Spatial Online Sampling and Aggregation

Lu Wang[1], Robert Christensen[2], Feifei Li[2], Ke Yi[1]
[1]Hong Kong University of Science and Technology, [2]University of Utah
{luwang, yike}@cse.ust.hk {robertc, lifeifei}@cs.utah.edu

ABSTRACT

The massive adoption of smart phones and other mobile devices has generated humongous amount of spatial and spatio-temporal data. The importance of spatial analytics and aggregation is ever-increasing. An important challenge is to support *interactive exploration* over such data. However, spatial analytics and aggregation using all data points that satisfy a query condition is expensive, especially over large data sets, and could not meet the needs of interactive exploration. To that end, we present novel indexing structures that support spatial online sampling and aggregation on large spatial and spatio-temporal data sets. In spatial online sampling, random samples from the set of spatial (or spatio-temporal) points that satisfy a query condition are generated incrementally in an online fashion. With more and more samples, various spatial analytics and aggregations can be performed in an online, interactive fashion, with estimators that have better accuracy over time. Our design works well for both memory-based and disk-resident data sets, and scales well towards different query and sample sizes. More importantly, our structures are dynamic, hence, they are able to deal with insertions and deletions efficiently. Extensive experiments on large real data sets demonstrate the improvements achieved by our indexing structures compared to other baseline methods.

1. INTRODUCTION

The increasing presence of smart phones and other mobile devices has generated humongous amounts of spatial and spatio-temporal data. Spatial and spatio-temporal analytics and aggregation over such data has become a building block for many such applications. As a result, their importance cannot be overemphasized. Even though various forms of spatial and spatio-temporal analytics and aggregations have been extensively studied in the field, the ever-increasing size of spatial and spatio-temporal data sets introduces new challenges. In particular, when the underlying data set is large, reporting all points that satisfy a query condition could be expensive, since there could be simply too many points that satisfy a query. The CPU cost of performing an analytical task or computing an aggregation using all these points adds additional overhead, and may not scale well with large numbers of points. Hence, waiting for the exact analytical or aggregation results may take a long time.

Proceedings of the VLDB Endowment, Vol. 9, No. 3

The good news is that often users do not need exact results; instead, they are happy with approximations, especially if these approximations come with quality guarantees. It is even better if the approximation quality gradually improves over time in an online fashion, while the query is being executed. This observation motivates us to design a *knob* that allows users to adjust the trade-off between query cost and approximation quality. It is important to make this knob online and let users be able to tune the knob in real time as needed, which enables *interactive exploration/analytics* over large spatial and spatio-temporal data sets.

Interactive spatial exploration and analytics. Spatial and spatio-temporal data are particularly suited for interactive exploration, given the popularity of different online map services.

For example, a user wants to understand the sale operations in NYC over the first quarter. But s/he wants to understand his/her data for different area and time range combinations in this spatial and temporal region. So s/he could zoom in to a particular area from NYC on a map and specify between January 1 to March 2, and ask for the average price for all sale transactions. In interactive exploration, or formally interactive analytics, the user wants to be able to change his/her query condition without the need of waiting for the current query to complete. In other words, in the above example, the user may change to a different area in NYC and/or adjust the time range to between January 15 and March 12, while the first query is still being executed.

On big spatial and spatio-temporal data sets, as explained above, waiting for exact results may take a while. The user faces a *dilemma*: *either* waits for the current query to complete *or* terminates the current query and issues the new query. Another possibility is to issue multiple queries in parallel and wait for all of them to complete eventually. But this does not solve the latency issue, and is clearly a waste of resources. Suppose the user is interested in finding a region with a particular feature (e.g., high average salary). With interactive exploration, s/he would be able to do some kind of binary search, using only a logarithmic number of queries. Without interactivity, a linear number of parallel queries would be needed, which is an *exponential* blowup.

The *knob* we have envisioned above solves the dilemma. The system provides a user with real-time feedback to his/her query from the start of query execution, and the quality of the feedback improves over time. The user can terminate the current query execution anytime (e.g., when s/he is satisfied with the quality of the query result) and change to a new query, without the worry that time spent on the current query so far was wasted. Alternatively, a user may specify a desired accuracy requirement and the system will stop the query execution automatically whenever the approximation quality has reached the specified level for a given query.

In the above example, assume that after 1 second into the execution of the first query, system reports that the average price is $278 with a standard deviation of $15 and 95% confidence, if the user is happy with the quality of this estimation, s/he can immediately change the query condition to stop the first query and start the second query. S/he could also wait a bit longer for better quality, say, using 1.5 seconds, system now reports the average price for the 1st query as $276 with a standard deviation of $5 and 98% confidence. This happens online in a continuous fashion. So the user could monitor the feedback in real time and make such decision anytime during the query.

Furthermore, for queries of his/her choice (e.g., a mission critical task), s/he also has the option of waiting for the complete execution of a query to get exact results. Lastly, note that parallel execution of multiple queries at once is an orthogonal approach. Users still have that option, but with the added flexibility of terminating/updating any query at any time instance.

Spatial online sampling and aggregation. Random sampling is a fundamental and effective approach for dealing with large data sets, with a strong theoretical foundation in statistics supporting its wide usage in a variety of applications that do not require completely accurate answers. The use of random sampling for approximate query processing in the database community also has a long history, notably with line of work on *online aggregation* [13]. In online aggregation, instead of evaluating a potentially expensive query until the very end, we repeatedly take samples from all tuples that satisfy the query condition, and continuously compute the required aggregate based on the sampled tuples returned so far. The accuracy of the computed aggregate gradually improves as we get more and more samples, which is measured by *confidence intervals*, and the user may stop the query processing as soon as the accuracy has reached a satisfying level. Recently, online aggregation has received revived attention [17, 30], as an effective tool for answering "big queries" that touch a huge number of tuples but the user can often be satisfied with just an accurate enough aggregate.

However, past work on online aggregation has focused on relational aggregates, group-by, and join queries [11, 13, 17, 30], on relational data. In this paper, motivated by the needs for *interactive spatial exploration and analytics*, we study this problem over spatial and spatio-temporal data. Since the statistical side of online aggregation is relatively well understood [11, 13, 17, 23, 30], which we will discuss briefly in Section 6, the problem essentially reduces to that of *online query sampling*, i.e., how to repeatedly sample a tuple from the query until the user says "stop". On spatial and spatio-temporal data, this problem can be formally defined as follows.

Definition 1 (Spatial online sampling) Given a set of n points P in a d-dimensional space, store them in an index such that, for a given range query Q, return sampled points from $Q \cap P$ (with or without replacement) until the user terminates the query.

Spatial online aggregation is a direct product of spatial online sampling, where online estimators for different types of spatial and spatio-temporal aggregates, like sum or average, are built using spatial online samples. A spatial online estimator ideally should be an *unbiased estimator*, and its estimation quality, characterized by confidence intervals, improves over time in an online fashion while more spatial samples are obtained. This concept can be further generalized beyond simple aggregates to a wide range of analytical tasks, like spatial clustering, spatial kernel density estimate (KDE). More details on this topic are provided in Section 6.

Our contributions. We formalize the problem of spatial online sampling (and aggregation) in this paper. We show the limitations of existing approaches when they are adapted to this setting, and propose novel indexing structures that are much more efficient and scalable. They also support dynamic updates. In summary,

- We formalize the problem of spatial online sampling and review how one may adapt existing methods to solve this problem in Section 2. We show that these approaches suffer from various types of limitations.
- We present two baseline methods QueryFirst and SampleFirst in Section 3.
- We design a new indexing structure, LS-tree, in Section 4 that is based on the idea of "level sampling". It is a collection of R-trees where each R-tree indexes a set of samples from the original data set. The sample rates for these sets of samples form a geometric series.
- The LS-tree needs to maintain and query multiple trees, which in practice may not be ideal. We design another new indexing structure, RS-tree, in Section 5 that needs to build, maintain, and query a single R-tree, by embedding samples into the R-tree. We show how we can tailor this structure for memory-based, disk-resident, and hybrid scenarios.
- We extend the discussion to spatial online aggregation and analytics in Section 6.
- We present extensive experimental results using two real data sets in Section 7. The results confirm the superior performance achieved by LS-tree and RS-tree compared to baseline methods. In particular, RS-tree performs the best in practice for nearly all experiments. We also show that both LS-tree and RS-tree support dynamic updates efficiently.

In addition, the paper reviews other related work in Section 8, and concludes the work in Section 9.

P	The raw data set in $\mathbb{R}^d$.
k	The number of samples to report.
N	$\|P\|$, the size of the raw data set.
Q	A range query in $\mathbb{R}^d$.
P_Q	$P \cap Q$, elements in the query range.
q	$\|P_Q\|$, the number of elements in the query range.
$u, v, \cdots$	Tree nodes.
$T(u)$	The subtree rooted at node u.
$P(u)$	The set of all data points covered by $T(u)$.
$R(u)$	The MBB of $P(u)$.
$h(T)$	The height of the subtree T.
$f(T)$	The fanout of tree T.
R_Q	The canonical set for Q.
$r(N)$	The size of a canonical set in a R-tree of size N.
B	The size of a disk block.
s	The sample buffer size in RS-tree.

Table 1: Notation used in the paper.

2. PRELIMINARIES

Sampling with and without replacement. There are two commonly used random sampling methods: sampling with replacement and sampling without replacement. The former repeatedly samples a point from an underlying population, and the sampled point is then immediately put back to the population (so the same point may be sampled multiple times). The latter, on the other hand, does not put a sampled point back. Strictly speaking, most formula on computing confidence intervals hold only for sampling with replacement, but the statistical difference between the two methods is quite small, especially when the sample size is much smaller than the population size. If needed, a sample obtained by sampling without replacement

Algorithm	Query Time	Query I/O	Update Time	Update I/O
QueryFirst	$r(N)+q$	$r(N)+q/B$	$\log N$	$\log_B N$
SampleFirst	kN/q	kN/q	$\log N$	$\log_B N$
RandomPath	$k\log N$	$k\log_B N$	$\log N$	$\log_B N$
RandomShuffle	kN/q	$kN/(qB)$	$\log N$	$\log_B N$
LS-tree	$\sum_{j=\log(q/k)}^{\ell} r\left(\frac{N}{2^j}\right)+k$	$\sum_{j=\log(q/k)}^{\ell} r\left(\frac{N}{2^j}\right)+k/B$	$\log N$	$\frac{1}{B}\log N$
RS-tree	$r(kN/q)+k$	$r(kN/q)+k/B$	$\log N$	$\frac{1}{B}\log N$

Table 2: Comparison of various algorithms.

can be converted to a sample with replacement, provided that we are also given the population size. Thus, mostly we will consider both methods acceptable.

R-trees. The R-tree is a classic and popular index structure for multi-dimensional data [10], which is often used to answer spatial range queries. A large number of R-tree variants exist in the literature. We will also use an R-tree to index our point set P. As our methods can be applied to any R-tree variant, we will not be specific about which R-tree variant is used.

Recall that an R-tree is similar to a B-tree in terms of the structure. All the points are stored on the leaf level, while an internal node stores the *minimum bounding box (MBB)* of all the points below. For a node u in an R-tree, we use $T(u)$ to denote the subtree rooted at u, $P(u)$ the set of data points stored below u, and $R(u)$ the MBB at u. Please see Table 1 for a list of symbols used in this paper.

To answer an (ordinary) range reporting query Q with an R-tree, we start from the root and make a top-down traversal of the tree, and visit a node u if and only if $R(u) \cap Q \neq \emptyset$. The cost of this query consists of two components. For any node u such that $R(u) \subseteq Q$, all the points stored in its subtree are reported, so the total cost of accessing all such nodes is $O(q)$, where $q = |P_Q|$ is the number of points in the query range. Thus this part of the cost is "mandatory" as q is the output size. The other type of nodes u accessed are those such that $R(u) \cap Q \neq \emptyset$ but $R(u) \not\subseteq Q$. These are the nodes with their MBBs on the boundary of Q or entirely contains Q. The cost of accessing these nodes is the "overhead" we pay to answer the query. We call these nodes the *canonical set* for Q, denoted as R_Q. On adversarial inputs, there can be as many as $O(N)$ canonical nodes. But on most real-world data, $|R_Q|$ is actually very small when the R-tree is properly constructed. Therefore, it is meaningless to do any worst-case analysis and we will simply use $r(N)$ to denote the size of the canonical set in an R-tree built on a set of N points in the "typical" case[1].

Using these notations, the cost of executing a range reporting query is $O(r(N)+q)$. For the spatial sampling problem, which is the focus of this paper, the purpose is exactly to avoid executing the query in its entirety, i.e., the $O(q)$ term. The $O(r(N))$ term, on the other hand, is the cost of "locating" the query in the R-tree, so is probably unavoidable. Thus, the goal is to solve the spatial online sampling problem in time $O(r(N)+k)$, where k is the number of sampled points returned to the user. One additional challenge is that the value of k is not known to the algorithm in advance; instead, it is decided by the user on-the-fly.

As it turns out, we can actually do slightly better than $O(r(N)+k)$, the intuition being that to return a sample of Q, we do not have to locate the boundary of Q precisely in the R-tree. Table 2 lists the query and update costs of various solutions that will be presented in the rest of the paper.

External memory indexing. We will consider both internal and external memory indexing schemes for the spatial sampling problem. For an R-tree in internal memory, the size of a node is a constant, while it equals the disk block size B if it is stored in external memory. For an index stored in external memory, we are primarily interested in the I/O cost of answering queries and performing updates. When the R-tree is stored in external memory, we assume that part of the R-tree is cached in main memory, and that all the k samples can fit in memory. We feel that this is a reasonable assumption as for nearly no aggregation tasks will we ever need more than a few million samples.

In order to facilitate query processing for spatial sampling, for each node u in the R-tree, in addition to the MBB, we also assume that $|P(u)|$, i.e., the number of points stored below u, is stored at u. This count can be easily computed at construction time, as well as maintained under insertions and deletions. Note that for a memory-resident R-tree, this count (as well as the MBB) can be stored at the node u itself. However, for a disk-based R-tree, this metadata has to stored at its parent together with the pointer pointing to u, so as to avoid unnecessary I/Os.

3. BASELINE ALGORITHMS

In this section, we first present three baseline methods that are very natural solutions to the problem of spatial online sampling.

Two most straightforward methods for this problem would be QueryFirst and SampleFirst:

QueryFirst Calculate $P \cap Q$ first, then repeatedly extract a sample from the set upon request.

SampleFirst Upon request, pick a point randomly from P and test if it is within Q. Return the sample if so, otherwise dispose it and repeat.

The running time of QueryFirst is $O(r(N)+q)$, the same as a full range reporting query. For SampleFirst, because a randomly picked point falls inside Q with probability q/N, we expect to draw $O(N/q)$ samples in order to see one inside. Thus, the expected cost of SampleFirst is $O(kN/q)$. This could be good for very large q, say, a query that covers a constant fraction of P. However, for most queries, this cost can be extremely large. If $q = 0$, it never terminates.

A better solution is to adapt the random sampling method of Olken [26] to R-trees. His method takes a sample from P_Q by walking along a random path from the root down to the leaf level. When deciding which branch to take, the subtree sizes $|P(u)|$ need to be taken into consideration so that the probabilities can be set appropriately. More specifically, it finds the canonical cover for the input range query using a count tree over P (e.g, an aggregate R-tree). For each subtree within the canonical cover, it is sampled in proportion to the number of data elements included by its leaf level.

[1]We note that if one uses the priority R-tree [4], then $r(N) = O(\sqrt{N})$ in the worst case. However, the priority R-tree is not as widely used as other R-tree variants, such as the R*-tree and Hilbert R-tree, which have a smaller $r(N)$ in practice, although $r(N) = O(N)$ in the worst case. Thus, we still prefer to use a generic function $r(N)$ rather than instantiating it into any worst-case bound.

Once such a subtree T is sampled, the same process is recursively carried out within T.

This way, a sample can be obtained in $O(\log N)$ time. Over k samples, the total time is $O(k \log N)$. We call this method RandomPath. It is reasonably good, but only in internal memory. When the R-tree resides on disk, each random path may involve a traversal in a completely different part of the R-tree, resulting in at least $\Omega(k)$ I/Os in total, which is very expensive.

A variant of SampleFirst, which we call RandomShuffle, is to perform a linear scan on a random permutation of the data, until k points in the query range have been found. The expected query cost is then reduced to $O(kN/(qB))$. The algorithm also enjoys a speedup factor in practise, as the elements are always read sequentially. On the other hand, unlike SampleFirst, RandomShuffle cannot be applied on an existing indexing structure, and a shuffled copy of the data set must be stored.

Because of that, update cannot be performed efficiently this way in RandomShuffle, especially to remove an element we might need to scan the whole data set. To fix this, instead of truly randomly shuffling the data, we assign a random ID for each data element, and build a B-tree on it. The random ID of an element may be obtained by hashing some attributes of the element so we will be able to locate and delete any item.

4. THE LS-TREE

In this section, we present the LS-tree, our first index structure for the spatial online sampling problem. It is based on the "level sampling" idea, which is perhaps known as a folklore.

Starting from $S_0 = P$, we build S_{i+1} by independently sampling each point from S_i with probability $1/2$, and stop until we have an S_ℓ that only has constant size. Then we build an R-tree T_i for each S_i, for $i = 0, 1, 2, \ldots, \ell$. Note that in expectation, we have $O(\log N)$ R-trees. Since their sizes form a geometric series, the total size is still $O(N)$.

Upon a query Q, we simply execute an ordinary range reporting query on the R-trees in turn $T_\ell, T_{\ell-1}, \ldots, T_0$. Note that from T_i, each reported point is sampled with probability $1/2^i$ independently, and all must fall inside Q. Thus, they form a probability-$(1/2^i)$ coin-flip sample of P_Q. To turn this into a sample without replacement, according to [7], we just need to perform a random permutation, and start to report the points to the user one by one, until the user terminates the query, or all samples are exhausted. In the latter case, we move on to the next R-tree T_{i-1}. Since $S_j \subseteq S_i$ if $j > i$, we need to make sure that any sample must not be reported twice, by maintaining a set of all the samples that have been reported.

Suppose the user terminates the query after receiving k samples. Then in expectation, we have reached tree T_j such that $q/2^j \approx k$, i.e., $j = \log(q/k)$. Thus, the total query cost (in expectation) is

$$\sum_{j=\log(q/k)}^{\ell} \left(r\left(\frac{N}{2^j}\right) + \frac{q}{2^j} \right) = O(k) + \sum_{j=\log(q/k)}^{\ell} r\left(\frac{N}{2^j}\right).$$

The $O(k)$ term is quite satisfying, though the second term looks ugly. If $r(N) = O(\sqrt{N})$ as in the priority R-tree, then the sum (asymptotically) reduces to the largest term, i.e., $O(r(Nk/q))$. However, for other R-tree variants, we cannot make this simplification, and that part of the cost can be as large as $O(\log(q/k))$ times the overhead of one R-tree.

This solution works well in external memory, since the query on each R-tree is just a normal R-tree range query.

Query (non-)independence. One issue with the LS-tree is that of query independence. From the construction of the LS-tree, it is clear that the samples returned for one query are independent. But if the user issues the same query again, the same samples will be returned, i.e., samples returned for different queries are not independent. However, we argue that this should not be a serious issue. First, since sampling is online, if a user issues the same query again and wants different and independent samples, s/he could just continue the previous query with a larger k. Secondly, if two different users issue the same query, then it is probably OK to return the same samples to the two different users. Third, precomputing the samples, as done in LS-tree, is important for query efficiency and scalability. If queries are to be independent, then samples must be taken on-the-fly, which is extremely slow, especially in external memory (as done as in the baseline solutions). Very recently, Hu et al. [14] proposed indexing structures that achieve independent query sampling without expensive I/O costs (theoretically), but their structures work only on one-dimensional data, are very complicated and of only theoretical value. In addition, the structure of their index changes during the querying process, which means that only one query can be processed at a time, resulting in poor concurrency. Finally, we can always reconstruct the samples periodically (say, every evening), as done in BlinkDB [2], which is another query sampling system that is also based on precomputed samples.

5. THE RS-TREE

The LS-tree is a simple structure that supports spatial online sampling, but it suffers from a couple of drawbacks. First, it consists of multiple R-trees, which may lead to additional system overhead. And second, its query performance is not ideal, as shown in Table 2.

In this section, we introduce our second index structure, the RS-tree, which addresses these drawbacks.

5.1 In-memory structure

The design of the RS-tree is based on the following three ideas.

Sample buffering. In LS-tree, we build a separate R-tree on each level of samples, which are stored at the leaves of the R-tree. This introduces additional overhead in retrieving these samples. In RS-tree, we attach a buffer of sampled points to each internal node u of the R-tree, which are taken randomly from u's subtree with replacement. The sample buffer attached at node u is denoted as $Samples(u)$, and we set its size at s, for some s to be decided later. This way, we integrate searching and sampling in one go, which improves performance and removes the need to have multiple R-trees.

Note that for a node u with $|P(u)| \leq 2s$, we will not attach a sample buffer, since its subtree is already small enough, and we can simply go directly to its leaves to get the actual data points instead. Therefore, the total size of all the sample buffers is still $O(N)$.

Rejection sampling. Having a sample buffer attached to each internal node of the R-tree means that we can start reporting samples in the very beginning, even at the root of the R-tree. However, since the samples in the buffer are taken from the entire R-tree, many of them may not be inside the query Q. Thus, we only return samples that are actually inside Q, and reject the rest. This idea, known as *rejection sampling*, is a common technique to draw values from some arbitrary distribution. Similarly, we apply this technique at each internal node u that we encounter in the query process.

Lazy exploration. The first two ideas naturally leads to the third idea, that of *lazy exploration*. As the sampling is online, we should not do more than what the user asks for. Thus, we start from the root of the R-tree, and visit its children only after its sample buffer is exhausted (either reported or rejected). We do recursively for each internal node visited in the top-down traversal. This way, we make sure only the nodes we have to access get visited. This idea exactly captures the intuition that for query sampling, we do not have to

locate the query Q accurately in the R-tree. In particular, if the node has a small subtree, even if its MBB intersects the boundary of Q, it may not get visited, as the probability of reaching that node is small. So we do not even need to visit all the canonical nodes R_Q, which is necessary for a range-counting query.

Query algorithm

With the ideas above, the query algorithm follows quite straightforwardly. The algorithm essentially mimics SampleFirst, but in a much more efficient manner.

More precisely, we maintain a list *Frontier* of nodes that we want to take samples from. Initially, only the root of the R-tree is in *Frontier*. Then, we repeatedly sample a node from *Frontier*, with probabilities proportional to the $|P(u)|$'s. After a node u is selected, we extract the next sample point stored in $Samples(u)$. If it is inside Q, we report it; otherwise we reject it. When $Samples(u)$ is exhausted, we remove u from *Frontier* and add all its children to *Frontier*. The process repeats until user termination. The detailed algorithm is described in Algorithm 1.

Algorithm 1: Querying an RS-tree

Input: tree T; range query Q
Output: random samples from $P \cap Q$

1 $Frontier \leftarrow \{$root node of $T\}$;
2 **while** *need more samples* **do**
3 **if** $Frontier = \emptyset$ **then return** $P \cap Q = \emptyset$;
4 $u \leftarrow$ a randomly picked node from *Frontier* with probability proportional to $|P'(u)|$, $P'(u) = P(u)$ *initially and is defined in line 8*;
5 **if** $R(u) \cap Q = \emptyset$ **then**
6 Remove u from *Frontier*;
7 **else if** $Samples(u)$ *does not exist* **then**
8 $e \leftarrow$ a random sample from $P'(u)$, where $P'(u)$ contains *only non disabled* elements in $P(u)$;
9 **if** $e \in Q$ **then** Report e as a sample ;
10 **else if** $e \notin Q$ **then**
11 Flag e as "disabled" in $P(u)$
12 **else**
13 $e \leftarrow$ the next sample from $Samples(u)$;
14 **if** $e \in Q$ **then** Report e as a sample ;
15 **if** *there are no more samples in* $Samples(u)$ **then**
16 Remove u from *Frontier*;
17 Add u's children to *Frontier*;

Query cost analysis

Suppose the algorithm is terminated after k samples have been reported. Then all the nodes visited by the algorithm form a tree rooted at the root of the R-tree, with *Frontier* at the bottom. Consider any R-tree node u visited. It is either a canonical node, or its MBB is entirely inside Q. There are $r(N)$ canonical nodes. This bound can be further improved since not all canonical nodes will be visited, especially those down in the tree. If only k samples are returned, then in expectation we only need to reach the level of the R-tree which has kN/q nodes. Thus, the number of canonical nodes is actually only $r(kN/q)$ in expectation.

For those visited nodes whose MBBs are entirely inside Q, we further classify them into two types, depending on whether they are in the *Frontier* list or not when the query terminates. If a node is not in the *Frontier* list, then it must have contributed s samples to the user, so there are at most k/s such nodes. If a node is in the *Frontier* list, then it may not have contributed any, so we cannot bound the number of such nodes in terms of k. The idea is to consider the parent of such a node. We note that the parent must be a canonical node, or a node of the first type, so there are at most $(r(kN/q) + k/s) \cdot f$ of them, where f is the fanout of the R-tree. Thus, the total number of nodes visited is $O((r(kN/q) + k/s) \cdot f)$. Since we pay a cost of $O(s)$ per node to examine its sample buffer[2], the total query cost is $O(sf \cdot r(kN/q) + kf)$. By setting both s and f to some constant, this is $O(r(kN/q) + k)$, as claimed in Table 2.

Batch sampling: an optimization

In the query cost analysis above, we have implicitly assumed we can pick a node from the *Frontier* list randomly in $O(1)$ time in line 4 of Algorithm 1. However, since the sampling is not uniform, and *Frontier* is changing, it is not as easy. The naive implementation will take time $O(\log |Frontier|)$, by building a binary tree on top of the probabilities which are proportional to the $|P(u)|$'s. Below, we describe a *batch sampling* technique that can drive the cost down to $O(1)$ amortized.

The idea, as the name of the technique suggests, is not to sample the nodes one by one. Instead, we will take $|Frontier|$ samples with one single scan of the *Frontier* list. More precisely, for each node $u \in Frontier$, we calculate how many samples would be taken from u in the next $|Frontier|$ draws with replacement. Note that this is simply a binomial distribution. If this number is more than the number of samples remaining in $Samples(u)$, then we know this node would be exhausted in the next $|Frontier|$ draws. Suppose we need b more samples than what $Sample(u)$ has, then we visit u's children, and allocate these b samples to these children according to their subtree sizes. This process may take place recursively until we have drawn enough random nodes.

After this process, we have $|Frontier|$ nodes randomly drawn with replacement with the right probabilities. To supply the decision to line 5 of Algorithm 1 one at a time, we need to perform a random permutation. Overall, the cost of one batch of sampling is $O(|Frontier|)$, which will last for the next $|Frontier|$ iterations of the main algorithm. Thus the amortized cost is $O(1)$. Meanwhile, by doing so we at most overshoot k by $O(|Frontier|)$. But from the previous analysis, this is at most $O(r(kN/q) + k)$, so it does not affect the overall cost asymptotically.

Maintaining the sample buffers

Given the R-tree, we can build the sample buffers easily by a DFS traversal. We maintain the invariant that, after the DFS completes its visit to a node u, we have filled $Samples(u)$ with s elements, as well as found enough samples for all its ancestors. The detailed procedure is described in Algorithm 2.

The sample buffers can also be maintained easily as the R-tree itself performs insertion or deletion of points. We first perform a normal insertion or deletion in the R-tree. Then we update all the affected sample buffers. To ensure a low amortized cost, we allow the buffer size to vary between $s/2$ and $2s$.

When a point e has been inserted to a subtree $T(u)$, each existing point in $Samples(u)$ should be replaced with e with probability $|P(u)|^{-1}$ (this can be done efficiently using binomial distribution, instead of doing a separate test for each point in the buffer). When a point e is removed from $T(u)$, all occurrences of e in $Samples(u)$ must be removed. When the sample buffer is less than half-full due to many deletions, we replenish it by drawing more samples from its children's buffers, which may trigger recursive replenishing.

[2]When $Samples(u)$ does not exists for a node u, it must be $|P(u)| = O(s)$, so it takes $O(s)$ time to access all the elements in $P(u)$.

Algorithm 2: BuildSamples

Input: a tree node u; d, which is 0 by default
Output: $Samples(v)$ is filled for any node v in $T(u)$

```
1  if u is a leaf node then
2      return {d random samples from P(u) with replacement};
3  if |Samples(u)| < s then
4      d ← d + 2s − |Samples(u)|;
5  S ← ∅;
   // We want to get d samples from T(u)
6  foreach child node v of u do
7      d' ← the number of samples we need from T(v);
8      S ← S ∪ BuildSamples(v, d');
9  Fill up Samples(u) with elements in S;
10 return the rest of S;
```

When nodes are split or merged, we similarly merge and split the sample buffers. We remove extra points if the sample buffer has overflowed, and replenish it if it is too empty.

5.2 RS-tree in external memory

The RS-tree naturally extends to a disk-based index by simply setting $f = \Theta(B)$ and $s = \Theta(B)$, such that each R-tree tree node, together with its sample buffer, fits in one disk block. Inserting a point in a disk-based R-tree costs $O(\log_B N)$ I/Os, by traversing a root-to-leaf path.

To support fast insertions, we adopt the buffer tree idea [3,18], and associate with each internal node u of the R-tree with an *insertion buffer* of size B that temporarily keeps the points to be inserted into u's subtree. The buffer of the root is always kept in main memory. To insert a new point, e, instead of finding the path all the way down to the corresponding leaf node, we just put it into the insertion buffer of the root node. When the insertion buffer is full, we flush these points to the insertion buffers of its children. This might make these buffers overflow, which may trigger recursive flushings down the tree. When a point is flushed to a leaf node, it is stored there and no more flushing will be done. Instead, if the leaf node is full, the leaf block is split into two. After a point has been inserted into the insertion buffer of a node u, we also update $Samples(u)$ to take the new insertion into consideration, as we need to maintain the invariant that $Samples(u)$ is always taken from all points stored below u (including u). The procedure is the same as in "maintaining the sample buffers" previously described.

By an analysis similar to that in [3], we can show that the amortized I/O cost for each insertion is $O\left(\frac{f}{B}\log_f N\right)$. The detailed proof can be found in [3], while the basic intuition is that an inserted point starts from the root and then gradually follows a root-to-leaf path consisting of $O(\log_f N)$ nodes. On each node, the amortized I/O cost of moving this point from its parent to this node is $O(f/B)$. By tweaking the value of f, we may trade off between insertion performance and query performance. In particular, if f is a constant, then insertions can be done in $O\left(\frac{1}{B}\log N\right)$ I/Os amortized, which means that most insertions can be done without incurring any I/O cost. Meanwhile, since the query I/O cost is $O(r(kN/q)+fk/B)$ I/Os, it is desirable to set f to be a constant, so that the query I/O cost is $O(r(kN/q)+k/B)$, as claimed in Table 2. Note that, however, using a constant f may increase the number of canonical nodes slightly, by roughly an $O(\log B)$ factor, since the height of the R-tree increases from $O(\log_B N)$ to $O(\log_f N) = O(\log N)$. Nevertheless, since the $O(k/B)$ term dominates the cost, using a small fanout is beneficial overall.

Even with the insertion buffers, the query algorithm actually remains the same, with the following clarification. First, recall that $P(u)$ is defined to be the set of points stored below u, so it now also includes those stored in the insertion buffers below u. Second, we will consider the insertion buffer at u to be an extra leaf directly below u, so that not all leaf nodes are on the same level. In line 15 of Algorithm 1, one of u's children may be its insertion buffer, and we just add it to *Frontier* list.

Deletion is handled slightly differently. To delete a point e, we first search for e. Note that e might reside in a leaf block *or* in one of the insertion buffers on a root-to-leaf path. So we follow the search path and remove e whenever it is encountered. We also remove e from any sample buffer where it is kept.

5.3 RS-tree: The Hybrid

The external memory version of the RS-tree tree described above requires only one block of main memory space. In general, there can be more memory available, which calls for a hybrid version that optimizes the total cost by fully utilizing all available memory.

As shown in Figure 1, the whole data structure is still an R-tree. Nodes in the top layers are kept in the memory, which are called mem layers. The nodes in the lowest mem layer are called leaf mem nodes, while the other higher nodes are called internal mem nodes. Similarly for the disk-resident layers, we classify them into internal disk nodes and leaf disk nodes.

As before, we attach a sample buffer to each node u, but the size differs for the mem nodes and the disk nodes. For a mem node, the sample buffer has a constant size, but $s = B$ for a disk node. As before, for a node u with $|P(u)| \le 2s$, we do not attach a sample buffer. The fanout is a constant for all nodes, mem or disk. The insertion buffers are associated to all the leaf mem nodes and the internal disk nodes. The rationale is that, since we do not incur any I/O in memory, we do not have to buffer the insertions at the root. Instead, an insertion can go as deep as possible until we reach the mem/disk boundary. This way, effectively, the subtree rooted at each leaf mem node is the same as the external memory version of the RS-tree. While the memory-resident part is the same as the internal memory version of the RS-tree.

5.4 Extending to GiST Indexes

While the R-tree was chosen as the underlying tree structure for RS-tree to simplify the analysis and implementation, the ideas behind RS-tree can be naturally extended to any GiST index on either spatial or non-spatial data. Recall that a GiST index defines a generic tree-structured hierarchical partitioning of the data domain, where raw data is stored on the leaves. By using the same algorithms of the RS-tree on any GiST index, we can maintain samples attached to the internal nodes of the GiST tree, and use them to answer queries approximately. Nevertheless, the bounds claimed in Table 2 may vary according to the particular instantiation of a GiST index.

6. SPATIAL ONLINE AGGREGATION AND ANALYTICS

Recall that both the LS-tree and RS-tree report random samples uniformly chosen from P_Q, either with or without replacement[3] Generally speaking, any aggregate of the whole population can be estimated from a sample, and the accuracy improves as more samples are obtained and the sample size increases. Suppose each point e in our data set is associated with an attribute $e.x$ of interest. Then for example, it is well known that the sample mean is an

[3]The statistical difference between the two sampling methods is insignificant; see [13].

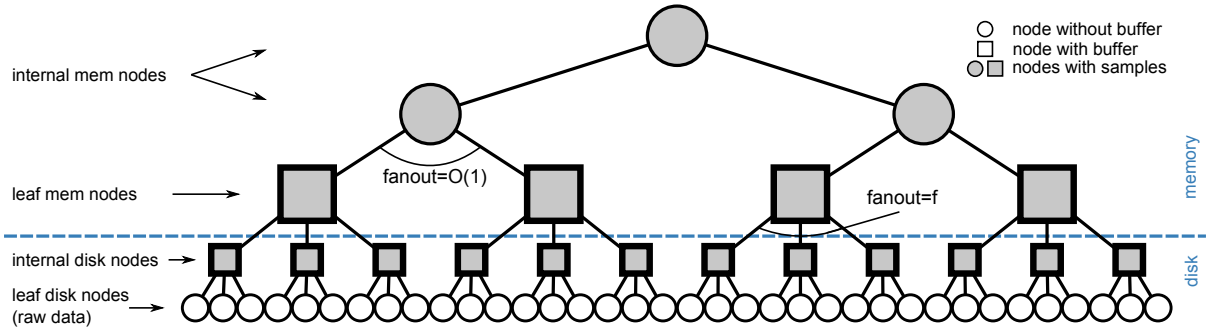


Figure 1: An example of the hybrid version of RS-tree.

unbiased estimator of the population mean, i.e., letting S be the set of k samples returned, we have

$$E[\bar{X}] = E\left[\frac{1}{k}\sum_{e\in S} e.x\right] = \mu = \frac{1}{q}\sum_{e\in P_Q} e.x.$$

By the central limit theorem, $X-\mu$ approaches $Normal(0,\sigma^2/k)$, where σ is the population standard deviation. This means that sample variance is inversely proportional to the sample size, and we expect the accuracy to improve quite rapidly as k. In addition, we can estimate σ^2 also from the sample, and further compute the confidence intervals, as in standard online aggregation [12, 13].

Estimating query size

Many aggregates other than average, like sum or count (when $e.x$ is either 0 or 1 depending on a predicate), require the knowledge of the population size, which is $q = |P_Q|$ in our case. One way to obtain q is to perform a range-count query using the R-tree, costing time $O(r(N))$. However, our sampling algorithm actually returns a very good estimate of q as a by-product. Recall that the *Frontier* list in the query algorithm always maintains a list of R-tree nodes that do not have ancestor-descendant relationship, i.e., the points stored below these nodes do not overlap (their MBBs may overlap, though). Thus, the total query size q is simply the sum of the numbers of points below each node $u \in Frontier$ that fall inside Q. There are two cases: If $R(u) \subseteq Q$, then we know exactly the number of points inside Q, which is $|P(u)|$, associated with u. Otherwise, not all points in $P(u)$ are inside Q. But recall that we also have a sample buffer $Samples(u)$ associated with u, which contains sampled points from $P(u)$ with replacement. Then we simply check how many of them are actually inside Q, and estimate $|P(u)\cap Q|$ as $\frac{|Samples(u)\cap Q|}{|Samples(u)|}\cdot|P(u)|$. This is an unbiased estimator with standard deviation at most $|P(u)|/\sqrt{|Samples(u)|}$. Adding the estimates from all nodes in *Frontier*, we obtain an estimate of q as well as the variation. If it is not accurate enough, we can always keep pushing *Frontier* down to explore more nodes, which eventually becomes a standard range-counting query. However, for most cases, the accuracy of the estimate is already high enough before reaching the leaf level.

Spatial analytics

In the spatial setting, there are more complicated statistics than simple aggregates like sum or mean. A widely used one is the *kernel density estimation (KDE)*, which constructs a continuous spatial distribution from discrete points. Specifically, the distribution density at some point p is defined as $f(p) = \frac{1}{q}\sum_{e\in P_Q}\kappa(d(e,p))$, where $d(\cdot,\cdot)$ is the distance between two points, and $\kappa(\cdot)$ is the *kernel function* that models the "influence" of e at p. Then we can compute $f(p)$ at regularly spaced points (say, all grid points), and construct a density map of the underlying spatial distribution. We observe that the distribution density at each point, $f(p)$, is still an average, so we can, as before, compute an approximated density map by drawing a sample from P_Q, and derive the confidence interval (for each point p).

Other spatial analytics tasks, such as clustering, can also be performed on a sample of points. Intuitively, the clustering quality also improves as the sample size increases, but in this case, it is unclear how a confidence interval should be defined and computed, which may remain as an interesting question for further investigation.

7. EXPERIMENTS

Implementation. We implemented 5 methods in C++: RandomPath, LS-tree, RS-tree (hybrid), RandomShuffle and RangeReport, where the last one is to extract and store P_Q with a range reporting query on the tree, then choose random samples from P_Q upon request.

A random shuffle of the original data is used by RandomShuffle, while all other algorithms use a Hilbert R-tree as the underlying tree structure, which is essentially a B+-tree where nodes are sorted by their Hilbert values [21] (the Hilbert value of a point is the order imposed by a Hilbert space filling curve). Hilbert R-tree usually preserves locality well [21], which is crucial to answer range queries promptly, yet it is also easy to implement. In this section we will simply refer to our implementation of Hilbert R-tree as R-tree, unless specified otherwise.

Fanout of each node is between 4 and 16. Other parameters are set such that each of the following fits into a 8KB disk block:

- Elements inside a leaf disk node.
- (Up to 16) child node entries and all buffered insertions of an internal disk node.
- Samples associated with an internal disk node.

In this way, each leaf disk node occupies 1 disk block, and each internal disk node occupies 2 disk blocks. In case insertion buffers are not used, we may combine the child node entries and samples such that only 1 disk block is occupied by an internal disk node. Using different page size value does have a modest impact on query performance. We have tested different page size values, and observed that increasing page size initially leads to better query performance, but it will then negatively affect the query efficiency, and 8KB is roughly a sweet spot for different methods.

The data set is sorted with the disk based sorting algorithm before the corresponding R-tree is built. An extra pass over the tree is taken to build the samples for RS-tree. For LS-tree, we scan and sample the sorted data set and build R-tree's until the data set becomes small enough (less than 256k nodes).

	Low Memory Machine	High Memory Machine
CPU	i7-960	i7-3820
Memory	6GB	64GB
Hard Disk	2TB Western Digital RE4	
OS	Ubuntu 12.04 LTS	
Compiler	GCC 4.8.1 (-O3)	

Table 3: Experiment environment.

Data	N	binary	R-tree	RS-tree	LS-tree
GEO	25 million	569MB	878MB	933MB	1.7GB
OSM	2.2 billion	49.5GB	75GB	80GB	151GB
4xOSM	8.8 billion	198GB	322GB	342GB	643GB

Table 4: Data set statistics.

The first few layers of RS-tree is loaded into memory, as well as for RandomPath and RangeReport. For LS-tree, to make it fair, layers from the few smallest R-trees are loaded into memory such that LS-tree uses roughly the same amount of memory as the other methods. Note that we did this because queries always start with smaller trees in LS-tree. We did the same for RandomShuffle so that enough blocks are loaded into the same amount of memory.

Environment. Table 3 lists the specifications of the machines we used to perform the experiments. Experiments were preformed using the low memory machines except where we state we are using the high memory machines.

Data sets. Two data sets are used as input. GeoLife GPS trajectories from Microsoft Research [37], abbreviated as GEO in this paper. This contains 17,621 GPS trajectories tracking various outdoor movements including shopping, hiking, sightseeing, and traveling to and from work. Each trajectory contains information such as latitude, longitude, time stamp and altitude, but only the first three are used in the experiments.

The other data set, abbreviated as OSM, is obtained from OpenStreetMap [29,31]. This collection of data comes from a community of volunteers from around the world mapping the globe. This includes *nodes* and *ways*. A *node* represents a specific location on the earth's surface described by latitude and longitude coordinators. A *way* is described as a collections of *nodes* which describes a polyline. A *way* can be used to describe roads, boarders, shorelines etc. Only latitude, longitude, and time stamp of each *node* in the OSM data set is used in the experiments.

To see how the algorithms perform with a much larger data set, the OSM data set was replicated four times to create a much larger data set. We call this data set 4xOSM.

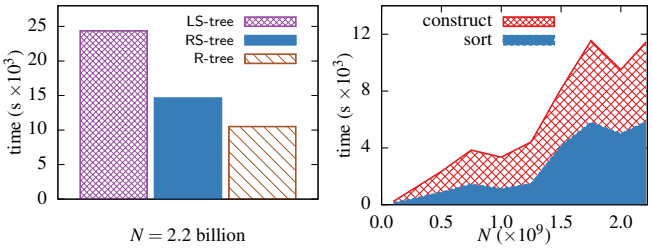
After each data set was processed and filtered they were stored into a csv file. The sizes of the raw data files are shown in Table 4.

For each data set, each element is represented by three 32-bit numbers internally, two floating point numbers for latitude and longitude, and one integer for timestamp. A unique 12-byte ID is also included in each element to uniquely identify a data element.

7.1 Index construction cost

We first compare the index construction cost for different methods. Both RandomPath and RangeReport simply build a single R-tree, while LS-tree needs to build multiple R-trees (of decreasing size). RS-tree needs to build a single R-tree, but with carefully chosen samples embedded into each node.

Figure 2(a) compares their total construction costs, including the cost of sorting the data sets based on Hilbert values since Hilbert R-tree is used. The complete OSM data set is used in this case. The



(a) construction time on full OSM (b) scalability of RS-tree.

Figure 2: Construction time of different indexes.

extra cost for RS-tree to build samples on top of a standard R-tree is shown to be not much, comparing with the construction time of R-tree itself, while LS-tree takes more than twice longer than R-tree, as a number of R-trees are built, whose sizes are decreasing geometrically.

Note that the scalability of an R-tree is well understood in the literature. Different variants of R-tree, including Hilbert R-tree, all have excellent scalability with respect to the size of the data sets. RS-tree is the only structure that has made changes to an R-tree, hence, we investigate its scalability in further details. For this experiment we constructed RS-tree from the OSM data with various amounts of the complete data set to see how the time to construct RS-tree would vary when the number of elements changed.

Building happens in three phases. In the *reading* phase the data is read from the raw data file and each record is tagged with a Hilbert value. Next the elements are sorted by their Hilbert value. Finally the RS-tree is constructed using the sorted data. We timed the construction of RS-tree for various amounts of data.

The results of this experiment is shown in Figure 2(b). The lower part represents how much time was spent in the *sort* phase, while the upper part represents the amount of time taken during the *construct* phase. The time of the *reading* phase is dominated by the *sort* and *construct* phases, hence, is omitted in the figure for clarity.

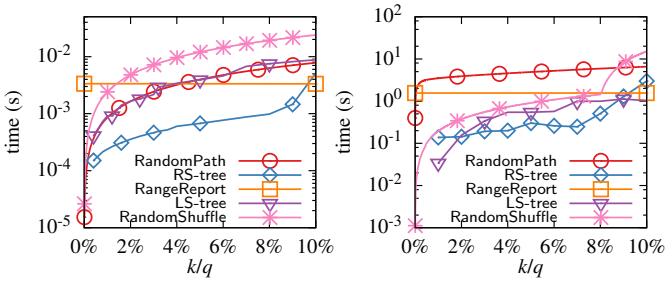
Clearly, RS-tree demonstrates excellent scalability as data grows. Its construction cost is slightly worse than being linear to the growth of the data size. Each phase of construction takes about 50 minutes on the full OSM data set. By calculating the amount of data read and written to the disk and considering the throughput of the disk used, this construction time is very reasonable.

Lastly, we also compared the size of different index structures on the full GEO and OSM data sets respectively. The results are shown in Table 4. It is well known that R-tree has linear size with respect to an input data set, which is reflected in our result. Not surprisingly, RS-tree also has linear size since the only change it has made to a R-tree is to embed some carefully selected samples (into a single R-tree). LS-tree also has linear size, but due to the multiple R-trees it builds over different sets of samples, its size is the largest, which is roughly twice as large as plain R-tree.

Lastly, the construction cost of RandomShuffle is to sort the data based on the random IDs that were assigned to each record. In addition, as explained in Section 3, to support efficient update, an B+ tree needs to be built over the randomly assigned IDs. Hence, its construction cost and index size is very similar to that in an R-tree (which is simply a B+ tree over Hilbert values in our case using Hilbert R-tree), hence, we have omitted these results.

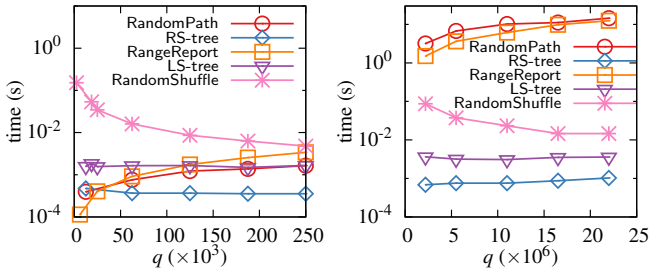
7.2 Query cost: vary k

The efficiency of performing a spatial online sampling query is of primary importance. A typical usage of spatial online sampling query is to keep asking for samples in an interested region Q until suffecient samples have been collected. We study the performance



(a) **GEO data,** $q = 250000$. (b) **OSM data,** $q = 2.2$ **million.**

Figure 3: Vary k**, number of samples**



(a) **GEO data,** $k = 5000$. (b) **OSM data,** $k = 10000$.

Figure 4: Vary q**, fixed** k

of the algorithms by asking for up to $k = 0.1q$ samples for a fixed Q. Recall that q equals $|P_Q|$, the total number of data elements in the complete query result for a range query Q.

Figure 3(a) shows the performance of all 5 methods on the full GEO data set, where q is 0.25 million. The result is the average of 10 random Qs with the same value of q (within a relative error of 0.1%). The dataset was loaded completely into main memory.

The time of RangeReport appears as a horizontal line in the figures. This is what is expected because it is always gathering all elements in the region regardless of the number of samples requested. RandomPath starts much faster than RangeReport because it can relatively quickly find a few elements in the query region. The cost of RandomPath is totaled per sampled taken. As the number of samples increases the benefit of using RandomPath decreases.

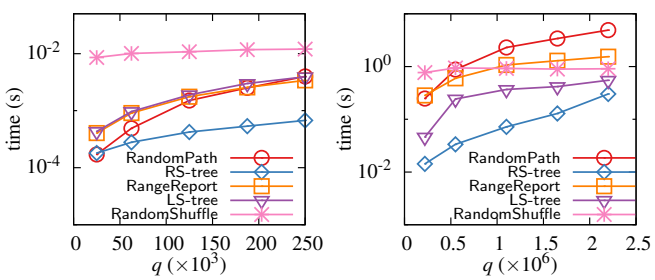
In this experiment, LS-tree takes approximately the same amount of time as RandomPath does to return samples for the GEO dataset. The cost of RandomShuffle is highest out of the algorithms tested, when k/q starts to exceed 2%. Overall RS-tree performs the best when $q = 0.25$ million because it can quickly return samples from the region when requested due to its block structure.

The same experiment is repeated on the full OSM data set except for q is set to 2.2 million. The result is presented in Figure 3(b). The experiment was performed on the same machine as before. Because the full OSM dataset is so large, only 5GB of the dataset could be loaded into RAM. Recall that we use the low memory machine by default unless otherwise specified.

When k is small RandomShuffle was able to quickly return samples because scanning the portion of the data in main memory is fast. But RandomShuffle suffers as k increases, and eventually it starts reading from hard disk and its performance drops quickly.

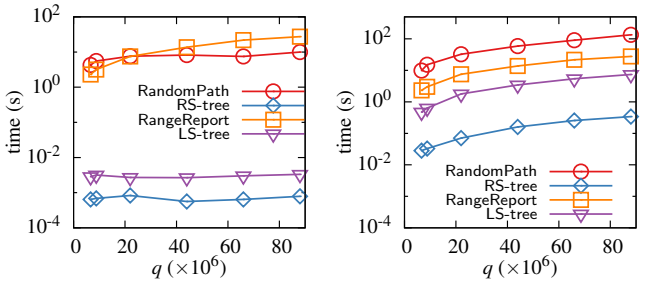
In Figure 3(b), when k is small LS-tree is slightly faster than RS-tree. This is because LS-tree first queries very small trees, which can be searched very quickly. As k increases this advantage is no longer present. Overall RS-tree responds the fastest.

We performed this same experiment using the high memory machines on the 4xOSM data set, allowing full use of the available memory. We found the results of this experiment to be similar to the results for GEO and OSM.



(a) **GEO data,** $k/q = 5\%$. (b) **OSM data,** $k/q = 5\%$.

Figure 5: Vary q**, fixed** k/q



(a) **4xOSM data,** $k = 10000$. (b) **4xOSM data,** $k/q = 5\%$.

Figure 6: 4xOSM, Vary q**, fixed** k **or** k/q

7.3 Query cost: vary q

It is also interesting to study how q affects the performance of the algorithms when k is fixed. We tested the algorithms when q changes. Ten Qs are randomly chosen for each value of q, within a relative error of 0.1%.

Figures 4(a) and 4(b) present the results on both data sets, where k is set to 5000 and 10000 respectively.

For both datasets, RS-tree and LS-tree stay mostly constant when q varies. This is expected because the most significant contributor to the complexity of each of the algorithms is k. Even though they are relatively constant RS-tree has less overhead than LS-tree, so it is consistently faster in this experiment.

RandomShuffle becomes faster as q increases because fewer elements must be scanned to find k matching elements. But it is still much more expensive than RS-tree and LS-tree on both data sets. RangeReport becomes very expensive when q is large, because all elements in Q must be retrieved, regardless of the value of k.

We also performed a similar experiment where we varied q but fixed k/q. The results are show in Figures 5(a) and 5(b). Once again, RS-tree has the best performance on both data sets and has roughly outperformed other methods by 1-2 orders of magnitude.

We ran a similar experiment on **4xOSM using the high memory machine**. The results of the experiments on this dataset is shown in Figure 6. The results are similar to the results of the other figures. As expected, RS-tree and LS-tree outperform the other methods, and RS-tree has the best performance consistently which outperforms the other methods by several orders of magnitude. Despite the dataset containing over 8.8 billion elements, samples of the dataset can be returned with extremely low latency using the RS-tree. Note that because of the size of the data RandomShuffle has much worse performance compared to other methods and was omitted from the experiments on 4xOSM.

7.4 Query cost: vary s

Recall that the parameter s controls the size of the sample buffer attached to each R-tree node. For an R-tree node stored on disk, it is quite natural to choose an $s = \Theta(B)$ such that the sample buffer has the same size as a disk block. For an R-tree node stored in

main memory, however, s could be potentially set to any constant. In this set of experiments, we explore how s should be set for the memory-resident nodes. We built 3 RS-tree's with different values of s. Meanwhile, we correspondingly change the number of memory-resident layers such that the amount of memory space used remains the same. This means that a larger s will lead to fewer memory-resident layers of the R-tree. To clearly see different s values' impacts to the query cost and IO, we tested for a wider range of query areas and a large number of samples in this set experiments. In particular, we used the GEO data set, and we fixed the value of $k = 0.4$ million and varied q from 0.4 million to 2 million.

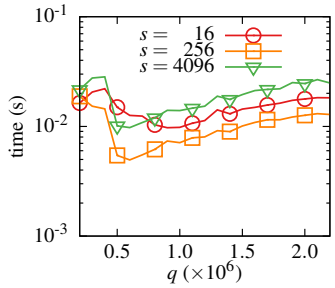


Figure 7: Vary q and sample buffer size, GEO, $k = 0.4$ million.

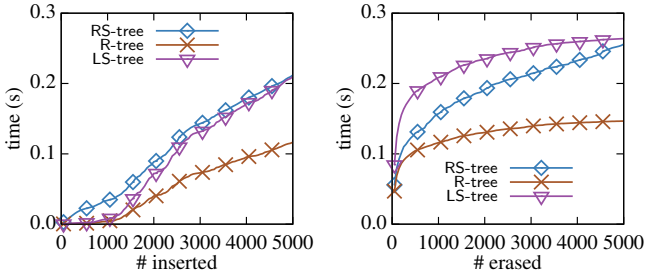
The results are shown in Figure 7. Overall, we see the value of s does not have a significant impact on the query performance, and different query sizes may benefit from different values of s. Larger queries are better with smaller values of s, while smaller queries prefer larger s. That said, we recommend a value of s that is the same or slightly (1 to 16 times) larger than the fanout f. The default s value for all other experiments is 256.

7.5 Update cost

Inserting and erasing elements were tested together. Ideally we would perform each sample of inserting or deleting on a fresh data structure, but this would take an unreasonable amount of time for our data set. To circumvent this problem, we started with a pristine structure. For each insertion experiment performed, we immediately followed that timing experiment by erasing the recently inserted elements. The erasing was timed and reported.

The experiment was setup similar to the query experiment setup. Three data structures are tested, including RS-tree, LS-tree, and R-tree, which is the underlying Hilbert R-tree without samples and used by both RandomPath and RangeReport. As explained earlier, RandomShuffle would use an B+tree to support efficient updates, and its update performance is similar to our R-tree results (which is an B+ tree over Hilbert values).

In the previous experiment, the few smallest trees of LS-tree are loaded into memory, leaving the largest tree completely resident on disk, which leads to significantly bad performance for updates, as the largest tree is always the first one to visit while inserting or deleting



(a) **5000 insertions** (b) **5000 deletions**

Figure 8: Updates on GEO, $q = 18$ million.

elements. In this experiment, we compensate this by allocating more memory for the largest trees (but this leads to worse query performance). In practice, the user should balance the memory usage when LS-tree is used, to trade off between query performance and update performance. A fair choice is to allocate the same amount of memory for the few largest trees and the few smallest ones.

For simplicity, we used the same set of Qs used in the query experiments to test the performance of insertion and deletion. For each Q and each data structure we did the following steps:

1. Choose 5000 elements in the region Q uniformly at random.
2. Insert all 5000 elements into the tree, recording the time and I/O cost every 50 elements inserted.
3. Erase all 5000 elements from the tree which were just inserted, recording the time and I/O cost every 50 elements inserted.

System cache is cleared before each set of experiments and regions with similar values of q were averaged.

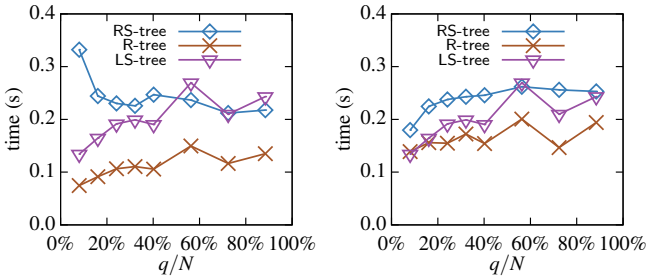
Figures 8(a) and 8(b) show the performance of the algorithm when q is fixed to 18 million, using the GEO data set.

RS-tree is about twice slower than R-tree, which is expected because sample blocks must be updated. LS-tree is slightly faster than RS-tree in insertion, and slower in deletion. Indeed LS-tree always needs to visit one extra tree to make sure that the element does not exist in the following (smaller) trees.

Overall all methods are shown to have good scalability, the insertion time appears linear and the deletion time appears sublinear.

The same experiment was repeated for various values of q, to explore how q may affect the performance when we randomly selected insertions and deletions from q elements, we recorded the total cost of 5000 insertions or deletions for each value q.

The results are presented in Figure 9(a) and 9(b). Again, every point is the average of 10 runs with Qs that share the same value of q (within 0.1% relative error).



(a) **5000 insertions** (b) **5000 deletions**

Figure 9: Updates on GEO, vary q.

LS-tree is about twice slow as R-tree. RS-tree is slightly faster than LS-tree for large values of q, but slower when q is small. The reason is that when q is small new elements are concentrated into a small subtree (or forest), tree nodes are more likely to be split during insertions. In this case sample blocks are updated more often by RS-tree, which incur many I/Os. Because the data is stored with good locality, the additional I/Os does not impact RS-tree as much as other algorithms.

Regarding data deletion, all 3 algorithms slowly turn slower as q increases, as the elements are more likely to be scattered on the disk. RS-tree is about twice as slow as R-tree, while LS-tree is in between.

The difference between R-tree and RS-tree shows that the bookkeeping incurs relatively little penalty compared to the overall run time and I/Os. It is a reasonable choice of twice slow update performance, in exchange for significantly faster query performance.

Lastly, the same set of experiments were also carried out on the OSM data set, and nearly identical trends have been observed, hence,

those results are omitted for brevity.

7.6 Estimating q

The size q of a query result may be estimated as a side effect of online sampling, to evaluate the quality and the performance, we compare the result with standard range counting queries on R-trees.

Five random queries with various values of q were chosen for this test. For each query Q, the exact value of q is determined with a range counting query, and the time used is measured as t. Next we issue an online sampling query with Q, and terminate it when $10\% \times t$ time has been elapsed, meanwhile we estimate the size of the query result using the method mentioned in Section 6.

q (million)	Time (ms, $10\% \times t$)	Error (%)
1.96	41.49	0.03
3.85	19.60	0.007
7.85	36.94	0.0006
9.75	26.68	0.0007
17.68	32.12	0.03

Table 5: Estimating the size of query results.

Table 5 presents the time elapsed before we terminate the online sampling queries, as well as the relative error of the estimation. We can see q may be estimated quite accurately with a small amount of time. Note that when q is large, the estimation could be faster and more accurate, because when Q is decomposed in the tree, most regions of Q may be covered by subtrees whose MBBs are completely inside Q, and these subtrees do not contribute any error in the estimation!

We believe the quality of the estimation is acceptable in practice, and the exact value of q, when necessary, can always be determined with range counting queries using more time.

7.7 From sampling to aggregation

We further did experiments on using the returned samples for online aggregation. We used an extended data set of GEO where the altitude is the attribute for aggregation. Specifically, we issued the following query and computed the average altitude of the returned sample points as time progresses.

Latitude	1.04–39.85
Longitude	-39.27–180.00
Timestamp	1.55×10^8–3.97×10^8
q (query size)	4297219
average (altitude)	1801.67
σ (altitude)	4832.7

Table 6: The query

The results are shown in Figure 10, where the curves are the estimates as time goes on. We also plot the 95% confidence interval after every 80,000 samples are reported. From the figure, we see LS-tree is slightly faster at reaching the first 80,000-sample point, but RS-tree quickly catches up, and becomes faster later on. This agrees with the earlier experimental results in Section 7.2. At about 100ms, the relative error of the estimate is already below 0.4%. On the other hand, computing the exact average altitude for this query takes about 3 seconds, i.e., 30 times slower.

We also see from the figure that, for the same sample size, the confidence intervals of the two methods have similar length. This is expected, since the length of the confidence interval directly depends on σ^2/k. As both methods return uniform samples from the same query region, the distribution is the same, so is σ^2. Thus, for the same sample size k, the confidence intervals should have the same length. Of course, since σ^2 is estimated from the sample, there could be some small variations as the samples are randomly generated.

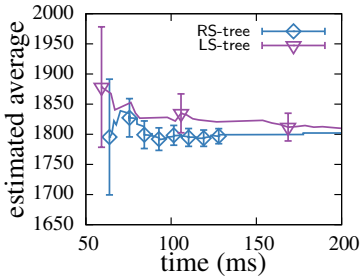


Figure 10: Spatial online aggregation: avg**(altitude).**

Other methods were again too slow compared to LS-tree and RS-tree so their results are not plotted.

8. RELATED WORK

The concept of online aggregation was first proposed in the classic work by Hellerstein et al. in [13], and has been revisited for different operators (e.g., join [11], group-by [32]) for relational data models, and computation models (e.g., MapReduce [30]). The standard approach for online aggregation is to produce online samples and build estimators that improve accuracy gradually over time using more and more samples [11–13, 30]. The translation from query accuracy (especially for standard aggregations) and estimation confidence to sample size is mostly well understood, see [11–13, 23, 30, 33] and many other work in the literature on building various kinds of estimators using random samples. Nevertheless, to the best of our knowledge, online aggregation has not been formally studied before for spatial and spatio-temporal databases.

Since the main technique for solving online aggregation is online sampling, naturally, our work is closely related to online sampling and sampling from a database in general. One of the earliest ideas was the RandomPath algorithm proposed by Olken [26]. This idea works for both B-tree in one dimension and R-tree in higher dimensions [26–28]. However, as explained in Section 3 and confirmed in our experiments, this method is too expensive for generating online samples in large spatial databases. Joshi and Jermaine [20] proposed the ACE tree that uses a binary tree (a k-d binary tree in higher dimensions) with samples stored in the leaf nodes. Samples in a query range can be obtained by combining samples from different leaf nodes. This design makes the structure inherently static, i.e., it does not support insertion or deletion of points in the data set. Lastly, Hu et al. [14] investigated the problem of producing samples for range queries with a new constraint that samples must be independent with respect to both intra-query and inter-queries. However, their result is purely theoretical, and is too complicated to be implemented or used in practice.

Our work is also related to producing random samples in an I/O-efficient matter. Existing works in this category concentrate on maintaining a large, disk-resident sample set over a continuous data stream [9, 19, 24], which are clearly different from our study.

There is an increasing interest in integrating sampling as an operator in a database management system; see recent efforts in [1, 2, 15, 16, 22, 25, 35, 36]. Nevertheless, none of them has investigated spatial and spatio-temporal online sampling and aggregation in large spatial and spatio-temporal databases. Our novel indexing structures are orthogonal to these efforts and can be implemented in these systems to support interactive spatial and spatio-temporal exploration through random samples.

The problem of finding samples from a collection of geographic points for displaying on a map is described in [8]. Samples are taken from an underlying data set such that the samples will be evenly distributed when the sampled data is drawn on a map. This differs from the problem solved in this paper, where we are concerned with producing random samples for spatial range queries such that we can perform statistical aggregations on the samples. Their definition of *spatial sampling* has a different objective which is to produce a better visual representation of the underlying data set.

Other than using random sampling, queries on spatial data can also be processed quickly and approximately using techniques such as data summaries, sketches, and signatures [5, 6, 34]. However, these techniques do not support online aggregation, i.e., the accuracy is fixed beforehand and will not improve as more time is spent.

Lastly, this work focuses on spatial range queries. Spatial kNN and joins are important queries worth of further studying. These are very difficult problems. In fact, even for natural joins on relational data, the problem of drawing a random sample without evaluating the join in full is already a difficult problem, with some negative results [7]. Further exploration of these problem represents interesting future work of this study.

9. CONCLUSION

This paper investigates spatial online sampling and aggregation. By designing novel indexing structures, we show that it is possible to produce online random samples efficiently for large spatial and spatio-temporal databases. Our designs are much more efficient and scalable than existing methods. The new designs also support dynamic updates efficiently. Using our novel indexing structures, one can produce spatial online samples and use these online samples to perform various spatial online aggregation and analytics. Our work leads to a number of interesting and important future directions to explore. The first challenge is how to ensure inter-query independence, which is recently proposed in the concept of independent range sampling (IRS), in spatial online sampling. Another challenge is to build more sophisticated, complex spatial online analytics and to support more types of spatial and spatio-temporal queries.

10. ACKNOWLEDGMENT

Robert Christensen and Feifei Li are supported by NSF grants 1443046, 1251019 and 1302663. Feifei Li is also supported in part by NSFC grant 61428204 and a Google research award. Lu Wang and Ke Yi are supported by HKRGC grants GRF-621413 and GRF-16211614, and by a Microsoft grant MRA14EG05.

11. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[2] S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden, and I. Stoica. Blink and it's done: Interactive queries on very large data. In *PVLDB*, volume 5, 2012.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] L. Arge, M. de Berg, H. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Transactions on Algorithms*, 4(1), 2008.

[5] L. Azevedo, G. Zimbro1, and J. de Souza. Approximate query processing in spatial databases using raster signatures. In *Advances in Geoinformatics*, pages 69–86. Springer Berlin Heidelberg, 2007.

[6] A. Belussi, B. Catania, and S. Migliorini. Approximate queries for spatial data. In *Advanced Query Processing*, volume 36 of *Intelligent Systems Reference Library*, pages 83–127. Springer Berlin Heidelberg, 2013.

[7] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, 1999.

[8] A. Das Sarma, H. Lee, H. Gonzalez, J. Madhavan, and A. Halevy. Efficient spatial sampling of large geographical tables. In *SIGMOD*, New York, New York, USA, 2012.

[9] R. Gemulla and W. Lehner. Deferred maintenance of disk-based random samples. In *EDBT*, 2006.

[10] A. A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[11] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, 1999.

[12] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, 1997.

[13] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *SIGMOD*, 1997.

[14] X. Hu, M. Qiao, and Y. Tao. Independent range sampling. In *PODS*, 2014.

[15] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. The monte carlo database system: Stochastic analysis close to the data. *ACM TODS*, 36(3):18, 2011.

[16] P. Jayachandran, K. Tunga, N. Kamat, and A. Nandi. Combining user interaction, speculative query execution and sampling in the DICE system. *PVLDB*, 7(13):1697–1700, 2014.

[17] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems*, 33(4), Article 23, 2008.

[18] C. Jermaine, A. Datta, and E. Omiecinski. A novel index supporting high volume data waresshouse insertion. In *VLDB*, pages 235–246, 1999.

[19] C. Jermaine, A. Pol, and S. Arumugam. Online maintenance of very large random samples. In *SIGMOD*, 2004.

[20] S. Joshi and C. Jermaine. Materialized sample views for database approximation. *IEEE Transactions on Knowledge and Data Engineering*, 20(3):337–351, 2008.

[21] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB*, 1994.

[22] A. Klein, R. Gemulla, P. Rösch, and W. Lehner. Derby/s: a DBMS for sample-based query answering. In *SIGMOD*, 2006.

[23] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[24] S. Nath and P. B. Gibbons. Online maintenance of very large random samples on flash storage. *PVLDB*, 1(1):970–983, 2008.

[25] S. Nirkhiwale, A. Dobra, and C. M. Jermaine. A sampling algebra for aggregate estimation. *PVLDB*, 6(14):1798–1809, 2013.

[26] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[27] F. Olken and D. Rotem. Random sampling from B+ trees. In *VLDB*, 1989.

[28] F. Olken and D. Rotem. Sampling from spatial databases. In *ICDE*, 1993.

[29] OpenStreetMap. OpenStreetMap.org, 2014.

[30] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. In *PVLDB*, volume 4, 2011.

[31] P. Weber. User-Generated Street Maps. *Pervasive Computing, IEEE*, 7(4):12–18, 2008.

[32] F. Xu, C. M. Jermaine, and A. Dobra. Confidence bounds for sampling-based group by estimates. *ACM TODS*, 33(3), 2008.

[33] Y. Yan, L. J. Chen, and Z. Zhang. Error-bounded sampling for analytics on big sparse data. *PVLDB*, 7(13):1508–1519, 2014.

[34] K. Yi, L. Wang, and Z. Wei. Indexing for summary queries: Theory and practice. *ACM Transactions on Database Systems*, 39(1), 2014.

[35] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo. ABS: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*, 2014.

[36] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, 2014.

[37] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma. Understanding mobility based on GPS data. In *UbiComp*, 2008.