Cost-Efficient Partitioning of Spatial Data on Cloud

Afsin Akdogan[1] Saratchandra Indrakanti[2*] Ugur Demiryurek[1] Cyrus Shahabi[1]

[1] Computer Science Dept.
University of Southern California
Los Angeles, CA, USA
{aakdogan, demiryur, shahabi}@usc.edu

[2] eBay Inc.
San Jose, CA, USA
sindrakanti@ebay.com

***Abstract*—With the rise of mobile technologies (e.g., smart phones, wearable technologies) and location-aware Internet browsers, a massive amount of spatial data is being collected since such tools allow users to geo-tag user content (e.g., photos, tweets). Meanwhile, cloud computing providers such as Amazon and Microsoft allow users to lease computing resources where users are charged based on the amount of time they reserve each server, with no consideration of utilization. One key factor that affects server utilization is partitioning method especially in data-driven location-based services. Because if the data partitions are not accessed, the servers storing them remain idle but the user is still charged. Whereas, existing spatial data partitioning techniques aim to 1) cluster spatially close data objects to minimize disk I/O and 2) create equi-sized partitions. On the contrary, the objective is different for cloud given the current pricing models. In this paper, we propose a novel cost-efficient partitioning method for spatial data where an increase in the servers' utilizations yields less number of servers to support the same workload, thus saving cost. Extensive experiments on Amazon EC2 infrastructure demonstrate that our approach is efficient and reduces the cost by up to 40%.**

***Keywords*-spatial databases; data partitioning; cloud computing**

I. INTRODUCTION

Cloud computing has become the common practice across businesses. According to Cisco, the majority of the workload at global scale has shifted to cloud infrastructures for the first time in history - 51% of all workloads are processed in cloud versus 49% in traditional IT space [8]. Cloud computing enables a considerable reduction in operational expenses by providing flexible resources that can scale up and down. On the other hand, the total cost proportionally increases with the total number of servers even if they are not fully utilized. This is because the largest cloud computing service providers (e.g., Amazon EC2 [1], Microsoft Azure [11], Google Compute [14]) charge users based on the total amount of time for each server with no consideration for the percentage of utilization, as long as the servers are running. Thus, it is crucial to fully exploit each and every server as much as possible.

The partitioning method significantly impacts server utilization in data-driven location-based applications. This is because if the data partitions are not accessed by users/applications, it is likely that the servers storing those partitions remain idle. On the other hand, most of the existing partitioning approaches adapt schema-based strategy aiming to generate equi-sized partitions [6, 9]. As a practical example, MongoDB (a popular NoSQL database) only balances the amount of data at each server without considering whether the data is accessed or not [7]. There are a few recent studies that utilize the workload to co-locate frequently accessed data objects together [10]. However, they aim to increase query throughput by minimizing distributed transactions.

With location-based services, workload skew becomes a severe problem as there is a very high correlation between geo-coordinates (latitude, longitude) and access patterns of the data objects. A practical example is Yelp -a popular online urban guide with over 100 million monthly users. Users typically use Yelp during daytime in their time zone to find local businesses around them and may not use this service as frequently during nighttime. When a conventional spatial partitioning technique is used to distribute such a geospatial dataset across multiple cloud servers, this correlation causes a remarkable amount of underutilization of the servers. Suppose we distribute Yelp's data across 100 servers using Voronoi diagram [6], where each server stores a disjoint subset of the data. Suppose as a result of this partitioning 50 servers are allocated to store the data associated with USA and the rest for Europe. In such a case, either batch of 50 servers will be idle almost half of the day due to infrequent accesses at midnight in their time zone.

We would like to note that there are several cloud data stores where users pay only for the resources they actually consume, unlike the *time-based* pricing model. For example, Amazon SimpleDB and DynamoDB adapt pay-per-use pricing model and charge users for each read and write operation. However, these services are not appropriate to manage multi-dimensional spatial data because they mostly support simple key-value based put-get operations with either no or very limited spatial query support. Whereas, with spatial data, it is expected and crucial to provide ***advanced querying*** features (e.g., range, *k* nearest neighbor, skyline). These reasons compel the location-based services to partition, index and query their own data. Another practical solution to reduce server cost is utilizing *elasticity* of cloud infrastructures. For example, Amazon's Auto Scaling engine *(AAS)* [4] allows

[*] Work was performed while the author was at University of North Texas

users to 1) reduce the number of running servers and 2) replace large servers with small ones based on user-defined rules (e.g., scale down to small server if CPU utilization is less than 40%). Even though this reactive scaling model works well for many applications, there are several problems with it when applied to data-driven applications. 1) Amazon EC2 provides coarse-grained servers, where each server is twice more powerful than a lower configured one in the same server group [4] (see Table 1) and waiting for a certain utilization threshold to scale down is sub-optimal from cost perspective. 2) Once scaled-down, the chances of running small servers underutilized is high. One option is to migrate the data from an underutilized server and shut that server down. However, limited storage capacity of servers might eliminate this solution or even prevent the server from scaling down. For example, smallest general-purpose server (m3.medium) in Amazon has only 4GB of storage. 3) It has been shown that this model performs poorly on variable traffic patterns where different times of the day have different workload characteristics and fleet sizes which are likely occur in location-based applications [12]. These reasons argue that partitioning method causes workload skews and manually defining optimal scaling rules is *non-trivial*.

In this paper, we propose a novel cost-efficient partitioning (***CEPS***) method for spatial data that by increasing the servers' utilizations yields less number of servers to support the same workload, thus saving cost. Our approach considers both 1) spatial proximity among the data objects and 2) access patterns (workload) associated with the objects (e.g., how many times an object is accessed at a particular time) that can be easily obtained as described in [10]. More specifically, our three-step approach first splits the dataset into small partitions based on spatial proximity, and then effectively merge the partitions with diverse access patterns to mitigate workload skew using a hierarchical clustering technique (*AHR*). The problem with *AHR* is that it achieves a uniform workload locally at each server but it might get stuck in local optimum and leave the server underutilized. To overcome this issue, we further improve these clusters by adapting an intelligent tabu search approach which **avoids local optimum**, and hence minimizes the total net idle of all servers.

II. RELATED WORK

A. Spatial Databases

Most of the existing approaches are designed for a centralized paradigm where all spatial operations are performed on a single server. Among these tree-based, R-tree [16] and Quad-tree [13] are the most prominent index structures which are implemented in commercial products. Conversely, several other techniques employ a flat structure such as Voronoi diagram [5, 6, 19] which decomposes the space into disjoint subsets. Likewise, the method in [9] converts spatial objects into a graph and divide the graph using min-cut algorithm to maximize the distance among the partitions. The main problem with these techniques is that they all suffer from workload skew as the main objective is clustering spatially close objects. In addition, tree-based methods cannot directly be applied in a distributed system as whichever server holds the higher levels of the hierarchies (e.g., root) becomes the system bottleneck [2].

There are few recent studies that handle spatial data in the context of distributed systems. RT-CAN [2] integrates a CAN-based [3] routing protocol with an R-tree based indexing scheme to support multidimensional query processing in a cloud system. However, CAN adapts a storage-oriented balancing strategy which leaves approximately half of the cluster idle if the application is used in different time zones.

B. Parallel Databases

The key differences among the existing partitioning techniques in parallel DBMSs are in 1) identifying the partitioning attributes and 2) the search process used to find the optimal partitioning strategy. For example, Schism seeks to minimize distributed transactions [10]. For a given database, Schism populates a graph containing a separate vertex for every tuple and creates an edge between two vertices if the tuples that they represent are co-accessed together in a transaction. The edge values proportionally increase with the common transactions. It then produces balanced boundaries using a graph partitioning technique that minimize the number of cross partition edges. The work in [17] employs a branch-and-bound algorithm to search for table partitioning and replication choices for shared-nothing, disk-based DBMSs. The main problem with these approaches is that they put co-accessed tuples into the same partition which reduces the network I/O and query coordination cost resulting in increased query throughput. In addition, these techniques only run on static datasets, where updates are not taken into account.

C. NoSQL Databases

Recently, several NoSQL database products such as MongoDB and HBase provide scalable solutions by giving up on strong consistency (ACID properties). These products have *geospatial* data support as well; however, they still use similar partitioning techniques as parallel databases (e.g., hash or range partitioning). With MongoDB, dataset is partitioned into smaller chunks and scattered across the servers. MongoDB employs a static *storage-oriented* partitioning strategy, with which if too many partitions are assigned to a server, some of those partitions are migrated to other servers. The main problems with this approach are as follows. First, data migration impacts the performance. Second, it only balances the amount of the data at each server without considering if the data is accessed or not. Big data processing frameworks such as SpatialHadoop might be relevant; however, the main objectives in such systems are achieving scalability, reducing network I/O and balancing the load across servers given a long-running data processing task.

III. CEPS:COST-EFFICIENT PARTITIONING

In this section, we present our workload-based partitioning approach. Given a dataset *D* consisting of spatial data objects

in 2D space and a set of N servers with capacity θ_k, $0 \leq k < N$, we would like to divide D into N disjoint partitions where each partition is assigned to a server, has a flat (uniform) access pattern and fully utilizes its server. This optimization problem manifests itself into an NP-hard problem [21], with a solution space of N^O where O is the number of objects in D. In addition, spatially close objects need to be stored together which adds a new dimension to the problem and makes it even more complicated. To overcome these challenges, we propose a *three* step approach.

In the **first** phase, we split the dataset into p number of small partitions based on spatial proximity using PR kd-tree [20] where $p > N$ (p is much larger than N) and each p consists of certain number of objects (e.g., 10,000). Generating such small partitions is sufficient to group co-accessed data objects together that reduces the local disk I/O at each server during query processing. For example, in real-world applications such as Yelp, users only query a limited region around them (e.g., show me the restaurants within 5 miles). We note that spatial clustering techniques such as *k-means* can be employed at this phase as well; however, it would be inefficient to maintain the partition borders as the dataset is updated over time.

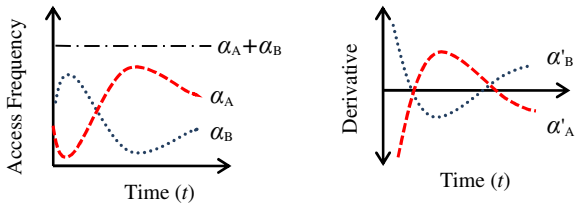
In the **second** phase, we represent each partition p with its access pattern $\alpha_p(t)$ at time t. Subsequently, we generate N number of super partitions (cluster) using an agglomerative hierarchical clustering (*AHR*) approach that combines partitions into clusters, where each cluster c_k corresponds to a server $k \in N$. Now the challenge is how to mitigate the spikes and create uniform clusters with uniform workloads at each cluster. To this extent, we group complementing (diverse) patterns in the same cluster. As shown in Figure 1(a), diverse patterns α_A and α_B form a flat pattern when summed up ($\alpha_A+\alpha_B$) and the sum of derivatives (see Figure 1b), $\alpha'_A(t)+\alpha_B'(t)$, a measure of their *diversity*, is *0*. Therefore, we define a *flatness* metric δ, which is the cumulative slope (derivatives) of the patterns, to identify diverse patterns and maximize diversity within each cluster. The local objective specific to the cluster c_k is to minimize the *flatness* metric δ_k, which is defined as:

$$\delta_k = \left| \sum_{i \in c_k} \frac{d\alpha_i(t)}{dt} \right| \quad (1)$$

In the **third** phase, we improve the solutions generated in the previous phase. The problem with *AHR* is that it might get stuck at *local optima*. More specifically, minimizing δ_k for each cluster c_k itself only achieves a balanced workload on the corresponding server. However, we also aim to maximize utilization or in other words, minimize net idle-time. The idle-time τ_k for the server k is defined as the server time unutilized and is given by:

$$\tau_k = \theta_k - \sum_{i \in c_k} \int_t \alpha_i(t) \quad (2)$$

The net idle-time of all the servers in the partition plan is measured by global fitness metric, T, for the partition plan. *AHR* aims at maximizing diversity of the access patterns



a) Access patterns **b)** Derivatives

Figure 1. a) Two diverse access patterns α_A and α_B with workload skew. b) Derivatives of α_A and α_B (α'_A, α'_B), which have a 0 sum, ensuring maximum diversity.

within a cluster and does not primarily focus on improving the global fitness metric. Although the constraint of server capacity drives *AHR* towards a balanced utilization across servers, it may not be optimal with respect to the global fitness metric, as will be explained shortly. In order to maximize utilization, idle-time across all servers is minimized. The global fitness metric for the optimization problem, T, is computed as:

$$T = \sum_{k \in S} \tau_k \quad (3)$$

Tabu search facilitates computing a near-optimal partition plan that minimizes the global fitness metric by improving the solution produced by *AHR*. It prevents the case of a local minimum being considered an optimal solution by ensuring that a series of ensuing successive iterations do not improve the solution. In addition, it reduces the search space by avoiding swapping those access patterns which did not yield improvements to the fitness metric during neighborhood computation. In the following sections we will discuss our clustering and tabu search techniques, respectively.

A. *Agglomerative Hierarchical Clustering*

The algorithm (Algorithm 1) starts with an input set of p singleton clusters, where each cluster c_i, where $0 \leq i < p$, corresponds to its access pattern. The clusters are iteratively merged to form N resulting clusters. During each merge operation, two clusters c_i, c_j are merged if the flatness metric δ_m for the cluster c_m formed by merging c_i, c_j is the minimum among all feasible pairs, i.e. $\delta_m = min(\delta_{pq})$, where p and q are clusters. A pair c_i, c_j is deemed feasible if the combined

Algorithm 1: Agglomerative hierarchical clustering (*AHR*)

Input: Set of p singleton clusters $C = \{c_i : \alpha_i(t) \in c_i, 0 \leq i < p\}$, desired number of clusters N
Output: Set C_N of N resultant clusters

```
num_clusters = p
C_N = C
while num_clusters > N do
    pf = compute_pairwise_flatness(C_N)
    (c_i, c_j) = get_minimum(pf)
    if exceeds_capacity(c_i, c_j) then
        (i, j) = get_minimum(C_N – max(c_i, c_j))
    end if
    c_m = merge_clusters(c_i, c_j)
    C_N = C_N + c_m - c_i - c_j
    num_clusters = num_clusters – 1
end while
return C_N
```

access frequency of c_i, c_j at any time is lesser than the capacity of a server. This hierarchical merging of clusters is iteratively performed until N clusters are obtained. This greedy approach produces clusters that satisfy local constraints by merging diverse access patterns. However, the solution could be further improved to obtain a reduced net idle-time by improving the global fitness metric.

B. Tabu Search

The clustering C_N of N clusters produced by Algorithm 1 is used to initiate Algorithm 2, based on Tabu search algorithm. Local search methods tend to repeatedly cycle through a set of visited solutions and get stuck at local optima. Tabu search uses the idea of short-term memory to prevent cycling when moving away from local optima through non-improving moves. Recent search history is maintained as a list of *tabu* (forbidden) moves, to forbid the search from revisiting recent moves. Non-improving moves are allowed due to the tabu list, in contrast to classic local search, thus avoiding the pitfalls of local optima.

Algorithm 2 improves the current best solution by exploring its neighborhood in the solution space for better solutions subject to certain tabu criteria. Improvement made to a solution is determined by the global fitness metric T (see Equation 3). The neighborhood of a solution is computed by applying local point-wise transformations to the current solution to generate a set of candidate solutions that constitute the neighborhood. Those transformations that have been applied in the recent past are added to a memory structure referred to as the tabu list, and are avoided in computing newer solutions. Maintaining the tabu list prevents cycling through solutions visited in the recent past and being limited to local optima. The current solution is replaced by an improved solution when the global fitness metric improves subject to the criteria of tabu search. The best solution discovered before the search terminates is returned. Below we first describe the important aspects of the algorithm namely neighborhood, tabu list and termination conditions.

Neighborhood: The neighborhood of a solution s is defined as the set of solutions S_l obtained by performing move transformations on a subset l of access patterns. A move transformation on a solution s corresponding to an access pattern a is defined as moving pattern a from its assigned server in solution s to another randomly chosen server, resulting in a solution $s' \in S_l$, the neighborhood of s. The set S_l is selected such that the flatness metric of clusters that are a part of this operation, changes by less than η% where η>0.

Tabu List: Tabu list is the list of recent moves. A move m is added to the tabu list if the solution that is a consequence of m is selected to be the current solution. A solution $i \in S_l$, the neighborhood of the solution s, is selected to be the current solution if its fitness metric T_i is an improvement from that of the current solution T_s, i.e., $T_i < T_s$ for this minimization problem and the move transformation that results in i is not part of the tabu list. A move m is added to the tabu list if the solution that is a consequence of m is selected to be the current solution. A move is dropped from the tabu list when the size of the list is greater than a pre-defined value on a first-in-first-out basis.

Algorithm 2: Tabu search

```
Input: Set C_N of N clusters
Output: Set C_N of N clusters of improved fitness metric T
tabulist = null
best_solution = C_N
current_solution = C_N
1: while not stopping_criterion
2:     neighbors = neighborhood(current_solution)
3:     for neighbor in neighbors
4:         if move(neighbor) in tabulist then
5:             if T(neighbor) < T(best_solution) then
6:                 best_solution = neighbor
7:                 current_solution = neighbor
8:                 break
9:         else if T(neighbor) < T(current_solution) then
10:            current_solution = neighbor
11:            if T(neighbor) < T(best_solution) then
12:                best_solution = neighbor
13:            add_to_tabulist(move(neighbor))
14:            if sizeof(tabulist) > max_size then
15:                pop_tabulist()
16:            break
17:    end for
18: end while
19: return best_solution
```

Termination: The stopping criterion for the algorithm is the failure to produce a feasible improved solution in a series of s, $s \in \mathrm{N}^+$, consecutive iterations after an improvement.

IV. EXPERIMENTAL EVALUATION

In this section, we first present our experimental setup that includes the details about the hardware specifications, metrics, datasets, workloads, and competitor approaches. Subsequently, we present the results and discussions.

A. Experimental Setup and Methodology

We conducted our experiments on Amazon EC2's general-purpose servers. Table 1 (see Section 1) shows the hardware specifications and prices of available servers. We use s*erver cost* to evaluate the effectiveness of CEPS algorithms.

Dataset & Workload: We use two different datasets: 1) real-world Gowalla dataset (GD) [15] and 2) synthetic dataset (SD) which can capture the real-world scenarios where objects cluster around certain location (e.g., cities, point of interests). Gowalla is a location-based social network allowing users to check in and GD includes 6442890 check-ins at 1280969 different locations. Since the number of check-ins is relatively small and it span across almost a two year period, we increase the check-in number 1000 times for every timestamp while keeping other parameters fixed, thus preserving the existing workload distribution over time.

With SD dataset, we use Gaussian distribution where the objects are clustered around hotspots (hs) which are initialized at random locations on land coordinates excluding seas and oceans. We vary the number of hotspots from 10 to 1000 where the dataset with 10 hotspots is the most skewed and the one with 1000 hotspots is uniform. The distance from

the object to the hotspot follows a Gaussian distribution. In addition, to simulate crowded and less crowded cities, we assign a number s (denoting size) to each hotspot in the range of [1, 100]. Higher s value yields more crowded cities and the largest one will be 100 times larger than the smallest. Furthermore, rather than randomly assigning an s value to a hotspot, we consider the number of internet users [18] in the city where the hotspot is located. Specifically, s value of a hotspot is calculated based on the normalized value of its corresponding internet user population. Obviously, this approach yields more realistic data distributions since it avoids to generate large clusters where no one lives in or there is no internet user. Finally, SD dataset consists of 40 billion objects in 2D space where each object has an x and y value.

We generate workloads with cosine and zipf distributions. With cosine, we fix the wave frequency to 1 representing the workload pattern of a day. With zipf, we vary the skew factor (sf) from 0 (uniform) to 3 (skewed). Workloads are associated with the data density. Therefore, as the density increases the magnitude of the workload increases proportionally and we obtain skewed workload at different locations.

1) Competing Approaches: We evaluate the following approaches in our experiments.

- **GP:** Grid based (spatial) partitioning where each server corresponds to a partition and the workload *is not considered*. The state-of-the-art distributed spatial index, RT-CAN [2], adapts this strategy.
- **GP-R:** Grid based partitioning where more *granular* partitions are distributed across the servers **randomly**. This strategy is commonly applied in practice.
- **GP-AAS:** Grid based partitioning combined with Amazon's Auto Scaling engine (AAS) which considers *work load* to adjust the amount of resources. This state-of-the-art reactive model is Amazon EC2's primary solution to scale up and down to reduce server cost. AAS requires users to 1) select a monitoring metric and 2) specify their own scaling policy (e.g., when to shut down a server or scaling up to a higher configured server). For our experiments, we monitor *CPUUtilization*, *DiskWriteOps and DiskReadOps* to trigger scaling actions. We also define an effective scaling policy consisting of the following rules. We downgrade to a lowered configuration if the usage is less than 50% for the past 30 minutes. This threshold (50%) is chosen in accordance with EC2 server configurations since a lower profiled server is always half as powerful as the current one (see Table 1 in Section 1). We wait for 30 minutes to avoid workload fluctuations resulting in contradictory scaling up and down requests. If any of the monitoring metrics attains 100% capacity, we can either upgrade to a higher level server or split the data and migrate it to another server. Since the higher level server is twice more costly, in order to save cost we split the data into two, start a new server with the lowest configuration possible which can store the data. This strategy is adapted by distributed hash tables as well that aim to avoid hotspots by sharing data zones [3]. Finally, in order to shut down underutilized servers, we check the last 30 minutes and merge two servers if one of them can handle the workloads of both servers and has enough capacity to store the data.
- **CEPS-:** Our approach where tabu search is disabled.
- **CEPS+:** Our approach where tabu search is enabled.

We discretize the workload patterns into 5 minute intervals for our approaches.

B. Experimental Results

Given real-world GD dataset, we first investigate how the server cost varies for competing solutions. In this specific experiment, we use Monday workload patterns and set the number of objects at each partition to 5,000 for our approach. For GP, we set the grid capacity to 100,000, which is a relatively small number due to the fact that the size of the actual data is small as well. On the other hand, the workload on each partition is large enough to saturate the computation capacity of the servers. We ensure that each partition is stored on the server with lowest configuration that can handle the workload to reduce the cost and eliminate *over-capacity* problem. As shown in Figure 2, CEPS+ outperforms other approaches and conventional grid-based partitioning significantly increases the server cost due to severe workload skew. GP-AAS is insufficient to adjust itself to fluctutions. During morning and afternoon, servers run a little over 50% capacity for several hours which is the worst case for GP-AAS as underutilized servers cannot be downgraded.

We now vary the number of hotspots on SD dataset, fixed the cluster size (*s*) and investigate the impact of data skew. As the results in Figure 3 show, CEPS+ reduces server cost approximately 25% and 40% with respect to GP and GP-AAS. The reason is cosine workload is the best case scenario for CEPS+ since access patterns are perfectly complementing each other. On the contrary, CEPS- suffers from cosine workload as it directly minimizes the intra-cluster diversity due to these matching patterns. We also observe that, hotspot count does not have a substantial impact on the server cost since the patterns still balance each other.

In the next experiment, we create clusters with various sizes by assigning *s* values in the range of [1, 100]. As Figure 4 shows, for small number of hotspots, CEPS+ still outperforms other approaches but the total saving on cost is lower. This is because there are not enough patterns that can completely balance the crowded regions. However, as we increase the number of hotspots, CEPS+ still provides approximately 20-25% cost saving.

Now we experiment with a different workload (zipf), fix the cluster size, set the number of hotspots to 1000 and vary the skew factor from 0 (constant) to 3 (skew). Skewed workload indicates that the data is accessed for a short period of time throughout a day. As Figure 5 illustrates, when the workload is constant, CEPS+ outperforms other approaches since all the patterns complement each other. Similar to cosine workload CEPS- suffers from maximized intra-cluster

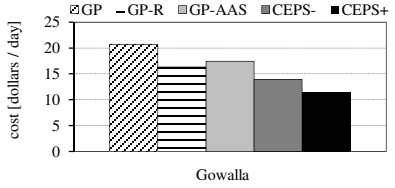


Figure 2. Server cost of GD.

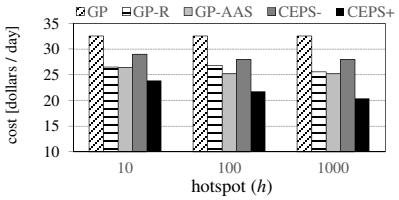


Figure 3. Impact of data skew (SD, Cosine, *s*= fixed)

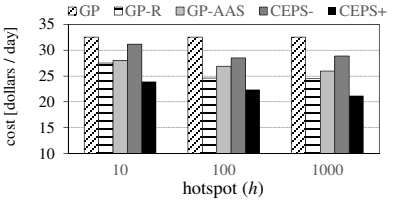


Figure 4. Impact of cluster size (SD, Cosine, s=[1,100])

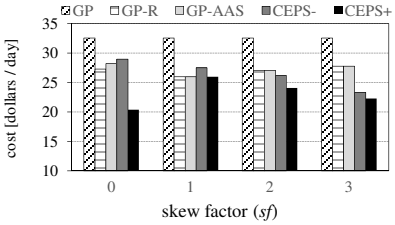


Figure 5. Impact of workload skew (SD, Zipf, *hs*=1000)

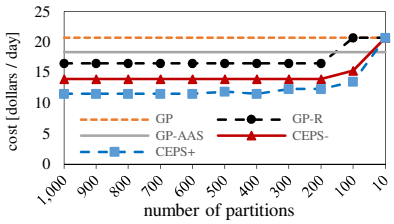


Figure 6. Impact of partition number (GD)

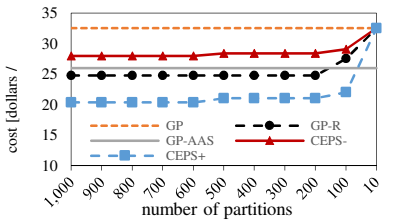


Figure 7. Impact of partition number (SD, *hs=1000*, *s*=fixed)

diversity. We also observe that with low-skew (*sf*=1), the gain of CEPS+ slightly decreases with respect to GP-AAS.

In the next experiment, we investigate the impact of the number of partitions (p) on server cost by varying the number of objects at each partition on GD dataset. As shown in Figure 6, the total cost slightly increases similar to a step function as the number of partitions decreases and when the p is very small (e.g., 10) the total costs are the same. This is because fine-granular partitions (access patterns) improve the quality of AHR clustering. In addition, CEPS- is slightly better than GP-R. We also observe that after a few hundred partitions the result does not change substantially. One may argue that small partitions degrades query performance; however, with spatial data, small partitions are sufficient to group co-accessed data objects together that reduces the local disk I/O at each server during query processing. For example, in real-world applications, users only query a limited region around them (e.g., show me the Starbucks shops within 15 miles). We repeat the same experiment on SD dataset with cosine workload where h=1000 and s is fixed. As Figure 7 illustrates, as long as we generate more than 100 partitions, we can reduce the server cost up to 40%.

V. CONCLUSION

Our problem is motivated by several pragmatic considerations such as time-based pricing models of the cloud providers, limitations in storage services that do not support advanced queries required by location-based services, complex configuration of scaling engines and coarse-grained servers. Our experiments show that CEPS reduces cost by up to 40% without trading off query performance.

VI. ACKNOWLEDGMENT

This research has been funded in part by NSF grants IIS-1115153 and IIS-1320149, the USC Integrated Media Systems Center (IMSC), and unrestricted cash gifts from Google, Northrop Grumman, Microsoft, and Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the sponsors such as the National Science Foundation.

VII. REFERENCES

[1] S. Agrawal, S. Chaudhuri, A. Das, and V. Narasayya. Automating layout of relational databases. In *ICDE*, 2003.
[2] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. 2010. Indexing multi-dimensional data in a cloud system. In *SIGMOD, 2010.*
[3] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.
[4] Amazon EC2, docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html
[5] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based Geospatial Query Processing with MapReduce. In *CloudCom*, 2010
[6] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. Wiley, 2000.
[7] MongoDB, http://www.mongodb.org/
[8] Cisco, newsroom.cisco.com/release/1274405
[9] S. Shekhar, C. Lu, S. Ravada , S. Chawla. Optimizing Join Index Based Spatial-Join Processing: A Graph Partitioning Approach. In Symposium on Reliability in Distributed Software, 1998.
[10] C. Curino, Y. Zhang, E. Jones, and S. Madden. Schism: a workload-drive approach to database replication and partitioning. In *VLDB*, 2010.
[11] MS Azure, azure.microsoft.com/en-us/pricing/details/virtual-machines
[12] Netflix,http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html
[13] R. A. Finkel, J. L. Bentley. Quad trees a data structure for retrieval on composite keys. In Acta Informatica 4, 1974.
[14] Google Compute Engine,cloud.google.com/products/compute-engine/
[15] Gowalla, snap.stanford.edu/data/loc-gowalla.html
[16] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.
[17] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, 2011
[18] Internet statistics, www.internetworldstats.com/stats.htm
[19] A. Akdogan, C. Shahabi, and U. Demiryurek. ToSS-it: A Cloud-Based Throwaway Spatial Index Structure for Dynamic Location Data. In *MDM*, 2014.
[20] J. Orenstein. Multidimensional tries for associative searching. Information Processing Letters, 14(4): 150-157, 1982.
[21] N. Karmarkar and R. M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *FOCS*, 1982.