LIRA: Lightweight, Region-aware Load Shedding in Mobile CQ Systems

Buğra Gedik[◇♠] Ling Liu[◇] Kun-Lung Wu[♠] Philip S. Yu[♠]

[◇] CERCS, College of Computing, Georgia Tech
[♠] Thomas J. Watson Research Center, IBM Research
{bgedik,lingliu}@cc.gatech.edu, {klwu,psyu}@us.ibm.com

Abstract

Position updates and query re-evaluations are two predominant, costly components of processing location-based, continual queries (CQs) in mobile systems. To obtain high-quality query results, the query processor usually demands receiving frequent position updates from the mobile nodes. However, processing frequent updates oftentimes causes the query processor to become overloaded, under which updates must be dropped randomly, bringing down the quality of query results, negating the benefits of frequent position updates. In this paper, we develop LIRA *– a lightweight, region-aware load-shedding technique for preventively reducing the position-update load of a query processor, while maintaining high-quality query results. Instead of having to receive too many updates and then randomly drop some of them,* LIRA *uses a* region-aware *partitioning mechanism to identify the most beneficial* shedding regions *to cut down the position updates sent by the mobile nodes within those regions. Based on the number of mobile nodes and queries in a region,* LIRA *judiciously applies different amounts of update reduction for different regions, maintaining better overall accuracy of query results. Experimental results show that* LIRA *is vastly superior to random update dropping and clearly outperforms other alternatives that do not possess full-scale, region-aware load-shedding capabilities. Moreover, due to its lightweight nature,* LIRA *introduces very little overhead.*

1 Introduction

The proliferation of mobile devices and advances in wireless communications are creating an increasing interest in rich, value-added location-based services, which are expected to form an important part of the future computing environments that will seamlessly integrate into our lives [16]. A recent example from the industry is the Google Ride Finder [5] service, which provides mobile users with the capability to employ continual queries (CQs) to monitor nearby taxi services.

Mobile CQ systems serve as an enabling technology for location monitoring applications. Scalable CQ middleware for location monitoring has been an active area of research in the past, attested by several recent works, such as SINA [11], SRB [7], MAI [4], and others [3, 1, 12]. In almost all of these systems, there are two sources of bottleneck in providing high-quality query results. First, in order to provide fresh results the queries have to be re-evaluated frequently, consuming processing and disk IO resources. Second, in order to provide accurate results the position updates from mobile nodes have to be collected with high frequency, and be processed with high consumption of CPU, disk IO, and network resources.

The update problem associated with mobile CQ systems has received significant attention from the research community, resulting in several spatial index structures for efficiently integrating position updates into the system [17, 8, 10, 20]. Although indexing techniques can speed up the processing of position updates, they do not solve the fundamental problem of overload. When such overloads happen, the position updates will clog the system buffers and will cause the random dropping of the updates, which (as we show in this paper) is a very ineffective way to handle overload. Surprisingly, none of the previous works have addressed the problem of effective *update load shedding*. Hence, there is a cogent need for developing intelligent update load-shedding techniques for mobile CQ systems. By intelligent update load shedding we mean that the load shedding algorithms should prevent overloads by reducing the number of position updates received by the query processor in such a way that will minimally impact the accuracy of the query results.

In this paper, we develop a lightweight load-shedding technique for reducing the update load in mobile CQ systems, called LIRA. The main idea behind LIRA is that, given an update budget (which is either calculated automatically by LIRA or specified as a system-level *throttle fraction* parameter), LIRA creates a partitioning of the monitoring space into a set of *shedding regions* and associates an *update throttler* with each shedding region, where these update throttlers define the amount of load shedding to be performed for each region in accordance with the overall update budget. Both the partitioning and the settings of the update throttlers are performed with the objective of minimizing the negative impact of update load shedding on the accuracy of the query results.

The LIRA approach to update load shedding has four unique properties. First, the partitioning scheme employed by LIRA is *region-aware*, in the sense that contiguous geographical areas that have similar characteristics in terms of the density of mobile nodes and queries are grouped into the same load shedding regions. Second, the update throttlers are set according to the following principle: the regions in which applying update shedding may cut down a large number of updates while maintaining a minimal impact on the query-result accuracy are subjected to larger amounts of load shedding. Third, the LIRA approach provides an adjustable bound on the maximum difference between the update throttlers of different shedding regions, ensuring that all mobile nodes are tracked by the system, albeit with varying accuracies. This feature extends the applicability of LIRA to mobile CQ systems with snapshot and historical query support.

Last but not the least, LIRA introduces very little overhead and can be employed in conjunction with any CQ systems that employ update-efficient index structures, such as the TPR-tree [15].

We evaluate our load shedding approach using realistic location data synthetically generated using existing road maps and real-world traffic volume data. We devise a set of evaluation metrics to assess the effectiveness of LIRA and empirically show that LIRA is vastly superior to update dropping and clearly outperforms other alternatives that do not provide full-scale, region-aware load shedding capabilities.

2 Overview

In this section we describe the fundamental concepts underlying the LIRA load shedder, introduce some of the notations used in the paper, and present the system architecture.

2.1 Design Ideas

There are two primary types of load shedding techniques that can be used to reduce the number of position updates received from the mobile nodes: *server-actuated* and *source-actuated*. In server-actuated shedding, the position updates are dropped by the CQ server in order to match the update arrival rate with the service rate of the server. This has two major disadvantages. First, the dropped updates are unnecessarily transferred from the mobile nodes to the CQ server, wasting the network bandwidth of wireless medium. Second, these excessive updates still have to be received by the server (even though will be dropped later), and thus contribute to the processing load. On the other hand, the source-actuated approach requires some coordination between the server and the mobile nodes, since the load shedding decisions are made by the server.

A commonly used mechanism for actuating the position update reduction at the mobile node side is motion modeling, also known as *dead reckoning*. Motion modeling uses approximation for location update prediction. Concretely, instead of reporting their position updates each time they move, mobile nodes only report the parameters of a model approximating their motion when the model parameters change significantly. A significant change in the model parameters is decided based on an *inaccuracy threshold* Δ and the last reported model parameters. When the predicted position of the mobile node deviates from the actual position of the node by more than Δ, the new motion parameters are reported. A popular motion model is piece-wise linear approximation of the mobile node movement [19], whereas more advanced models also exist [2]. However, for the purpose of this paper the particular motion model used is not of importance. In the design of the LIRA load shedder, this inaccuracy threshold Δ is used as a control knob to adjust the number of position updates received. Without loss of generality, we adopt linear motion modeling in LIRA. Note that many of the existing mobile CQ systems have built-in support for linear motion modeling [15, 17, 4]. In this paper we take the source-actuated approach to develop LIRA, which provides a lightweight method to coordinate the server-initiated but source-actuated update load shedding.

A straightforward but naïve way of shedding update load is to have all mobile nodes to use a single system-controlled inaccuracy threshold. Let $\Delta_\vdash$ be the minimum value that the inaccuracy threshold can take, which defines the ideal resolution of

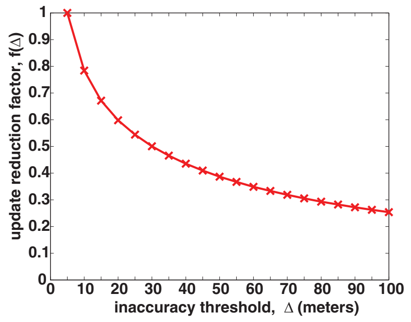

Figure 1: Reduction in the number of location updates received with different inaccuracy thresholds

position updates. Let $\Delta_\dashv$ be the maximum value that the inaccuracy threshold can take, which defines the lowest resolution of position updates required to achieve reasonable query result accuracy. The inaccuracy threshold Δ can be set to a value within $[\Delta_\vdash, \Delta_\dashv]$ in order to adjust the update expenditure of the system. By increasing Δ from $\Delta_\vdash$ to $\Delta_\dashv$, the number of updates will decrease even though this reduction is not linear as shown in Figure 1 [1]. The figure plots $f(\Delta)$, called the *update reduction factor*. For a given inaccuracy threshold $\Delta \in [\Delta_\vdash, \Delta_\dashv]$, $f(\Delta)$ gives the number of position updates received relative to the case of $\Delta = \Delta_\vdash$. As observed from Figure 1, the rate of reduction in the update expenditure is more pronounced while Δ is increased within the proximity of $\Delta_\vdash = 5$ meters, whereas it reduces to a fixed slope (linear decrease in the number of updates) as Δ gets closer to its maximum value of $\Delta_\dashv = 100$ meters.

A key observation we make in this paper is that different regions of the monitoring space exhibit different characteristics in terms of the densities of mobile nodes and queries and can benefit from differing amounts of load shedding. This observation suggests that a uniform Δ approach is significantly suboptimal. To understand this better, we plot the desirability of load shedding for regions with differing characteristics in Table 1.

n \ m	low	high
low	$<$	$\times$
high	✓	$>$

Table 1: Region characteristics and preference of load shedding

Let n be the number of mobile nodes and m be the number of queries within a region. When n is low and m is high for a region, load shedding should be avoided as much as possible (upper right quadrant in Table 1 is marked with $\times$ to show this). This is because a small number of nodes that generate a small number of updates are used for answering a large number of queries for this region, which implies that increasing Δ here will significantly impact the overall query accuracy, while bringing only a small reduction in the number of position updates received. In contrast, load shedding is very desirable when n is high and m is low for a region (lower left quadrant in Table 1 is marked with ✓ to show this). This is because a large number of nodes that generate a large number of updates are used for answering a small number of queries for this region. It implies that increasing Δ will minimally impact the overall query accuracy, while bringing a large reduction in the number of position updates received. Interestingly, the ratio m/n does not completely characterize the preference of one region over another for increasing the inaccuracy threshold Δ. This is because the over-

[1]The experimental setup and default parameters used to generate the graphs in Figure 1 are given in Section 4.2

all inaccuracy introduced in the mobile node positions increases linearly with increasing Δ, whereas the amount of update reductions increases non-linearly as Δ increases. This is why regions with small m and n are less attractive for load shedding compared to the regions with large m and n, but comparing to the scenario of high m and low n both being better choices for load shedding (the symbols $<$ and $>$ indicate this in Table 1).

This insight leads us to a region-aware approach to update load shedding. Concretely, in LIRA we partition the geographical area of interest into l shedding regions, denoted by $A_i, i \in [1..l]$. Furthermore, we associate an inaccuracy threshold with each shedding region A_i, denoted by Δ_i. We call Δ_i the update throttler of the region A_i. A simple way of determining the shedding regions is to partition the entire geographical space of interest into l regions evenly. However, such even partitioning of the space is unlikely to produce an effective solution, since the level of heterogeneity (in terms of the number of mobile nodes and queries) inside two given equally-sized regions may differ significantly. Intuitively, a region where further partitioning generates sub-regions of similar characteristics in terms of densities of mobile nodes and queries does not provide any gain with regard to reducing the number of position updates while minimizing the query result inaccuracy. Thus the design of the LIRA load shedder should address the following two challenges: (1) How to partition the geographical space of interest into a set of shedding regions effectively, and (2) How to set the update throttler for each region to minimize the inaccuracy introduced in query results while meeting our update budget constraint.

In LIRA we introduce the concept of throttle fraction to define the position update budget of the system, denoted by $z \in [0, 1]$. For instance, $z = 0.75$ means that the number of updates should be reduced by a quarter, compared to the case of using a common $\Delta = \Delta_{\vdash}$. The throttle fraction can be calculated automatically by the server in reaction to overload situations by observing the size of the system message queue (see Section 3.4). Alternatively, when the server is not overloaded but the wireless communication load of receiving updates are putting a heavy burden on the network, the throttle fraction can be manually set as a system-level parameter. In the next section we discuss the system architecture and describe how we compute the l shedding regions and how to set the update throttlers for the l regions within a given system-controlled position update budget, represented by the throttle fraction z.

2.2 System Architecture

Figure 2 illustrates the system architecture of LIRA, which consists of three layers. The first layer is formed by the mobile CQ server. The server has three main responsibilities. First, it sets the throttle fraction $z \in [0, 1]$ to define the position update budget of the system. Second, it is responsible for calculating the shedding regions and the associated update throttlers for a given update budget, that is for a given value of the throttle fraction z. Third, it is responsible for reporting to each base station in the second layer, the subset of shedding regions and update throttlers corresponding to the base station's coverage area.

The set of base stations that cover the space of interest form the second layer. The base stations are assumed to be connected to the mobile CQ server via the wired network. They provide wireless networking services to the mobile nodes. The base sta-

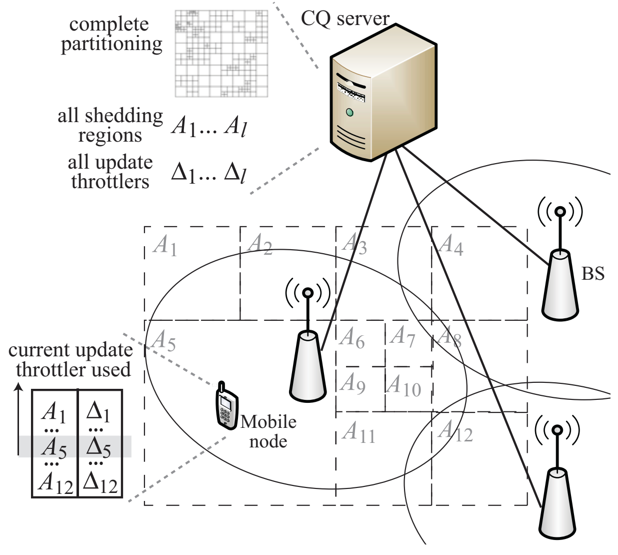

Figure 2: System Architecture

tions are responsible for broadcasting the subset of load shedding regions and update throttlers corresponding to their coverage area to mobile nodes, when the server reports a change in the partitioning or the update throttler values. The base stations are also responsible for sending the shedding regions and update throttlers to a mobile node entering into a new base station's coverage area during a hand-off.

The set of mobile nodes form the third layer of our system. The mobile nodes are responsible for reporting their positions to the mobile CQ server using dead reckoning. However, the inaccuracy threshold used by a mobile node is dependent on the region in which it resides. As a result, the mobile nodes store a subset of shedding regions and update throttlers corresponding to the coverage area of their current base station. As the mobile nodes move from one shedding region to another within their base station's coverage area, they use the update throttler corresponding to their current shedding region as the inaccuracy threshold. The update throttler to use is determined locally. When the mobile nodes switch base stations, they change the subset of shedding regions and update throttlers they store based on the information they receive from the new base station.

Factors Affecting the Number of Shedding Regions
From the description of the system architecture, one may observe an interesting trade-off in setting the number of shedding regions l. On the one hand, the larger the number of shedding regions is, the more fine grained is the partitioning, leading to more fully exploiting the potential heterogeneity existent in the geographical space of interest in terms of different characteristics of regions with respect to the number of mobile nodes and queries. On the other hand, as the number of shedding regions increase, the average number of update throttlers and shedding regions per base station coverage area grows. This implies that the mobile nodes should know about a larger number of shedding regions and update throttlers. This increases the cost of finding the correct update throttler to use at the mobile node side, as well as the communication cost of disseminating the new set of throttle fractions and shedding regions to the mobile nodes once they are updated by the server. Thus a careful setting of l is critical for the overall system scalability in terms of both

service quality and wireless communication bandwidth.

We below describe in detail our design of Lira load shedder in two steps. In Section 3, we discuss how the Lira load shedder works to shed loads effectively, including how to set the throttle fraction z, how to partition the geographical space of interest into l shedding regions, and how to determine the update throttle for each of the shedding regions. In Section 4, we discuss in detail how the LIRA system sets l to achieve a sufficiently fine granularity in partitioning while minimizing the inaccuracy in query results, and yet putting very little load on the mobile nodes and the wireless network.

3 The LIRA Load Shedder

In this section we describe the main technical components of the LIRA load shedder, encompassing the three major server-side functionalities: (1) partitioning the geographical space of interest into l shedding regions for a given l, performed by the GRIDREDUCE algorithm, (2) determining the update throttler for each of the l shedding regions, performed by GREEDYINCREMENT algorithm, and (3) setting the throttle fraction z to adjust the system-wide position update budget, performed by THROTLOOP algorithm. These three algorithms work in cooperation to perform the load shedding. In particular, the THROTLOOP algorithm monitors the performance of the system under the current workload and resource availability to decide the throttle fraction z. Given z computed by THROTLOOP and the number l of shedding regions specified as a system-supplied parameter, the GRIDREDUCE algorithm creates a partitioning of the entire geographical space of interest and computes the set of l shedding regions, i.e., A_i ($i \in [1..l]$). Finally, given z, l, and A_i's, the GREEDYINCREMENT algorithm determines the update throttlers for the l shedding regions, i.e., Δ_i ($i \in [1..l]$).

A common thread that is shared by all three algorithms is the optimization problem of finding a partitioning and the associated set of update throttlers that defines the near-optimal load shedding strategy, which sheds the load within the given update budget z and yet minimizes the inaccuracy of the results of queries. Thus, in this section we first formally define our update load shedding problem, and then we describe the three key algorithms of the Lira load shedder.

3.1 Problem Formulation

The problem is to find a partitioning of $A_i, i \in [1..l]$, and an associated set of update throttlers $\Delta_i, i \in [1..l]$, such that certain constraints are met (e.g., the update budget is respected) and an objective function is optimized (i.e., inaccuracy in query results is minimized). We start with formulating the two constraints. Let n_i denote the number of mobile nodes within shedding region A_i. The following two constraints should hold:

$$\sum_{i \in [1..l]} n_i \cdot f(\Delta_i) \leq z \cdot n \cdot f(\Delta_{\vdash})$$

$$\forall_{i \in [1..l]} \; \Delta_{\vdash} \leq \Delta_i \leq \Delta_{\dashv}$$

The first constraint, which we call the *update budget constraint*, states that the number of updates received under the region-aware load shedding approach should not exceed z (throttle fraction) times the number of updates that would have been received if there were no load shedding applied, i.e., we were to use a uniform inaccuracy threshold of $\Delta_{\vdash}$ for all nodes. Note that we have $f(\Delta_{\vdash}) = 1$. The second constraint defines the domain of update throttlers (Δ_i's).

We now formulate the objective function of the problem we want to minimize, that is the inaccuracy in query results. For the purpose of this problem formalization we define the inaccuracy introduced by using an update throttler value of Δ_i for a given region A_i as the number of queries in the region A_i, denoted by m_i, times the inaccuracy threshold Δ_i, that is $m_i \cdot \Delta_i$. When computing m_i, queries partially intersecting the shedding region A_i are fractionally counted. The objective function that we want to minimize can be formulated as follows:

$$InAcc(\{A_i\}, \{\Delta_i\}) = \sum_{i \in [1..l]} m_i \cdot \Delta_i$$

Note that m_i and n_i are functions of the partitioning $\{A_i\}$. We now discuss a number of extensions to this basic problem.

3.1.1 The Fairness Threshold

The first extension to the basic problem formulation is to provide a system-level control over the difference in the inaccuracy thresholds used in different regions. We introduce a parameter called the fairness threshold, denoted by $\Delta_{\Leftrightarrow}$. In the original problem formulation, the shedding regions that do not contain any queries (i.e., $\{A_i : m_i = 0\}$) may be overly penalized by setting their update throttlers to maximum inaccuracy value of $\Delta_{\dashv}$, since the update reduction for those regions does not impact the query results. However, for mobile CQ systems supporting historic and ad-hoc queries this may be undesirable, thus $\Delta_{\Leftrightarrow}$ can be used to reduce this effect. Formally, we replace the second constraint in the basic problem formulation with the following constraint on the domain of update throttles:

$$\forall_{i,j \in [1..l]} \; |\Delta_i - \Delta_j| \leq \Delta_{\Leftrightarrow}$$

One extreme case of $\Delta_{\Leftrightarrow} = \Delta_{\dashv} - \Delta_{\vdash}$ represents the original formulation, whereas the other extreme case of $\Delta_{\Leftrightarrow} = 0$ represents the uniform Δ scenario.

3.1.2 The Speed Factor

Different regions may have different average speeds for the mobile nodes within, and thus may exhibit different behaviors with respect to the impact of the inaccuracy threshold on the number of updates received from the mobile nodes. As a result, the update budget constraint should be adjusted using speeds to achieve a more accurate modeling of the update cost. Let us denote the average speed of mobile nodes within the shedding region A_i by s_i and the overall average speed by $\hat{s}$ (i.e., $\hat{s} = \sum_{i=1}^{l} s_i \cdot (n_i/n)$). Assuming that the number of updates is linearly proportional to the average speed of mobile nodes, we modify our update budget constraint as follows:

$$\sum_{i=1}^{l} n_i \cdot s_i \cdot f(\Delta_i) \leq z \cdot n \cdot \hat{s} \cdot f(\Delta_{\vdash})$$

3.1.3 Final Problem Formulation

The final formulation with the two extensions is as follows:

$$\underset{\{A_i\},\{\Delta_i\}}{argmin} \quad InAcc(\{A_i\},\{\Delta_i\}) = \sum_{i\in[1..l]} m_i \cdot \Delta_i$$

$$\text{s.t.} \quad (i) \quad \sum_{i\in[1..l]} n_i \cdot \frac{s_i}{\hat{s}} \cdot f(\Delta_i) \le z \cdot n \cdot f(\Delta_\vdash)$$

$$(ii) \quad \forall_{i,j} \ |\Delta_i - \Delta_j| \le \Delta_\Leftrightarrow$$

$$(iii) \quad \forall_{i\in[1..l]} \ \Delta_\vdash \le \Delta_i \le \Delta_\dashv$$

We consider the partitioning and the settings of update throttlers as separate problems. In what follows, we first provide a heuristic-based partitioning algorithm for constructing the shedding regions and then give an optimal (under certain conditions) algorithm for setting the update throttlers for a given partitioning of the space. It is worth mentioning that the problem of setting update throttlers is not a linear program, since the update reduction function f is not linear and as a result the update budget constraints are not linear.

3.2 GRIDREDUCE: Partitioning the Space

The goal of the GRIDREDUCE algorithm is to partition the geographical space of interest into l shedding regions, such that this partitioning produces lower query result inaccuracy. For each shedding region A_i generated, the algorithm also determines the number of nodes n_i, the number of queries m_i, and the average speed s_i for that region. This information is later used by the GREEDYINCREMENT algorithm to set the update throttlers.

The GRIDREDUCE algorithm works in two stages and uses a *statistics grid* as the base data structure to guide its decisions. The statistics grid serves as a uniform, maximum fine-grained partitioning of the space of interest. In the first stage of the algorithm, which follows a bottom-up process, we create a region hierarchy over the statistics grid and aggregate the query and mobile node statistics for the higher-level regions in this hierarchy. This region hierarchy serves as a template from which a non-uniform partitioning of the space can be constructed. The second stage follows a top-down process and creates the final set of l shedding regions, starting from the highest region in the hierarchy (the whole space). The main idea is to selectively pick and drill down on a region using the hierarchy constructed in the first stage. The region to drill down is determined based on the expected amount of gain in the query result accuracy, called the *accuracy gain* (see Section 3.2.3), which is computed using the aggregated region statistics.

We now describe the details of the GRIDREDUCE algorithm. Its pseudo code is given in Algorithm 1.

3.2.1 The Statistics Grid

The statistics grid is an $\alpha \times \alpha$ evenly spaced grid over the geographical space, where α is the number of grid cells on each side of the grid. We describe the relationship between α and l later in this section. For each grid cell $c_{i,j}$ the statistics grid stores the average number of mobile nodes $n_{i,j}$, queries $m_{i,j}$, and average speed $s_{i,j}$ for that grid cell. The only data structure maintained by the LIRA load shedder is this grid.

The maintenance of the grid can be performed in a number of ways. For instance, if the mobile CQ server uses a grid-based index on mobile node positions [9, 11] the statistics grid can be trivially supported as a part of the grid index. Alternatively, the grid can be explicitly maintained by processing position updates. Note that it takes constant time to process an update for maintaining the grid. Moreover, all of the updates need not be processed, since the statistics can easily be approximated using sampling. In an off-line alternative, the average number of mobile nodes and average node speeds can be pre-computed for different times of the day based on historic data, in which case the maintenance cost is close to zero. In all three alternatives, maintenance of the statistics grid is a lightweight operation. The partitioning generated by the GRIDREDUCE algorithm using an $\alpha \times \alpha$ grid is called the (α, l)-*partitioning* of the space.

3.2.2 Stage I: Building the Region Hierarchy

In the first stage (see lines 1- 9 in Algorithm 1), we build a complete quad-tree over the grid. Each tree node corresponds to a different region in the space, where regions get larger as we move closer to the root node which represents the whole space. Each level of the quad-tree is a uniform, non-overlapping partitioning of the entire space. Through a post-order traversal of the tree, we aggregate the statistics associated with the grid cells for each node of the tree, i.e., we compute the number of mobile nodes, number of queries, and average speed for each tree node's region. The first stage of the algorithm takes $\mathcal{O}(\alpha^2)$ time and consumes $\mathcal{O}(\alpha^2)$ space, since the number of tree nodes is $\alpha^2 + (\alpha^2 - 1)/3$, assuming α is a power of 2.

3.2.3 Stage II: Drilling Down in the Hierarchy

In the second stage of the algorithm (see lines 10- 22 in Algorithm 1) we start with the root node of the tree, i.e., the entire space. At each step, we pick an explored tree node (initially only the root) and replace it with 4 tree nodes by partitioning the node's region into 4 sub-regions corresponding to its child nodes in the tree. This process continues until we reach l tree nodes (thus l shedding regions), assuming $l \mod 3 = 1$. The crux of this stage lies in how we choose the region to partition during each step. For this purpose we maintain a max-heap of all explored tree nodes based on the accuracy gain, a metric we introduce below, and at each step we pick the node with the highest accuracy gain.

Given a tree node, the accuracy gain is a measure of the expected reduction in the query result inaccuracy, achieved by partitioning the node's region into 4 sub-regions corresponding to its child nodes. For a tree node t, the accuracy gain $V[t]$ is calculated as follows. Let $E[t]$ be the average result inaccuracy if we only had one shedding region that is t's region. Formally, we have $E[t] \leftarrow min_\Delta\,(m[t]\cdot\Delta),$ s.t. $f(\Delta) \le z \cdot f(\Delta_\vdash)$. Now let $E_p[t]$ be the average result inaccuracy if we had 4 shedding regions that correspond to the regions of t's child nodes $t_i, i \in [1..4]$. Formally, we have $E_p[t] \leftarrow min_{\{\Delta_i\}} \sum_{i=1}^{4} \Delta_i \cdot m[t_i]$ subject to the constraint $\sum_{i=1}^{4} n[t_i] \cdot f(\Delta_i) \le z \cdot n[t] \cdot f(\Delta_\vdash)$. Then the difference $E[t] - E_p[t]$ gives us the accuracy gain $V[t]$.

The computation of $E[t]$ and $E_p[t]$, and thus the accuracy gain $V[t]$, requires solving the problem of update throttler setting for a fixed l of shedding regions. Concretely, computation of $E[t]$ requires to solve for node t with $l = 1$ and computation of $E_p[t]$ requires to solve for the four child nodes of t with

Algorithm 1: (l, α)-partitioning of the space

```
Input: $\alpha$ is the number of cells on each side of the grid. It is a power of
2. $l$ is the number of grid areas desired, where we have $l \mod 3 = 1$.
$z$ is the throttle fraction.
Output: $A_i$, $i \in [1..l]$ is the $i$th grid area. $n_i$ is the number of mobile
nodes under $A_i$ and $m_i$ is the number of queries under $A_i$.
GRIDREDUCE($\alpha$, $l$, $z$)
  {the following steps take $\mathcal{O}(\alpha^2)$ time and $\mathcal{O}(\alpha^2)$ space}
1) Construct $\log_2 \alpha + 1$-level quadrant tree over the $\alpha \times \alpha$ grid
2) foreach tree node $t$ in post-order
3)     if $t$ is a leaf node, {initialize # objs., # qrys., and speeds}
4)         $c_{i,j}$: corresponding grid cell of $t$
5)         $n[t] \leftarrow n_{i,j},\ m[t] \leftarrow m_{i,j},\ s[t] \leftarrow s_{i,j}$
6)     else {$t$ is not a leaf, aggregate # objs., # qrys., and speeds}
7)         $t_i$: $i^{\text{th}}$ childeren of $t$, $i \in [1..4]$
8)         $n[t] \leftarrow \sum_{i=1}^{4} n[t_i],\ m[t] \leftarrow \sum_{i=1}^{4} m[t_i]$
9)         $s[t] \leftarrow \sum_{i=1}^{4} (n[t_i]/n[t]) \cdot s[t_i]$
  {the following steps take $\mathcal{O}(l \cdot \log l)$ time and $\mathcal{O}(l)$ space}
10) $H$: empty max. heap of nodes, based on $V$ (accuracy gain) values
11) $L$: empty list of tree nodes, $i \leftarrow 0$
12) $t \leftarrow$ root of the tree, $H$.INSERT($t$)
13) while $L$.SIZE()+$H$.SIZE() $< l$ {$l$ regions not reached}
14)     $t \leftarrow H$.POPMAX() {region to partition}
15)     if $t$ is not a leaf {further partitioning possible}
16)         for $i = 1$ to 4 {partition the region}
17)             $g \leftarrow$CALCERRGAIN($t_i$), $H$.INSERT($\langle t_i, g \rangle$)
18)     else {$t$ is a leaf node} {no further partitioning}
19)         $L$.INSERT($t$) {store the region in $L$}
20) foreach $t \in L \cup H$ {process the regions}
21)     $n_i \leftarrow n[t],\ m_i \leftarrow m[t],\ s_i \leftarrow m[t]$ {set the region stats.}
22)     $A_i \leftarrow$ Area of $t$'s quadrant, $i \leftarrow i + 1$ {set the area}
CALCERRGAIN($t$)
1) $E \leftarrow min_{\Delta}\ (m[t] \cdot \Delta)$, s.t. $f(\Delta) \leq z \cdot f(\Delta_{\vdash})$
2) $E_p \leftarrow min_{\{\Delta_i\}} \sum_{i=1}^{4} \Delta_i \cdot m[t_i]$,
                         s.t. $\sum_{i=1}^{4} n[t_i] \cdot f(\Delta_i) \leq z \cdot n[t] \cdot f(\Delta_{\vdash})$
3) $V[t] \leftarrow E - E_p$ {accuracy gain is the difference in error}
```

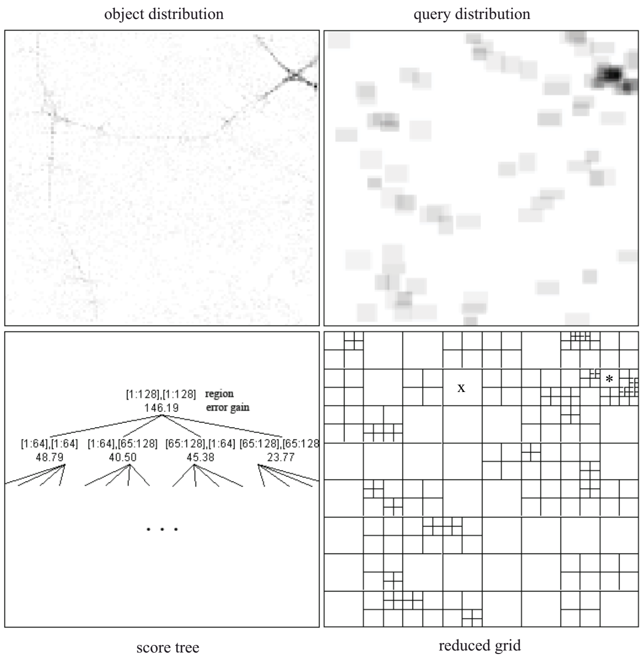

Figure 3: Illustration of (α, l)-partitioning

$l = 4$. As we will show in Section 3.2.4, this general problem can be solved in loglinear time on l. As a result, the accuracy gain is computed in constant time for a tree node t. The second stage of the GRIDREDUCE algorithm takes $\mathcal{O}(l \cdot \log l)$ time and consumes $\mathcal{O}(l)$ space, bringing the combined time complexity to $\mathcal{O}(l \cdot \log l + \alpha^2)$ and space compexity to $\mathcal{O}(\alpha^2)$.

3.2.4 Illustration of the Partitioning

Figure 3 depicts an example (α, l)-partitioning. The mobile node distribution (generated from a road map) is shown on the top left corner, whereas the query distribution is shown on the top right corner. The top three layers of the quad-tree built over the statistics grid is shown on the bottom left corner and the final (α, l)-partitioning is shown on the bottom right corner. It is important to note that the regions are not being further partitioned when the further partitioning will not benefit the query result accuracy. Here are the two interesting examples: the shedding regions marked with $\times$ and $*$ in Figure 3, which we denote by $A_\times$ and A_*. We see that $A_\times$ is larger than some of the nearby regions. This is because the number of queries is zero for $A_\times$ and as a result further partitioning is not needed. A_* is also larger than some of the nearby regions, but in contrast to $A_\times$ the number of queries is large for A_*. However, what matters is the heterogeneity of the region in terms of the number of mobile nodes and queries within. In the case of A_* further partitioning of the region results in sub-regions of similar characteristics, implying that partitioning is unnecessary due to low heterogeneity.

3.2.5 The Relationship Between l and α

To find a pragmatic way of configuring the statistics grid parameter α, we first observe the relationship between l and α. Assume that the partitioning is performed such that all the shedding regions are evenly sized. This will yield a grid partitioning with $\sqrt{l}$ number of cells on each side, which we refer to as the *l-partitioning*. Our aim is to have a statistics grid that is fine grained enough to provide us with an (α, l)-partitioning whose non-uniformly sized shedding regions are sufficiently flexible in terms of the size of their area compared to the case of l-partitioning in which all regions are equal-sized. The side length of the minimum possible shedding region in (α, l)-partitioning is proportional to $1/\alpha$ (the shedding region is equal to a cell of the statistics grid), whereas the side length of a region in l-partitioning is proportional to $1/\sqrt{l}$. To achieve around x^2 times difference in the areas of minimum possible shedding regions of l-partitioning and (α, l)-partitioning, we should determine α using the formula $\alpha = 2^{\lfloor \log_2(x \cdot \sqrt{l}) \rceil}$. Having $x = 10$ provides around 100 times difference in size. In our experimental studies we have found that this setting gives effective results.

3.3 GREEDYINCREMENT: Setting the Δ_i's

The goal of the GREEDYINCREMENT algorithm is to find the optimal setting of the update throttlers associated with the l shedding regions produced by the GRIDREDUCE algorithm, so that the inaccuracy in query results is minimized (while respecting the fairness thresholds). We first consider this problem without the fairness threshold constraints. The main idea is to increase the update throttlers in order to match the update budget. The update throttlers that bring a larger reduction in the update expenditure of the system in return for a smaller reduction in the result accuracy are preferred for increment.

3.3.1 The Greedy Steps

As the name suggests, the algorithm is a greedy one. It starts by setting all Δ_i's to $\Delta_{\vdash}$, the current update expenditure U to $n \cdot \hat{s} \cdot f(\Delta_{\vdash})$ and the update budget $U_{\dashv}$ to $z \cdot U$. Note that the initial setting is an infeasible solution since the update expenditure is higher than the update budget, that is $U > U_{\dashv}$. At each greedy step one of the update throttlers is selected based on the *update gain*, a criterion to be defined in the next subsection, and is increased by c_Δ, called the *increment* (or by a smaller value in the case that we undershoot the update budget). When Δ_i is incremented by c_Δ, the current update expenditure is decreased by $n_i \cdot s_i \cdot (f(\Delta_i) - f(\Delta_i + c_\Delta))$. This process continues until the current update expenditure decreases to match the update budget (i.e., $U = U_{\dashv}$), or all the update throttlers reach their maximum bound (i.e., $\Delta = \Delta_{\dashv}$). The former condition implies that the update expenditure is reduced to a value equal to throttle fraction times the maximum update expenditure associated with the case of $\forall_{i\in[1..l]}\, \Delta_i = \Delta_{\vdash}$. This means that the update constraint is satisfied. On the other hand, the latter condition implies that the update budget can not be met for the given throttle fraction z and the update throttler range $[\Delta_{\vdash}, \Delta_{\dashv}]$, leading to the solution $\forall_{l\in[1..l]}\, \Delta_i = \Delta_{\dashv}$.

3.3.2 Update Gain Calculation

The key point of GREEDYINCREMENT is the selection of the update throttler to use at each greedy step. We pick the update throttler that has the highest update gain. The update gain is defined as the ratio of the decrease in update expenditure to the additional inaccuracy introduced in the query results. We denote the rate of decrease in the update expenditure at a point Δ by $r(\Delta)$, and define it as the negative of the update reduction function f's derivative at point Δ. Formally, we have:

$$r(\Delta) = -\frac{d(f(x))}{dx}\bigg|_{x=\Delta}$$

Based on this definition, making a dx increase in update throttler Δ_i will reduce the update expenditure by $n_i \cdot s_i \cdot r(\Delta_i) \cdot dx$, and will decrease the query result inaccuracy by $m_i \cdot dx$. As a result, the update gain for the i^{th} update throttler Δ_i, denoted by S_i, is:

$$S_i(\Delta) = (n_i/m_i) \cdot s_i \cdot r(\Delta)$$

In each step of the GREEDYINCREMENT algorithm, an update throttler Δ_j is selected such that we have $j = argmax_{i\in[1..l]}\, S_i(\Delta_i)$. If the update gain for Δ_j is larger than the update gain for Δ_k, then increasing Δ_j provides better update reduction compared to Δ_k for the same amount of increase in query result inaccuracy.

3.3.3 Optimality and Setting of the Increment c_Δ

To provide an optimality guarantee and to guide the setting of c_Δ, we approximate the update reduction function f by a non-increasing, piece-wise linear function of κ segments, each of size $(\Delta_{\dashv} - \Delta_{\vdash})/\kappa$. This enables us to prove the following result:

Theorem 3.1. *For $c_\Delta = (\Delta_{\dashv} - \Delta_{\vdash})/\kappa$, the* GREEDYINCREMENT *algorithm is optimal for the non-increasing piece-wise linear approximation of the update reduction function f with κ segments of size c_Δ each.*

Proof. See Appendix A □

The time complexity of the GREEDYINCREMENT algorithm is given by $\mathcal{O}(\kappa \cdot l \cdot \log l)$ or by $\mathcal{O}(l \cdot \log l)$ if κ is constant. The derivation follows from the simple fact that in the worst case it will take κ steps to increase Δ_i from $\Delta_{\vdash}$ to $\Delta_{\dashv}$. There are l number of update throttlers and each greedy step takes $\mathcal{O}(\log l)$ time due to heap operations. As a result, the overall time complexity stated above is achieved. The space complexity is $\mathcal{O}(l)$.

3.3.4 Supporting the Fairness Threshold

In order to support the fairness constraints dictated by the fairness threshold $\Delta_{\Leftrightarrow}$, we make the following changes to the base algorithm. At each greedy step, the update throttler with the highest update gain, say Δ_i, is incremented by *at most* c_Δ, making sure that it does not go beyond a value that will violate the fairness constraint. Concretely, if the minimum update throttler we have is $\Delta_{\trianglerighteq} = min_{j\in[1..l]}\, \Delta_j$, then Δ_i is not increased beyond $\Delta_{\trianglerighteq} + \Delta_{\Leftrightarrow}$. When an update throttler Δ_i reaches the limit, that is we have $\Delta_i = \Delta_{\trianglerighteq} + \Delta_{\Leftrightarrow}$, then it is moved to a *blocked list* and is not considered for the following steps of the algorithm until it is removed from the blocked list. Whenever the minimum update throttler $\Delta_{\trianglerighteq}$ is changed, the set of update throttlers in the blocked list that are no more on the limit are removed and are included in the following steps of the algorithm. The pseudo code of the GREEDYINCREMENT algorithm with the extensions is given in Algorithm 2. [2]

3.4 THROTLOOP: Setting the Throttle Fraction

The throttle fraction z can be adaptively adjusted by the LIRA load shedder, when it is not set as a fixed system-level parameter to retain only a pre-defined fraction of position updates. The adjustment of the throttle fraction is performed by the THROTLOOP, which observes the position update queue and periodically decides the fraction of position updates that should be retained (throttle fraction z). The aim is to reduce the system load so that the rate at which the position updates are received (λ) and the rate at which these updates are processed (μ) are balanced to prevent dropping updates from the input queue. The *utilization* of the system, denoted by ρ, is given by λ/μ. Let us denote the maximum size of the input queue by B. Assuming an $M/M/1$ queuing model, we should have the following relationship between ρ and B to make sure that the average queue length is no more than the maximum queue size [14]: $\rho = 1 - 1/B$. If the utilization is larger than $1 - B^{-1}$, it represents an overload situation and thus the throttle fraction z should be decreased. On the other hand, if the utilization is smaller than $1 - B^{-1}$, it implies that the system is not fully utilized and the throttle fraction z should be increased. This understanding leads to the following procedure that describes the operation of THROTLOOP:

$$\begin{aligned} \text{Initially: }\ & i \leftarrow 0,\ z^{(i)} \leftarrow 1 \\ \text{Periodically: }\ & u \leftarrow \rho/(1 - B^{-1}),\ i \leftarrow i + 1 \\ & z^{(i)} \leftarrow min(1,\ z^{(i-1)}/u) \end{aligned}$$

[2] Note that in order to maintain $\Delta_{\trianglerighteq}$ for each step of the algorithm without changing the time complexity, a sorted tree of update throttlers is maintained and provides $\mathcal{O}(\log l)$ time insertion and removal operations.

Algorithm 2: Setting the update throttlers

```
Input: $z$: throttle fraction, $c_\triangle$: increment, $\Delta_\Leftrightarrow$: fairness threshold
Output: $\Delta_i, i \in [1..l]$: shedding thresholds
GreedyIncrement($z$, $c_\triangle$, $\Delta_\Leftrightarrow$)
1)  $H$: empty, max heap of $S_i$'s (update gains)
2)  $D$: empty, sorted tree of $\Delta_i$'s (update throttlers)
3)  $L$: empty, list of blocked $\Delta_i$ (throttlers at fairness limit)
4)  $U \leftarrow n \cdot \hat{s} \cdot f(\Delta_\vdash)$, $U_\dashv \leftarrow z \cdot U$ {set update budget, expend.}
5)  for $i = 1$ to $l$ {initialize $H$ and $D$}
6)      $\Delta_i \leftarrow \Delta_\vdash$, $D$.Insert($\Delta_i$) {update throttler}
7)      $S_i \leftarrow (n_i/m_i) \cdot s_i \cdot r(\Delta_i)$, $H$.Insert($S_i$) {update gain}
8)  repeat {start increment loop}
9)      $S_i \leftarrow H$.PopMax() {next $\Delta_i$ to increment}
10)     $\Delta'_i \leftarrow \Delta_i$, $\Delta'_\trianglerighteq \leftarrow D$.Min() {backup $\Delta_i$,$\Delta_\trianglerighteq = min_j\, \Delta_j$}
11)     $c'_\triangle \leftarrow min(c_\triangle \cdot \lfloor \Delta_i / c_\triangle + 1 \rfloor, \Delta'_\trianglerighteq + \Delta_\Leftrightarrow) - \Delta_i$
12)     $c'_\triangle \leftarrow min(c'_\triangle, (U - U_\dashv)/(S_i \cdot m_i))$ {set step size}
13)     $\Delta_i \leftarrow min(\Delta_i + c'_\triangle, \Delta_\dashv)$ {increment $\Delta_i$}
14)     $U \leftarrow U - (\Delta_i - \Delta'_i) \cdot (S_i \cdot m_i)$ {adjust the update budget}
15)     $D$.Update($\Delta'_i, \Delta_i$), $\Delta_\trianglerighteq \leftarrow D$.Min() {update min $\Delta$}
16)     if $\Delta_i - \Delta_\trianglerighteq = \Delta_\Leftrightarrow$ {fairness limit reached}
17)         $L$.Insert($\Delta_i$) {store $\Delta_i$ in the blocked list}
18)     else if $\Delta_i \neq \Delta_\dashv$ {upper bound not reached}
19)         $S_i \leftarrow (n_i/m_i) \cdot s_i \cdot r(\Delta_i)$, $H$.Insert($S_i$){update gain}
20)     if $\Delta'_\trianglerighteq \neq \Delta_\trianglerighteq$ {min $\Delta$ changed}
21)         foreach $\Delta_j \in L$ {iterate the blocked $\Delta$ list}
22)             if $\Delta'_j - \Delta_\trianglerighteq < \Delta_\Leftrightarrow$ {no more on the limit}
23)                 $S_j \leftarrow (n_j/m_j) \cdot s_j \cdot r(\Delta_j)$ {update gain}
24)                 $L$.Remove($\Delta_j$), $H$.Insert($S_j$) {move to $H$}
25) until $U \leq U_\dashv$ or $H$.Size() $= 0$ {budget reached or all maxed}
```

This completes our discussion of the LIRA load shedder. We have described how LIRA sheds load effectively using an (α, l)-partitioning strategy and the three key algorithms for partitioning the geographical space of interest into l shedding regions for a given l (GRIDREDUCE), for determining the update throttle for each of the l shedding regions (GREEDYINCREMENT), and for setting the throttle fraction z (THROTLOOP). In the next section, we present the experimental evaluation of the Lira load shedder, including a discussion through experimental results on how to determine l in order to achieve a sufficiently fine granularity in partitioning and at the same time minimizing the inaccuracy in query results, while putting very little load on the mobile nodes and the wireless network.

4 Experimental Evaluation

In this section we present experimental results on the effectiveness of the LIRA load shedder in cutting the cost of receiving and processing position updates in mobile CQ systems, while minimally affecting the accuracy of the query results. Before describing the experimental setup, we first discuss the set of evaluation metrics we define to assess the effectiveness of LIRA.

4.1 Evaluation Metrics

We define two sets of evaluation metrics. The first set of evaluation metrics are used to measure the accuracy of the query results under load shedding and the second set of metrics deal with the cost of performing load shedding.

4.1.1 Query Result Accuracy

Mean Containment Error, denoted by E^C_{rr}, defines the average containment error in query results. Containment error for a query result is defined as the ratio of the number of missing and extra items in the result to the correct result set size. Let Q denote the set of queries, $R(q)$ denote the result set for a query $q \in Q$ under load shedding, and $R^*(q)$ denote the correct result set under $\forall_{i \in [1..l]}\ \Delta_i = \Delta_\vdash$. Then:

$$E^C_{rr} = \sum_{q \in Q} \frac{|R^*(q) \setminus R(q)| + |R(q) \setminus R^*(q)|}{|Q| \cdot |R^*(q)|}$$

Mean Position Error, denoted by E^P_{rr}, defines the average position error in query results. Position error for a query result is defined as the average error in the positions of mobile nodes in the query result compared to the correct positions. Let $p(o)$ denote the position of a mobile node o in a query result q under load shedding and $p^*(o)$ denote the correct position of o under $\forall_{i \in [1..l]}\ \Delta_i = \Delta_\vdash$. We have:

$$E^P_{rr} = \sum_{q \in Q} \sum_{o \in q} \frac{|p(o) - p^*(o)|}{|Q| \cdot |R(q)|}$$

Standard Deviation of Containment Error, denoted by D^C_{ev}, and *Coefficient of Variance of Containment Error*, denoted by C^C_{ov}, are fairness metrics that measure the variation among the query results in terms of their containment error. We have $C^C_{ov} = D^C_{ev}/E^C_{rr}$. These two metrics can also be extended to the position error.

4.1.2 Cost of Load Shedding

To evaluate the cost incurred by load shedding, we measure *i*) the time it takes to execute the adaptation step that involves running the THROTLOOP, GRIDREDUCE, and GREEDYINCREMENT algorithms and *ii*) the number of shedding regions that should be known by a mobile node on average. The former metric measures the cost of load shedding from the perspective of the server, whereas the latter measures it from the perspective of the mobile node as well as the wireless network.

4.2 Experimental Setup

The experiments were performed using an hour long car (mobile node) position trace [3] generated from real-world road networks available from the National Mapping Division of the United States Geological Survey (USGS) [18] and traffic volume data taken from [6]. We used a map from the Chamblee region of the state of Georgia in the USA (which covers a rich mixture of expressways, arterial roads, and collector roads) to generate the trace used in this paper. The map covers a region of $\approx$ 200km^2. The trace is generated by simulating the cars going on roads in accordance with the traffic volume data.

The queries used in the experiments are range CQs. The side length for the range queries are randomly selected from the interval $[w/2, w]$ where w is called the *side length parameter*. We use three different distributions for the locations of the queries, namely *Proportional*, *Inverse*, and *Random*. When the query distribution is Proportional, the locations of the queries follow the mobile node distribution. Similarly, they follow the inverse of the mobile node distribution when the query distribution is

[3]The trace generator we have developed is available at http://www.prism.gatech.edu/ gtg470c/research/research.html#kanom

Inverse, and are randomly distributed when the query distribution is Random.

In the experiments presented in this paper we compare our LIRA load shedder with the following alternatives:

– **Random Drop**: The excessive position updates are not admitted to the input FIFO queue and are dropped.

– **Uniform** Δ: A uniform inaccuracy threshold Δ is used to retain only throttle fraction times the original number of location updates. The THROTLOOP algorithm is still used, but the approach is not region-aware and thus space partitioning and update throttler setting are not performed.

– **Lira-Grid**: A downgraded version of the LIRA load shedder, lacking the GRIDREDUCE algorithm which determines the shedding regions based on (l, α)-partitioning. Instead, it uses equally-sized shedding regions based on an l-partitioning, yet still employs the GREEDYINCREMENT algorithm for setting the update throttlers.

Table 2 presents the set of experimental parameters used and the default values they take when not stated otherwise. As we show in this section, the default setting $l = 250$ of the number of shedding regions provides sufficient granularity in partitioning (for a region of size $\approx 200\text{km}^2$) to improve the query result accuracy significantly, while putting very little load on the mobile nodes and the wireless network.

Parameter	Description	Default Value
l	number of shedding regions	250
α	statistics grid side cell count	128
z	throttle fraction	0.5
$\Delta_{\vdash}$	minimum inaccuracy threshold	5 meters
$\Delta_{\dashv}$	maximum inaccuracy threshold	100 meters
c_{Δ}	increment	1 meter
$\Delta_{\Leftrightarrow}$	fairness threshold	50 meters
m/n	# of queries to # of nodes ratio	0.01
w	query side length	1000 meters

Table 2: Experimental parameters

All experiments presented in this paper are performed on an IBM PC with 512MB main memory and 2.4Ghz Intel Pentium4 processor, using Java with Sun JDK 1.5.

4.3 Experimental Results

We present the set of experimental results in two groups. The first group of results are on the query result accuracy and highlight the superiority of LIRA compared to competing approaches for shedding position update load in mobile CQ systems. The second group of results are on the additional cost brought by the LIRA load shedder, and show that the overhead is minimal.

4.3.1 Query Result Accuracy

We study the impact of several system and workload parameters on the query result accuracy and the relative advantage of LIRA over competing approaches.

Impact of the Throttle Fraction: The graphs in Figures 4 and 5 plot the mean position error E^P_{rr} and mean containment error E^C_{rr} as a function of the throttle fraction z, for the proportional query distribution. The left y-axis is used to show the relative values (solid lines) with respect to the error of LIRA and the right y-axis is used to show the absolute errors (dashed lines). Both y-axes are in logarithmic scale. We make three observations from the figure.

First, the LIRA load shedder outperforms all other approaches throughout the entire throttle fraction range. Random Drop performs the worst, followed by Uniform Δ and Lira-Grid. At $z = 0.75$, Random Drop has 300 times the mean position error of LIRA, Uniform Δ has 40 times that of LIRA, and Lira-Grid has 2 times that of LIRA. At $z = 0.5$, Random Drop, Uniform Δ, and Lira-Grid has 10, 2, and 1.08 times the E^P_{rr} of LIRA. The results for the mean containment error E^C_{rr} are similar. Second, we observe that as the throttle fraction z gets smaller, the relative errors approach to 1, while at the same time the absolute errors increase and finally merge. The increasing errors are the result of decreasing update budget, whereas the relative errors decrease to 1 due to the maximum inaccuracy bound $\Delta_{\dashv}$. When the update budget gets smaller than the minimum update expenditure of the system achieved at $\forall_{i \in [1..l]}\, \Delta_i = \Delta_{\dashv}$, all of the three approaches that use inaccuracy thresholds converge at this same solution. For this experimental setting, this convergence occurs around $z = 0.25$. Last, we observe very high (in the order of 10^3's) relative errors for Random Drop and Uniform Δ as z gets closer to 1. This seems surprising at first, as for the case of $z = 1$ (not shown in the figures) all approaches have zero error. However, a slight decrease in the throttle fraction, that is when we have $z = 1 - \epsilon$, introduces some error in the query results for the case of Random Drop and Uniform Δ, whereas it introduces almost no error in the case of LIRA. This is because LIRA cuts the required fraction of position updates from the regions that do not contain any queries. Close to none error of LIRA near $z = 1$ boosts the relative error results for Random Drop and Uniform Δ.

The graphs in Figures 6 and 7 plot the mean containment error E^C_{rr} (relative and absolute) as a function of the throttle fraction z, for the inverse and random query distributions, respectively. The errors of competing approaches relative to LIRA are slightly less for the case of Inverse and Random query distributions compared to that of Proportional query distribution. Otherwise, the results are very similar. In the rest of the experiments, when not stated otherwise, we assume the Proportional query distribution.

Impact of the Number of Shedding Regions: The graphs in Figure 8 plot the relative mean containment error E^C_{rr} of Lira-Grid with respect to LIRA as a function of the number of shedding regions l, for different query distributions. The throttle fraction is set as $z = 0.5$. We observe that Lira-Grid has up to 35% higher containment error in query results compared to LIRA. The improvement provided by LIRA is more pronounced when Inverse query distribution is used and is smallest for the case of Proportional query distribution. As l increases, the flexibility provided by having a larger number of shedding regions improves the error incurred by LIRA at a better rate than Lira-Grid, since LIRA utilizes an intelligent space partitioning algorithm. However, when l gets too large the grid partitioning of Lira-Grid achieves enough granularity to catch Lira in terms of the query result inaccuracy, as observed form the figure. This is because after a certain level of granularity is reached, more fine-grained partitioning is of no use, since the accuracy gain is close to zero for all of the shedding regions. The graphs in Figure 9 attest to this latter intuition. They plot the mean contain-

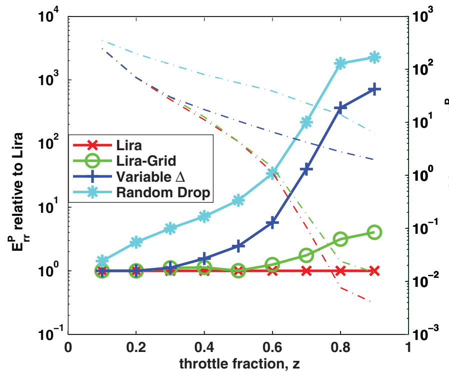

Figure 4: Position Error vs. throttle fraction, Query Dist.: Proportional

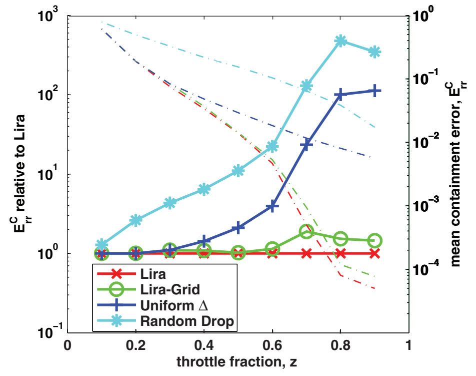

Figure 5: Containment Error vs. throttle fraction, Query Dist.: Proportional

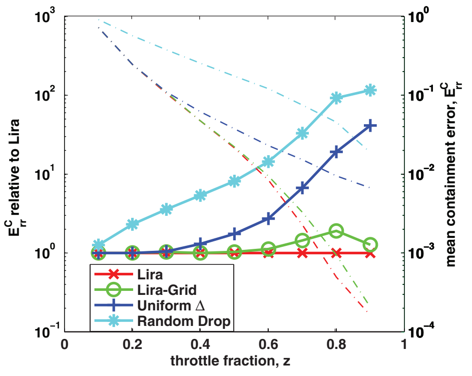

Figure 6: Containment Error vs. throttle fraction, Query Dist.: Inverse

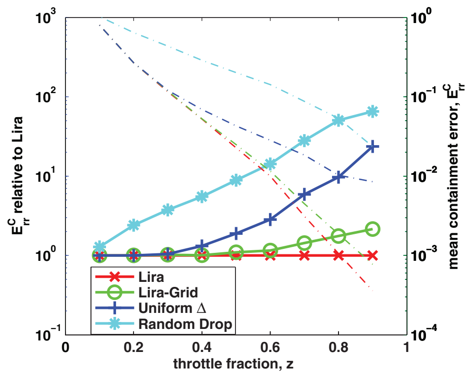

Figure 7: Containment Error vs. throttle fraction, Query Dist.: Random

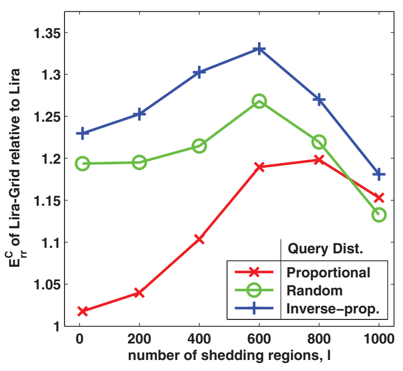

Figure 8: E^C_{rr} of Lira-Grid w.r.t. to LIRA vs. # of shedding regions, $z = 0.5$

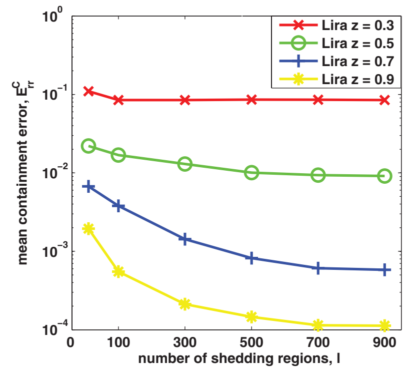

Figure 9: Containment Error of LIRA vs. # of shedding regions

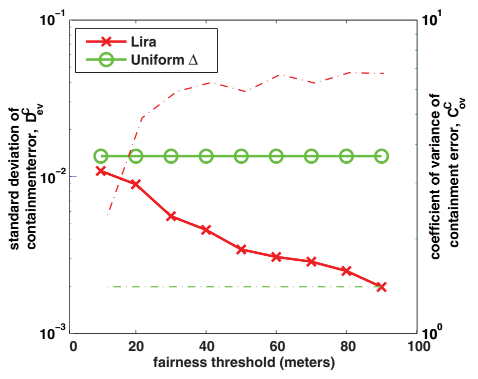

Figure 10: Fairness in query result accuracy for LIRA and Uniform Δ, $z = 0.75$

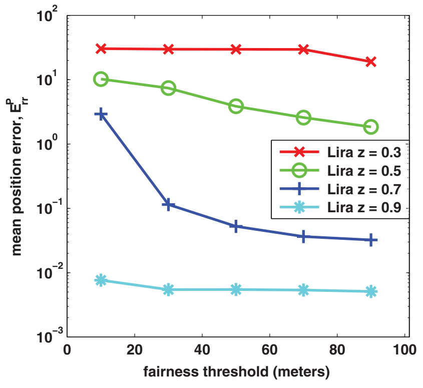

Figure 11: Impact of fairness threshold on E^P_{rr} for different z and l

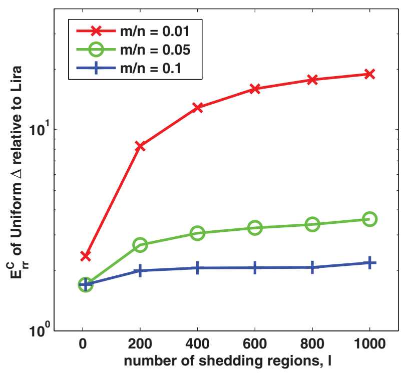

Figure 12: Impact of query to node ratio on containment error, $z = 0.5$

ment error E^C_{rr} of LIRA as a function of the number of shedding regions, for different throttle fractions. We see that the error reduction rate decreases with increasing l and the errors stabilize. The reduction in error is more pronounced for larger z values. Note that the default setting of $l = 250$ for the number of shedding regions is rather conservative based on Figure 9, yet it still performs significantly better than the competing approaches as illustrated by Figures 6 and 7. This conservative setting of l also results in a very lightweight load shedding solution, as we will prove later in this section.

Impact of the Fairness Threshold: The graphs in Figure 10 plot the standard deviation of containment error D^C_{ev} (on the left y-axis corresponding to solid lines) and coefficient of variance of containment error C^C_{ov} (on the right y-axis corresponding to dashed lines) for LIRA and Uniform Δ as a function of the fairness threshold $\Delta_{\Leftrightarrow}$. Note that Uniform Δ does not use a fairness threshold, thus the evaluation metrics stay constant. The surprising observation from the figure is that, with increasing fairness threshold the standard deviation in containment error decreases for LIRA and at all times stays smaller than the D^C_{ev} of Uniform Δ. Even though larger $\Delta_{\Leftrightarrow}$ values imply less fairness, the resulting relaxed constraints in setting the update throttlers enable smaller containment errors and thus the standard deviation also gets smaller. If we look at the coefficient of variance of containment error, which is a better measure of fairness, we see that increasing $\Delta_{\Leftrightarrow}$ increases C^C_{ov} in LIRA and Uniform Δ is more fair compared to LIRA. To put this into simple terms, we can say that on average the difference in errors of two query results will be smaller for LIRA compared to Uniform Δ, yet when judged based on the relative average query error of LIRA and Uniform Δ respectively, the error in query results is more fair among different queries in the case of Uniform Δ.

The graphs in Figure 11 plot the mean position error E_{rr}^{P} for different throttle fractions, as a function of the fairness threshold $\Delta_{\Leftrightarrow}$. We expect that for both small z close to 0.25 (point of reduction to Uniform Δ for this setup) and large z close to 1, fairness threshold will play an insignificant role in determining the mean position error. This is because, for small z the solution reduces to $\forall_{i\in[1..l]}\Delta_i = \Delta_{\dashv}$, making it independent of the fairness threshold. On the other hand, for large z the number of position updates to be cut is small and as a result the small increase in some of the update throttler values does not violate fairness constraints, unless the fairness threshold is close to zero. Figure 11 confirms this understanding, as we observe that for $z = 0.3$ and $z = 0.9$ the error E_{rr}^{P} is marginally sensitive to the fairness threshold $\Delta_{\Leftrightarrow}$, whereas for throttle fraction values in-between, the containment error is more sensitive to the changes in $\Delta_{\Leftrightarrow}$.

Impact of # of Queries and Query Ranges: The graphs in Figure 12 plot the mean containment error E_{rr}^{C} of Uniform Δ relative to LIRA for different number of queries to number of mobile nodes ratios (m/n's) as a function of the number of shedding regions l. We observe that the relative E_{rr}^{C} of Uniform Δ with respect to LIRA is an order of magnitude larger for the case of $m/n = 0.01$ compared to $m/n = 0.1$. This is because LIRA is more effective when the ratio of number of queries to number of nodes is smaller, which implies that there are more regions that contain none or a small number of queries and thus can be used to shed the update load while minimally impacting the result accuracy. However, LIRA has around half the containment error of Uniform Δ even when we have $m/n = 0.1$.

The graphs in Figure 13 plot the mean position error E_{rr}^{P} (using the left y-axis) and the mean containment error E_{rr}^{C} (using the right y-axis) for LIRA as a function of the query side length parameter w. The position and containment errors behave differently under changing query side length. As the average query areas increase, the queries cover a larger region in the space. This makes it harder to reduce the number of updates without increasing the inaccuracy in the positions of the mobile nodes that are included in the query results. As a result the mean position error increases with increasing w. On the other hand, the containment error is a set-based metric and since the result set size increases with increasing w, the percentage of nodes that are correctly included in the result set also increases. This explains the decrease in E_{rr}^{C} as w increases.

4.3.2 Cost of Load Shedding

The cost of load shedding consists of i) configuring the parameters of LIRA on the server side, which includes setting the throttle fraction, shedding regions, and update throttlers, ii) broadcasting the subset of shedding regions and update throttlers that correspond to the coverage area of each base station, and iii) installing the new set of shedding regions and update throttlers on the mobile node side.

Server Side Cost: The graphs in Figure 14 plot the time it takes to execute the THROTLOOP, GRIDREDUCE, and GREEDYINCREMENT algorithms as a function of the number of shedding regions l, for different numbers of cells (α^2) for the statistics grid. For the default parameters of $l = 250$ and $\alpha = 128$, the configuration of LIRA takes around 40 msecs. This will enable frequent adaptation, even though for most applications that involve monitoring cars or pedestrians it is unlikely that the update load will fluctuate with a period less than tens of minutes. Even for an adaptation period of 10 minutes, the configuration of LIRA will take only $6.6\cdot 10^{-5}$ fraction of the adaptation period. Note that these values are for a region of size 200km^2. If we have a 16 times larger region of size 3200km^2 (≈ 10 times the size of Atlanta, the capital city of the state of Georgia, USA), then we should have $l = 16 \cdot 250 = 4000$, and from $\alpha = 2^{\lfloor \log_2(10\cdot\sqrt{l})\rceil}$ we should have $\alpha = 512$. For this setting the configuration of LIRA takes 500 msecs. This corresponds to $8 \cdot 10^{-4}$ fraction of a 10 minute adaptation period. These numbers show that LIRA is indeed lightweight and introduces very little overhead on the server side.

Messaging Cost: Table 3 shows the average number of shedding regions that should be known to a base station as a function of the base station coverage area radius. However, in reality base stations have smaller coverage regions at places where the number of users is large (urban areas) and larger coverage regions at places where the number of users is small (suburban areas) [13]. This nature of base stations match perfectly with LIRA's space partitioning scheme, since the number of partitions are usually larger for dense areas and the small base station coverage areas help decreasing the average number of shedding regions known to a mobile node. Following this logic, we have used a node density dependent base station placement scheme and found that on the average each node and thus each base station should know around 41 shedding regions. Assuming a shedding region (which is square in shape) is represented by 3 floats and an update throttler is represented by a single 4 byte float, the size of the broadcast data sent by a base station to all nodes in its coverage area to install the shedding regions and update throttlers is around $41 \cdot (3 + 1) \cdot 4$ bytes = 656 bytes on average. To asses the messaging cost of LIRA, compare this number to 1472 bytes, which is the maximum payload available to an UDP packet over Ethernet with a typical MTU of 1500 bytes. When LIRA reconfigures the load shedding parameters, the new information is installed on all mobile nodes by using an average of one wireless broadcast packet per base station.

Mobile Node Side Cost: Since the total number of shedding regions known to a mobile node at any time is only around 41, LIRA does not put a major burden on mobile nodes in terms of memory consumption or processing load. By employing a tiny 5×5 grid index on the mobile node side, the shedding region that contains the current position of the mobile node can be found quickly. As a result, LIRA will work on computationally weak mobile nodes without any problem.

5 Related Work

To the best of our knowledge, this is the first work on position update load shedding in mobile CQ systems. Several works have appeared in the literature on handling the position updates efficiently in mobile CQ systems [17, 8, 10, 20] or using motion modeling to reduce the number of position updates received [19, 2]. The first set of works do not directly address the update load shedding problem, but instead aim at decreasing the IO and CPU cost of integrating the position updates into spatial index structures. This does not involve suppressing or dropping

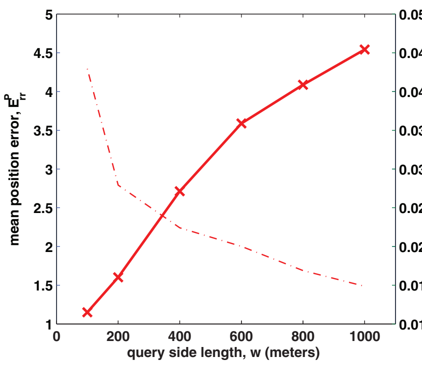

Figure 13: Impact of query side length on E^P_{rr} and E^C_{rr}, $z = 0.5$

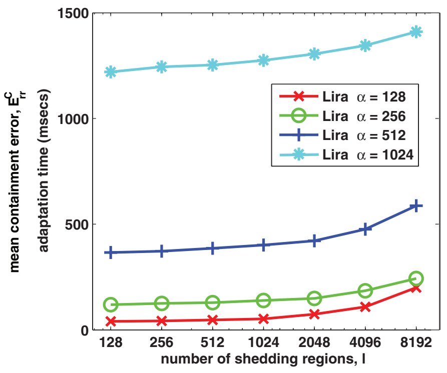

Figure 14: Server side cost of configuring LIRA, $z = 0.5$

41 Δ_i's on average, which takes $41 \cdot (3+1) \cdot 4$ bytes = 656 bytes. Compare this to 1472 bytes, which is the maximum payload available to an UDP packet over Ethernet with a typical MTU of 1500 bytes.

Table 3: Number of shedding regions per base station

base station radius (km)	1.0	2.0	3.0	4.0	5.0
# of Δ_i's per node	3.1	12.5	28.2	50.2	78.5

the position updates from the mobile nodes, which is inevitable when the current resources of the system are not sufficient to handle the update load. Our work is complementary in nature to this line of previous work. The second set of previous work use motion modeling to cut the update load, and ensure that the resulting position updates do not have inaccuracy beyond a prespecified threshold. A key difference is that, our work is driven by the update budget enforced by the load on the system. We adjust the inaccuracy thresholds to reduce the update expenditure of the system to meet the update budget. In other words, our work utilizes the previous work on motion modeling at the mobile node side for actuating the position update suppressing. However, the core of our solution is to find a partitioning and a set of inaccuracy thresholds to associate with each partition, so that the position updates received from the mobile nodes can answer the queries installed in the system accurately.

There have also been a number of distributed solutions to evaluate CQs in mobile systems [1, 7, 3]. In these systems, the position updates are only received if they affect a query result. Even though these systems do not provide any load shedding capability, their update load is expected to be significantly lower compared to solutions that track all mobile nodes. It is worth to note that these solutions cannot support historic queries, since the location updates are not received from all objects. The adhoc snapshot queries are also expensive to evaluate. Interestingly, LIRA can be configured to mimic the behavior of these systems by setting the maximum inaccuracy bound $\Delta_\dashv$ to a large value. However, our system has the additional advantage of not being tied to any specific query processing technique and has very little overhead.

6 Conclusion

We presented LIRA, a position update load shedder for mobile CQ systems. The primary feature of LIRA is its region-awareness, which enables it to partition the space into a set of shedding regions and apply differing amounts of update throttling for different shedding regions. We formalized the update load shedding problem as an optimization one, where the aim is to minimize the inaccuracy introduced in the query results and the constraint is to meet a given or an automatically calculated update budget. We developed a heuristic algorithm to discover a partitioning of the space that leads to reduced error in query results, and an optimal algorithm that sets the update throttlers associated with each shedding region to minimize the query result inaccuracy. We showed that the LIRA load shedder is significantly superior to random update dropping and uniform inaccuracy threshold schemes. Moreover, LIRA is lightweight by design and can be used in conjunction with many of the existing update indexing and mobile CQ processing techniques.

References

[1] Y. Cai and K. A. Hua. An adaptive query management technique for efficient real-time monitoring of spatial regions in mobile database systems. In *IEEE International Performance Computing and Communications Conference*, 2002.

[2] A. Civilis, C. S. Jensen, and S. Pakalnis. Techniques for efficient road-network-based tracking of moving objects. *IEEE Transactions on Knowledge and Data Engineering*, 17(5):698–712, 2005.

[3] B. Gedik and L. Liu. Distributed processing of continuously moving queries on moving objects in a mobile system. In *International Conference on Extending Database Technology*, 2004.

[4] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Processing moving queries over moving objects using motion adaptive indexes. In *IEEE Transactions on Knowledge and Data Engineering*, volume 18, pages 651–668, 2006.

[5] Google RideFinder home page. http://labs.google.com/ridefinder, Febuary 2006.

[6] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *ACM International Conference on Mobile Systems, Applications, and Services*, 2003.

[7] H. Hu, J. Xu, and D. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *ACM International Conference on Management of Data*, 2005.

[8] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *International Conference on Very Large Data Bases*, 2004.

[9] D. V. Kalashnikov, S. Prabhakar, S. Hambrusch, and W. Aref. Efficient evaluation of continuous range queries on moving objects. In *International Workshop on Database and Expert Systems Applications (DEXA)*, 2002.

[10] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *International Conference on Very Large Data Bases*, 2003.

[11] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable

incremental processing of continuous queries in spatio-temporal databases. In *ACM International Conference on Management of Data*, 2004.

[12] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[13] QualComm. Wireless access solutions using 1xEV-DO. http://www.qualcomm.com/technology/1xev-do/webpapers/wp_wirelessaccess.pdf, 2005.

[14] S. Ross. *A First Course in Probability*. Prentice Hall, 2005.

[15] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *ACM International Conference on Management of Data*, 2000.

[16] C. Science and T. Board. *IT Roadmap to a Geospatial Future*. The National Academics Press, November 2003.

[17] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An optimized spatio-temporal access method for predictive queries. In *International Conference on Very Large Data Bases*, 2003.

[18] U.S. Department of the Interior. U.S. geological survey web page. http://www.usgs.gov/, November 2003.

[19] O. Wolfson, P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Springer Distributed and Parallel Databases*, 7(3):257–387, 1999.

[20] X. Xiong and W. G. Aref. R-trees with update memos. In *IEEE International Conference on Data Engineering*, 2006.

APPENDIX

A Proof of Theorem 3.1

Proof. The proof is by contradiction. Assume that the optimal solution is denoted by $\{\Delta_i^*\}$ and is different than the solution given by GREEDYINCREMENT, which we denote as $\{\Delta_i^+\}$. There must be at least one $j \in [1..l]$ such that $\Delta_j^+ < \Delta_j^*$. Otherwise we would have $\forall i, \Delta_i^+ \geq \Delta_i^*$ which will introduce a contradiction, since it implies that the solution of GREEDYINCREMENT has not consumed its update budget completely (which is not possible due to algorithm design). Similarly, we should have at least one $k \neq j \in [1..l]$ such that $\Delta_k^+ > \Delta_k^*$. Otherwise we would have $\forall i, \Delta_i^+ \leq \Delta_i^*$ which will introduce a contradiction, since it implies that the solution of GREEDYINCREMENT has overshot the update budget.

Since we have $\Delta_k^+ > \Delta_k^*$, there must be a step in GREEDYINCREMENT in which Δ_k is incremented from value a to b such that $a \leq \Delta_k^* < b$. Let v be the value of Δ_j when this step is taken by GREEDYINCREMENT. We should have $S_k(a) > S_j(v)$, since Δ_k is selected to be incremented but not Δ_j. Since f is piece-wise linear and a is a brake point of f (because $c_\Delta = (\Delta_\dashv - \Delta_\vdash)/\kappa$), S_k is constant throughout $[a, a + c_\Delta]$); and since $c_\Delta \geq b - a$, we have $S_k(a) = S_k(\Delta_k^*)$. This leads to $S_k(\Delta_k^*) > S_j(v)$. Since we have $\Delta_j^+ < \Delta_j^*$, we must have $v < \Delta_j^*$ and since S_j is a decreasing function, we have the following:

$$S_k(\Delta_k^*) > S_j(\Delta_j^* - \epsilon) \geq S_j(\Delta_j^*), \epsilon \in (0, \Delta_j^* - v] \quad (1)$$

Pick a sufficiently small $\alpha \in (0, \epsilon]$ such that Δ_k^* and $\Delta_k^* + \beta$ are within the same segment of f, where we have

$$f(\Delta_k^*) - f(\Delta_k^* + \beta) = f(\Delta_j^* - \alpha) - f(\Delta_j^*) \quad (2)$$

We now show that the solution $(\{\Delta_i^*\} \setminus \{\Delta_j, \Delta_k\}) \cup \{\Delta_j - \alpha, \Delta_k + \beta\}$ has less average query inaccuracy than the optimal solution, thus a contradiction. This can also be stated as $m_k \cdot \beta - m_j \cdot \alpha < 0$, which is the change in average query inaccuracy going from the original optimal solution to the one we have constructed. Since Δ_k^* and $\Delta_k^* + \beta$ are within the same segment of f, we have:

$$\begin{aligned} S_k(\Delta_k^*) &= \frac{f(\Delta_k^*) - f(\Delta_k^* + \beta)}{m_k \cdot \beta} \\ S_j(\Delta_j^* - \alpha) &> \frac{f(\Delta_j^* - \alpha) - f(\Delta_j^*)}{m_j \cdot \alpha} \end{aligned}$$

Plugging these values into Equation 1 we have:

$$\begin{aligned} S_k(\Delta_k^*) &> S_j(\Delta_j^* - \alpha) \\ \frac{m_k \cdot \beta}{m_j \cdot \alpha} &< \frac{f(\Delta_k^*) - f(\Delta_k^* + \beta)}{f(\Delta_j^* - \alpha) - f(\Delta_j^*)} \end{aligned} \quad (3)$$

Using Equation 2 and 3 we get $m_k \cdot \beta < m_j \cdot \alpha$. This proves the contradiction and thus completes the proof. □