Exploiting Taxi Demand Hotspots Based on Vehicular Big Data Analytics

Lu Zhang, Cailian Chen, Yiyin Wang, Xinping Guan
Department of Automation, Shanghai Jiao Tong University,
Key Laboratory of System Control and Information Processing, Ministry of Education of China
Shanghai, P.R.China
{tianlangmushui, cailianchen, yiyinwang, xpguan}@sjtu.edu.cn

***Abstract*—In the urban transportation system, the unbalanced relationship between taxi demand and the number of running taxis reduces the drivers' income and the levels of passengers' satisfaction. With the help of vehicular global positioning system (GPS) data, the taxi demand distribution of city can be analyzed to provide advice for drivers. A clustering algorithm called Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is suitable for discovering demand hotspots. However, the execution efficiency is still a big challenge when DBSCAN is applied on big databases. In this paper, we propose an improved density-based clustering algorithm called Grid and Kd-tree for DBSCAN (GD-DBSCAN), which integrates partitioning method with kd-tree structure to improve the computational performance of DBSCAN. Furthermore, this algorithm can take advantages of multi cores and shared memory to parallelize related functions. The experiment shows GD-DBSCAN is efficient, it has an improvement of at least 10% in performance compared with DBSCAN.**

I. Introduction

With the development of economy and the growth of urban population, the urban transport problems are becoming serious. Taxi has become an important part in urban transport system, because of its fast, convenient and a wide range of services. However with the increase of taxi, extensive management mode results in the decrease of working efficiency of taxi, traffic jams, energy waste and environment pollution. Passengers often wait for a long time to take a taxi, while the taxis often drive with no passenger and the empty rate is high.

To solve the problem, getting accurate and full traffic information is a key. Advances in Global Position System (GPS), most taxis are now equipped with an infrastructure to record the current and historical taxi traces, producing a new source of rich spatio-temporal information. Over the last decade, GPS-location systems have attracted the attention of both researchers and industries because of the new type of information they provide [1]. Du [2] uses vehicular sensor networks to sense urban traffic. Zhang [3] proposes an effective service strategy with GPS data. An on-line recommendation model is presented in [4] to offer drivers advice about the best stand to head in each moment. With current and historical data uploaded from taxi, we can learn the running traces and distribution of taxi. Besides, these data helps exploit when and where people use taxi. Recognizing these demand hotspots can help drivers acquire jobs more effectively. [5].

For discovering demand hotspots, a Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm is proposed in [6]. DBSCAN is more suitable than other classical clustering approaches which will be discussed in the section II. However, when DBSCAN is applied on big databases, the execution efficiency is still a big challenge. There are few papers researching about improving the computational efficiency of DBSCAN. In [7] the author presents a GPU parallel version of DBSCAN. A scalable DBSCAN algorithm using MapReduce is presented in [8]. Moreover, a new algorithm based on DBSCAN using the Resilient Distributed Datasets approach is operated in a fully distributed fashion. Although they have good performance, they need more hardware support such as GPUs.

In this paper, we focus on algorithm optimization by changing algorithmic scheme and data structures instead of using more powerful hardware. For two-dimensional GPS data, we integrate partitioning method with kd-tree structure to improve the computational performance of DBSCAN. Kd-tree is a structure suitable for low-dimensional search. It can save much time when it is applied for querying the nearest points. The method of partitioning is also a good way to accelerate computation. On the one hand, partitions help avoid unnecessary searches. On the other hand, dividing data into many partitions is good for parallel computing, hence we can use multi cores and shared memory in a single machine to parallelize the algorithm based on two-dimensional data.

The following paper is organized as follows. Section II presents the process framework to extract pick-up events and DBSCAN clustering algorithm. Next, Section III proposes our improved clustering algorithm to sense hotspots. Then Section IV performs an experimental evaluation of the effectiveness and efficiency using real data. Finally, conclusions are drawn and future work topics are discussed in Section V.

II. Overview of Process Framework

The taxi GPS trace data set used in this paper is provided by Shanghai Qiangsheng Intelligent Navigation Technology Company. Shanghai has high population density and complicated traffic network. The taxi GPS traces were recorded from April 1, 2015 to April 30, 2015. The GPS data of a taxi is recorded every 10 Sec. Over 123,000 taxis upload about one billion records everyday. The data set has big volume of

978-1-5090-1701-0/16/$31.00 ©2016 IEEE

300 GB. As shown in Fig. 1, these records are uploaded to data center. On the one hand, data is stored in the hadoop distributed file system (HDFS) which is suitable for big data storage. On the other hand, these data is transmitted to the computation layer. Because data is huge, data processing will be a time-consuming task. Fortunately, there exists several latest distributed frameworks such as storm and spark-streaming. These tools are designed to process massive data, hence the total processing can be completed in a reasonable time such as several seconds. Then under the framework, a serial of data processing procedures are performed, including extracting pick-ups and drop-offs from unsorted raw record data. As a result, the distributions of taxis and pick-up locations can be generated. Clustering these pick-ups helps exploit demand hotspots. Hotspots split the city into different regions. For different regions, we compute their pick-up rate using real-time and historical information. Finally, these results help provide guidances for drivers to select locations of waiting passengers.

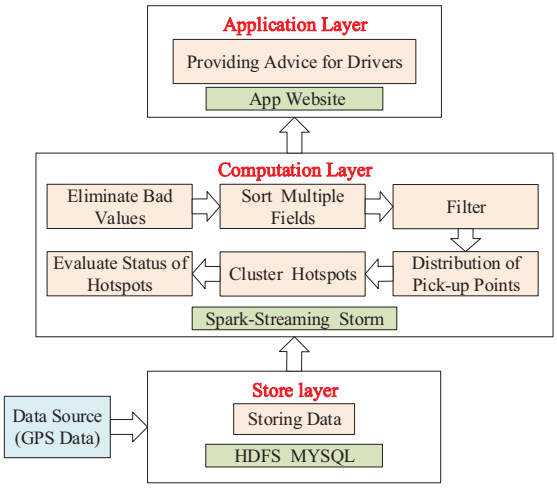


Fig. 1: Process Framework

Fig. 2: Distribution of pick-ups in Shanghai

A. Spatial Distribution of Pick-up spots

With preprocessing, pick-ups information can be generated easily. These pick-up records are separated from all records. For example, there are 112,348,953 uploaded records on April 1, 2015. After preprocessing, we get only 13220 pick-up records. These records have information about location and time of pick-ups, which are useful for following research. As shown in Fig. 2, we can research the spatial distribution of the pick-up spots through matching these points to Shanghai map. These block points represent the pick-up locations on April 1, 2015. Apparently, the distribution of pick-up locations is uneven and the number of pick-ups has a closely relationship with locations.

TABLE I: Main notations

Notation	Physical interpretation
$MinPts$	The number of proximity of a specific point
Eps	Spatial distance threshold
S	Data set
n	The number of all points
W_P, L_P	The width and length of every partition
m	The number of total partitions
p_i	The i-th data point
D_i	The ID of partition the i-th data point located in
lon_i, lat_i	The longitude and latitude of i-th data point
$d_i^t, d_i^b, d_i^l, d_i^r$	The distance between point p_i and top, bottom, left, right boundary of partition th point located in.
$P_{i,j}$	Partition located in the i-th row, the j-th column of all partitions
$D_{i,j}$	The ID of $P_{i,j}$

B. Clustering Algorithms

In order to sense demand hotspots, clustering algorithm is used to find areas of high density. Clustering methods can be classified into several categories, such as partitioning method, hierarchical method, density-based method. In the situation, the density-based method called Density-based spatial clustering of applications with noise (DBSCAN) is the most suitable. First, this density-based algorithm deals with noises better than partitional clustering or hierarchical clustering, therefore it can erase noise points to improve accuracy. As the result of K-Means method shown in Fig. 3, the blue clustering center deviates from its accurate location because of several noise points above it. Second, DBSCAN also has the ability in discovering clusters with arbitrary shapes. Finally, in contrast to some clustering algorithms, it does not require the predetermination of the numbers of clusters. As we know, we can not know the number of hotspots in our city in advance. It is natural and changeable. If we set it improperly, we may get inaccurate clustering centers as the red points shown in Fig. 3 because the parameter K in K-Means algorithm is not big enough. In DBSCAN, a spatial distance threshold Eps is used to define the proximity of two points. If the number of proximity of a specific point exceeds a predefined parameter $MinPts$, the point is considered as the core point of one cluster, and its proximity belongs to the same cluster. If the number of proximity is less than the parameter $MinPts$,

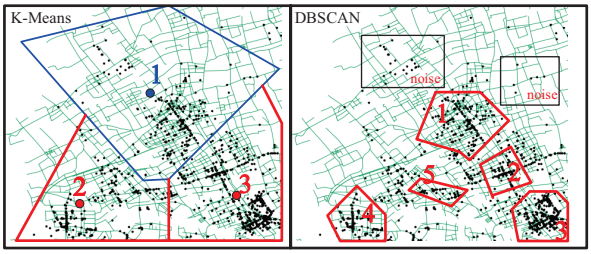


Fig. 3: Comparison between K-Means and DBSCAN

the point may be at the border of one cluster, or a noise point. Some main notations are described in Table I and the pseudocode of DBSCAN is given in Algorithm 1.

Algorithm 1 DBSCAN algorithm

```
1:  procedure DBSCAN(S, Eps, MinPts)
2:    for each unvisited points p ∈ S do
3:      mark p as visited
4:      N ← G(p, Eps)
5:      if |N| < MinPts then
6:        mark p as noise
7:      else
8:        C ← p
9:        for each point p' ∈ N do
10:         N ← N \ p'
11:         if p' is not visited then
12:           mark p' as visited
13:           N' ← G(p', Eps)
14:           if |N'| ≥ MinPts then
15:             N ← N ∪ N'
16:         if p' doesn't belong to any cluster then
17:           C ← C ∪ p'
```

III. An Improved Clustering Algorithm GD-DBSCAN

Although DBSCAN has been proven in its ability of processing very big databases, it costs lots of time. The function $G(\cdot)$ (shown in Algorithm 1) takes up most time spent in DBSCAN. $G(\cdot)$ is a function for neighborhood search. Our improved algorithm named Grid and KD-tree for DBSCAN (GD-DBSCAN) has the same idea with original DBSCAN. However, GD-DBSCAN uses a method of partitioning points to improve the speed of function $G(\cdot)$. Partition is a class consisting of ID, Neighbor and Data. Neighbor is a set of eight partitions around it including Top, Left, Right, Bottom, Top-Left and so on. Points of every partition are organized as kd-tree structure which helps accelerating searching. The method of partitioning is given in Algorithm 2.

Algorithm 2 PARTITION algorithm

```
1:  function P(S, W_P, L_P)
2:    mlon ← Min(lon), mlat ← Min(lat)
3:    X ← (Max(lon) − mlon)/L_P + 1
4:    Y ← (Max(lat) − mlat)/W_P + 1
5:    for all p_i ∈ S do
6:      D_i ← (lon_i − mlon)/L_P + (lat_i − mlat)/W_P * X
7:    end for
8:    while i < X, j < Y do
9:      D_{i,j} ← i + j * X
10:     for p_m ∈ P_{i,j} do
11:       store p_m as a kd-tree structure in parallel
12:     end for
13:   end while
14: end function
```

Partitioning the dataset reduces the search space of each point instead of scanning the whole dataset. For every iteration,

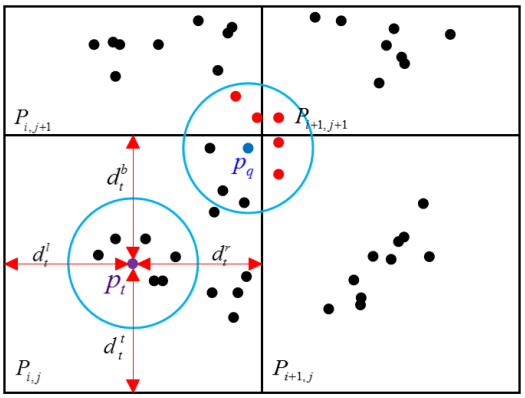


Fig. 4: Partition and Neighborhood search

it does not need to traverse all data. How many partitions to read is judged by comparing the distance between core point and the four boundaries of partition with parameter Eps. As shown in Fig. 4, when the purple core called p_t searches neighborhood points, because the distance d_t^t, d_t^b, d_t^l, d_t^r between p_t and the four boundaries are all less than the distance threshold Eps, it just needs to search the partition $P_{i,j}$ where it is located, ignoring points in other partitions. For another blue point called p_q, its bottom, right and bottom-right partitions should be selected into the searching queue apart from the partition that it is located in. So the searching range is partitions marked as $P_{i,j}$, $P_{i+1,j}$, $P_{i,j+1}$, $P_{i+1,j+1}$. This allows us to eliminate all the redundant points, and thus speeding up the searches of neighbor points. Kd-tree is a useful data structure for low-dimension space searches. Because points in different partitions has no relationship with each other, the process of constructing kd-tree and querying in kd-tree can both be executed in parallel. Details about method of getneighbors in GD-DBSCAN are described in Algorithm 3.

Suppose dividing all raw data into m partitions and there are N_i points for the ith partition. The time complexity of DBSCAN without kd-tree and partitions is $O(n^2)$. For DBSCAN without kd-tree but with partitions, q is the number of partitions every point searches. q is usually one or two, at most 4. So the Eq. 1 is correct.

$$\sum_{i=1}^{q} N_i < \sum_{i=1}^{m} N_i = n \tag{1}$$

The time complexity C of DBSCAN without kd-tree but with partitions is less than $O(n^2)$ whose proving is shown in (2).

$$C = \sum_{i=1}^{n} \sum_{j=1}^{q} N_j < n^2 \tag{2}$$

When we use kd-tree, the complexity has a decline and an overall runtime complexity of $O(n \log n)$ is obtained. The main time spent in DBSCAN with kd-tree is constructing tree and querying points within a range. Inserting a new point into a balanced kd-tree takes $O(\log n)$ time [9]. The complexity of range-searching operation in a kd-tree is still $O(\log n)$. Without partition, because of n points, the inserting process will cost $O(n \log n)$ and querying will cost $O(n \log n)$ as well. If we divide all data into m partitions, the time complexity C_I

Algorithm 3 GETNEI algorithm

function G(p_t, eps)
  $i \leftarrow (lon_t - \text{Min}(lon))/L_P$
  $j \leftarrow (lat_t - \text{Min}(lat))/W_P$
  compute distance between p_t and four boundaries of
  partition located p_t in $d_t^t, d_t^b, d_t^l, d_t^r$
  add $P_{i,j}$ to sp.
  if $d_t^t < eps$ **then** add $P_{i,j-1}$ to sp.
  else if $d_t^b < eps$ **then** add $P_{i,j+1}$ to sp.
  else if $d_t^l < eps$ **then** add $P_{i-1,j}$ to sp.
  else if $d_t^r < eps$ **then** add $P_{i+1,j}$ to sp.
  end if
  if $d_t^t < eps$ and $d_t^l < eps$ **then**
    add $P_{i-1,j-1}$ to sp.
  else if $d_t^t < eps$ and $d_t^r < eps$ **then**
    add $P_{i+1,j-1}$ to sp.
  else if $d_t^b < eps$ and $d_t^l < eps$ **then**
    add $P_{i-1,j+1}$ to sp.
  else if $d_t^b < eps$ and $d_t^r < eps$ **then**
    add $P_{i+1,j+1}$ to sp.
  end if
  for all $P_{m,n} \in sp$ **do**
    $pts \leftarrow$ find-in-kd-tree($P_{m,n}, eps$) in parallel
    add pts into $NeighborP$
  end for
  return $NeighborP$
end function

for inserting is computed in (3) which is obviously less than the original complexity.

$$\begin{aligned} C_I &= \sum_{i=1}^{m} N_i \log N_i = \sum_{i=1}^{m} \log N_i^{N_i} \\ &= \log \prod_{i=1}^{m} N_i^{N_i} < \log \prod_{i=1}^{m} \max(N_i)^{N_i} \\ &= \log \max(N_i)^{\sum_{i=1}^{m} N_i} = \log \max(N_i)^n \\ &= n \max(N_i) < n \log n \end{aligned} \tag{3}$$

For querying, the time complexity which is computed in (4) is less than that without partitioning when q equals one. When q does not equal one, we can also query in parallel to get the same result. Both inserting and querying with partitioning have a lower time complexity compared with algorithm without partitioning. Therefore the performance has been improved and this algorithm is proved to save time.

$$\begin{aligned} C_Q &= \sum_{i=1}^{n} \sum_{j=1}^{q} (\log N_j) \\ &< \sum_{i=1}^{n} \log \sum_{j=1}^{m} (N_j) = n \log n \quad \text{if } (q = 1) \end{aligned} \tag{4}$$

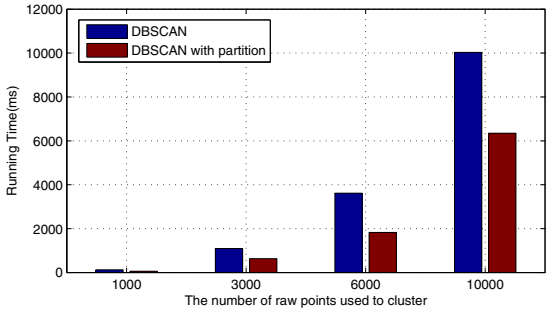
IV. Experiment Result

In this section, we evaluate the performance of GD-DBSCAN. We compare it with DBSCAN. We have implemented GD-DBSCAN based on an implementation of kd-tree and different versions of DBSCAN in C++. All experiments have been run on a workstation equipped with 120 GB memory and two Intel Xeon 2.60GHz CPUs consisting of 12 cores. We select different numbers of location records on April 1, 2015 as a test database and use it as a benchmark. Since GD-DBSCAN and DBSCAN are clustering algorithms of same types, their clustering results are almost same, we compare their running time. As shown in Table II, the running time of

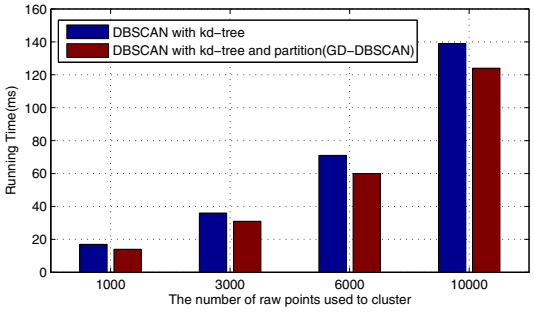
TABLE II: Running time

Number of points	1000	6000	10000	90000
DBSCAN (ms)	126	3617	10030	785758
GD-DBSCAN (ms)	14	60	124	3120

GD-DBSCAN has a big decline compared with DBSCAN and the gap of running time between them is becoming wider with the growing of data set. For DBSCAN without kd-tree, the use of partitioning saves about 30% of time as shown in Fig. 5(a). For DBSCAN with kd-tree, the time complexity declines from $O(n^2)$ to $O(n \log n)$, if also adding the method of partitioning, GD-DBSCAN will save about 10% of time again as shown in Fig. 5(b). In detail, we compare the running time of inserting



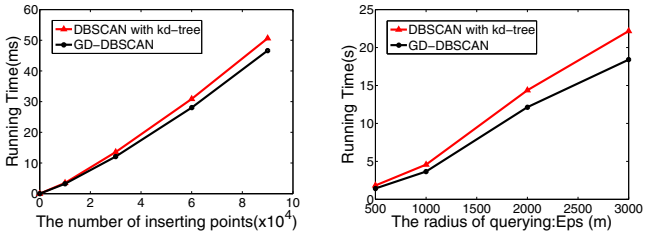
(a)



(b)

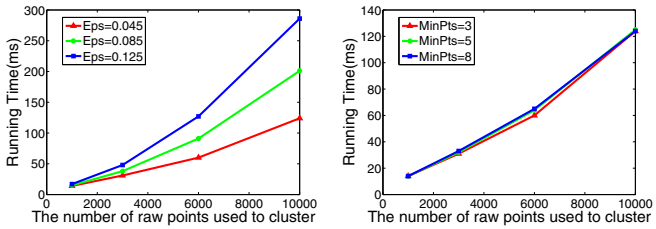
Fig. 5: Comparison of Running time between GD-DBSCAN and others

and querying process in GD-DBSCAN with that in DBSCAN. As shown in Fig. 6, the running time of inserting process in GD-DBSCAN is always less than that in DBSCAN and the running time of querying process in GD-DBSCAN is less than that in DBSCAN. Moreover, we research the effect of the parameter $MinPts, Eps$. As Fig. 7 shows, Eps has a great influence in running time while $MinPts$ has little effect in it. Another advantage of GD-DBSCAN is that this algorithm



(a) (b)

Fig. 6: (a) Running time of inserting data to kd-tree (b) Running time of querying for 90000 records



(a) (b)

Fig. 7: (a) The effect of different Eps ($MinPts = 3$) (b) The effect of different $MinPts$ ($Eps = 0.045$)

is suitable for parallel computing in multi cores because data set can be divided into lots of parts.

In order to sense hotspots, we use this algorithm to cluster about ten thousand pick-up records which have been extracted from all uploaded records on April 1, 2015. Since GD-DBSCAN will not generate cluster center automatically, we use the average of longitude and latitude in a cluster as the center. As the Fig. 8 shows, the red points are centers of the top 100 clusters.

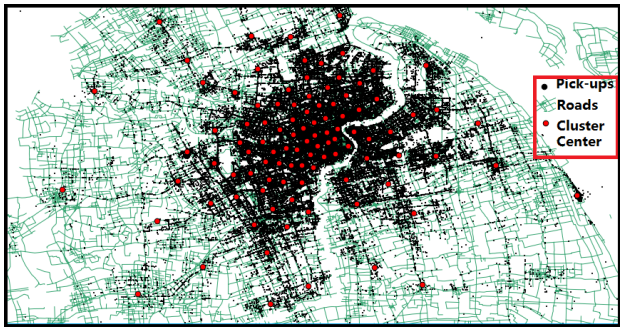


Fig. 8: Clustering centers on April 1, 2015 Shanghai

V. Conclusion

This paper has presented the whole framework of exploiting taxi demand hotspots by extracting pick-ups from taxi trajectory information and clustering pick-ups to get hotspots. For big vehicular data, we propose an improved clustering algorithm called GD-DBSCAN. This algorithm can not only keep the advantages of DBSCAN but also reduce the computational complexity. The experiment shows GD-DBSCAN has an improvement of at least 10% in performance compared with original fastest DBSCAN. In the future, with the help of hotspots we continue to research about predicting pick-up rate for different hotspots and modeling taxi dispatching.

VI. Acknowledgements

This work was partially supported by NSF of China under the grant nos. U1405251, 61521063 and 61273181, and by New Century Excellent Talents in University Program under the grant no. NCET-13-0358.

References

[1] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "Predicting taxi–passenger demand using streaming data," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 3, pp. 1393–1402, 2013.

[2] R. Du, C. Chen, B. Yang, N. Lu, X. Guan, and X. Shen, "Effective urban traffic monitoring by vehicular sensor networks," *IEEE Transactions on Vehicular Technology*, vol. 64, no. 1, pp. 273–286, 2015.

[3] D. Zhang, L. Sun, B. Li, C. Chen, G. Pan, S. Li, and Z. Wu, "Understanding taxi service strategies from taxi gps traces," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 1, pp. 123–135, 2015.

[4] L. Moreira-Matias, R. Fernandes, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "An online recommendation system for the taxi stand choice problem (poster)," in *Proc. IEEE on Vehicular Networking Conference (VNC)*, 2012, pp. 173–180.

[5] D. Liu, S.-F. Cheng, and Y. Yang, "Density peaks clustering approach for discovering demand hot spots in city-scale taxi fleet dataset," in *Proc. IEEE 18th International Conference on Intelligent Transportation Systems (ITSC)*, 2015, pp. 1831–1836.

[6] G. Pan, G. Qi, Z. Wu, D. Zhang, and S. Li, "Land-use classification using taxi gps traces," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 1, pp. 113–123, 2013.

[7] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-dbscan: A gpu accelerated algorithm for density-based clustering," *Procedia Computer Science*, vol. 18, pp. 369–378, 2013.

[8] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan, "MR-dbscan: a scalable mapreduce-based dbscan algorithm for heavily skewed data," *Frontiers of Computer Science*, vol. 8, no. 1, pp. 83–99, 2014.

[9] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o (n log n)," in *Proc. IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 61–69.