any partition).

If o is indeed an outlier, in most cases we can verify it with Inequality 8 because (i) S is typically empty (an outlier tends to be faraway from all clusters), and (ii) $n^{noPA}_{removed}$ is often 0. To understand (ii), remember that very few objects belong to no partition: the number of them is given by Formula 2, and its value is usually less than 1% of n as shown in Figure 1. As long as the value does not exceed $b \cdot m/2$ (the number of removed objects in each flushing), all the "no-partition" objects are necessarily retained in memory, due to their high priorities.

Provided that r is not very close to the lower end of its interesting range (defined in Section 3.2), Type-3 objects often do not exist (the third term of Formula 4 upper bounds the number of such objects under a very low value). Otherwise, it implies the presence of a small cluster whose number of objects is at the order of k. This is rare because the cardinality n is larger than k by orders of magnitude (more than 1000 times), and thus, the size of a cluster is not likely to be comparable to k. However, if type-3 objects do exist, there is no effective way we can verify them without another scan of R. This is reasonable because determining such objects as non-outliers demands extremely accurate neighbor counters.

Having performed the above procedures, SNIF terminates if $n^{noPA}_{removed} = 0$, all type-4 objects have been verified as outliers, and there is no type-3 object. Otherwise, we execute a *verification step* as follows.

We read the verification file, checking the partition-IDs of the retrieved objects. If the object belongs to a partition in S_{good}, it

[6]If we use the optimization stated in Footnote 4, then we should revise the statement here: if the neighbor counter of o is less than k, x_{est} is computed from a sample set with size at least $b \cdot m/2$.

2	2	2	2	2	2	2
2	2	2	2	2	2	2
2	2	1	1	1	2	2
2	2	1	c	1	2	2
2	2	1	1	1	2	2
2	2	2	2	2	2	2
2	2	2	2	2	2	2

Figure 2: Idea of CELL (applicable only to point data)

is discarded right away. In other words, the objects that remain in memory are those (i) in partitions whose densities are less than k, or (ii) in no partition at all. SNIF keeps scanning the verification file, until the file has been exhausted, or $m - 1$ pages of memory have been occupied by objects. In either case, we scan R for a second time to precisely decide whether these memory-resident objects are outliers. If the verification file has not been completely scanned (which is rare; the verification file is empty in almost all our experiments), we discard all the objects in memory, resume the scan, and repeat the above process.

5. RELATED WORK

Except BNL, no existing solution can solve our problem and satisfy the generality requirements in Section 1.1. However, for the restricted scenario where objects are multi-dimensional points with mutual distance measured by the L_2 norm, Knorr and Ng [8] develop an alternative method CELL[7]. Since we will compare SNIF with CELL for Euclidean datasets in the experiments, next we explain the rationale of CELL.

Given a distance threshold r, CELL partitions the data space regularly into a grid, where each cell is a multi-dimensional square whose diagonal has length $r/2$. Then, CELL hashes the objects into cells, by reading and writing the dataset R once, respectively. Meanwhile, each cell is associated with a *counter*, equal to the number of points it covers.

Based on the grid and the counters, it is possible to quickly determine some outliers and non-outliers. To illustrate this in 2D space, consider cell c in Figure 2, which shows part of a grid. The cells labeled with '1' constitute the *level-1* cells with respect to c, and those labeled with '2' the *level-2* (note that the level-2 is "thicker" than level-1). CELL obtains two numbers n_1 and n_2, where n_1 is the total number of points in c and its level-1 cells, and n_2 is the the number of points in c, its level-1, and -2 cells. It is not hard to observe that (i) if $n_1 \geq k$, all points in c must be non-outliers, and (ii) if $n_2 < k$, all points in c must be outliers. In either case, c is marked as "colored"; otherwise, c remains "white", indicating that the identities of the points covered by c are currently unknown.

For each white cell c, CELL loads the points in it (from its hash bucket) into memory, and verify whether they are outliers, by scanning the data in its level-1 and -2 cells. Obviously, as long as there is available memory, multiple white cells can be processed together to improve I/O efficiency. Specially, if the points in all the white cells fit in memory (as is an assumption in [8]), CELL terminates by scanning R another time in the worst case.

The problems of CELL are two-fold. During hashing, at least one memory page must be allocated to each cell as an input buffer, so that a page of objects can be written to the hash bucket at a time. This seriously limits the range of r that can be supported. For example, for dataset *CA* (as shown in Figure 1a), a meaningful r can be as small as 2.6% of a dimension. In this case, each cell in the grid has a side length of $2.6\%/(2\sqrt{2})$ = 0.92%, i.e., the grid contains more than 11830 cells! Assuming a disk size of 1k bytes, *CA* (with 62k two-dimensional points) occupies around 720 pages[8]. Hence, CELL requires a memory size 16.4 times that of the dataset!

Unlike BNL and SNIF, it incurs a significantly larger number of random accesses, due to its reliance on hashing. Specifically, every time a buffer is flushed, the disk head is forced to move from (and then back to) its original position in reading the dataset file, necessitating at least two random accesses. This problem is particularly serious, if each cell's input buffer has a single memory page (which, unfortunately, is usually true, as the number of cells is large). In this case, every I/O writing and most I/O reading are random.

CELL can be extended to higher dimensionalities, however, at the cost of severely aggravating the above defects. The reason is that, with the same r, the number of cells increases exponentially with the dimensionality l (each cell has a side length of $r/(2\sqrt{d})$). Furthermore, while the level-1 of a cell c still includes those cells adjacent to c, the level-2 becomes an l-dimensional "rectangular ring" with a thickness of $\lceil 2\sqrt{l} - 1 \rceil$ (e.g., if l = 9, the ring has 5 cells on either side of c along each dimension). As a result, as l grows, each white cell must be inspected against a higher number of hash buckets. Finally, CELL is clearly inapplicable to non-Euclidean domains, where grid partitioning is simply undefined.

Although our algorithm SNIF is also based on "partitions", it significantly outperforms CELL both in applicability (SNIF can be applied as long as the distance function is a metric), and efficiency (SNIF is faster than CELL even in Euclidean space). This is achieved by leveraging several problem characteristics based on random sampling (see Section 3.1), and integrating these characteristics with prioritized flushing. In particular, unlike CELL, SNIF prunes a majority (more than 99%) of the objects directly after the first database scan, and for some datasets, even terminates right after the first scan without missing any outlier.

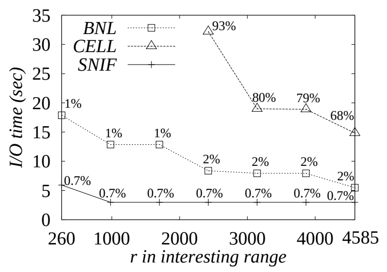
It is worth mentioning that other definitions of outliers have also been proposed in the literature. The earliest definition appears in statistics, where data values are known to obey a probability model, and a value is captured as an outlier if it should have occurred only with a very low probability [2] (calculated from the model). Johnson et al. [7] identify outliers as points on the convex hull (or, in general, the "out-most" layers of convex hulls). Ramaswamy et al. [14] present a definition based on the distance between an object and its k-th nearest object in the dataset. Breunig et al. [4] propose the concept of "local outliers", according to which an object is an outlier if it demonstrates behavior significantly different from the behavior of its nearby objects. This concept is extended by Jin et al. [6] to "top-k local outliers". Aggarwal and Yu [1] analyze Euclidean outliers in high-dimensional space, whereas Lazarevic and Kumar [11] approach this issue with a technique called "feature bagging". Finally, Papadimitriou et al. [13] discuss outliers based on "local correlation integrals".

6. EXPERIMENTS

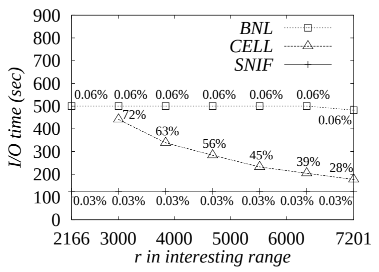
The organization of this section is as follows. First, we compare the proposed algorithm SNIF against BNL and CELL, deploying the real datasets *CA*, *Household*, and *Server* described in Section 3.2 (they contain Euclidean points in a data space where all dimensions have a domain of [0, 10000]). Then, we examine the scalability of SNIF with respect to the dataset cardinality, using

[7]Knorr and Ng [8] also propose an index-based algorithm, which, however, is substantially slower than CELL, and hence, is omitted from our discussion.

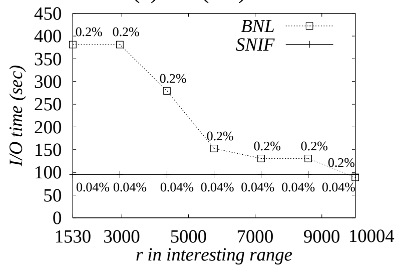
[8]For each point, 3 values must be stored, i.e., id and x-, y- coordinates.



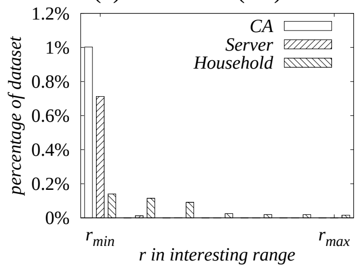
(a) *CA* (2D)



(b) *Household* (3D)



(c) *Server* (5D)



(d) Number of objects verified by SNIF in the 2nd scan

Figure 3: Performance vs. r (k = 0.05% of cardinality, memory size = 10% of database)

synthetic non-Euclidean data (whose generation will be clarified later). In the above experiments, the value of s (the number of centroids) for SNIF is fixed to 1000, because the behavior of our technique is not sensitive to s, as demonstrated in the last experiment.

The default memory size equals 10% of the space occupied by the underlying database. The default k is 0.05% of the dataset cardinality n. As discussed in Section 3.1, for any k, there exists an "interesting range" of r such that, as r distributes in the range, the number of outliers is between 1 and $0.1\% \cdot n$. We define the *median* of r, also the default of r, as the value of this parameter when the number of outliers equals exactly $0.05\% \cdot n$ (as with the interesting range, the median of r also depends on k). For $k = 0.05\% \cdot n$, the default values of r are 375, 4200, 1688 for *CA*, *Household*, and *Server*, respectively.

We measure the performance of each method by its I/O cost, including the time of both random and sequential accesses. The disk page size equals 1024 bytes.

Performance vs. r. The first experiment inspects the efficiency of SNIF, BNL, and CELL with respect to the distance threshold r. For this purpose, we set k and the memory size to their default values, and measure the cost of all algorithms at 7 values of r that evenly partition the interesting range of each dataset.

Figures 3a-3c illustrate the cost as a function of r. There is no result of CELL for some experiments on *CA*, *Household*, and all experiments on *Server* because, in these experiments, the memory requirement of CELL exceeds 10% of the database (e.g., for *CA* and r = 260, CELL requires memory 16 times larger than the database, as explained in Section 5). Each percentage along the curves indicates the percentage of random-access cost in the overall overhead. We will use the same style to illustrate the cost fraction of random I/Os in the following diagrams.

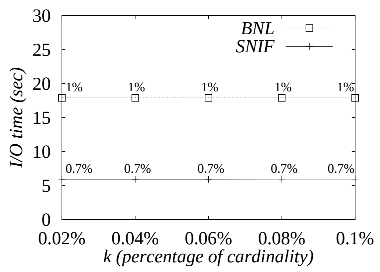
SNIF outperforms its competitors significantly, especially when r is small (i.e., more outliers are retrieved). Our technique terminates by scanning the dataset at most twice. In particular, for *CA*, when r is different from the lower end of its interesting range, SNIF returns all outliers by performing a single scan, as indicated by its cost decrease in Figure 3a. Furthermore, as analyzed in Section 5, most I/O accesses by CELL are random, whereas SNIF and BNL perform (almost) only sequential I/Os.

The verification file produced by SNIF after the first scan is empty in the above experiments, i.e., all the objects removed from memory during prioritized flushing have neighbor counters $\geq k$ (since this is true for most of the subsequent experiments, we will explicitly discuss the size of the verification file, only if it is not zero). Equivalently, the objects that are verified in the second scan are retained in memory at the end of the first scan. Figure 3d demonstrates the number of such objects (in percentages of the corresponding dataset cardinality) in the experiments of Figures 3a-3c. Observe that, in all cases, *SNIF prunes at least 99% of a dataset after the first scan.*

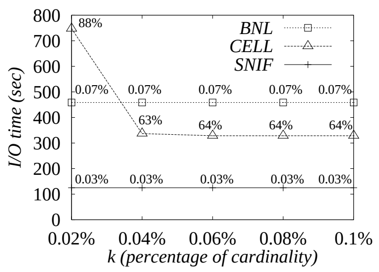
Performance vs. k. Figures 4a-4c evaluate the efficiency of alternative solutions when k distributes from 0.01% to 0.1% of the cardinality, using default values for r and the memory size. CELL is applicable only to *Household*, again due to its excessively large memory consumption for the other datasets. The performance of all methods remains stable for the entire range of k tested. This phenomenon implies that each algorithm incurs similar cost as long as the number of fetched outliers is the same (remember that, for a median r, the number of outliers is always 0.05% of the cardinality). Similar to Figure 3d, we present in Figure 4d the number of objects verified by SNIF in the second scan, confirming the observation that the number is less than 1% of the dataset cardinality.

Performance vs. memory size. Next, we study the impact of memory size on the efficiency of outlier mining. Towards this, we use the default values for both k and r, but measure the performance of all algorithms, as the amount of memory changes from 1% to 20% of the database. The results are illustrated in Figure 5. CELL is inapplicable to *CA* and *Server*, and applicable to *Household* only if the memory accounts for at least 5% of the database. There is no result of *CA* at memory size 1%, because in this case the memory contains less than 10k bytes, which is unrealistically small.

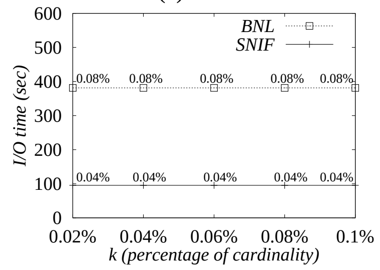
It is clear that SNIF is by far the best method, if memory is scarce. Particularly, for *Household* and *Server*, SNIF is faster than BNL by a factor over an order of magnitude at memory size 1%. The cost of our method is the same regardless of the memory capacity, with one exception: *Server* and 1% memory. In this case,



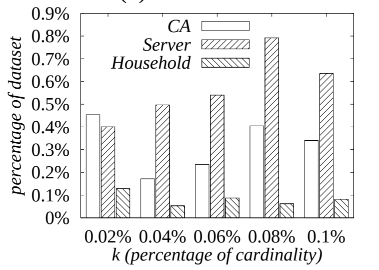
(a) *CA*



(b) *Household*

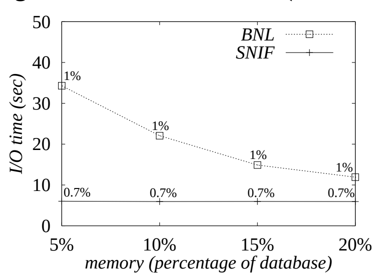


(c) *Server*

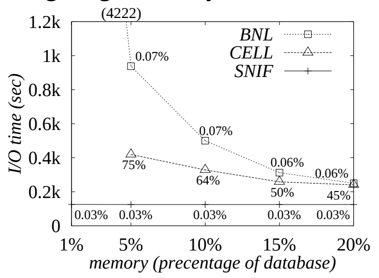


(d) Number of objects verified by SNIF in the 2nd scan

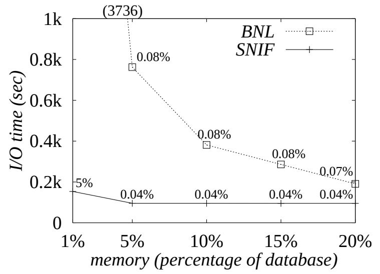
Figure 4: Performance vs. k (r = median of interesting range, memory size = 10% of database)



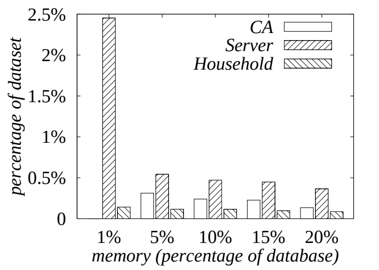
(a) *CA*



(b) *Household*



(c) *Server*



(d) Number of objects verified by SNIF in the 2nd scan

Figure 5: Performance vs. memory size (r = median of interesting range, k = 0.05% of cardinality)

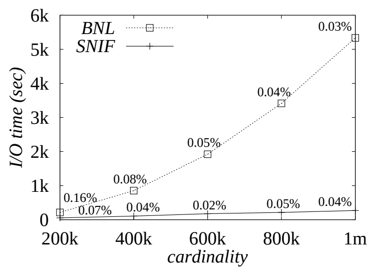
the verification file is not empty, but contains 16k objects, i.e., 3.2% of the database. As a result, SNIF incurs additional cost for scanning the file, and accordingly, the cost percentage of random I/Os increases, because random accesses must be performed whenever objects are appended to the verification file.

Performance vs. cardinality. To test the scalability of our solution with respect to the dataset size, we create several synthetic non-Euclidean *Signature* datasets with various cardinalities. Each object in *Signature* is a string containing 30 English letters. First, 50 "pivot" strings are randomly generated. Each pivot defines a cluster, in which an object is obtained by modifying a number x of letters in the pivot, where x uniformly distributes in [1, 10]. We continuously generate objects this way (randomly picking a pivot for each object), until the number of objects reaches 99.95% of the target cardinality n. Finally, the remaining $0.05\% \cdot n$ objects are again randomly generated, i.e., they are outliers. The distance metric for *Signature* datasets is the edit distance.

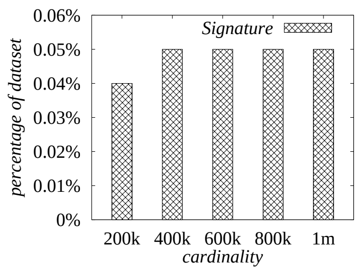
We vary the cardinality from 200k to 1 million, but fix the amount of available memory to 10% of the database with the *smallest* cardinality 200k. Setting r and k to their default values, Figure 6a demonstrates the cost of SNIF and BNL as the cardinality grows (CELL cannot be applied to non-Euclidean data). As expected, the I/O-time of BNL demonstrates clear quadratic behavior (confirming our analysis in Section 2), whereas that of SNIF increases linearly (always terminating after 2 scans). At the highest cardinality, SNIF again outperforms BNL by more than an order of magnitude. Figure 6b shows the number of objects verified by SNIF in the second scan.

SNIF sensitivity to s. Finally, we examine the performance of SNIF when the number s of centroids changes. In this experiment, we set r, k, and the memory size to their default values, but vary s from 1000 to 3000. Figure 6a plots the cost of SNIF for the real datasets, and a synthetic dataset *Signature* with cardinality 500k.

The behavior of our algorithm is not affected as s changes: SNIF

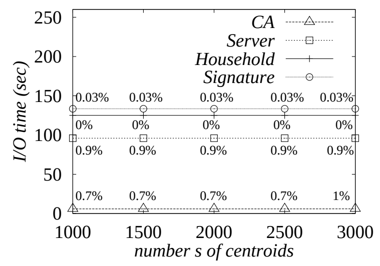


(a) I/O cost

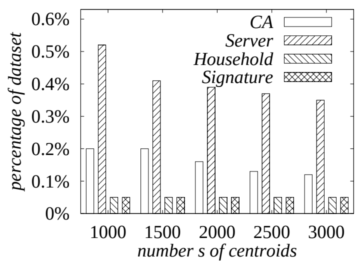


(b) Number of objects verified by SNIF in the 2nd scan

Figure 6: Performance vs. cardinality (non-Euclidean dataset; r, k at default values, memory size 10% of the 200k-dataset)



(a) I/O cost



(b) Number of objects verified by SNIF in the 2nd scan

Figure 7: SNIF behavior as s varies (r, k, memory size at default values)

always scans a dataset twice. To explain this, recall that the number of objects that need to be verified after the first scan is given by Formula 4, and a third scan is necessary only if these objects do not fit in memory. After s has reached a reasonably large value (independent of the dataset cardinality), the second and third terms of Formula 4 are already very low, such that further increasing s leads to only marginal decrease of those terms. Figure 6b verifies this, by showing that, for all datasets, the number of objects for verification in the second scan decreases only slightly (by less than 0.15% of the cardinality) as s grows from 1000 to 3000. The phenomenon implies that selection of s is simple in practice — we recommend $s = 1000$.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we developed a novel algorithm SNIF, which reports all outliers by scanning the dataset at most twice. Our solution is general, and can be applied as long as the distance function satisfies the triangle inequality, regardless of the function itself (it can be Euclidean distance, edit distance, etc.) and the types of data (e.g., points, strings, and so on). SNIF has solid theoretical justifications, and can be easily implemented in a commercial DBMS.

This work also indicates several promising directions for future investigation. The first one concerns "probabilistic outliers", where the goal is to identify objects that may be outliers with at least a certain probability. Compared to "exact outliers" (as retrieved by SNIF), deriving probabilistic results may be achieved with lower cost (e.g., we may never have to scan the database twice). Another exciting direction is to address outlier detection on streams [5], where the objects (credit card transactions) are received by the system at a fast rate, and the objective is to catch outliers continuously. In this scenario, the algorithm must operate in strict time bounds, in order to avoid jamming the subsequent data traffic.

ACKNOWLEDGEMENTS. This work was accomplished when Yufei Tao and Xiaokui Xiao were with the City University of Hong Kong, and were supported by CERG Grant CityU 1163/04E from the Research Grant Council of the HKSAR government. Shuigeng Zhou was supported by grants numbered 60373019 and 90612007 from the National Natural Science Foundation of China, and the Shuguang Scholar Program of Shanghai Municipal Education Committee. We would like to thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] C. Aggarwal and S. Yu. An effective and efficient algorithm for high-dimensional outlier detection. *The VLDB Journal*, 14(2):211–221, 2005.

[2] V. Barnett and T. Lewis. *Outliers in Statistical Data, 3rd Edition*. John Wiley, 1994.

[3] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *SIGKDD*, pages 29–38, 2003.

[4] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. In *SIGMOD*, pages 93–104, 2000.

[5] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. pages 359–366, 2000.

[6] W. Jin, A. K. H. Tung, and J. Han. Mining top-n local outliers in large databases. In *SIGKDD*, pages 293–298, 2001.

[7] T. Johnson, I. Kwok, and R. T. Ng. Fast computation of 2-dimensional depth contours. In *SIGKDD*, pages 224–228, 1998.

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

[9] E. M. Knorr and R. T. Ng. Finding intensional knowledge of distance-based outliers. In *VLDB*, pages 211–222, 1999.

[10] E. M. Knorr, R. T. Ng, and V. Tucakov. Distance-based outliers: algorithms and applications. *The VLDB Journal*, 8(3-4):237–253, 2000.

[11] A. Lazarevic and V. Kumar. Feature bagging for outlier detection. In *SIGKDD*, pages 157–166, 2005.

[12] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB*, pages 160–169, 1986.

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

[14] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *SIGMOD*, pages 427–438, 2000.