COLR-Tree: Communication-Efficient Spatio-Temporal Indexing for a Sensor Data Web Portal

Yanif Ahmad
Brown University
yna@cs.brown.edu

Suman Nath
Microsoft Research
sumann@microsoft.com

***Abstract*— We present COLR-Tree, an abstraction layer designed to support efficient spatio-temporal queries on live data gathered from a large collection of sensors. We use COLR-Tree in a publicly-available sensor web portal to separate the concerns of sensor data management from the web portal application. COLR-Tree uses two techniques to optimize end-to-end latencies of users' queries by minimizing expensive data collection from sensors. First, it uses a novel technique to effectively cache aggregate results computed over sensor data with different expiry times. Second, it incorporates an efficient one-pass sampling algorithm with its range lookup to utilize cached data and compensate for occasional unavailability of sensors. We evaluate our implementation of COLR-Tree on SQL Server 2005 with a real, large workload from Windows Live Local. Our experiments demonstrate that COLR-Tree significantly improves both the end-to-end query performance and the number of sensors accessed compared to existing techniques.**

I. Introduction

With a rapidly increasing number of large scale sensor network deployments, the vision of a world wide sensor web is closer to becoming a reality. Ranging from camera or loop-sensor networks to monitor traffic on highways to weather sensors to report live weather conditions, these deployments generate tremendous volumes of useful data. To better harvest the potential of the data generated by these sensors, we have built SensorMap[1], a web portal that can host data generated by millions of sensors and lets users query that live data directly on a map in many useful ways. One example service SensorMap provides is a Restaurant Finder [1]. Supposing restaurants in a city periodically publish their current waiting times, users can query SensorMap for restaurants *with low waiting times* in a geographic region. SensorMap also supports multi-resolution aggregates: if the queried area is large and contains a large number of restaurants, SensorMap groups near-by restaurants together, and shows a distribution of waiting times for each group (to keep the on-screen information minimal). Users can zoom in on a particular area to get more detailed live data about the restaurants in that area. Moreover, a user can combine different types of live data, such as traffic conditions of roads leading to the restaurants, on the same map, to get an estimate of the total time required for driving to a restaurant and waiting there before dinner is served.

There are many issues that must be addressed to successfully design and implement such an application, and in this paper, we focus on one of the central challenges: *query processing*. In one implementation, the application (e.g., SensorMap) could naively use a custom program (*data collector*) to periodically collect sensor data[2] and populate a database that could process user queries. However, such decoupling of the database and the data collection program is inefficient since some data may be pulled into the database even if no queries use it. Such a solution is awkward too; applications on top of the database cannot effectively deal with user-specified data staleness—if the data collector has not collected data from some sensors within the staleness period specified by the user query, the cached sensor data cannot be used. We address these problems via a new abstraction called COLR-Tree (short for *Collection R-Tree* and pronounced "color-tree"), a novel spatio-temporal index based on the classic R-Tree [2]. It serves as an *independence* layer between sensors and applications. Applications issue relational queries on static metadata and realtime sensor data in COLR-Tree as if sensor data were already collected and stored in persistent tables. COLR-Tree transparently probes relevant sensors and collects data from them *on-demand*.

Coupling data collection with query processing presents a few challenges. First, collecting data from sensors on demand is expensive in terms of latency and bandwidth, especially when the query involves a large number of sensors. Second, sensors are largely heterogenous in terms of their availability. While some sensors can be probed for data almost any time (e.g., those connected to the Internet), some may only be probed when they are connected, working properly, and have the resources required to sense and communicate (e.g., a sensor on a cell phone). Finally, dynamically aggregating sensor data at different zoom levels of the map is computation intensive, resulting in a high end-to-end latency. A database system suitable for our target set of applications, unlike previous query processing systems, should optimize query

[1] `http://atom.research.microsoft.com/sensormap`

[2] Most publicly deployed sensors do not support continuously *pushing* data to a sink; rather, data needs to be *pulled* from them on demand.

processing to address all these challenges.

COLR-Tree uses two key ideas to solve these problems. First, it augments indexing with caching, so that query planning can take advantage of cached data and optimize the cost of collecting data from sensors. To reduce query latency, COLR-Tree precomputes sensor groups with different spatial resolution and dynamically caches individual sensor data as well as aggregate results at different resolutions. However, effectively caching aggregate data becomes extremely challenging due to the fact that different sensors publish data with different expiry times and a cached aggregate must be expired when at least one of the corresponding raw data expires. COLR-Tree uses a novel mechanism, called a *slot cache*, to effectively cache aggregate data over different temporal resolutions.

Second, to bound the data collection cost per query, COLR-Tree samples a subset of sensors (instead of all the sensors) within the query region to compute aggregate results. In the presence of COLR-Tree's caching and occasional sensor failures, selecting a target number of random sensors and probing them are not sufficient; for example, more sensors need to be probed in sub-regions with less data in the COLR-Tree cache, and more sensors than the target sample size need to be probed in parallel to compensate for occasionally unavailable sensors. We provide an efficient one-pass sampling algorithm that deals with these challenges and provides provable guarantees on the expected number of successfully probed sensors and the sensing workload uniformity.

We implement COLR-Tree in Microsoft SQL Server 2005 as an abstraction layer on top of the database engine. It is being used as the live database backend of SENSORMAP, which has been publicly available for over a year.

In summary, this paper makes the following contributions.

1) We present COLR-Tree, a database abstraction layer that hides the messy details of data collection from applications and acts as an independence layer between applications and sensor data.
2) We present slot-cache, a novel technique for effectively caching aggregate results computed over sensor data with different expiry times. We show how to efficiently maintain slot-caches in COLR-Tree nodes and how to utilize the cache during lookup.
3) We incorporate an efficient one-pass sampling algorithm with COLR-Tree's range lookup that leverages cached data and handles occasional unavailability of sensors. The algorithm provides provable guarantees on the number of successfully probed sensors and the uniformity of sensing workload.
4) We evaluate our COLR-Tree implementation on Microsoft SQL Server 2005 with a real workload. Our evaluation shows that caching and sampling are able to reduce processing latency to approximately 20% and provide more than a factor of 30 reduction in the number of sensors accessed in comparison to collection-agnostic techniques.

The rest of the paper is organized as follows. Section II discusses related work. Section III describes SENSORMAP and its requirements that are addressed by COLR-Tree. Section IV and Section V describe COLR-Tree's caching and sampling techniques. Section VI and Section VII describe the design and evaluation of our index. Finally, we conclude in Section VIII.

II. RELATED WORK

Several recent works have tackled query processing challenges for wide-area sensing systems. IrisNet [3] provides techniques to process queries over a distributed XML document containing sensor data. IrisNet builds hierarchical indices of sensors and applications need to explicitly specify the parts of a hierarchy that the query will traverse. In contrast, COLR-Tree takes a relational approach over a flat sensor collection. This enables us to insulate applications from several complexities such as hierarchical organization of the sensors. IceDB [4] presents a delay-tolerant query processor to address intermittent and variable connectivity. It focuses on individual sensors prioritizing its *push* of query results to a central server. In contrast, COLR-Tree selectively *pulls* data from an appropriate subset of sensors and focuses more on spatio-temporal query processing at the central server. MauveDB [5] supports model-based user views in database systems. Such modeling is orthogonal to our work, and could be used in COLR-Tree (e.g., COLR-Tree can maintain a model from its cached data).

The database literature contains many examples of index structures, each customized for various query types, including spatial and temporal queries. An overview of recent spatio-temporal access methods by Mokbel et al. may be found in [6]. Our work is inspired by the R-Tree by Guttmann [2] and the bulk-loaded R-Tree of Kamel and Faloutsos [7]. Both the RT-Tree and the 3D R-Tree have added temporal search to the R-Tree by including temporal metadata at each tree node. The multiple-resolution aggregation (MRA-) tree by Laziridis and Mehotra [8] is an index structure maintaining standard aggregates such as min, max, sum, count at each node in the tree structure. However the authors do not account for real-time and as such do not discuss how to manage frequently changing data. The aRB-Tree by Papadias et al. [9] is similar in nature to the MRA-Tree with the exception that multiple aggregates are maintained over time for every internal R-Tree node, and the temporal dimension is indexed with a standard B-Tree. The SB-Tree [10] describes an index structure to maintain temporal aggregates and service time window queries. Each node in the tree maintains aggregates for multiple time segments, in a similar manner to our aggregation per slot. However the SB-Tree time segments are of arbitrary lengths, and are determined by the insertion order into the index. The SB-Tree does not consider any spatial search component in its design, nor does it investigate the use of sampling techniques. To the best of our knowledge, COLR-Tree is the only index structure focusing on data collection issues by tightly coupling both a sampling and caching algorithm with index seek.

COLR-Tree's sampling algorithm significantly differs from existing algorithms [11], [12], [13] in that they do not deal with sensor unavailability and cached data, which make the sampling problem more complex. Moreover, in a single pass, many of these algorithms select one random data point from a spatial database, while our algorithm selects a user-specified number of random sensors, even in the presence of sensor unavailability.

III. The SensorMap Portal

COLR-Tree is designed to address unique requirements of SensorMap, a web portal for live sensors. SensorMap consists of two high-level components: the portal service and a back-end database.

A. SensorMap Usage

The portal acts as the rendezvous point for sensor data publishers and users. Data publishers publish their sensors in SensorMap by registering them with static metadata such as sensor locations, data types, data expiry times, etc. SensorMap users see a map that they can zoom and pan (like Google Maps [14] or Windows Live Local [15]) to locate the geographic area of interest. They can also query for sensor data by specifying polygonal regions of interest, types of sensors, and a few keywords describing the sensors of interest. Upon receiving a query, SensorMap collects live data from the relevant sensors and presents them to the user by overlaying them as icons on top of the map.

B. Back-end Database

The back-end database of SensorMap is required to process queries with the following requirements.

Users issue spatial queries with a fixed-size viewport. Displaying results of a query made over a large geographic region (e.g., a whole state) in a fixed-size viewport (i.e., computer monitor) causes some sensor icons to visually overlap with near-by sensors, hiding them from users. Moreover, too many icons representing individual sensors make it difficult for users to infer underlying information. To deal with these problems, SensorMap requires the database to group near-by sensors (those who would overlap with each other in the display) and to compute aggregate information for each group. Moreover, the database must optimize data collection, grouping, and aggregation to reduce the latency perceived by users.

Sensor data becomes stale. Users may also specify the maximum staleness of sensor data. If no data of sufficient freshness is present in the cache, it must be collected from relevant sensors.

Data collection costs must be constrained. SensorMap is configured with the maximum number of sensors that can be contacted per query; so, if someone asks for a query for the whole world, SensorMap will not try to collect data from all the sensors in the world; rather it will collect data from the maximum number of sensors distributed roughly uniformly over the world.

We now present an example query that SensorMap issues to the back-end database.

```
SELECT count(*)
FROM sensor S
WHERE S.location WITHIN Polygon(<lat,long>)
AND S.time BETWEEN now()-10 AND now() mins
CLUSTER 10 miles
SAMPLESIZE 30
```

Here, the query asks for a count of sensors in a region, defined by a polygon whose vertices are given by latitude and longitude locations, within the specified time window. Furthermore, the query requests that live data is obtained from a subset of 30 sensors inside the region, and that sensors within 10 miles of each other are grouped together and a count aggregate result should be computed for each group.

C. The COLR-Tree Index

COLR-Tree is designed to meet the above query requirements. It is based on an R-Tree, a classic multidimensional index [2], with several extensions. COLR-Tree needs to aggregate data in different spatial granularities (corresponding to different zoom levels of the map or `CLUSTER` values in queries). We assume the locations of sensors do not change often, allowing COLR-Tree to be built bottom-up, in batch mode, by iteratively computing sensor clusters with a k-means algorithm [16] to construct a hierarchy. We periodically reconstruct the COLR-Tree index to reflect any change in sensor locations.

The basic query processing in COLR-Tree leverages the containment relationship between parent and child nodes. Starting at the root of the tree, the lookup algorithm prunes nodes whose location ranges do not overlap with the query range, since these are guaranteed not to contain any entries that could be a result for the query. Given that COLR-Tree nodes may spatially overlap, the algorithm descends down multiple paths through the tree. The lookup algorithm completes its descent along any path when it encounters a *terminal node* which is below a threshold level (depending on the query's zoom level) and is contained entirely within the query region. At this point, COLR-Tree collects readings from the terminal node's descendant sensors.

In our context, the algorithm above would incur high end-to-end response time due to high data collection latency from sensors. To avoid this, COLR-Tree uses two techniques. First, sensor readings and aggregates are cached in both leaf and non-leaf nodes of COLR-Tree. Second, COLR-Tree supports computing approximate aggregate results by probing a subset of sensors of interest. We elaborate on these two mechanisms in the next two sections.

IV. COLR-Tree Caching

Due to the presence of spatio-temporal locality in query workloads, caching allows COLR-Tree to re-use already collected data (avoiding expensive communication) and to deal

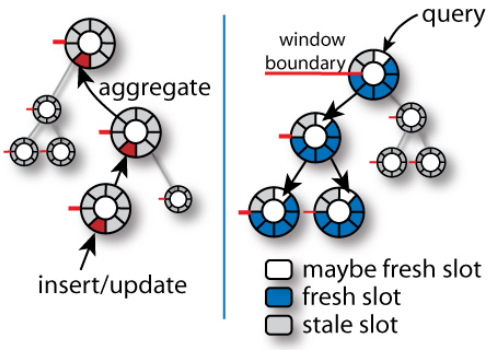

Fig. 1. COLR-Tree's slot cache mechanism.

with temporarily disconnected sensors. However, unlike a traditional database architecture where caching is decoupled from indexing, COLR-Tree combines them: leaf nodes cache raw sensor readings, while internal nodes cache aggregates computed over the set of sensor readings from their descendents. Incorporating caching with indexing has two advantages. First, since non-leaf index nodes cache aggregate data, top-down query processing can stop at non-leaf nodes, reducing query latency. Second, the data collection plan can be optimized based on the cached data at non-leaf nodes. We elaborate this in Section V. Note that caching aggregate data has the added benefit of reducing the computational overhead of query processing in addition to any benefits from accessing data.

While caching raw sensor data is straightforward, caching aggregate data is challenging due to the fact that different sensor readings collected by COLR-Tree expire after times specified by their sources. An expiry time is a fixed range indicating the validity of the reading. Consider n sensor readings with expiry times in the range $[t_{min}, t_{max}]$. If we cache an aggregate computed over these n readings, it needs to expire after t_{min}, because it is the point when at least one of the constituent values expires, and, depending on the aggregation function, the expired data can affect the aggregate in arbitrary ways. Since a typical aggregate includes a large number of sensors, t_{min} can be very small (i.e., stale), seriously limiting the usefulness of aggregate caching. COLR-Tree addresses this problem with a *slot-cache*.

A. Slot Cache

A slot-cache in a non-leaf COLR-Tree node maintains $m > 1$ aggregates in m slots, where $m = t_{max}/\Delta$, t_{max} is the maximum expiry time of sensors, and Δ is slot size. Slot i maintains the (partial) aggregate of the subset of sensor readings whose expiry times are within the range $((i-1)\Delta, i\Delta]$. In effect, a slot-cache maintains multiple partial aggregates over multiple time windows. COLR-Tree queries are answered considering all partial results stored in all slots. We allow slots to *slide*, factoring in the continuously changing nature of sensor data. When sliding, the slot-cache advances one slot at a time in every Δ units of time, expiring all entries lying in the oldest slot $((m-1)\Delta, m\Delta]$. Aggregates in other slots remain useful for subsequent queries. We now describe a slot cache's primitive operations.

Insert. Sensor readings are inserted into a node's slots using a hash function on timestamps. During the insertion, we detect if the reading's timestamp lies beyond the newest slot. In this case, we slide the slot-cache until the youngest slot covers the reading's timestamp. Cache insertions of large sets of readings may also violate a cache size constraint. Here, our replacement policy is to evict the *least recently fetched* readings from the cache lying in the oldest slot, in just as the entries are evicted following a slide.

Lookup. A query accesses the slot-cache according to its freshness requirement. The timestamp corresponding to the freshness bound is hashed using the same function as with insertions, yielding the *query slot*. The query slot determines the useful readings in the current cache as those entries lying in slots which are strictly younger than itself, through to the window's boundary.

B. Slot Cache Tree

COLR-Tree maintains a slot-cache at every tree node to provide cached aggregates at multiple spatial and temporal resolutions. For an internal node, each slot in the cache represents an aggregated value over the sensor readings in the same slot-caches in the node's descendents. Thus the slot maintains a spatio-temporal aggregate at a temporal resolution corresponding to the size of a slot, and at a spatial resolution according to the bounding box of the internal node.

Insert, update. Insertions of new sensor readings at the leaf level result in bottom-up updates through the tree's slot caches. A new or updated entry in a leaf's slot triggers an update to the aggregate value in the same slot, in the parent's cache. For inserts, the aggregate update may be performed incrementally. However in the case of updates to an existing value, we must decrement the old value from any aggregate using the reading. This may or may not be performed incrementally depending on the aggregate function (e.g. sum and count support a decrement operation, while min and max do not). We are only able to perform this per-slot aggregation given a globally aligned slotting scheme for all of the slot-caches in the tree.

Lookup. Lookups first take advantage of the global temporal alignment of slots to perform a pre-traversal elimination of slots containing stale entries. Then, during tree traversal, the test at a node is extended to terminate early if the sensor reading or aggregate is indeed cached at the node. Any leaf entries lying in the query slot itself must be inspected further by comparison of the entries' timestamps against the freshness bound timestamp, to determine whether the cached readings may be used to answer the query.

Figure 1 illustrates basic operations.

C. Optimal Slot Size

Even though a slot can be of any size Δ within the range $(0, t_{max}]$, the performance of a slot cache depends on Δ. We now use a utility-cost analysis to find a slot size that optimizes the overall performance of COLR-Tree. A similar analysis has

been previously used to find the optimal node size of an index stored in a disk [17] or a flash device [18].

Intuitively, larger slots have the benefit that fewer partial results need to be combined to produce the final result. Without loss of generality, we normalize t_{max} to 1. When the slots are of size $\Delta < t_{max}$, answering a query with a time window T would require combining $\lfloor T/\Delta \rfloor$ slots. This would also require updating $\lceil T/\Delta \rceil \times f$ slots with new data, where f is the fraction of times data for a specific slot is collected from sensors. Finally, the data from sensors in $T - \lfloor T/\Delta \rfloor \times \Delta$ slots, outside the usable ones, need to be collected. Thus, the total *cost* incurred by a query due to slots of size Δ is given by:

$$cost \sim \lfloor T/\Delta \rfloor + \lceil T/\Delta \rceil \times f + (T - \lfloor T/\Delta \rfloor \times \Delta) \times c$$

Where c is the collection cost normalized to the processing cost of a slot. Note that the cost depends on the actual query workload: T depends on the query workload and f depends on how often queries arrive.

On the other hand, smaller slots provide the benefit that partially aggregated information can stay in the cache for longer, before it gets discarded (remember that slots are discarded one by one). We define the *utility* of a slot size as the average time the data from a sensor, in aggregated form, can remain valid in slot cache. Suppose we have k slots $< s_1, s_2, \cdots, s_k >$, where $k = \lceil 1/\Delta \rceil$. Suppose, sensor expiry time distribution is such that expiry times of n_i sensors on average fall within the slot s_i. Then all the aggregated information from n_i sensors will remain valid in the cache for time $(i-1)\Delta$. In other words, after time $(i-1)\Delta$, data cached in slot s_i will be discarded because at least one of its constituent data may expire at that time. Thus, the total utility of slots of size Δ is given by:

$$utility \sim \sum_i n_i(i-1)\Delta$$

Like cost, utility also depends on the workload since n_i depends on the distribution of sensor expiry times.

The optimal slot-size relies on the sweet-spot within the above two tradeoffs; i.e., the ratio of utility and cost is maximized at the optimal slot size. Figure 2 shows the utility-cost ratio of three workloads: Uniform is a hypothetical sensor deployment with expiry times uniformly distributed within the range [0,1], USGS denotes the expiry times of $\approx 10,000$ United States Geological Survey sensors (collected from `www.usgs.gov`), and Weather denotes the expiry times of $\approx$ 1000 personal weather stations (collected from `www.WeatherUnderground.com`). For all scenarios, we use a real query workload described in Section VII. As shown in the figure, the utility-cost ratio is optimal at different slot sizes for different scenarios. For Uniform, optimal slot size is 0.5 (i.e., the slot cache has 2 slots), for USGS, it is 0.8, and for Weather, it is 0.2. COLR-Tree can be configured with the optimal slot size found by using the target workload in the above framework.

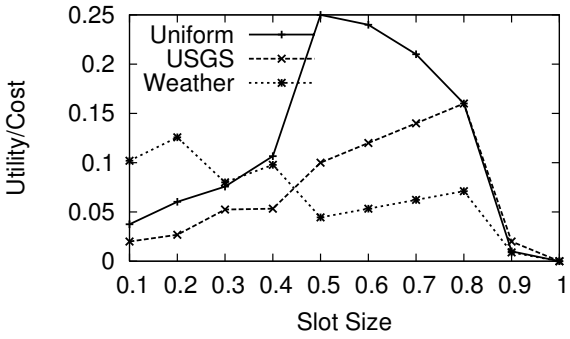

Fig. 2. Utility/cost ratio with different sensor expiry times.

V. COLR-Tree Sampling

To reduce the number of sensors probed for readings, and thus the communication cost and end-to-end latency, COLR-Tree provides the option to compute approximate answers by probing a subset of sensors within the query region. One simple approach to select a sample (i.e., subset) is to first obtain the complete sensor list by using the standard range query, and then to randomly sample from the result. However this approach is inefficient since it offers no advantage in terms of latency over the regular range lookup. We present a one-pass algorithm that samples sensors *during* range lookup in COLR-Tree.

COLR-Tree samples an application specified number R of sensors with several goals in mind. First, sensors are selected uniformly randomly to both distribute the sensing load on sensors, and to provide more sensors from "interesting" areas with denser deployment. The second goal of sampling deals with sensor unavailability. Since some of the probed sensors occasionally fail to return required data (e.g., due to failure or disconnection), COLR-Tree selects $R' > R$ sensors such that probing all R' sensors would provide readings of the target size R. Sampling utilizes aggregate and raw data in cache as much as possible. Existing spatial sampling algorithms [11], [12], [13] do not address caching and over-sampling.

A. Layered Sampling

Algorithm 1 shows the pseudocode of COLR-Tree's sampling algorithm, called *layered sampling*. In addition to a target sample size R and a query region A, it takes two threshold levels[3]: O and T. In our implementation, one sample (or aggregate computed over the sample) is returned for each non-leaf node at level T, and it can be adjusted based on the zoom level of the map. For simplicity, the pseudocode in Algorithm 1 returns only the union of all the samples. The other threshold O is used during over-sampling (described below). The algorithm has the following key ideas.

Weighted partitioning of sample size. The basic idea of COLR-Tree's layered sampling is to allow siblings in the tree to independently choose their samples from their descendents.

[3]The root node is at level 0

Algorithm 1 SAMPLE(R, A, O, T)

Require: A number $R \geq 0$ of sensors to probe, an area of interest A, an oversampling level O, and a result threshold level T.
Ensure: A sample.

Definitions:
w_i: weight of the node i
c_i: cached sensors at node i
a_i: mean availability of sensors below node i
$BB(i)$: bounding box of node i
$Overlap(A_1, A_2)$: fraction of A_1 overlapping with A_2

1: $sample \leftarrow \emptyset$
2: $nodes \leftarrow PriorityQueue()$
3: $insert(nodes, R, root)$
4: **while** $|nodes| > 0$ **do**
5: $r, n \leftarrow pop(nodes)$ {r is the priority, i.e., target sample size, of node n}
6: $totalFetched \leftarrow 0$
7: **for** child i in $children(n)$ **do**
8: **if** $BB(i)$ is inside A **and** $Level(n) > T$ **then**
9: $r_i \leftarrow r \times \frac{w_i \times Overlap(BB(i),A)}{\sum_i w_i \times Overlap(BB(i),A)} - |c_i|$
10: **if** $Level(n) < O$ **then**
11: $r_i \leftarrow r_i / a_i$
12: $s \leftarrow r_i$ random sensors under i
13: $totalFetched \leftarrow totalFetched + |s|$
14: $d \leftarrow$ probe sensors in s and successfully collect data from available sensors
15: $sample \leftarrow sample \cup d \cup c_i$
16: **else if** $BB(i)$ overlaps with A **then**
17: $r_i \leftarrow r \times \frac{w_i \times Overlap(BB(i),A)}{\sum_i w_i \times Overlap(BB(i),A)}$
18: **if** $Level(n) = O$ **then**
19: $r_i \leftarrow r_i / a_i$
20: $totalFetched \leftarrow totalFetched + r_i$
21: $insert(nodes, r_i, i)$
22: **if** $totalFetched < r$ **then**
23: REDISTRIBUTE$(nodes, r - totalFetched)$
24: **return** $sample$

Algorithm 2 REDISTRIBUTE(N, F)

Require: A priority queue N of tree nodes with priority as the number of sensor probes assigned to the nodes, a number of additional probes F to distribute amongst the given tree nodes.

Definitions:
$priority(i)$: the priority for a a node $i \in N$

1: $incr \leftarrow F \times \frac{priority(i)}{\sum priority(i)}$
2: **for** node i in N **do**
3: $priority(i) \leftarrow priority(i) + incr$

The difficulty with independent sampling lies in the ability to precisely control the size of the resulting sample. We adopt the following strategy. Starting at the root, with a sample target size specified by the user, we descend along nodes relevant to the query, splitting the target size recursively amongst children. Thus each child is asked to return a sample smaller than the original target size, so that consequently when the samples from each child is combined, we meet our target size. Line 17 of Algorithm 1 shows how a node partitions its sample size among its children. Each child node i gets a target size which is proportional to its weight w_i normalized by the fraction of its bounding box overlapping with the query region. The weight w_i can be defined to suit the desired semantics of the sampled answer. We assume applications want uniformity over sensors, and set w_i as the number of descendant sensors at node i.

Oversampling. To cope with sensor unavailability, a non-leaf COLR-Tree node scales up the target sample size to $R' > R$ such that when a random R' of its descendent sensors are probed, R sensors are found to be available. To reduce probing complexity, R' should be as small as possible. However, an absolute guarantee of R out of R' successful probes is not feasible in practice since non-leaf nodes scale up the target size before sensors are actually probed and individual sensors may be found unavailable in nondeterministic ways. Moreover, nodes independently scale up their target sizes, and do not block while other sensors are accessed by other nodes. We therefore provide a probabilistic guarantee: R' is chosen such that when all of them are probed, an expected number of R sensors will be available to provide data.

To determine R', we use the historical availability of individual sensors which has proved to be effective in predicting the future availability of the sensor. Suppose, the target sample size is R over m sensors $(s_1, s_2, \ldots, s_m)$ with availabilities $(p_1, p_2, \ldots, p_m)$. Then, the probability that a randomly probed sensor will be available to produce readings is $a = 1/m \times \sum_{i=1}^{m} p_i$. The probability that exactly R sensors will be available out of R' probed sensors follows a negative binomial distribution, with an expected value of $R' = R/a$.

The value of a could be computed with a range query on a COLR-Tree built over sensor availability information. However, this would result in a two-pass algorithm: first computing a over the query region, and then using it during lookup. Instead, during lookup, we scale up the target size by computing a at nodes whose bounding boxes BB are entirely within A (line 8 of Algorithm 1). Such scaling up is done at nodes within a threshold level O such that the nodes have enough sensors under them to oversample. Finally, we make sure that the sample size is scaled up exactly once in any path from the root to a node probing sensors, either at the first node below level T whose bounding box is entirely inside A, or the node at level O if no node above level O has its bounding box entirely inside A. This is necessary to ensure correctness of the algorithm (Section V-B).

Redistribution of sample sizes. The above oversampling algorithm provides a probabilistic guarantee of achieving a target sample size and may sometimes fail to provide the target size. This may happen due to nondeterministic sensor unavailability and holes and nonuniform distribution of sensors in bounding boxes. In such cases, if the sample size lags behind the target size for some nodes of the tree, the lag is compensated by the REDISTRIBUTE subroutine by evenly distributing it among nodes yet to be probed. This increases the probability that a target sample size is achieved even in the presence of sensor deployment irregularity.

Exploiting leaf and non-leaf caches. Before probing sensors, a node checks its cache for sensors that satisfy the query, fol-

lowing which, only the additional number of sensors required to satisfy the target sample size are probed (line 9 and line 15).

B. Sampling Properties

The following two theorems show the desirable properties of our sampling algorithm.

Theorem 1: Algorithm 1 returns a sample with an expected size of R.

Proof: Suppose the priority queue Q created in line 2 of Algorithm 1 contains two logical classes of nodes: (1) S, nodes whose sample sizes have already been scaled up (i.e., if a node n is in S, it or one of its ancestors has executed line 11 or 19.), and (2) NS, nodes who are not in S. We will now show that after each round of the while loop of Algorithm 1, the algorithm maintains the following invariant:

$$\sum_{n \in NS} r_n + \sum_{n \in S} r_n \cdot a_n + sample \approx R \qquad (1)$$

where $sample$ is the variable defined in line 1 of Algorithm 1 and $\approx$ denotes the expected size. At the end of the algorithm $S = NS = \emptyset$; therefore, according to the invariant, $sample \approx R$, which proves the theorem.

The invariant holds in the beginning of the while loop because NS contains only the root node with priority R, $S = \emptyset$, and $sample = 0$. Now suppose, a node $n \in NS$ is popped out in the beginning of a while loop. This reduces the value of the left hand side (LHS) of Equation 1 by r_n. Suppose, n's children are C with $|C| = m$. Without loss of generality, assume that the sample size r_n is partitioned for first k children (line 17), partitioned and scaled up for next l children (line 17 and line 19), and scaled up for actual probing for next (m-k-l) children (line 11 and line 12). In the first case, k nodes are added to NS, increasing the value of LHS by $\sum_{i=1}^{k} r_n O_i$, where $O_i = \frac{w_i \times Overlap(BB(i),A)}{\sum_{j \in C} w_j \times Overlap(BB(j),A)}$. In the second case, l nodes are added to S, increasing the value of LHS by $\sum_{i=k+1}^{k+l}((r_n O_i)/a_i) \times a_i = \sum_{i=k+1}^{l} r_n O_i$. In the third case, the expected value of $sample$ increases by $\sum_{i=l+1}^{m}[(r_n O_i - c_i)/a_i] \times a_i + c_i = \sum_{i=l+1}^{m} r_n O_i$. Thus the total increase of the value of LHS is given by $\sum_{i=1}^{m} r_n O_i = r_n \sum_{i=1}^{m} O_i = r_n$, exactly offsetting the decrease of LHS due to popping out the node n.[4] Thus the invariant is maintainted. Similar argument can be given for the case when $n \in S$.

Theorem 2: Suppose sensors are uniformly distributed within the bounding boxes that partially overlap with query region A and caching is disabled. Then, Algorithm 1 with a target sample size R successfully collects data from each sensor in A with the equal probability of R/N, where N is the total number of sensors in A.

Proof: Consider a path $P = (n_0, n_1, \ldots, n_{s-1}, n_s, \ldots, n_{t-1}, n_t)$, where n_0 is the root, n_s is the node where sample size is scaled up (line 11 or line 19), and n_t is the terminating node where sensors are probed (line 13). As mentioned before, the sample size is scaled up in exactly one node n_s in this path.

We now show that each sensor below n_t is successfully probed with the probability R/N. Under the above assumption, $w_i \times Overlap(BB(i), A)$ denotes the expected number of sensors η_i below node i that are within A. We denote the *sampling ratio* of a node n as x/y where x of total y sensors need to be selected below node n. Clearly, at n_0, the sampling ratio is $R/\eta_{n_0} = R/N$. At n_1, $x = R \times \frac{\eta_{n_1}}{\sum_{i \in child(n_0)} \eta_i} \approx R\eta_{n_1}/\eta_{n_0}$ and $y = \eta_{n_1}$. Thus, the sampling ratio at n_1 is $x/y = R/\eta_{n_0} = R/N$. Similarly, it is easy to show that the sampling ratio remains the same R/N in all nodes $n_0, \ldots, n_{s-1}$. Similarly, the sampling ratio remains $\frac{R/a_i}{N}$ in all nodes $n_s, \ldots, n_t$. At n_t, only a_i fraction of the probed sensors, on expectation, return data, and therefore, the ratio of sensors successfully returning data to all the sensors below n_s is given by $\frac{R/a_i}{N} \times a_i = R/N$. Since the sensors are selected uniformly randomly below n_t, each sensor below n_t has the equal probability of R/N.

VI. COLR-Tree Implementation

COLR-Tree is implemented entirely on top Microsoft SQL Server 2005 using relations to represent the tree and cache structures, and T-SQL to manipulate these structures declaratively. Our reasons for choosing a relational implementation included reducing communication and context switching between the front-end portal service and the back-end database by implementing access methods as a single query, in addition to rapid prototyping through the use of a high-level language, and extensive tool reuse.

A. Index Schema and Access Methods

The database schema implementing our index uses a layered approach similar to work by Bohm et al. [19]. Here, each tree layer is represented by a table (denoted the node table) whose schema includes the following attributes:

layer = {node id, child id, child bounding box, child weight}

Child identifiers in an upper level layer table are present as node identifiers in a lower level layer table. We traverse the tree by joining two adjacent layers' node tables on a child identifier. Each layer table has a corresponding cache table representing cached sensor readings of all nodes within the layer. Cache tables contain the following attributes:

cache = { node id, slot id, value, value weight}

The value represents an aggregate over the node's descendents and the value weight maintains the size of the aggregation set producing this value.

Sensor Selection Access Method. The sensor selection access method returns a set of sensor identifiers for the frontend to probe for fresh readings. Our implementation of this algorithm is as a multiway join on the layer tables, executed as a left deep join tree that joins each layer's node table and cache table from root to leaf layer. At each layer we check for sufficiently cached nodes, otherwise we perform sampling. This requires that we compute the total number of relevant cached nodes for the sampling heuristic, by filtering spatially with a join

[4] If, in the third case, less than a_i fraction of probes are successful, the value of $sample$ can be smaller. However, in that case the REDISTRIBUTE subroutine distributes the lag among nodes in S and NS, keeping the invariant true.

predicate and aggregating cache value weights across slots. The sampling heuristic further reduces the nodes we consider traversing at lower layers.

Cache Read Access Method. A cache read is a union of multiple join trees, which individually are similar to the join trees for sensor selection. Each join tree declaratively specifies that we want to retrieve cached entries at a specific layer that lie entirely within the query region and have a timestamp of newer than our freshness requirement. We also specify that no contained cached entry exists in a higher level to eliminate duplicate readings. Retrieving cached entries from the leaf layer requires an additional predicate on the timestamp of the sensor readings, where we directly compare the readings' timestamps and the query's freshness timestamp in addition to a comparison of slot identifiers.

B. Cache Maintenance

We now describe our mechanism to manage the contents of the slot cache. This includes the window and caching policies' implementation, and the aggregate maintenance within each slot and cache. Our basic design is to leverage SQL triggers to perform these tasks, and we present four triggers which execute after the insertion of new readings into the leaf cache level.

Roll trigger. This trigger is responsible for managing the window extents, the slot identifiers, and the alignment of all tree nodes' cache slots. This trigger advances the start time in increments of slot periods, until the latest insertion lies in the most recent slot, and only fires on insertions to the leaf cache level. The roll trigger implements the window expiration cache replacement policy, by expunging the values in any slot the window slides over.

Slot insert trigger. This trigger handles new sensor readings that are not already present in the index structure. The trigger increments all aggregates in the same slot as the reading, in the level directly above the leaf cache level. The slot insert trigger implements the cache size constraint, and the least recently fetched replacement policy, by checking the size of the cache following an insertion.

Slot delete trigger. This trigger fires on deletions in the leaf cache level, specifically on slot rolls or on violations of the cache size constraint. Its primary function is to update the sampling weight of the cache table and to initiate expirations at the tree layer above the leaves.

Slot update trigger. This trigger is the only trigger firing on cache tables above the leaf layer. Note that the above slot roll, insert and delete triggers only execute update statements above the leaf layer, which modify the sampling weight and the cached value. The update trigger's responsibility is to propagate this update through all cache tables to the root.

VII. Experimental Evaluation

In this section we experimentally evaluate the performance of our COLR-Tree implementation in SQL Server 2005. While the majority of experiments are performed on this setup, we prototyped an equivalent design in Java on top of the Berkeley DB storage engine so that we may directly instrument internal COLR-Tree statistics. We ran these experiments on a desktop class machine with an AMD Athlon 3200+, 2GB RAM and a Western Digital WD160 drive.

A. Windows Live Local Dataset

For our evaluation, we used a workload consisting of two real datasets from Windows Live Local [15]: a query set and a sensor set. The query set consists of 106,000 Windows Live Local queries, each query asking for restaurants in a rectangular geographic region in the USA. The sensor set consists of approximately 370,000 restaurants in Windows Live Local YellowPages directory. These two datasets represent the workload for a Restaurant Finder Service mentioned in Section I: each restaurant periodically publishes their current waiting times for tables and users query restaurants with small waiting time in a region. We use the above workload to understand the effectiveness of COLR-Tree's sampling and caching strategy.

B. Query Processing Complexity

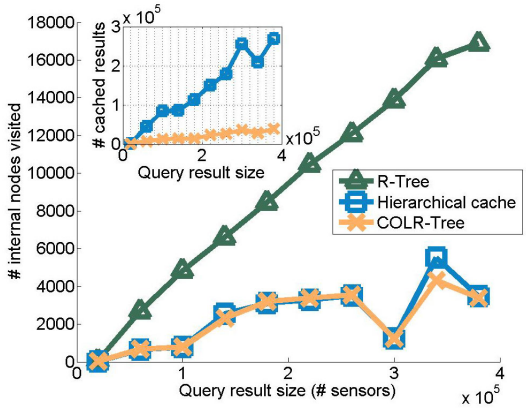

Fig. 3. COLR-Tree internal node traversal analysis.

We first analyze the performance of COLR-Tree using a set of internal data structure statistics collected during query processing. For this experiment, we configured COLR-Tree in three ways, first as a regular R-Tree (i.e. with no caching or sampling), next as a *hierarchical cache* (i.e. with slot-caches and a standard R-tree range query), and finally in its full-fledged form with both caching and sampling enabled.

Figure 3 shows the number of index nodes traversed for each query processed from the Live Local workload, as function of the query's ideal result set size (i.e the total number of sensors lying in the query region). Queries are binned according to their result set size, and we compute an average node traversal count per bin. Figure 3 verifies that for a standard R-Tree query the number of internal nodes visited grows linearly with the number of leaf nodes accessed. This occurs as a result of the uniformity of our clustering, where we verified near uniform distributions of internal node weights (i.e., number of descendents) per layer at lower tree layers. Both the hierarchical cache and COLR-Tree configurations access extremely similar numbers of internal nodes. The nested

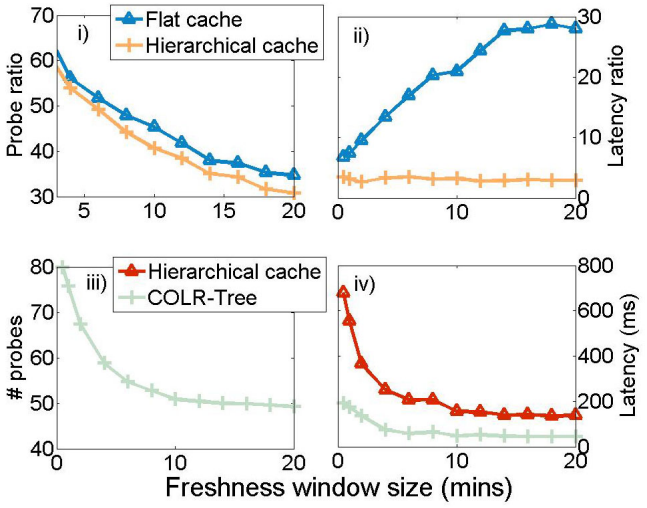

Fig. 4. **Ratio of:** i) sensor probes compared to COLR-Tree, ii) processing latency compared to COLR-Tree, and **absolute values of:** iii) sensor probes, iv) processing latencies, over varying freshness windows.

plot displays the number of cached nodes accessed by these two configurations and when considered with the number of node traversals, illustrates the tradeoff between caching and sampling. Slot cache is able to reduce nodes traversed due to the presence of sufficiently fresh cached aggregates. COLR-Tree on the other hand caches much fewer nodes (between 5-8x fewer nodes), but uses sampling on top of caching to reduce the number of nodes traversed.

C. End-to-end Performance

Figure 4 compares end-to-end results for various COLR-Tree configurations. These configurations include a simple *flat cache* that maintains only raw sensor readings (and not aggregates) and is scanned for query processing, as well as the hierarchical cache and standard COLR-Tree setups from the previous section. Figures 4i and 4ii compare the number of sensors probes and the processing latency of the flat cache and hierarchical cache setups to COLR-Tree as ratios. COLR-Tree is clearly beneficial for communication costs relative to algorithms lacking sampling, as it is capable of reducing sensors probes by a factor of 30-100x fewer probes over varying freshness requirements. In terms of latency, both indexed configurations dominate the flat cache, and COLR-Tree further provides a 3-5x reduction over the hierarchical cache. This arises due to the extra cache lookups and maintenance performed by the hierarchical cache over COLR-Tree as we saw from Figure 3. Figures 4iii and 4iv display the corresponding absolute values of probes and processing latencies. In particular Figure 4iv indicates the viability of supporting high throughput query processing for SENSORMAP as we can provide a low processing latency of approximately 40ms per query. Figure 4iii shows that as we weaken freshness requirements, COLR-Tree is able to take advantage of a larger number of cached sensors to reduce the number of probes it performs, with the heel of the curve at a freshness of approximately 4 minutes.

D. Caching and Sampling Parameters

In this section, we investigate the caching and sampling tradeoffs in more detail.

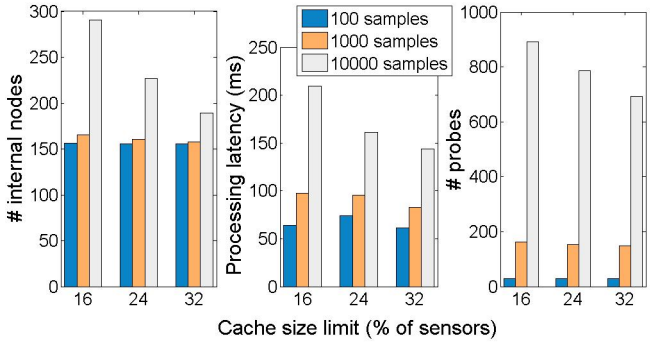

Fig. 5. Effects of varying cache size constraint, and sample size on: i) # sensor probes, ii) end-to-end processing latency, iii) # internal nodes traversed.

Cache limits and sample size targets. Figure 5 displays the effects of both a cache size constraint, and the sample size targets on the number of tree nodes traversed, the processing latency and the number of sensor probes. We vary the cache size limit from 16-32% of sensors (approx. 60k-120k sensors). This range was chosen based on preliminary experiments measuring the unconstrained cache size from the setup for Figure 3. We also consider three different sample sizes at each size constraint, ranging from 100-10000 sensor readings requested per query. For a large sample size, as the cache limit increases, we see the benefits of caching extra readings equally for all of our metrics. For smaller sample sizes, varying the cache limit has much less of an effect. The important trend to note is that as the cache limit increases, the sample size has a diminishing effect on all three aspects of performance. This can be seen by the smaller difference between all of our metrics across the different sample sizes at a cache limit of 32% than at a cache limit of 16%. This indicates sampling becomes a highly critical feature for systems desiring to support small caches (as may be the case for complex sensor readings such as images).

Sampling Accuracy. Figure 6 shows two metrics capturing the performance of our sampling algorithm for approximate query processing. The first metric is target accuracy which captures the percentage of the sensors requested in the `SAMPLESIZE`-clause that contribute to the result, defined as:

$$target\ acc = \frac{min(target\ size, sensors\ probed)}{min(target\ size, unsampled\ result\ size)}$$

The second metric is probe discretization error (pde) which we define as the relative error between the number of sensors probed and the target size at the terminal (i.e. probing) points of index access:

$$pde = \sum_{i \in terminals} \frac{target\ size(i) - \#results(i)}{target\ size(i)}$$

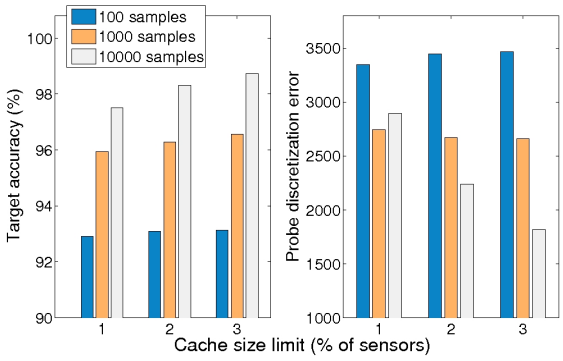

Fig. 6. Sampling accuracy and probe discretization error under varying cache and target sample size parameters.

where

$$\#results(i) = \begin{cases} cached\ result\ size(i) & \text{if } i \text{ is cached} \\ \#sensors\ probed(i) & \text{otherwise} \end{cases}$$

Note these terminal points are not always leaves due to the presence of cached aggregates at intermediate nodes. This metric reflects the spatial uniformity of the query result, under the assumption that the target size is itself distributed uniformly at the terminals.

Figure 6 shows that even at a small target of 100 sensors, COLR-Tree is able to accurately meet the desired sample size, achieving 93% accuracy with a small cache size. This validates that layered sampling can be effective despite the spatial "holes" and pruning that arises in the index during lookup, through its use of target weight redistribution. As both target sizes and cache limit increase, COLR-Tree improves on its target accuracy achieveing a maximum of 99% accuracy. In terms of probe discretization error, we see that with a sample size of 100 sensors the error increases with cache size, indicating the spatial bias induced by having aggregated cached results comprised from a greater number of sensors than necessary at the terminal. In contrast at the largest target, the probe error decreases significantly with larger cache size due to the generally higher targets at all query terminals, and hence lower bias incurred by using cached aggregates. In summary, the probe discretization error indicates the tension between caching aggregates and uniform sampling, and suggests further investigation for reversible aggregation materialization.

Result Accuracy. Note that since cached data is expired after expiry times defined by sensors, caching does not affect the accuracy of results. On the other hand, sampling provides only approximate answers. However, note that, since sensor data is often spatially correlated, sampling can provide a reasonably good approximation. We validate this hypothesis by using real-time water discharge data reported by 200 United States Geological Survey sensors in the Washington state (the data is available at `www.usgs.org`). We query for average water discharge reported by all these sensors, and COLR-Tree uses different sample sizes to answer the query. Figure 7 shows the relative error of the results as a function of sample size. As shown, an error within 10% can be achieved by sampling as few as 15 sensors.

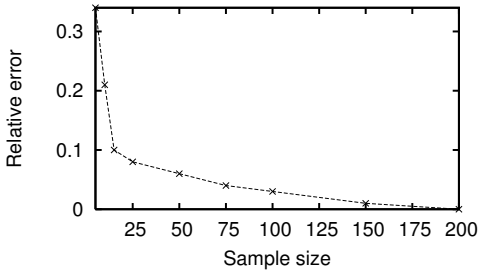

Fig. 7. Sensor reading approximation error due to sampling.

VIII. Conclusions

We have described COLR-Tree, an abstraction layer designed to support efficient spatio-temporal queries on live data gathered from a large collection of sensors. COLR-Tree promotes the efficient collection of data from live sensors during query processing. COLR-Tree attains collection-efficiency through two techniques. First, it augments caching with index structures and uses a novel mechanism that enables caching aggregates computed from data with different expiry times. Second, it allows computing approximate aggregate results by probing carefully chosen subset of sensors. Our evaluation with real workload shows significant benefits of these techniques.

References

[1] "Microsoft's plan to map the world in real time," MIT Technology Review, http://www.technologyreview.com/read_article.aspx?id=16781&ch=infotech, May 2006.

[2] A. Guttman, "R-trees: A dynamic index structure for spatial searching." in *SIGMOD*, 1984, pp. 47–57.

[3] A. Deshpande, S. Nath, P. Gibbons, and S. Seshan, "Cache-and-query for wide area sensor databases," in *ACM SIGMOD*, 2003.

[4] Y. Zhang, B. Hull, H. Balakrishnan, and S. Madden, "ICEDB: Intermittently-Connected Continuous Query Processing," in *ICDE*, 2007.

[5] A. Deshpande and S. Madden, "MauveDB: Supporting Model-based User Views in Database Systems," in *ACM SIGMOD*, 2006.

[6] M. F. Mokbel, T. M. Ghanem, and W. G. Aref, "Spatio-temporal access methods." *IEEE Data Engineering Bulletin.*, vol. 26, no. 2, pp. 40–49, 2003.

[7] I. Kamel and C. Faloutsos, "On packing r-trees." in *CIKM*, 1993.

[8] I. Lazaridis and S. Mehrotra, "Progressive approximate aggregate queries with a multi-resolution tree structure." in *SIGMOD*, 2001.

[9] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang, "Indexing spatio-temporal data warehouses," 2002.

[10] J. Yang and J. Widom, "Incremental computation and maintenance of temporal aggregates." *VLDB Journal.*, vol. 12, no. 3, 2003.

[11] D. Gross and M. de Rougemont, "Uniform generation in spatial constraint databases and applications," in *ACM PODS*, 2000.

[12] A. Nanopoulos, Y. Manolopoulos, and Y. Theodoridis, "An efficient and effective algorithm for density biased sampling," in *CIKM*, 2002.

[13] F. Olken and D. Rotem, "Sampling from spatial databases," in *ICDE*, 1993.

[14] "Google Maps, http://maps.google.com/," Google, Inc.

[15] "Windows Live Local, http://local.live.com/," Microsoft Corporation.

[16] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, 1999.

[17] J. Gray and G. Graefe, "The five-minute rule ten years later, and other computer storage rules of thumb," *SIGMOD Rec.*, vol. 26, no. 4, pp. 63–68, 1997.

[18] S. Nath and A. Kansal, "FlashDB: Dynamic Self-tuning Database for NAND Flash," in *IPSN*, 2007.

[19] C. Böhm, S. Berchtold, H.-P. Kriegel, and U. Michel, "Multidimensional index structures in relational databases." *Journal of Intelligent Information Systems.*, vol. 15, no. 1, pp. 51–70, 2000.