Hierarchically Clustered LSH for Hierarchical Outliers Detection

Konstantinos Georgoulas and Yannis Kotidis

Athens University of Economics and Business, Department of Informatics
Patission 76, 10434 Athens, Greece
{kgeorgou,kotidis}@aueb.gr

Abstract. In this work we introduce hierarchical outliers that extend the notion of distance-based outliers for handling hierarchical data domains. We present a novel framework that permits us to detect hierarchical outliers in a consistent manner, providing a desired monotonicity property, which implies that a data observation that finds enough support so as to be disregarded as an outlier at a level of the hierarchy, will not be labelled as an outlier when examined at a more coarse-grained level above. This way, we enable users to grade how suspicious a data observation is, depending on the number of hierarchical levels for which the observation is found to be an outlier. Our technique utilizes an innovative locality sensitive hashing indexing scheme, where data points sharing the same hash value are being clustered. The computed centroids are maintained by our framework's scheme index while detailed data descriptors are discarded. This results in reduced storage space needs, execution time and number of distance evaluations compared to utilizing a straightforward LSH index.

1 Introduction

An outlier is an observation that differs so much from others so as to arouse suspicion that it was generated by a different process than the rest of the data. In order to put this intuition into a context where outliers can be formally defined and computed many alternative definitions have been proposed. One of the most commonly used approach is the distance-based outlier definition, which suggests that given a dataset P, a positive integer N and a positive real number r, a data object p of P is a $O(N, r)$-outlier, if less than N objects in P lie within distance r from p, for some appropriate distance metric.

Outliers detection is critical for many modern applications such as decision support (OLAP), customer behavior analysis and network management. However, none of the well known outlier detection techniques takes into consideration the hierarchical nature of the data domains that is inherent in such applications. The natural aggregation of atomic values along a domain hierarchy is a critical summarization technique that can be used to detect different *grades* of abnormal behavior by looking at all levels of the hierarchy.

As an example, we consider the case of an electronic store. There are several ways to categorize products (*ProductId, Group, Class* categories) that a customer purchases. Table 1 presents an example of customers and the products they purchased. Distance-based computations of outliers in this example can be performed by mapping each

customer into a point in a high-dimensional domain (e.g. dimensions being the productIds). The values of the coordinates on each dimension (i.e. productId) can be boolean values (indicating whether the user has purchased the product), or may be derived from different statistics (e.g. number of times the customer purchased a product, her rating, etc).

Independently of the details of this mapping, if we compare customers based on the productIds of the products they purchased, then John and Mary show no apparent similarity. However, if we look at the *Group* category of the products, it is obvious that they both purchased Smart Phones. Similarly, John and Jim look dissimilar until they are observed at the upper level of the product's hierarchy (*Class* category). Consequently, distance-based outliers derived by looking at the data domain that corresponds to the leaves of the product's domain hierarchy (Product→Class→Group→ProductID) may find support when these observations are aggregated further up the hierarchy.

User	ProductId	Group	Class
John	Samsung Galaxy S4	Smart Phones	Computers
John	Apple iPhone 6	Smart Phones	Computers
Tim	Nikon Camera D750	Cameras	Tvs-Cameras
Jim	Apple iPad Air 2	Tablets	Computers
Mary	LG Nexus 5	Smart Phones	Computers

Table 1. Product Purchases

Given that domain hierarchies are commonly used in many applications, in this work we first look at the problem of deriving an intuitive definition that extends the notion of distance-based outliers over hierarchical domains. A straightforward independent computation of distance-based outliers over all hierarchical levels may yield inconsistent results that complicate data analysis. As an example, depending on the selected threshold values N and r, an observation that is not an outlier at the leaves of the hierarchy may be deemed as such at an intermediate level. This goes against intuition, which suggests that as atomic values are being aggregated via the hierarchy, data observations tend to look similar.

In this work, we introduce the notion of hierarchical outliers for handling hierarchically organized data domains. Our proposed definition computes outliers in a consistent manner, which implies that a data observation that finds enough support so as to be disregarded as an outlier at a level of the hierarchy, can not be labelled as an outlier when examined at a more coarse-grained level above. This intended monotonicity property not only leads to conclusions that are not surprising to the user analyst but also enable us to grade how suspicious a data observation is, depending on the number of hierarchy levels for which the observation is found to be an outlier.

In addition to providing a proper definition of hierarchical outliers, in this work we also look at efficient techniques that enable us to compute such outliers in large datasets. Locality sensitive hashing (LSH) is a popular technique that partitions a high-dimensional dataset into buckets so as to avoid performing all-pair computation of item distances. Direct application of LSH for hierarchical outliers identification is pro-

hibitively expensive as independent indexes need to be constructed for each level of the hierarchy, leading to increased computational and storage overhead.

Thus, we propose an innovative LSH index scheme, termed as hierarchically clustered LSH (cLSH), which instead of storing the data items at an independent index for every level of the hierarchy, it only maintains the centroids of clusters, which are constructed performing a clustering technique among the data items that share a common hash value at every index. As a result, both the computational and storage overhead for the cLSH is reduced compared to the original LSH structure.

The contributions of our work are:

- We introduce the notion of hierarchical outliers and provide an intuitive framework for detecting hierarchical outliers over hierarchically organized data domains. Our framework assigns a simple and intuitive statistic called *grade* for every data item identified as hierarchical outlier, which is a positive integer referring to the number of levels for which the specific item is outlier. The higher the *grade*, the more erroneous the item is.
- We propose an innovative indexing scheme based on locality sensitive hashing. This scheme maintains centroids of data clusters at LSH indexes of hierarchical levels, making it less space demanding compared to the case of independently created original LSH indexes at every level.
- We introduce a bottom up computation via the hierarchy of data domain in order to detect hierarchical outliers. At each level, our method utilizes results from previously performed computations resulting in faster computation of outliers.
- We present an experimental evaluation for our framework measuring the accuracy and the efficiency (in terms of space and time) of our proposed techniques.

2 Related Work

Many previous works in different areas of data management have studied the problem of outlier detection. Different approaches for the definition of an outlier have been presented in case of multidimensional data. In [2, 9, 15] distance based outliers are discussed, while [3, 13] consider density-based outliers as well. In the first case, data items are considered as outliers based on the distances from their neighbors. In the second case data items are studied by computing the density of data around their local neighbors. The relative density of a data item compared to its neighbors is computed as an outlier score. Different approaches have discussed different variants for computing this score [8, 16]. A different outlier definition presented in [10] suggests that angles between data vectors are more stable than distances in high dimensional spaces. In this case items are compared using angle-based similarity metrics, like the cosine similarity metric. A data item is not considered as outlier if most of the objects are located in similar directions with it.

Many of the aforementioned solutions exploit well known indexing techniques (like the R-tree and its variants) in order to perform range or NN queries that are necessary for outlier detection. Conventional multidimensional indexes are inapplicable in large

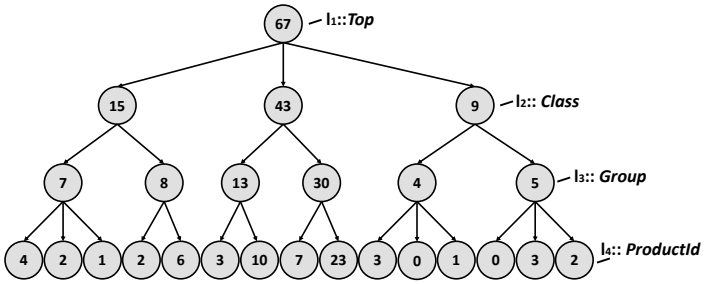

Fig. 1. Product Domain Hierarchy

data domains. For instance, the cardinality of the product dimension in a data warehouse can be in the order of tens of thousands. Instead, our technique utilizes a probabilistic indexing method termed LSH [4] that approximates the results of NN and range queries in high dimensional spaces and extends it in order to handle efficiently the hierarchical structure that data follows. The LSH scheme that we present in this work can be extended to support different distance metrics, including the cosine similarity for angle-based computation of outliers [6].

3 Motivational Example

Suppose that we would like to detect hierarchical outliers in the data warehouse of an electronic store. A set of data items could be derived by projecting every customer's purchases at the hierarchically organized *Product* domain space. Figure 1 shows a data item representing a customer's purchases. The hierarchy of the *Product* domain consists of four levels. The lowest level l_4 contains all *ProductIds*, which are used for the unique identification of the products. At level l_3, the *Group* category of the products is represented (i.e. *Home Theatre, TVs, Cameras, Smart Phones, Tablets, Laptops*). Level l_2 depicts the *Class* category of products (i.e. *Audio, Tvs & Cameras, Computers*), while l_1 contains the *Top* level representing all products. In this example, without loss of generality, at the lowest level of the hierarchy the values represent cumulative purchases of different productIds for this customer. Aggregated values at upper levels are obtained by the utilization of *sum* function (e.g. as in a typical roll-up aggregation).

Given this setting of data, we may focus on detecting outliers at every level of this hierarchy. We could try to identify outliers based on the productIds that the customer bought, or according to her aggregated purchases over product *Groups*. Someone else may take into consideration customers purchases at more abstracted summarization levels provided by the *Class* or *Top* category. We suggest a holistic approach that considers all abstraction levels of product purchases, based on the specified hierarchy. We introduce the notion of *hierarchical outlier* that takes into consideration the whole hierarchical tree that represents her complete purchasing history, supporting a more intuitive decision whether a customer is an outlier, or not.

Moreover, our framework provides an succinct measure termed *hierarchical outlier grade* that denotes the number of levels a customer is identified as an outlier. For instance, if a customer is regarded as hierarchical outlier with *grade*=2, this would suggest that her purchases based on the productIds and their Groups are significantly different from other people on the dataset. On the other hand, this result implies that her purchases when aggregated at the *Class* level are similar to many other customers.

4 Hierarchical Outliers

The hierarchical nature of the data domain motivates us to examine data at every level of the hierarchy they follow, in order to be identified as outliers. In our motivational example, if we check all customers at level l_4 and identify a specific customer as an outlier, we have no evidence to regard her as an outlier at upper levels too. It is likely that only few customers purchase the same exactly products as she does (in terms of productIds), while there are many who purchase similar quantities of products at the *Group* level of products categorization.

This observation leads us to propose a framework for the outlier detection problem that takes into consideration the hierarchical structure of the data domain. An obvious solution would be to compute distance-based outliers at all different abstracted levels in a completely separate way. Given the fact that data items are high dimensional, someone could construct an index for every hierarchical level, in order to retrieve the nearest neighbors of the queried item at every level and then according to the distance-based outlier definition she could decide whether it is outlier or not. However, it is quite possible a specific item in question to be identified as an outlier at some levels and not to be considered as an outlier at some others lower or higher to previous ones, depending on the selected distance thresholds.

This lack of coherence stems from the main drawback of an independent evaluation of distance-based outliers: it handles the different abstraction levels of a data item as independent observations, rather than different abstractions of the same data item, obtained through the hierarchy. By manipulating data in this manner, there could be no *consistent* results in order to characterize a customer's behavior in total.

In order to alleviate this inconsistency of results for hierarchically organized data we introduce the notion of *hierarchical outlier* $HO(N, r)$.

Definition 1 (hierarchical outlier $HO(N, r)$**).** *Given a dataset P over a hierarchically organized data domain with h hierarchical levels, a positive integer N (threshold) and a positive real number r, a data item $p \in P$ is a $HO(N, r)$-Hierarchical Outlier with grade L, if there are L levels of data hierarchy, at which less than N objects in P lie within distance r_i from p, where $1 \leq i < h$. $r_i = \sqrt{(2 * max(F^i) - 1)} * r_{i+1}$, $r_h = r$ and $max(F^i)$ denotes the maximum fanout of those hierarchical tree's nodes belong to hierarchical level l_i.*

Intuitively, the definition utilizes a certain method for computing the distance thresholds at the different levels. As will be explained in what follows this ensures that outliers' grade can be computed in a consistent manner following the desired monotonicity property.

We first present some preliminaries that we utilize to better describe our hierarchical outlier definition.

Lemma 1. *Given a data item* $X = \{x_1, x_2, x_3, \ldots, x_{d_{i+1}}\}$ *and a query point* $q = \{q_1, q_2, q_3, \ldots, q_{d_{i+1}}\}$ *in a domain organized by a hierarchy* H*, it holds that* $D_i(q, X) \leq \sqrt{(2 * max(F^i) - 1)} * D_{i+1}(q, X)$ *where* $D_{i+1}(q, X), D_i(q, X)$ *are the Euclidean distance between* q *and* X *at hierarchical levels* l_{i+1} *and* l_i*, respectively, where* $1 \leq i < h$ *and* $max(F^i)$ *is the maximum fanout of those hierarchical tree's nodes belonging to hierarchical level* l_i*.*

Lemma 1 ensures the consistency of the results that the proposed hierarchical outlier detection process provides. Based on this property, if a data item q has N items that lie within distance r_{i+1} from it at level l_{i+1}, then it will also have at least the same N items in distance $r_i = \sqrt{(2 * max(F^i) - 1)} * r_{i+1}$ at higher level l_i. By utilizing the popular distance-based outlier definition, we disregard q as an outlier at a level l_{i+1} and furthermore we also do not consider it as outlier at any upper level l_i with the condition of defining distance thresholds r_i based on Lemma 1. In Figure 2, we graphically depict how the distance threshold r_4 at the lowest level l_4 is "expanded" at the upper levels of hierarchy H. When thresholds are increased in a manner consistent to Lemma 1, the computation of distance-based outliers provides the desired consistency.

Proof. Here, we prove that $D_i(q, X) \leq \sqrt{(2 * max(F^i) - 1)} * D_{i+1}(q, X)$. For every level l_i, where $1 \leq i < h$ we know that

$$D_i^2(q, X) = \sum_{j=1}^{d_i} \Big((q_{k_j+1} - x_{k_j+1}) + \cdots + (q_{k_j+f_j} - x_{k_j+f_j}) \Big)^2 = \sum_{j=1}^{d_i} Value(j).$$

where f_j is the fanout of j-th node at level l_i of hierarchical tree. d_i is dimensionality of level l_i and $k_j = \sum_{w=0}^{j-1} f_w$.

$$Value(j) = (q_{k_j+1} - x_{k_j+1})^2 + \cdots + (q_{k_j+f_j} - x_{k_j+f_j})^2 + 2 * \sum_{w=k_j+1}^{k_j+f_j-1} \sum_{y=w+1}^{k_j+f_j} (q_w - x_w)(q_y - x_y)$$

and thus,

$$D_i^2(q, X) = D_{i+1}^2(q, X) + \sum_{j=1}^{d_i} extra(j) \tag{1}$$

where $extra(j) = 2 \sum_{w=k_j+1}^{k_j+f_j-1} \sum_{y=w+1}^{k_j+f_j} (q_w - x_w)(q_y - x_y)$.

Bounding the $\sum_{j=1}^{d_i} extra(j)$ of Equation 1, we are able to express distance $D_i(q, X)$ as a factor of $D_{i+1}(q, X)$. It is

$$\begin{aligned}\sum_{j=1}^{d_i} extra(j) &\leq 2 * \sum_{j=1}^{d_i} \sum_{w=k_j+1}^{k_j+f_j-1} \sum_{y=w+1}^{k_j+f_j} |q_w - x_w|\,|q_y - x_y| \\ &\leq 2 * (max(F^i) - 1) \sum_{j=1}^{d_i} \sum_{w=k_j+1}^{k_j+f_j} (q_w - x_w)^2 \\ &\leq 2 * (max(F^i) - 1) \sum_{j=1}^{d_{i+1}} (q_j - x_j)^2 \\ &\leq 2 * (max(F^i) - 1) * D_{i+1}^2(q, X)\end{aligned}$$

and thus we prove that: $D_i^2(q, X) \leq (2max(F^i) - 1) * D_{i+1}^2(q, X)$

Although our techniques are tailored for the popular Euclidean metric, they can be adapted appropriately for different distance metrics and aggregation functions applied to the data domain's hierarchy.

In the following sections, we present in detail our adopted LSH indexing structure that is tailored to identify hierarchical outliers, as well as our algorithm for their efficient detection based on the proposed index.

5 Hierarchically Clustered LSH Indexing

Given that we need to compare high-dimensional data when looking for hierarchical outliers, we adapt a powerful dimensionality reduction technique called LSH [1]. LSH generates an indexing structure by evaluating multiple hashing functions over each data item. Using the LSH index, we can identify the nearest neighbors of each customer and compute outliers based on the distances from her neighbors.

We utilize hash functions that are based on 2-stable distributions and create several different hash tables in order to increase the effectiveness of the LSH indexing schema. There have been many proposals on how to tune and increase performance of LSH (e.g. [11, 6]), however such techniques are orthogonal to the work we present here.

A direct approach for indexing a data set over a hierarchical domain would be the construction of independent LSH hash schemes (one per hierarchical level). Each hash scheme would contain T hash tables, named $HT_1^1 \ldots HT_1^T$, $HT_2^1 \ldots HT_2^T$, $\ldots$, $HT_h^1 \ldots HT_h^T$, which would maintain the data items of levels $l_1, l_2, \ldots, l_h$, respectively. As we have already denoted, storing the whole dataset at independent LSH indexes for every hierarchical level is not an efficient way of indexing. Instead, we introduce a more space-saving index by creating T_i hash tables at every level l_i, named $HCT_1^1 \ldots HCT_1^{T_1}$, $HCT_2^1 \ldots HCT_2^{T_2}$, $HCT_h^1 \ldots HCT_h^{T_h}$. Each HCT_i is a hierarchically clustered hash table and contains a small number of centroids, which are computed by clustering the data items that falls in the bucket with the same id for the HCT_{i+1} hash table of the immediate lower level l_{i+1}.

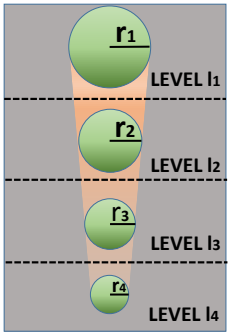

Fig. 2. Bounding r for different hierarchical levels.

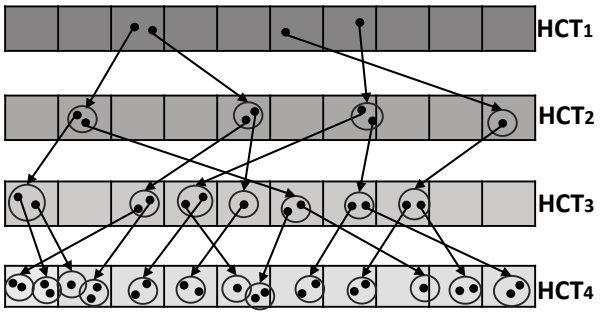

Fig. 3. Hierarchical Clustering of our LSH Scheme.

At most k centroids (to be stored in HCT_i) are computed by clustering the data items belonging to the same bucket of HCT_{i+1}. k is a user-defined parameter, which affects the space cost of our proposed LSH scheme. Its value could vary from one to the exact number of items stored every time at a bucket. The more centroids per bucket we maintain, the higher the space requirements of the index would be. A more flexible option that we apply in our framework, is the derivation of different values for k at every bucket so as a target space reduction ratio rr is achieved for the whole index.

Thus, we compute k_{B^j} (where $k_{B^j} = \frac{|B^j|}{rr}$) centroids for every B^j bucket of HCT_{i+1}. We compute the hash values for the computed centroids and store them in the appropriate bucket of HCT_i. We have to notice that, both the number T_i of *HCT* tables that are created at every level l_i and the hash functions utilized to hash the centroids, are selected following the same parametrization process [14] as in the case of building independent hash schemes for every level. Similarly, for hash table HCT_{i-1}, we compute the centroids after the clustering of the centroids maintained to each bucket of HCT_i. As a result, HCT_{i-1} maintains centroids of the clusters constructed over the centroids stored at each bucket of HCT_i. Following the same procedure, we create the *HCT* tables for all the remaining levels up to l_1, in a bottom-up process. The higher the hierarchical level, the fewer centroids need to be indexed to its corresponding hash table *HCT*. The aforementioned procedure can also be performed at the lowest level l_h. In this case a primary LSH scheme for level l_h is constructed. The clusters and their centroids for every bucket are computed and stored to newly created *HCT* hash tables, while the primary LSH scheme is not required any more and is, thus, discarded.

In more detail, our LSH indexing structure construction requires the following steps:

- We initially construct a temporary LSH scheme for indexing the real data items of level l_h. These hash tables are auxiliary (i.e. used for the construction of the *HCT* hash tables at level l_h) and they are discarded immediately after the next step of the process is completed.
- We compute $k_{B^j} = \frac{|B^j|}{rr}$ centroids for every bucket B^j of the hash table HT_h^1. In our framework, we utilize k-means for clustering, however this choice is orthogonal to our scheme. This set of centroids are hashed to a set of T_h hash tables using hash

functions $g_h^1 \dots g_h^{T_h}$, where g_h^i for $1 \leq i \leq T_h$ is a family of a $2-$stable distribution functions [5].

– We repeat the previous step for every level l_i, with $1 \leq i \leq h-1$. Each time, we perform a $k-$means clustering at the centroids stored to the buckets of the HCT_{i+1}^1 hash table. These centroids are abstracted to the upper hierarchical level l_i, forming a much smaller dataset (in terms of cardinality) than the real dataset, for the level l_i. Based on the centroids' hash values, they are stored at the corresponding buckets of $HCT_i^1 \dots HCT_i^{T_i}$ hash tables.

In Figure 3, we show an instance of our indexing structure for the case of our running example's hierarchy. For ease of presentation, we create only one table, instead of T_i, at every hierarchical level l_i. The hierarchy consists of 4 levels and, thus, we create four tables $HCT_4, HCT_3, HCT_2, HCT_1$, one per level. Every arrow links a centroid, that is maintained at HCT_{i-1}, with the cluster of a HCT_i hash table at level l_i which members it represents. There can be one or multiple clusters in the same bucket, for example the two clusters at the first bucket of HCT_4. At hash table HCT_3, we can see thirteen centroids derived from HCT_4's data. As we mentioned previously, these centroids are assigned to buckets of HCT_3, based on their hash value during index's construction. Only these thirteen centroids are maintained in the hierarchically clustered hash table HCT_3 of level l_3. Similarly, at HCT_2 seven centroids are constructed based on the hash values of the centroids of the seven clusters created at HCT_3. Finally at the top HCT_1 hash table, we notice only four entries.

It is clear that the number of data items/centroids stored at each level are quite fewer than the data maintained in the case of storing the whole dataset at every hash table of every hierarchical level. Consequently, our index requires significantly smaller space compared to the original LSH scheme. It maintains hash tables, consisted only of a small number of centroids performing a per-bucket clustering of the data items. These centroids, as we will explain in Section 6, are utilized in order to compute the support score to a query point during the hierarchical outlier detection process. This evaluation leads to reduced number of distance computations when querying the index resulting to even faster outlier identification compared to the baseline approach. In our experimental analysis, we depict several figures proving our aforementioned claims. Encapsulated information in centroids, such as the number of data items that the cluster contains and the cluster range (i.e. distance of the centroid to its furthest cluster member), is a key factor for the reduction of the computation cost of hierarchical outlier identification, as we show in the next section.

6 Efficient Identification of Hierarchical Outliers

Given the proposed cLSH indexing scheme, we are able to identify hierarchical outliers *HO(N,r)* based on Definition 1 and compute their *grade* according to the procedure described below.

Formally, given a query point q, we would like to compute its grade. Notice that q may be part of the data set, or an arbitrary point (e.g. a new customer). The identification process begins at the lowest level l_h of the hierarchy. Firstly, a nearest neighbor (NN) query is executed for the query point q utilizing the HCT_h hash tables created

Algorithm 1 $HO\_Query(q, L, support, grade)$

Input: q is the query point
l_i is i-th level of hierarchy H
$support$ is the support score of q at level l_{L+1}
$cur\_support$ is the support score of q at level l_L
$grade$ is Hierarchical Outlier Grade for q

1: $SupCand_L(q) = \emptyset$ $cur\_support = 0$
2: **for** $j = 1 \dots T_L$ **do**
3: $SupCand_L(q) = SupCand_L(q) \cup lsh(q, HCT_L^j)$
4: **end for**
5: **for** $\forall c \in SupCand_L(q)$ **do**
6: $sup(c)_q^L = ComputeSupport(q, c)$
7: $cur\_support = cur\_support + sup(c)_q^L$
8: **if** $pred(c) \notin bucket(q, HCT_{L+1}^1)$ **then**
9: $support = support + sup(c)_q^L$
10: **end if**
11: **end for**
12: **if** $support < N$ OR $cur\_support < N$ **then**
13: $grade++$
14: **if** $L \geq 2$ **then**
15: $HO\_Query(q, L-1, support, grade)$
16: **end if**
17: **end if**

for indexing data at level l_h. Following the original LSH scheme's way of *NN* evaluation [7], we compute the hash value of q by applying the g_h^i hash function for every one of the T_h hash tables at level l_h, where $1 \leq i \leq T_h$. We retrieve from every HCT_h hash table the content from those buckets which id value is the same with the computed hash value of q. A set of items is returned from each bucket. These sets are merged, removing any duplicates, forming a resulted set, named $SupCand_h(q)$. $SupCand_h(q)$ set contains all the centroids of the data clusters containing data items that are candidates to lie within distance r from q.

A query item q *gains* support (i.e. increases its support score), if a centroid lies within distance r_i from it at level l_i. In order to compute the support that a centroid *gives* to a query item we proceed to an approximation technique, based on which a centroid c, with radius r_c and $rep(c)$ (where $rep(c)$ are the number of data items a cluster contains), gives support $sup(c)_q$ to a query point q according to the following formula:

$$sup(c)_q^i = rep(c) \times \frac{V(Sphere(c, r_c) \bigcap Sphere(q, r_i))}{V(c, r_c)} \tag{2}$$

where $Sphere(q, r_i)$ is the the hyper-sphere having as center the point q and radius r_i. Figure 4 provides a visualization of this process.

Algorithm 1 shows the algorithm for computing whether q is a hierarchical outlier and return its grade. Firstly, we compute the set $SupCand_L(q)$ of candidate centroids (Lines 2-4), that may give support to q at level L. Function *lsh* returns those centroids

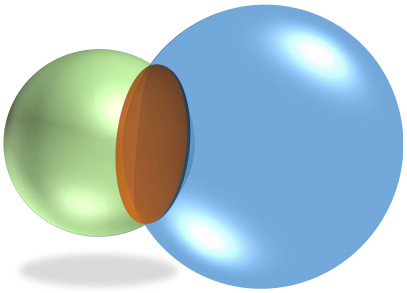

Fig. 4. Computing support that a centroid c "gives" to a query point q

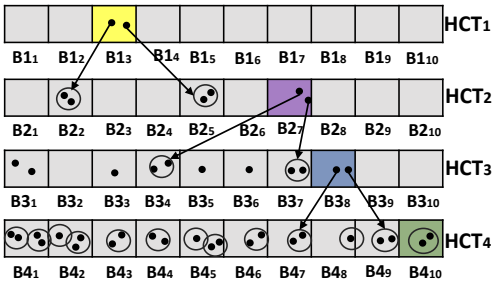

Fig. 5. Buckets Visited during Query Execution.

from all the hierarchical hash tables at level l_L, which have the same hash value with q. We then update (Lines 5-10) the support score of q at level l_L based on every centroid c that belongs to $SupCand_L(q)$. Function *ComputeSupport(q,c)* (Line 6) approximates (as it implied by Equation 2) the support that c provides to q. In Line 7 the support $sup(c)_q^L$ increases the $cur\_support$ of q at level l_L. In case the centroid c represents a cluster of centroids (this information is derived by function *pred(c)*) that belong to a bucket HCT_{L+1}^1 of level l_{L+1} that it has not been processed during the query evaluation at level l_{L+1}(i.e. members of c's cluster do not fall in the same bucket of HCT_{L+1}^1 with the one that q's hash value implies - $bucket(q, HCT_{L+1}^1)$), its providing support to q is also added to the support that q has already gained by the previous levels (Lines 8-10). If the $support$ or $cur\_support$ do not exceed threshold N the q's grade is increased by one and we recursively call the algorithm for level l_{L-1}. Our process terminates whenever the obtained support at a level l_i exceeds threshold N or level l_1 is reached.

In our running example, given a query point q, we first compute its hash values for hierarchical levels l_4, l_3, l_2, l_1 and then we assign it to the corresponding buckets that our index maintains. As it is depicted in Figure 5, q falls in buckets $B4_{10}, B3_8, B2_7$ and $B1_3$ of HCT_4, HCT_3, HCT_2 and HCT_1 respectively. For ease of presentation, we only depict one table per level, instead of T^i copies for every level l_i that our method suggests. However, query execution utilizing T^i tables per level is straightforward to what we discuss here. We only need to merge the sets of items retrieved from the buckets of T^i tables of a specific level and then proceed as we describe below.

For a given threshold value N=10, our method starts at level l_4 evaluating the Euclidean distance between q and centroids stored in bucket $B4_{10}$ of HCT_4. Based on these evaluations we approximate q's support score at hierarchical level l_4. In case support value is greater than N, we terminate query's evaluation and answer that q is not a hierarchical outlier, otherwise we set its grade value to 1 and we continue checking q at level l_3. Assuming in this example that support score at hierarchical level l_4 is two, we continue at level l_3 retrieving the two centroids (c_{310}, c_{311}) stored at bucket $B3_8$ of HCT_3 based on q's hash value. For each one of these two centroids we approximate the support that they provide to q by utilizing Equation 2. Suppose that $sup(c_{310})_q^3 = 3$ and $sup(c_{311})_q^3 = 2$ we conclude that q is also an outlier at level l_3 and increase its grade by one. Continuing at level l_2 we obtain centroids c_{25}, c_{26} from $B2_7$ bucket,

which represent clusters that its members are stored in bucket $B3_4$ and $B3_7$ respectively, that has not been processed during the lower levels query evaluation and thus could be added to the already computed support score of q. The support that c_{25} provides (e.g. $sup(c_{25})_q^2 = 6$) is added both to the $cur\_support$ for level l_2 and $support$ that q have already gained from levels l_3 and l_4. $support$ exceeds threshold N and query execution terminates (without accessing bucket $B1_3$ of level l_1). As a result, our algorithm replies that q is identified as a hierarchical outlier with grade=2.

Concluding, we should notice that a hierarchical outlier detection query involves processing of several buckets of the HCT tables for levels l_h up to l_1. However, the higher the hierarchical level our method examines, the lower is the number of centroids obtained by these buckets, as the number of centroids at higher levels is reduced as an effect of the recursive clustering over the hierarchy during index construction. Moreover, our algorithm retains the value of support from previous (lower) levels, in order to expedite processing. Consider level l_1, where we have already processed three buckets ($B4_{10}$, $B3_8$, $B2_7$), that give the necessary support to q at l_2 and so the $B1_3$ *bucket* is not need to be accessed. Finally, the monotonicity property of hierarchical outliers, permits us to terminate the query, when enough support is gained at a specific level.

7 Experimental Evaluation

In this section, we present an experimental evaluation of the proposed hierarchical outlier detection framework. All algorithms are implemented in Java and the experiments run on a desktop PC with an i7 CPU (4 cores, 3.4 GHz), 8GB RAM, and a 128GB SSD.

7.1 Experimental Setup

Data sets. In the experimental study, we employ two data sets. In the first dataset, we created a hierarchy of products consisting of six levels with dimensions (cardinality) 2654, 380, 51, 13, 3, 1 from the leaves to the root of hierarchical tree, respectively. We generated data for 50000 customers with their purchases over the 2654 different products at the lowest level of the hierarchy. In order to generate the purchases of a customer, we first set the number of cumulative purchases for every customer by selecting uniformly from the range 30000 - 80000. We then selected randomly 20% of the 2654 products belonging at the lowest level of product's domain hierarchical tree. These 20% of products are considered as high interest products for customers and 80% of her total purchases are uniformly distributed to these products. The remaining 20% of a customer's purchases are distributed randomly to the rest of products (that span 80% of the produce domain) that are considered as low interest. We created several clusters of customers where customers of the same cluster have the same sets of high and low interest products.

We also used the OLAP Council APB-1 benchmark generator [12] to create a second dataset which contains 5300 customers. For every customer, the generator produced a vector representing her cumulative purchases over a period of 17 months on a domain of 6050 products. The products' domain hierarchy consists of six hierarchical levels.

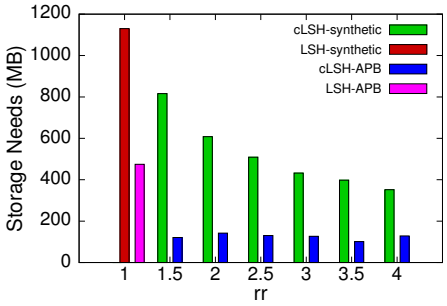

Fig. 6. Storage Requirements

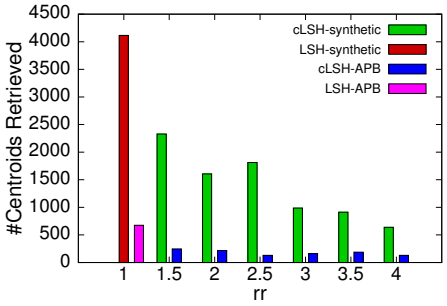

Fig. 7. Index Points Retrieved per Query

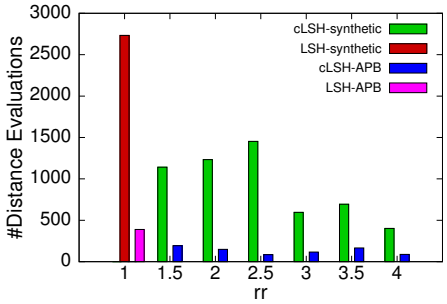

Fig. 8. Average Distance Evaluations per Query

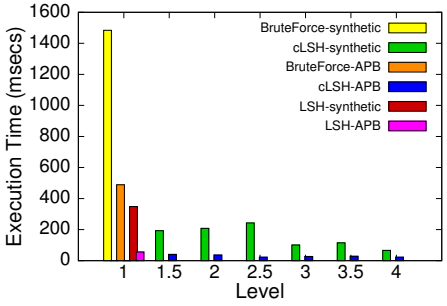

Fig. 9. Average Execution Time per Query

Algorithms. We evaluate our hierarchical outlier detection algorithm that utilizes the hierarchically clustered LSH (cLSH) index and we compared it to an alternative implementation of the same algorithm that utilizes independent LSH indices for every level of the hierarchy. All indices are parametrized as described in [14].

Metrics. Our main metrics include: a) the average number of distance evaluations for a hierarchical outlier detection query, b) the average query execution time, c) the average number of candidates points that the indices return per query execution, d) the storage needs for both implementations, and e) the precision of the results of both techniques computed as

$$precision_{level(i)} = \frac{|customers\ retrieved_{level(i)} \cap real\ outliers_{level(i)}|}{|customers\ retrieved_{level(i)}|}$$

Queries. We present average values over 100 queries, where all query points are outliers at lowest level of products hierarchy on both datasets. For the remaining levels the number of outliers range from 4 to 74. The higher the hierarchical level, the smaller the number of queries that are outliers. For instance at fifth level there are 74 queries identified as outliers for APB dataset and 13 for the synthetic one while at the highest hierarchical level there are only 6 and 4 outliers respectively.

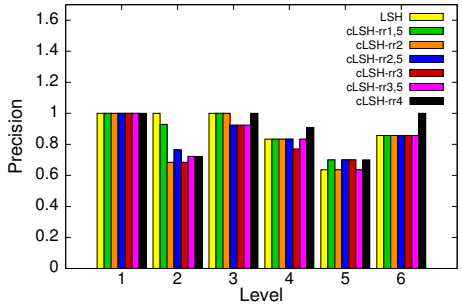

Fig. 10. Synthetic dataset

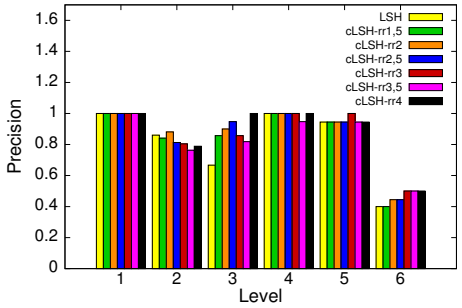

Fig. 11. APB dataset

Parameters. We conduct experiments varying the reduction ratio rr (1.5-4) that defines $k_i = \frac{|B_i|}{rr}$ for the $k-$means clustering evaluation on every bucket B_i.

7.2 Experimental Results

Space Cost. In Figures 6, we depict the storage needs of our technique for various values of rr. The higher the requested value of rr the lower the space cost because larger clusters are constructed and, thus, fewer centroids are stored at the cLSH index. Given that only centroids are maintained by cLSH it is expected that we gain in terms of space compared to the original LSH scheme.

Distance Evaluations and Index Points Retrieved. In Figure 7, we show the average number of the points (LSH)/centroids (cLSH) returned as candidates by each index to provide support to a query point. This number is significantly smaller for cLSH, as a result of the recursive clustering over the hierarchy and the way these centroids are used to increase the support of a query point.

Figure 8 depicts the average number (over 100 queries) of distance evaluations required in order to compute the hierarchical outliers and their grade. Our method using cLSH performs up to 80% fewer distance evaluations in order to detect a hierarchical outlier, compared to the straightforward LSH scheme. This significant reduction is attributed to the use of centroids in order to calculate the support from a whole cluster of points to the query point in a single step, instead of a per-data-item calculation.

Execution Time. Figure 9 shows that our method is up to 75% faster compared to the original LSH scheme and 15 times faster than a brute-force method that does not use any index. The main factor that increases the execution time is the number of distance evaluations. This is evident by the fact that the evaluations and execution time graphs follow the same trend.

Precision. Figures 10, 11 depict the precision of both indices (LSH/cLSH) in hierarchical outlier identification. Both techniques are very accurate and provide high precision results. We do not provide a similar graph for the recall because it was 100% for all levels of the hierarchy, in these experiments (i.e. our approximation technique-summarized in Formula 2- overestimates the support score). Even though cLSH is significantly more condense than the LSH index, it provides equally accurate results.

8 Conclusions

In this work we introduced a framework for detecting outliers in hierarchically organized domains. Key to our method is a monotonicity property that enables us to grade in an intuitive manner how erroneous a data item seems with respect to the rest of the data. We also discussed a novel indexing scheme that computes hierarchical clusters of data items and embeds them in a LSH index. Using this index we can quickly identify hierarchical outliers with reduced storage and computation cost compared to using a straightforward LSH index. The benefits of our techniques stem from the hierarchical organization of the LSH buckets that permits us to reuse distance computations while exploring a data item.

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: FOCS. pp. 459–468. IEEE Computer Society (2006)
2. Bhaduri, K., Matthews, B.L., Giannella, C.R.: Algorithms for speeding up distance-based outlier detection. In: Proceedings of the 17th ACM SIGKDD international conference on Knowledge Discovery and Data Mining. pp. 859–867. ACM (2011)
3. Breunig, M., Kriegel, H., Ng, R., Sander, J., et al.: Lof: identifying density-based local outliers. Sigmod Record 29(2), 93–104 (2000)
4. Charikar, M.: Similarity estimation techniques from rounding algorithms. In: STOC (2002)
5. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-Sensitive Hashing Scheme Based on p-stable Distributions. In: Proccedings of SCG (2004)
6. Georgoulas, K., Kotidis, Y.: Distributed Similarity Estimation using Derived Dimensions. VLDB J. 21(1), 25–50 (2012)
7. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC (1998)
8. Jin, W., Tung, A., Han, J., Wang, W.: Ranking Outliers using Symmetric Neighborhood Relationship. In: Proceedings PAKDD. pp. 577–593 (2006)
9. Knorr, E., Ng, R., Tucakov, V.: Distance-based outliers: algorithms and applications. The VLDB Journal 8(3), 237–253 (2000)
10. Kriegel, H., Zimek, A., et al.: Angle-based outlier detection in high-dimensional data. In: Proceeding of ACM SIGKDD. pp. 444–452 (2008)
11. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In: VLDB. pp. 950–961 (2007)
12. OLAP Council APB-1 Benchmark., `http://www.olapcouncil.org/research/resrchly.htm`:
13. Papadimitriou, S., Kitagawa, H., Gibbons, P., Faloutsos, C.: LOCI: Fast Outlier Detection Using the Local Correlation Integral. In: Proceedings of ICDE. pp. 315–326 (2003)
14. Slaney, M., Lifshits, Y., He, J.: Optimal parameters for locality-sensitive hashing. Proceedings of the IEEE pp. 2604–2623 (2012)
15. Sugiyama, M., Borgwardt, K.: Rapid distance-based outlier detection via sampling. In: Advances in Neural Information Processing Systems. pp. 467–475 (2013)
16. Tang, J., Chen, Z., Fu, A., Cheung, D.: Enhancing effectiveness of outlier detections for low density patterns. Advances in Knowledge Discovery and Data Mining pp. 535–548 (2002)