STATS - A Point Access Method for Multidimensional Clusters

Giannis Evagorou(✉) and Thomas Heinis

Imperial College London, SW7 2AZ, London, UK
{g.evagorou15,t.heinis}@imperial.ac.uk,

Abstract. The ubiquity of high-dimensional data in machine learning and data mining applications makes its efficient indexing and retrieval from main memory crucial. Frequently, these machine learning algorithms need to query specific characteristics of single multidimensional points. For example, given a clustered dataset, the *cluster membership* (*CM*) query retrieves the cluster to which an object belongs.
To efficiently answer this type of query we have developed *STATS*, a novel main-memory index which scales to answer *CM* queries on increasingly big datasets. Current indexing methods are oblivious to the structure of clusters in the data, and we thus, develop STATS around the key insight that exploiting the cluster information when indexing and preserving it in the index will accelerate look up. We show experimentally that STATS outperforms known methods in regards to retrieval time and scales well with dataset size for any number of dimensions.

Keywords: High-dimensional indexing · Clustering

1 Introduction

Machine learning algorithms have received considerable attention from the research community, demonstrating their usefulness of extracting knowledge from data. Applications of machine learning are vast and include image segmentation, object or character recognition and many more. In image segmentation, for example, machine learning is used to identify tumors and other pathologies to determine the best surgical plan or diagnosis [5]. Executing machine learning efficiently is hence key to analyzing big amounts of data.

Not only do machine learning algorithms require a vast number of iterations to converge, but in each iteration, they execute a vast number of queries on high-dimensional data. Machine learning algorithms [13] thus, crucially depend on the efficient execution of CM (cluster membership) queries — testing to which cluster an object belongs — during their execution. This is particularly true for algorithms that assume background knowledge, i.e., semi-supervised learning algorithms like constrained K-means [14] which uses partial or complete cluster membership (CM) information to improve the accuracy of the clustering.

With these applications in mind, we develop STATS, a new index to answer *CM queries* efficiently. Formally, STATS takes a point query $Q = (q_1, ..., q_n)$ and, if Q belongs to any cluster C, it returns the id of C. Known indexes for CM queries store the cluster ID separately in the index which increases the size of the data structure, making retrieval slower.

STATS is a main-memory index that uses statistics to index multidimensional points along with their cluster ID. The cluster ID is an integral part of the

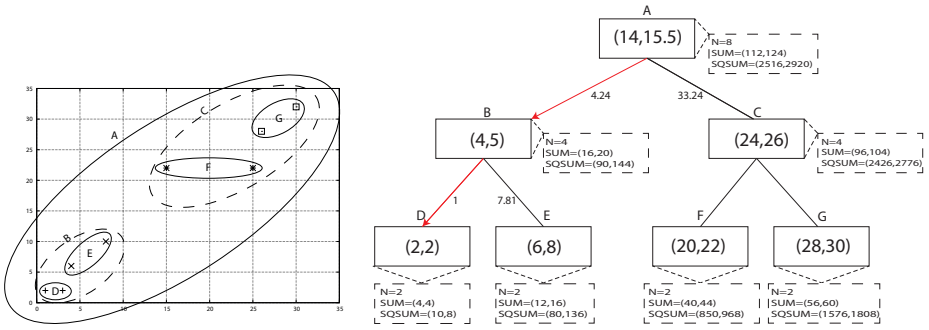


Fig. 1: STATS indexing points

index, which makes STATS more efficient when answering CM queries. STATS uses bottom-up clustering to produce a hierarchical data structure, which is the equivalent of a dendrogram in agglomerative clustering approaches [9]. The dendrogram guides the search to a particular cluster (which contains the points).

Our experimental evaluation shows that STATS outperforms other approaches in terms of retrieval time and it scales well for datasets of increasing size. Furthermore, its performance does not deteriorate as dimensionality increases.

2 STATS

In this paper, we develop STATS, a novel dimension agnostic main-memory index that uses *statistics* to represent clustered points in space. STATS uses *agglomerative clustering* (i.e. bottom-up clustering) to cluster data points in a recursive manner. In case of clustered data, a large number of points can be effectively indexed using a representative point - the cluster's centroid.

During its building phase, STATS uses agglomerative clustering to produce a hierarchical structure of statistics that can be used to calculate distances and guide the search. The result of the building phase is an *unbalanced binary tree*. An internal node contains statistics that summarize the content of its descendants. An inner node is a cluster without the data but with statistics that represent the constituent points at lower levels. Leaf-level nodes contain the data points.

In this project, we propose the use of statistics, instead of other approximations (e.g. Minimum Bounding Rectangles used in R-Tree variants [2, 12, 3]), to effectively summarize points in k-dimensional space. These statistics can efficiently support query operations. Like the internal nodes of R-tree variants contain MBRs to summarize objects at lower levels, STATS uses statistics to summarize the objects at a lower level:

1. N: the number of data points (recursively) enclosed leaf-level nodes contain,
2. SUM: a vector whose i_{th} element is the sum of all points the node represents at dimension i,
3. $SQSUM$: a vector whose i_{th} element is the sum of the squares of all points the node represents at dimension i.

An internal node stores these statistics and can derive additional statistics that can be useful for searching the index. The derived statistics are:

1. C, the centroid of the node, can be derived by dividing the SUM vector with the scalar value N. C represents the node and is needed for queries,

2. VAR, a vector whose i_{th} element is the variance of the coordinates over all points that the node represents at dimension i. VAR_i is computed by calculating the expression: $\frac{SQSUM_i}{N} - (\frac{SUM_i}{N})^2$, for every dimension i,
3. STD, a vector whose i_{th} element is the standard deviation of the coordinates at dimension i, over all points that the node represents. Standard deviation is estimated by performing an element-wise square root of the vector VAR.

Let us assume the four clusters in 2D space described in Figure 1 (left); the resulting data structure is depicted in Figure 1 (right). STATS starts building the index structure based on the clusters in the dataset. Each cluster in the dataset is summarized into a leaf node of STATS. To build the upper levels of STATS nodes are recursively merged starting at the leaf level, i.e., the statistics of low-level nodes are combined into a higher-level node. Merging the nodes and their statistics continues bottom up until the root of the tree is created.

The example in Figure 1 illustrates this: the ancestor of the leaf nodes $(2, 2)$ and $(6, 8)$ contains the centroid $(4, 5) = (\frac{SUM_1}{N}, \frac{SUM_2}{N}) = (\frac{16}{4}, \frac{20}{4})$. The required information is shown in the dashed boxes, while (some of) the derived information is shown in the solid boxes. To avoid calculation of the derived information during the querying phase, we store both the derived and the required information. When main memory is scarce, however, derived information is not stored.

Leaf nodes also need to store information about single points. A Red-Black tree maintains the distance of each point from the cluster's centroid. To rapidly locate the point within a leaf node, we index the distance of each point from the centroid of its enclosing cluster/leaf-node. In particular, at the leaf-level we maintain a Red-Black tree that holds the following key-value pairs: $[d(p_i, c), p_i]$, where p_i is a point that belongs to the cluster/leaf-node c and $d(p_i, c)$ is the distance of p_i from c's centroid. Put differently, $d(p_i, c)$ is used as the key to index a Red-Black tree to retrieve p_i and thus, impose a total order on all points in the leaf node. The distance metric used (i.e., d) to index the points at the leaf level must be the same as the one used during the querying phase.

Building Phase

The building phase uses agglomerative clustering to create an unbalanced binary tree. At each step of the algorithm, the two nearest clusters are merged generating a new cluster, which is added to the collection of available clusters to be merged. Importantly, each cluster is represented by its centroid. Algorithm 1 explains the building phase (see Table 1 for the notation used).

The building phase starts by estimating the pairwise distances between each pair of clusters in CL (lines 1 to 5). The most important steps of the algorithm are explained next: 1) Line 5 inserts the pairwise distances between all clusters into the heap, H. 2) Line 7 retrieves from the heap the two clusters which have the closest centroids to each other. 3) If the two clusters have already been merged, they are ignored and removed from the set of all clusters (lines 8 to 10). 4) Line 11 merges the nearest clusters creating a new cluster, which only contains statistics summarizing points or clusters. 5) The heap is updated with the distances between the new cluster and the remaining clusters (line 13). 6) Nodes are created bottom-up to form an unbalanced binary tree (line 15).

Algorithm 1 Building phase

```
1: for c_i in CL do
2:     for c_j in CL do
3:         if c_i ≠ c_j then
4:             Calculate d(c_i, c_j)
5:             Insert(d,H)
6: while H is not empty do
7:     H.top()
8:     if merged(c_i,c_j) then
9:         H.pop()
10:        continue
11:    c_ij=merge(c_i, c_j)
12:    CL = CL − c_i − c_j
13:    update(c_ij,CL,H)
14:    CL = CL ∪ c_ij
15:    createNodes(c_i, c_j, c_i,j)
16:    H.pop()
```

Algorithm 2 Querying algorithm

```
   Input:qp          ▷ The query point
   Output:clusterId
1: node = root
2: while node do
3:     left = node.getLeft()
4:     right = node.getRight()
5:     leftDist = distance(left,qp)
6:     rightDist = distance(right,qp)
7:     if leftDist < rightDist then
8:         node = left
9:     else
10:        node = right
11:    if node.isLeaf() then
12:        leafDis = dist(node, qp)
13:        point = RBtree(leafDis)
14:        report point.clusterId()
```

Table 1: Building phase notation

Symbol	Description
H	Heap
CL	All clusters
c_{ij}	Merged cluster
c_i	Random cluster i
c_j	Random cluster j
$d(c_i, c_j)$	Distance between clusters i & j

Table 2: Building phase - heap contents

i

p_i	p_j	$d(p_i, p_j)$
D	E	$\sqrt{52}$
F	G	$\sqrt{128}$
E	F	$\sqrt{392}$
D	F	$\sqrt{724}$
E	G	$\sqrt{968}$
D	G	$\sqrt{1460}$

ii

p_i	p_j	$d(p_i, p_j)$
F	G	$\sqrt{128}$
F	B	$\sqrt{545}$
G	B	$\sqrt{1201}$

iii

p_i	p_j	$d(p_i, p_j)$
C	B	$\sqrt{841}$

As an example consider the index in Figure 1. The algorithm begins by estimating the distance between each pair of clusters (i.e., the leaf nodes) with a given distance metric. The initial distance calculation is shown in Table 2 (**i**). It finds nodes D and E closest to each other, and they are thus merged, creating node B - the parent of D and E. The distance between B and all other nodes has to be calculated and inserted into the heap (see line 13 of Algorithm 1). Table 2 (**ii**) (where we use eager removal of elements from the heap to make the example more concise) shows the updated heap after the merge of D and E. At the next step of the algorithm, nodes F and G are merged, creating node C. Finally, C and B are closest to each other and thus merged to create the root of tree A.

The merging process (line 11 in Algorithm 1) does not consider all the points of the two clusters to be merged. If an internal node, n, at a high level of the tree represents l_n leaf nodes and the size of the clusters at the leaf level is s, then the complexity for merging nodes n and m is $\mathcal{O}(l_n * l_m * s)$. For bigger nodes (and bigger datasets) merging nodes can become prohibitively expensive

and merging close to the root of the tree is more expensive because nodes involve more points. To mitigate this expensive merging procedure we merge the required statistics instead. Therefore, if we want to create cluster c by merging clusters a and b, then we estimate the expressions in Equation 1, for every $i \in [1, ..., d]$ where d is the number of dimensions. As an example consider nodes D and E in Figure 1. STATS can efficiently merge them and create node B by calculating the aforementioned expressions, as in Equation 2.

$$\begin{aligned} N_c &= N_a + N_b \\ SUM_i^c &= SUM_i^a + SUM_i^b \\ SQSUM_i^c &= SQSUM_i^a + SQSUM_i^b \end{aligned} \quad (1)$$

$$\begin{aligned} N_b &= 2 + 2 \\ SUM_b &= \begin{bmatrix} 4 \\ 4 \end{bmatrix} + \begin{bmatrix} 12 \\ 16 \end{bmatrix} = \begin{bmatrix} 16 \\ 20 \end{bmatrix} \\ SQSUM_b &= \begin{bmatrix} 10 \\ 8 \end{bmatrix} + \begin{bmatrix} 80 \\ 136 \end{bmatrix} = \begin{bmatrix} 90 \\ 144 \end{bmatrix} \end{aligned} \quad (2)$$

Distance Calculation and the Building Phase

Throughout this work, we represent a cluster with a *centroid*. A centroid is a point in a k-dimensional space, where its i_{th} dimension is the average over all points in the same dimension. Assuming a cluster with N points in a k-dimensional space, we can formally define the centroid to be: $[c_1, ..., c_k]$, where $c_i = \frac{1}{N} \sum_{j=1}^{N} x_{ji}$, with x_{ji} as the value of the i_{th} dimension of a random point j.

During the building phase, we estimate the distance between two clusters, a and b, by using the *euclidean* distance metric: $d(a, b) = \sqrt{\sum_{i=1}^{d} (c_i^a - c_i^b)^2}$. Although the *DED* (i.e. Default Euclidean Distance) metric is not appropriate for high dimensional data [1], especially when executing nearest neighbour queries, it is suitable when the centroids of the clusters are well separated.

STATS Querying

STATS uses a point as the key to retrieve the ID of the cluster the point belongs to. Algorithm 2 describes the procedure followed to find a query point. The search for a point, *qp*, starts at the root, where the distance between *qp* and the root's descendants is calculated (lines 5-6 of Algorithm 2). The algorithm then follows the path with the least distance (lines 7-10), until STATS reaches the leaf node where the point resides.

We illustrate the process with an example where we look for the point $(1, 2)$ that resides in leaf node D (see Figure 1 - left). The search algorithm follows the path $A \rightarrow B \rightarrow D$ shown in Figure 1 (the branch with the smallest distance).

When the search algorithm reaches the leaf node, the distance between *qp* and the leaf node's centroid is calculated. The distance is used to locate the point in logarithmic time (guaranteed by the Red-Black tree) without performing a linear search over all points in the cluster. Algorithm 2 uses Equation 3 to estimate the distance between *qp* and a cluster b. In Equation 3, c_i^b is the value of the centroid and σ_i^b is the standard deviation of a cluster b at dimension i. We can use the *DED* as illustrated in Equation 4 to estimate the distance between a query point and each node in the hierarchy. Nevertheless, the query accuracy of Algorithm 2 deteriorates when using the *DED* metric.

$$d(qp,b) = \sqrt{\sum_{i=1}^{d} \frac{(qp_i - c_i^b)^2}{(\sigma_i^b)^2}} \quad (3) \qquad d(qp,b) = \sqrt{\sum_{i=1}^{d} (qp_i - c_i^b)^2} \quad (4)$$

The default Euclidean distance is not suitable for high-dimensional data. The work in [1] has explored the behavior of the L_k norm for various values of k (including the L_2 norm, i.e., Euclidean distance) and has proven that the L_2 norm is *not meaningful* for high dimensional data. The problem with DED and high dimensional data is that the ratio of the distances of the farthest and nearest objects to a query object is almost 1 [1], i.e., the distance between a query object and its farthest neighbor is as big as the distance between the same query object and its nearest neighbor. This blurry distinction between farthest and nearest objects renders the default Euclidean distance *unsuitable* for the querying Algorithm 2. Normalized Euclidean Distance (*NED*) is suitable for clustered data as it considers the spread of a cluster in each dimension.

Table 3: Datasets

Series 1		Series 2		Series 3	
# of clusters	Size(GB)	Dimensions	Size	# of clusters	Cluster size
128	0.071	10	0.354 *MB*	100	5K
256	0.143	20	0.7 *MB*	200	5K
400	0.226	30	1.1 *GB*	300	5K
512	0.290	40	1.4 *GB*	400	5K
1000	0.564	50	1.7 *GB*	500	5K
1500	0.847	60	2.1 *GB*	600	5K
2000	1.1	70	2.4 *GB*	700	5K
3000	1.7	80	2.8 *GB*	800	5K
3500	2.0	90	3.1 *GB*	900	5K
4000	4.5	-	-	-	-

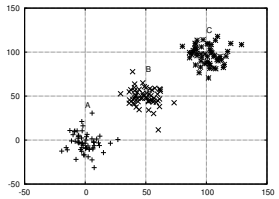


Fig. 2: Clustered Data - Structure

3 Experimental Evaluation

We compare the retrieval time of STATS with an R*-Tree [4] (bulkloaded R*-Tree using STR [7] and a plain R*-Tree [2]) and a KD-Tree [10]. We additionally evaluate the accuracy of the Querying Algorithm 2 for several distance measures. We assume a static clustered dataset and a workload of CM queries.

3.1 Experimental Setup

We run the experiments on a virtual machine (one core at 3.6GHz, 14GB of RAM and 100 GB of disk) with a 64-bit Debian-based OS - Linux Mint with a kernel version of 4.4.0.21. We use numactl [6] on Linux to ensure that experiments run on one processing core. The R-Tree [4] uses a branching factor of 100 and a fill factor of 70%. For both trees, we use the parameters providing the best performance. All the implementations are in-memory.

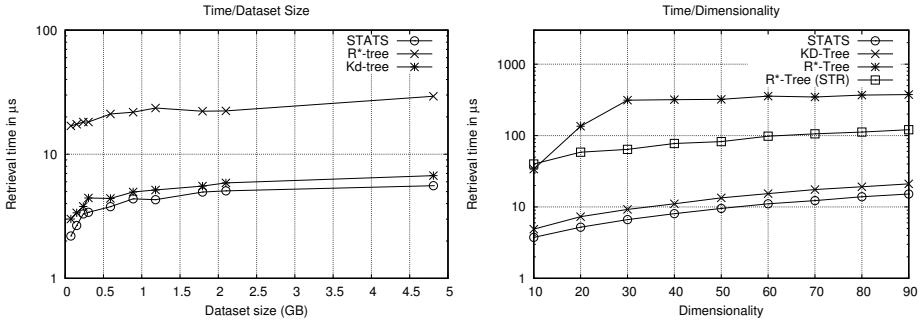
3.2 Experimental Methodology

To measure the retrieval time, we use synthesized datasets up to 90 dimensions containing well-separated clusters produced with ELKI [11]. Figure 2 shows an example of our synthesized datasets. Cluster A has a normal distribution with a mean of 0, and a standard deviation of 10, the two dimensions of cluster B have a normal distribution with a mean of 50 and a standard deviation of 10, etc.

Retrieval Time and Dataset Size In this series, we compare the scalability of STATS with the KD-Tree and the R*-Tree (see Figure 3a). This series assumes constant cluster size at 10'000 points, three dimensions and a varying number of clusters (see Table 3). Not only does the R-tree follow an upward trend, but its retrieval time for the smallest datasets is also high. STATS has the lowest

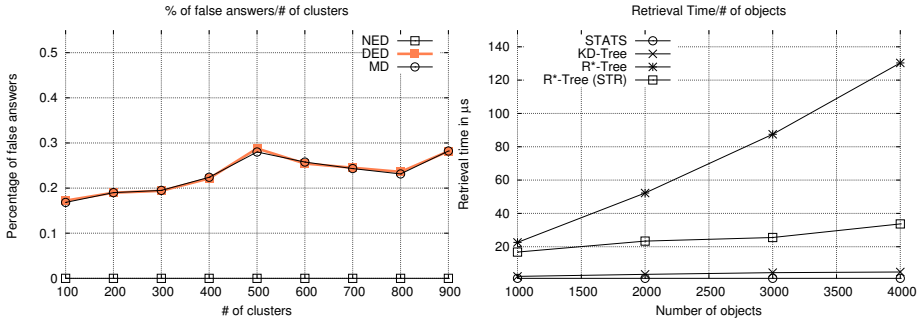
retrieval time and scales well with dataset size. The KD-tree has similar performance to STATS due to its very good behavior in low dimensions. Nevertheless, both experience a slight increase when querying the two biggest datasets.

Retrieval Time and Dimensionality This series measures the impact of dimensionality on retrieval time (Figure 3b). The number of clusters is 200 and with a size of 10'000, while we increase dimensionality (see Table 3). The R*-Tree has the worst performance with a mean retrieval time of 285 μs across any dimensionality. Additionally, we compare with an R*-Tree loaded with STR [7] and configured with a branching factor of 100 and a fill factor of 0.8. Using STR yields faster and more stable execution times. STATS and the KD-Tree experience a steady increase in retrieval time as dimensionality increases.



(a) (b)

Fig. 3: Cluster membership query time (μs) (logscale)



(a) Percentage of false answers % (b) Retrieval Time in μs

Fig. 4: Queering accuracy and performance on a real dataset

Distance Metric and Querying Accuracy STATS can calculate both *NED* and *DED* metrics. To illustrate the impact of choice of distance metric, we execute 50'000 queries to measure the percentage of false answers for three metrics, the DED metric, the NED metric and the Manhattan Distance (MD) metric. Generally, when using the L_k norm, the contrast between the farthest and nearest neighbours is bigger for small values of k rather than for big values of k. For this reason, we use the L_1 norm for Algorithm 2.

For this experiment (see Figure 4a), we use a dataset of dimensionality 30 with a varying number of clusters. Increasing the number of clusters results in more leaf nodes, which increases the number of paths in the tree. We show that choosing the wrong distance metric can create ambiguity in the choice of path.

The results show that the MD and the DED metric have almost the same accuracy. STATS' building phase creates the inner nodes of the hierarchy bottom-up. More centroids are thus created in the space apart from the centroids of the *actual clusters/leaf nodes*, i.e., the clusters that contain the points. These high-level centroids increase the ambiguity in the choice of path. The NED metric alleviates this problem by considering the extent of the clusters and thus, Algorithm 2 follows the cluster whose centroid is close enough but its extent is big enough to accommodate the query point in its circumference.

Retrieval Time on Real Data In the last experiment (Figure 4b), we index measurements of abalones [8]. The cluster ID for a group of abalones is their age, and each dimension is a measurement (diameter, height, etc.). STATS has the lowest retrieval time. When using the R*-Tree without bulk-loading, time increases rapidly with the number of objects indexed. Bulk-loading the R*-Tree yields a more stable and lower time, but it is still considerably higher than that of STATS. The KD-Tree is the only indexing approach that competes with STATS.

4 Conclusions

With STATS, we depart from the norm of approximating points using bounding based approximations, and we suggest the use of statistics to approximate multidimensional points. STATS stores cluster membership as an integral part of the index making it an efficient method to answer CM queries.

As our experiments show, STATS performs very well regardless of dimensionality and dataset size. It can index well separated clustered data and can efficiently answer CM queries. STATS is sensitive to the choice of distance metric but as we showed, NED provides very good performance on a range of datasets.

Acknowledgements

This work is supported by the EU's Horizon 2020 grant 650003 (Human Brain project), EPSRC's PETRAS IoT Hub and HiPEDS grant reference EP/L016796/1).

References

1. Aggarwal, C.C., Hinneburg, A., Keim, D.A.: On the Surprising Behavior of Distance Metrics in High Dimensional Space. In: ICDT '01
2. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD '90
3. Guttman, A.: R-trees: Dynamic Index Structure for Spatial Data. SIGMOD '84
4. Hadjieleftheriou, M.: libspatialindex (2014), https://libspatialindex.github.io/
5. Jain, A.K., Murty, M.N., Flynn, P.J.: Data Clustering: A Review. ACM Comput. Surv. 31(3), 264–323 (Sep 1999)
6. Kleen, A.: numactl(1) Linux User's Manual. SuSE Labs (September 2016)
7. Leutenegger, S.T., Lopez, M.A., Edgington, J.: STR: A simple and efficient algorithm for R-tree packing. In: ICDE '97
8. Lichman, M.: UCI Machine Learning Repository (2013)
9. Maimon, O., Rokach, L.: Data Mining and Knowledge Discovery Handbook. Springer-Verlag New York, Inc. (2005)
10. Muja, M., Low, D.G.: nanoflann (2016), https://github.com/jlblancoc/nanoflann
11. of Munich, L.M.U.: Elki data mining library (2016)
12. Sellis, T.K., Roussopoulos, N., Faloutsos, C.: The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. VLDB '87

13. Suthaharan, S.: Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning. Springer Publishing Company, Incorporated, 1st edn. (2015)
14. Wagstaff, K., Cardie, C., Rogers, S., Schrödl, S.: Constrained K-means Clustering with Background Knowledge. ICML '01