RC24214 (W0703-048) March 12, 2007
Computer Science

IBM Research Report

MobiQual: QoS-aware Load Shedding in Mobile CQ Systems

Bugra Gedik, Kun-Lung Wu, Philip S. Yu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Ling Liu
College of Computing
Georgia Institute of Technology
Atlanta, GA

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

MobiQual: QoS-aware Load Shedding in Mobile CQ Systems

Buğra Gedik[♠,◇]
bgedik@us.ibm.com

Kun-Lung Wu[♠]
klwu@us.ibm.com

Philip S. Yu[♠]
psyu@us.ibm.com

Ling Liu[◇]
lingliu@cc.gatech.edu

[♠] IBM T. J. Watson Research Center, Hawthorne, NY
[◇] College of Computing, Georgia Institute of Technology, Atlanta, GA

ABSTRACT

Freshness and accuracy are two key measures of quality of service (QoS) in processing location-based, mobile continual queries (CQs). Freshness necessitates the CQ server to perform frequent query re-evaluations. Accuracy demands the CQ server to receive and process frequent position updates from the mobile nodes. However, it is often difficult to provide both fresh and accurate CQ results due to (a) limited resources in computing and communication and (b) fast-changing load conditions caused by continuous mobile node movement. Thus a key challenge for a mobile CQ system is: How do we achieve the highest possible quality of the query results, in both freshness and accuracy, with currently available resources under rapidly changing load conditions? In this paper, we formulate this problem as a load shedding one, and develop MobiQual as a solution. It is a dynamic and QoS-aware framework for performing both update load shedding and query load shedding. MobiQual uses per-query QoS specifications to maximize the overall freshness and accuracy of the query results. In view of the QoS specifications, it employs query clustering and space partitioning mechanisms to apply differing amounts of query and update load shedding for different query and mobile node groups, respectively. We show that our solution is superior to competing approaches that lack the QoS-awareness properties of MobiQual, as well as solutions that perform query-only or update-only load shedding.

1. INTRODUCTION

With the ever increasing accessibility of wireless communications and the proliferation of mobile devices, we are experiencing a world where we can stay connected while on-the-go. Combined with the availability of low-cost positioning devices (such as GPS sensors), this has created a new class of applications and business opportunities in the area of mobile location-based services (LBSs). Examples include location-aware information delivery and resource management, such as transportation services (NextBus bus locator [13], Google ride finder [6]), fleet management, mobile games, and battlefield coordination.

A key challenge for LBSs is a scalable location monitoring system capable of handling a large number of mobile nodes and processing complex queries over those node positions. Although several mobile continual query (CQ) systems have been proposed to efficiently handle long-running location monitoring tasks in a scalable manner [5, 9, 10, 12, 18], the focus of these works is primarily on efficient indexing and query processing techniques, not accuracy or freshness of the query results.

Accuracy (inaccuracy) refers to the amount of mobile node position errors found in the query results *at the time of* query re-evaluation. The accuracy measure is strongly tied to the frequency of position updates received from the mobile nodes. Alternatively, accuracy can be measured by the amount of containment errors found in the query results, including both false positives and false negatives. However, a bound on the amount of containment errors can be approximated by a bound on the position errors, if the distribution of the mobile nodes around the query boundaries is at hand. In this paper, we choose the amount of node position errors as the accuracy measure, because such inaccuracy can be easily bounded by a threshold-based position reporting scheme [17, 2].

Freshness (staleness), on the other hand, refers to the age of the query results *since* the last query re-evaluation. It is dependent on the frequency of query re-evaluations performed at the server. As mobile nodes continue to move, there are further deviations in mobile node positions after the last query re-evaluation. However, such deviations are not attributed to inaccuracy, which only captures those at the time of query re-evaluation. Hence, freshness can be seen as a metric capturing the post-query-re-evaluation deviations in mobile node positions.

To obtain fresher query results, the CQ server must re-evaluate the continual queries more frequently, requiring more computing resources. Similarly, to attain more accurate query results, the CQ server must receive and process position updates from the mobile nodes in a higher rate, demanding communication as well as computing resources. However, it is almost impossible for a mobile CQ system to achieve 100% fresh and accurate results due to continuously changing positions of mobile nodes. A key challenge then is: How do we achieve the highest possible quality of the query results, in both freshness and accuracy, with currently available resources under fast changing workloads?

In this paper, we propose MobiQual as a solution. It is a resource-adaptive and QoS-aware load shedding framework for mobile CQ systems. MobiQual is capable of providing high-quality query results by dynamically determining the appropriate amount of update load shedding and query load shedding to be performed according to the application-level QoS specifications of the queries.

To the best of our knowledge, none of the existing work has considered the application-level freshness and accuracy of mobile queries, or the opportunities for performing dynamic update load shedding and query load shedding in the presence of limited resources or severe performance degradations. Yet, due to the rapidly changing nature of mobile node positions, freshness and accuracy are two critical quality of service (QoS) measures. More importantly, limited-resource scenarios are the norm, rather than the exception, in mobile CQ systems. The concepts of freshness and accuracy in mobile CQ systems are, to some extent, similar to those of timeliness and completeness, respectively, in the web information monitoring domain [14]. It is important to note that higher freshness does not necessarily imply higher accuracy and vice versa.

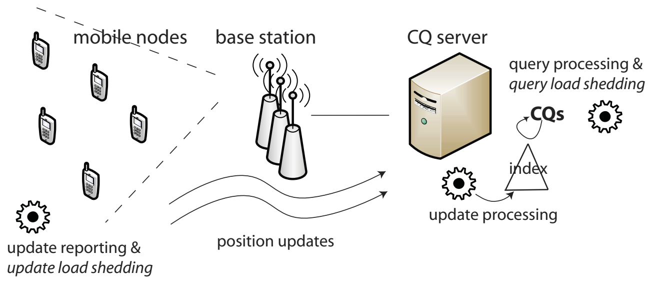

Figure 1: Mobile CQ system and load shedding

1.1 Load Shedding in Mobile CQ Systems

In a mobile CQ system, the CQ server receives position updates from the mobile nodes through a set of base stations (see Figure 1) and periodically evaluates the installed continual queries (such as continual range or nearest neighbor queries) over the last known positions of the mobile nodes. Since the mobile node positions change continuously, motion modeling [17, 2] is often used to reduce the number of updates sent by the mobile nodes. The server can predict the locations of the mobile nodes through the use of motion models, albeit with increasing errors. Mobile nodes generally use a threshold to reduce the amount of updates to be sent to the server and to limit the inaccuracy of the query results at the server side below the threshold. Smaller thresholds result in smaller errors and higher accuracy, at the expense of a higher load on the CQ server. This is because a larger number of position updates must be processed by the server, for instance, to maintain an index [16, 10]. When the position update rates are high, the amount of position updates is huge and the server may randomly drop some of the updates if resources are limited. This can cause unbounded inaccuracy in the query results. In MobiQual, we use accuracy-conscious *update load shedding* to regulate the load incurred on the CQ server due to position update processing by dynamically configuring the inaccuracy thresholds at the mobile nodes.

Another major load for the CQ server is to keep the query results up to date by periodically executing the CQs over the mobile node positions. More frequent query re-evaluations translate into increased freshness in the query results, also at the expense of a higher server load. Given limited server resources, when the rate of query re-evaluations is high, the amount of queries to be re-evaluated is vast and the server may randomly drop some of the re-evaluations, causing stale query results (low freshness). In MobiQual, we utilize freshness-conscious *query load shedding* to control the load incurred on the CQ server due to query re-evaluations by configuring the query re-evaluation periods.

In general, the total load due to evaluating queries and processing position updates dominates the performance and scalability of the CQ server and thus should be bounded to the capacity of the CQ server. Furthermore, the time-varying processing demands of a mobile CQ system entails that update and query load shedding should be dynamically balanced and adaptively performed in order to match the workload with the server's capacity, while meeting the accuracy and freshness requirements of queries.

1.2 The MobiQual Approach

The MobiQual system aims at performing dynamic load shedding to maximize the overall quality of the query results, based on per-query QoS specifications and subject to processing capacity constraints. The QoS specifications are defined based on two factors: accuracy and freshness. In MobiQual, the QoS specifications are used to decide on not only how to spread out the impact of load shedding among different queries, but also how to find a balance between query load shedding and update load shedding. The main idea is to apply *differentiated load shedding* to adjust the accuracy and freshness of queries. Namely, load shedding on position updates and query re-evaluations is done in such a way that the resulting impacts on freshness and accuracy are nonuniform among the queries.

From the perspective of query load shedding, we make two observations to show that nonuniform freshness in the query results can increase the overall QoS of the mobile CQ system: (1) Different queries have different costs in terms of the amount of load they incur. (2) Different queries have different tolerance to staleness in the query results. Thus it is more effective to shed load (by sacrificing certain amount of freshness) on a costly query than an inexpensive one. This is especially beneficial if the costly query happens to be less stringent on freshness, based on its QoS specification. Bearing these observations in mind, in MobiQual we use the query re-evaluation periods as control knobs to perform query load shedding, where the same amount of increase in query re-evaluation periods for different queries brings differing amounts of load reduction and QoS degradation with respect to freshness. We refer to the load shedding that uses query re-evaluation periods to maximize the average freshness of the query results under the QoS specifications as the *QoS-aware query load shedding.*

Similar to query load shedding, we make two observations regarding update load shedding to show that nonuniform result accuracy can increase the overall QoS. First, different geographical regions have different numbers of mobile nodes and queries. Second, different queries have different tolerance to position errors in the query results. This means that shedding more updates from a region with a higher density of mobile nodes and a lower density of queries can bring a higher reduction on the update load and yet have a smaller impact on the overall query result accuracy. This is especially true if the queries within the region have less stringent QoS specifications in terms of accuracy. Thus, in MobiQual we use the inaccuracy thresholds employed in motion modeling as control knobs to adjust the amount of update load shedding to be performed, where the same amount of increase in inaccuracy thresholds for different geographical regions brings differing amounts of load reduction and QoS degradation with respect to accuracy. We refer to the load shedding that adjusts the inaccuracy thresholds based on the densities of mobile nodes and queries to maximize the average accuracy of the query results under the QoS specifications as the *QoS-aware update load shedding*.

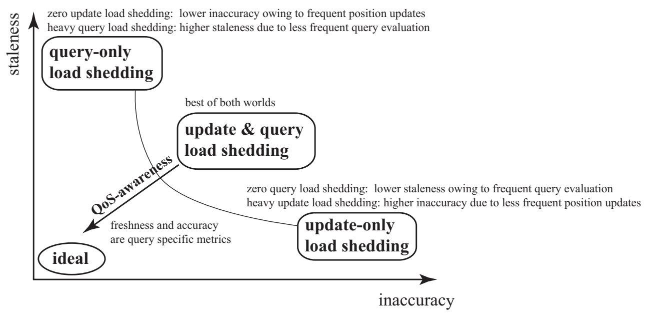
staleness
zero update load shedding: lower inaccuracy owing to frequent position updates
heavy query load shedding: higher staleness due to less frequent query evaluation
query-only load shedding
best of both worlds
update & query load shedding
QoS-awareness
zero query load shedding: lower staleness owing to frequent query evaluation
heavy update load shedding: higher inaccuracy due to less frequent position updates
freshness and accuracy are query specific metrics
update-only load shedding
ideal
inaccuracy

Figure 2: QoS-aware update load shedding and query load shedding

MobiQual dynamically maintains a *throttle fraction*, which defines the amount of load that should be retained. It performs both update load shedding and query load shedding to control the load of the system according to this throttle fraction, while maximizing the overall quality of the query results. As illustrated in Figure 2, MobiQual not only strikes a balance between freshness and accuracy by employing both query and update load-shedding, but also improves the overall quality of the results by utilizing per-query QoS specifications to capture each query's different tolerance to staleness and inaccuracy.

1.3 Contributions

MobiQual makes the following three major contributions:
− We combine query load shedding and update load shedding within the same framework by formalizing the problem as an optimization one. We provide a fast greedy algorithm that uses differentiated load shedding concept to configure query re-evaluation periods and inaccuracy thresholds, aiming at achieving high overall QoS with respect to the freshness and accuracy requirements of the queries.
− We use per-query QoS specifications to characterize the tolerance of queries to the staleness and inaccuracy in the query results. In order to deal with a large number of queries and mobile nodes, we introduce query grouping and space partitioning to reduce the adaptation time required to reconfigure the system upon changes in workload characteristics, and to enable low-overhead and frequent adaptation. We also impose certain structure on the QoS functions to make aggregation for query load shedding independent of aggregation for update load shedding.
− The MobiQual load shedding mechanisms are lightweight, enabling quick adaptation to changes in the workload, in terms of both the number of queries, the number of mobile nodes, and their changing movement patterns. We have conducted a detailed experimental study. Our experimental results show that MobiQual significantly outperforms the approaches that are based on query-only or update-only load shedding, or the approaches that lack the support of differentiated load shedding elements of the MobiQual solution, including the query grouping and the space partitioning mechanisms.

2. NOTATION AND FUNDAMENTALS

The set of continual queries installed in the system is denoted by Q. For each query $q \in Q$, it has an associated QoS specification $\mathcal{S}_q$. The QoS function $\mathcal{S}_q(\tau_q, \epsilon_q)$ takes a value in $[0, 1]$, where 1 represents perfect quality in terms of freshness and position error, and 0 represents the worst. τ_q and ϵ_q are used to denote the degree of staleness and inaccuracy in the query results, respectively. τ_q corresponds to the query re-evaluation period for q, whereas ϵ_q corresponds to the average of the inaccuracy thresholds used in motion modeling for the mobile nodes within the query result of q. At any given time, the result of query q can be at most τ_q seconds old and at the time of query evaluation the position of a mobile node in the query result can deviate from its actual position by ϵ_q meters on average. The mobile CQ system supports a minimum staleness value of $\tau_\vdash$ and a minimum position error of $\epsilon_\vdash$. For any query q, we have $\mathcal{S}_q(\tau_\vdash, \epsilon_\vdash) = 1$. Similarly, we introduce a maximum staleness value, denoted by $\tau_\dashv$, and a maximum position error, denoted by $\epsilon_\dashv$. The staleness in the query results cannot exceed the maximum threshold value of $\tau_\dashv$, at which point the results are assumed to be useless. Also the position error is bounded by $\epsilon_\dashv$. In summary, we have $\tau_q \in [\tau_\vdash, \tau_\dashv]$ and $\epsilon_q \in [\epsilon_\vdash, \epsilon_\dashv]$. The minimum and maximum staleness and position error thresholds are system parameters.

Since a scalable mobile CQ system should be able to handle tens of thousands of queries and hundreds of thousands of mobile nodes, it is inefficient, even if it is possible, to adjust and dynamically maintain the re-evaluation periods for queries and inaccuracy thresholds for mobile nodes individually. In MobiQual, we divide the set of m queries into k groups, denoted by $C_j, j \in [1..k]$, where the number of queries in C_j is denoted by m_j and $\sum_{j=1}^{k} m_j = m$. The queries within the same group C_j share the same re-evaluation period P_j, i.e. we have $\forall q_u \in C_j, \tau_u = P_j$. We denote the one-time cost of processing the set of queries in C_j as $f_c(C_j)$, which is simply the sum of one-time processing costs of individual queries. The usage of the cost model in MobiQual does not require absolute values of query costs and can work with relative values for cost-based analysis. A key question for query load shedding is how to divide the queries into k query groups and how to compute the re-evaluation period P_j for each query group C_j $(j \in [1..k])$.

Similarly, given a total number of n mobile nodes, we partition the geographical area of interest into l regions, denoted by $A_i, i \in [1..l]$, where the number of mobile nodes in A_i is denoted by n_i and $\sum_{i=1}^{l} n_i = n$. The mobile nodes within the same region A_i use the same inaccuracy threshold Δ_i. A query q_u whose result lies completely within region A_i will have $\epsilon_u = \Delta_i$. For queries whose results contain mobile nodes from different regions, ϵ_u is given by a weighted average of Δ_i values of the involved regions.

We denote the fraction of updates received from a region A_i, when using an inaccuracy threshold Δ_i, as $f_r(\Delta_i)$. f_r is relative to the ideal case where all Δ_i's are equal to the minimum position error $\epsilon_\vdash$. Thus we have $f_r(\epsilon_\vdash) = 1 > f_r(\epsilon_\dashv)$. f_r is a non-increasing continuous function with a positive second derivative. More detailed characterization of such functions exist for specific motion modeling and prediction schemes [17, 4]. A key challenge for update load shedding is how to partition the geographical area of interest into l regions and how to compute the inaccuracy threshold Δ_i for each region A_i ($i \in [1..l]$).

2.1 Trade-offs in Setting k and l

In general, the larger the number of query groups (k) we have, the higher the quality of the query results is in terms of freshness, as it enables performing differentiated load shedding with finer granularity. The only restriction in setting the value of k is the computational cost (which forms a major part of the adaptation cost) of finding an effective setting for the re-evaluation periods $P_j, j \in [1..k]$. Similar trade-off is observed in setting the number of regions (l) and thus the number of inaccuracy thresholds, with one exception. Since the changes in inaccuracy thresholds have to be communicated back to the mobile nodes through control messages (broadcasts from base stations), there is a second dimension to this trade-off: The larger the l value is, the higher the control cost of the adaptation step will be. In Section 7, we experimentally evaluate the benefit/cost trade-off in setting k to show that with lightweight adaptation we can achieve high quality query results. The details of setting l can be found in [4], whose results apply to the QoS-aware update load shedding problem presented here and are used for setting the value of l in the experimental evaluation of Section 7. We show that this facilitates a very lightweight solution and significantly improves the query results in terms of accuracy.

2.2 Solution Outline

There are three functional components in the MobiQual system: *reduction*, *aggregation*, and *adaptation*.

− **Reduction** includes the algorithm for grouping the queries into k clusters and the algorithm for partitioning the geographical space of interest into l regions. The query groups are incrementally updated when queries are installed or removed from the system. The space partitioning is re-computed prior to the periodic adaptation.

− **Aggregation** involves computing aggregate-QoS functions for each query group and region. The aggregated QoS functions for each query group represent the freshness aspect of the quality. The aggregated QoS functions for each region represent the accuracy aspect of the quality. We argue that the separation of these two aspects is essential to the development of a fast algorithm for configuring the re-evaluation periods and the inaccuracy thresholds to perform adaptation. QoS-aggregation is repeated only when there is a change in the query grouping or the space partitioning.

− **Adaptation** is performed periodically to determine: (*i*) the throttle fraction $z \in [0, 1]$, which defines the amount of load that can be retained relative to the load of providing perfect quality (i.e., $\forall_{j\in[1..k]} P_j = \tau_\vdash$ and $\forall_{i\in[1..l]} \Delta_i = \epsilon_\vdash$); (*ii*) the setting of re-evaluation periods $P_j, j \in [1..k]$; and (*iii*) the setting of inaccuracy thresholds $\Delta_i, i \in [1..l]$. The latter two are performed with the aim of maximizing the overall QoS. The computation of the throttle fraction is performed by monitoring the performance of the system and adjusting z in a feedback loop.

In the remaining sections, we first present the aggregation of QoS functions, assuming that the query grouping and space partitioning are performed (Section 3). We then present the formulation of the QoS-aware query load shedding problem and present the *quality loss based clustering* (QLBC) algorithm for clustering the queries into k groups (Section 4). Then we formalize the QoS-aware update load shedding problem and provide a brief description of the QoS-aware space partitioning algorithm for dividing the geographical space of interest into l regions (Section 5). Finally, we present the formulation of the problem of combining query load shedding with update load shedding, and present the *minimum quality loss per cost step* (MQLS) algorithm for performing the adaptation step (Section 6).

3. AGGREGATING THE QOS FUNCTIONS

The aim of QoS aggregation is to associate an aggregate function $\mathcal{V}^*_j(P_j)$ for each query group C_j, and an aggregate function $\mathcal{U}^*_i(\Delta_i)$ for each region A_i, such that the overall QoS of the system, denoted by Ψ, is maximized. We define Ψ as follows:

$$\Psi = \frac{1}{m} \sum_{q\in Q} \mathcal{S}_q(\tau_q, \epsilon_q) \qquad (1)$$

$\mathcal{S}_q(\tau_q, \epsilon_q)$ denotes the QoS specification for query q and can be defined as follows:

$$\mathcal{S}_q(\tau_q, \epsilon_q) = \alpha_q \cdot \mathcal{V}_q(\tau_q) + (1 - \alpha_q) \cdot \mathcal{U}_q(\epsilon_q)$$

In other words, $\mathcal{S}_q(\tau_q, \epsilon_q)$ is a linear combination of the freshness QoS function $\mathcal{V}_q(\tau_q)$ and the accuracy QoS function $\mathcal{U}_q(\epsilon_q)$. The parameter $\alpha_q \in [0, 1]$, called *freshness weight*, is used to adjust the relative importance of the two components, freshness and accuracy. $\mathcal{V}_q(\tau_q)$ and $\mathcal{U}_q(\epsilon_q)$ are non-increasing positive functions, where $\mathcal{V}_q(\tau_\vdash) = 1$ and $\mathcal{U}_q(\epsilon_\vdash) = 1$.

Since the query groups are non-overlapping, we have:

$$\mathcal{V}^*_j(P_j) = \sum_{q\in C_j} \alpha_q \cdot \mathcal{V}_q(P_j) \qquad (2)$$

We approximate the $\mathcal{V}_q$ functions using piece-wise linear functions of κ equal-sized segments along the input domain $[\tau_\vdash, \tau_\dashv]$. This enables us to represent the aggregate QoS functions ($\mathcal{V}^*_j$'s) as piece-wise linear functions of κ segments as well. Figure 3 gives and example of aggregating two piece-wise linear functions of 4 segments each.

Recall that the set of queries that intersect a region A_i can overlap with the set of queries that intersect a different region, since a query q can intersect more than one region. Let $m_q(i)$ denote the fraction of q's query region that lies within A_i and Q denotes the set of queries in the system.

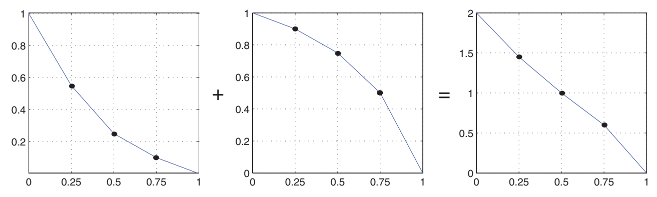

Figure 3: Example of QoS function aggregation

Then, we have:

$$\mathcal{U}_i^*(\Delta_i) = \sum_{q \in Q, \text{ s.t. } m_q(i)>0} (1-\alpha_q) \cdot m_q(i) \cdot \mathcal{U}_q(\Delta_i) \quad (3)$$

The equality in Equation 3 holds when (a) $\mathcal{U}_q$'s are linear functions,[1] or (b) $\mathcal{U}_q$'s are piece-wise linear functions and there are no queries crossing the region borders. However, it is still a good approximation for the general case of piece-wise linear functions if the crossings are not frequent. Because the size of a region is significantly larger than that of a query, query crossings are indeed infrequent. Like $\mathcal{V}_j^*$'s, we also represent $\mathcal{U}_i^*$'s as piece-wise linear functions with κ segments. Based on this analysis, the Equation 1 can be written in the following form:

$$\Psi = \frac{1}{m}\left(\sum_{j=1}^{k}\mathcal{V}_j^*(P_j) + \sum_{i=1}^{l}\mathcal{U}_i^*(\Delta_i)\right) \quad (4)$$

Note that, for a given $j \in [1..k]$, $\mathcal{V}_j^*$ is independent of Δ_i's ($i \in [1..l]$). Similarly, for a given $i \in [1..l]$, $\mathcal{U}_i^*$ is independent of P_j's ($j \in [1..k]$). This separation allows us to operate at the granularity of query groups for configuring query load shedding and at the granularity of regions for configuring the update load shedding.

It is critical to note that the queries within C_j may intersect a number of different areas, and similarly queries within area A_i may be contained in a number of different query groups. As a result, if $\mathcal{U}_i^*$'s were not independent of P_j's, altering the re-evaluation period P_j for queries within C_j may have altered more than m_j different aggregate QoS functions belonging to different regions. A similar argument is valid for altering the inaccuracy threshold Δ_i for A_i when $\mathcal{V}_j^*$'s are not independent of Δ_i's. Thus, without a clear separation of re-evaluation periods and inaccuracy thresholds in aggregated QoS functions, one may create a significant problem for the system optimization, as it will defy reduction by making $\mathcal{V}_j^*$'s and $\mathcal{U}_i^*$'s dependent on a large number of parameters, making their computation costly.

One downside of representing a query's QoS specification as a linear combination of a freshness-related QoS function and an accuracy-related QoS function is the loss of certain amount of expressiveness, compared to the case of an unrestricted QoS function of two parameters. Yet, the presented model still manages to capture a wide spectrum of QoS specifications, ranging from staleness insensitive ($\alpha_q = 0$) to inaccuracy insensitive ($\alpha_q = 1$) scenarios. As we will present in the rest of the paper, this way of modeling the QoS specifications lends itself to an efficient implementation of the adaptive load shedding optimization, making it possible to adapt more frequently, with minor overhead.

To better understand the problem of how to combine query load shedding and update load shedding, we first discuss the query load shedding and the update load shedding separately in the next two sections and then present our final solution for combining QoS-aware query load shedding and QoS-aware update load shedding.

4. QOS-AWARE QUERY LOAD SHEDDING

We now focus on the QoS-aware query load shedding problem, by only considering the freshness aspect of the quality and the cost of query re-evaluation.

4.1 Formalization of the Problem

The aim of the query load shedding problem is to maximize the first component of the overall quality from Equation 4, denoted by Ψ_v. Given k query groups, recalling that $\mathcal{V}_j^*(P_j)$ denote the aggregation function for the query group C_j, and P_j denote the setting of the re-evaluation period for C_j, we define Ψ_v as follows:

$$\Psi_v = \sum_{j=1}^{k}\mathcal{V}_j^*(P_j)$$

Assume that the throttle fraction z is given, which defines the fraction of query load to keep. The detail for computation of z will be described in Section 6.3. Under this assumption, the one-time re-evaluation cost of queries within C_j is given by $f_c(C_j)$ and since these queries are re-evaluated every P_j seconds, the overall cost is given by $f_c(C_j)/P_j$. As a result, the load under a given set of re-evaluation periods $\{P_j\}$ is $\sum_{j=1}^{k} f_c(C_j)/P_j$, which should be less than or equal to the throttle fraction times the load of the ideal case of $\forall_{q\in Q}, \tau_q = \tau_\vdash$, which is given by $z \cdot \sum_{q\in Q} f_c(\{q\})/\tau_\vdash$. In summary, the query load shedding algorithm should respect z as the query re-evaluation budget, while maximizing the freshness in the query results. This can be modeled by the following processing constraint:

$$\sum_{j=1}^{k} f_c(C_j)/P_j \leq z \cdot \sum_{q\in Q} f_c(\{q\})/\tau_\vdash$$
$$\forall_{j\in[1..k]},\ \tau_\vdash \leq P_i \leq \tau_\dashv$$

The second constraint defines the scope of the re-evaluation period P_j ($j \in [1..k]$). The key problem here is to define the set of query groups, so as to maximize Ψ_v.

4.2 Measuring Quality Loss Per Unit Cost

The first question for clustering queries is to find which metric should be used as a distance measure to define similar queries. One intuitive observation is that two queries are similar for the purpose of query load shedding if the amount of reduction in quality per unit decrease in cost is similar for the two queries. We call this measure ***quality loss per unit cost*** (*qlpc*) metric. Let $G(q,z)$ denote the quality loss per unit cost for a given query q and a given throttle fraction z. We define $G(q,z)$ using the following formula:

$$G(q,z) = \frac{\alpha_q \cdot \frac{d(\mathcal{V}_q(\tau))}{d\tau}\Big|_{\tau=\tau_\vdash/z}}{f_c(\{q\}) \cdot \frac{d(1/\tau)}{d\tau}\Big|_{\tau=\tau_\vdash/z}} \quad (5)$$

[1]For Equation 3 to hold, we should be able to write $\mathcal{U}_q(\epsilon_q) = \mathcal{U}_q(\sum_{i,m_q(i)>0} m_q(i)\cdot\Delta_i) = \sum_{i,m_q(i)>0} m_q(i)\cdot\mathcal{U}_q(\Delta_i)$. This can be done if and only if $\mathcal{U}_q$ is a linear function.

Note that $\mathcal{V}_q(\tau)$ is the freshness-related QoS function associated with q, whereas $f_c(\{q\})/\tau$ is the cost function of q. Setting the re-evaluation period to $\tau = \tau_{\vdash}/z$ reduces the overall cost of re-evaluating q to z times the cost for the ideal case of $\tau = \tau_{\vdash}$. Since queries within the same group will share the same re-evaluation period, Equation 5 captures the quality loss per unit cost for a $d\tau$ increase in the re-evaluation period.

Clearly, a query q with a small $G(q,z)$ value is a good choice for shedding the query load, as it brings a small loss in QoS for a large amount of decrease in load. Therefore, if two queries have similar $\mathcal{V}_q$ functions, then the one with the larger evaluation cost $f_c(\{q\})$ will be preferred for load shedding. However, if two queries have similar $f_c(\{q\})$ values, then the query with the smaller (absolute) derivative of its $\mathcal{V}_q$ function will be preferred for load shedding. Note that the derivative of the QoS function $\mathcal{V}_q$ is constant over each linear segment and thus Equation 5 can be simplified as follows, where $\mathcal{V}^a_q(i)$ denotes the slope of the ith ($i \in [1..\kappa]$) linear segment of $\mathcal{V}_q$:

$$G(q,z) = \frac{\alpha_q \cdot \mathcal{V}^a_q\left(\lceil \kappa \cdot \frac{\tau_{\vdash}/z - \tau_{\vdash}}{\tau_{\dashv} - \tau_{\vdash}} \rceil\right)}{-f_c(\{q\}) \cdot (z/\tau_{\vdash})^2} \qquad (6)$$

4.3 Grouping Queries with Qlbc

MobiQual uses the quality loss per unit cost (*qlpc*) metric to define the similarity of queries and the distance function used for clustering queries into desirable query groups in terms of load shedding effectiveness. We call this algorithm the Q*uality **l**oss **b**ased **c**lustering* algorithm, Qlbc for short.

It is obvious that putting queries that have diverse $G(q,z)$ values into the same group is very ineffective, because queries with larger $G(q,z)$ values are not good candidates for query load shedding compared to others. Hence there will be less overall benefit from increasing the common re-evaluation period. The Qlbc algorithm finds the similarity between two queries q_1 and q_2 in two steps. First, it models the quality loss per unit cost of each query at different z values using a *qlpc vector*, where each element of the vector corresponds to the $G(q,z)$ value at a different load shedding level z (κ different levels equally spaced between 0 and 1). Second, the Qlbc algorithm uses the Euclidean distance between the *qlpc* vectors of queries to define the similarity of queries. This similarity, denoted by $D(q_1,q_2)$, is defined as follows:

$$D(q_1,q_2) = \sum_{\iota \in ([1..\kappa]-0.5)/\kappa} \left(G(q_1,\iota) - G(q_2,\iota)\right)^2 \qquad (7)$$

The Qlbc algorithm uses k-means clustering [8] to form the final k set of query groups, based on Equation 7.

5. QOS-AWARE UPDATE LOAD SHEDDING

In this section we describe the QoS-aware update load shedding problem, by only considering the accuracy aspect of the quality and the cost of position update processing.

5.1 Formalization of the Problem

The goal of the update load shedding problem is to maximize the second component of the overall quality from Equation 4, denoted by Ψ_u. Given l regions of the geographical space of interest, recalling that $\mathcal{U}^*_i(\Delta_i)$ denote the aggregation function for region A_i, and Δ_i denote the inaccuracy threshold associated with A_i, we define Ψ_u as follows:

$$\Psi_u = \sum_{i=1}^{l} \mathcal{U}^*_i(\Delta_i)$$

Assume that the throttle fraction z is given, which defines the fraction of update load to keep. The number of updates and thus the relative cost of update processing for a given region A_i are proportional to $n_i \cdot f_r(\Delta_i)$. As a result, the load under a given set of inaccuracy thresholds $\{\Delta_i\}$ can be computed by $\sum_{i=1}^{l} n_i \cdot f_r(\Delta_i)$. This load should be less than or equal to the throttle fraction times the load of the ideal case of $\forall_{i\in[1..l]}, \Delta_i = \epsilon_{\vdash}$, which is given by $z \cdot n \cdot f_r(\epsilon_{\vdash})$. Thus, the following processing constraints must hold for the update load shedding problem:

$$\sum_{i=1}^{l} n_i \cdot f_r(\Delta_i) \leq z \cdot n \cdot f_r(\epsilon_{\vdash})$$
$$\forall_{i\in[1..l]},\ \epsilon_{\vdash} \leq \Delta_i \leq \epsilon_{\dashv}$$

The second constraint defines the domain of the inaccuracy threshold Δ_i ($i \in [1..l]$). The key question here is how to partition the space of interest into a number of regions such that the overall quality Ψ_u is maximized. To perform this we use an extension of the previously proposed GridReduce algorithm [4]. Our extensions are targeted toward building QoS-awareness into the algorithm. Here we briefly describe this extension.

5.2 The Space Partitioning Algorithm

The goal of the space partitioning algorithm is to partition the geographical space of interest into l shedding regions, such that this partitioning produces query results of higher accuracy. Concretely, the algorithm first builds a partition hierarchy over the space by constructing a quad-tree, where each tree node corresponds to a different region in the space. The partition hierarchy contains a single region at the top level and becomes more fine-grained as we go down in the hierarchy. This is achieved by dividing each partition into four quadrant partitions at the next lower level. Each level of the quad-tree is a uniform and non-overlapping partitioning of the entire space. In order to capture the QoS characteristics of each partition, we aggregate the QoS functions for each partition in the hierarchy through a post-order traversal of the quad-tree. Once the hierarchy is constructed and the QoS functions are aggregated for each partition, the selection of the l regions follows a top-down process. The algorithm starts from the top-most partition in the hierarchy. At each step one partition is picked and is replaced by its four quadrants. This process continues until l regions of possibly different sizes are reached. The criterion used to decide which partition to pick at each step forms the crux of the algorithm. With the QoS functions at hand, we compute how beneficial it is to divide a partition into its quadrants by solving the small scale version of the QoS-aware update load shedding problem we formalized in the previous subsection by restricting it to the four quadrants at hand. The partition that provides the highest gain in terms of the QoS measure Ψ_u is picked for further partitioning.

6. PUTTING IT ALL TOGETHER: THE MOBIQUAL SOLUTION

In this section we first formalize the problem of combining QoS-aware update load shedding and QoS-aware query load shedding. Then we present a fast greedy algorithm called the M*inimum **q**uality **l**oss per cost **s**tep* (MQLS) to configure the re-evaluation periods $P_j, j \in [1..k]$ and the result inaccuracy thresholds $\Delta_i, i \in [1..l]$ within the same framework, aiming at achieving high overall QoS and better satisfying the freshness and accuracy requirements of mobile location queries. Finally, we describe how to set the throttle fraction z using a feedback-based adaptive mechanism.

6.1 Problem Formalization

The objective of the combined load shedding problem is to maximize the overall quality $\Psi = \frac{1}{m}(\Psi_v + \Psi_u)$ given in Equation 4. We now restate the processing constraint by combining the load due to query re-evaluation and update processing.

Let z_v denote the fraction of the query load retained for a given set of re-evaluation periods $\{P_j\}$. We have: $z_v = \frac{\sum_{j=1}^{k} f_c(C_j)/P_j}{\sum_{q \in Q} f_c(\{q\})/\tau_\vdash}$. Similarly, let z_u denote the fraction of the update load retained for a given set of inaccuracy thresholds $\{\Delta_i\}$. We have: $z_u = \frac{\sum_{i=1}^{l} n_i \cdot f_r(\Delta_i)}{n \cdot f_r(\epsilon_\vdash)}$. With these definitions, we can state the processing constraint as follows:

$$z_v + z_u \cdot \gamma \leq z \cdot (1 + \gamma) \tag{8}$$

The parameter γ in Equation 8 represents the cost of performing update processing with the setting of $\forall_i, \Delta_i = \epsilon_\vdash$ compared to the cost of performing query re-evaluation with the setting of $\forall_j, P_j = \tau_\vdash$. In other words, for the ideal case the query re-evaluations costs 1 unit, whereas the update processing costs $\gamma \in (0, \infty]$ units. Note that γ is *not* a system specified parameter and is learned adaptively as follows. Let U be the observed cost of update processing and V be the observed cost of query re-evaluation during the last adaptation period. Then we have $\gamma = \frac{U/z_u}{V/z_v}$. This assumes that the workload does not significantly change within the time frame of the adaptation period. Recall that the load shedding parameters are configured after each adaptation period, thus yielding new values for z_u and z_v (by way of changing P_j's and Δ_i's). Thus the combined load shedding problem is formalized as follows:

$$\boxed{\begin{array}{l} \text{maximize } \Psi = \frac{1}{m}\left(\sum_{j=1}^{k} \mathcal{V}_j^*(P_j) + \sum_{i=1}^{l} \mathcal{U}_i^*(\Delta_i)\right) \\ \text{subject to} \\ \qquad \frac{\sum_{j=1}^{k} f_c(C_j)/P_j}{\sum_{j=1}^{k} f_c(C_j)/\tau_\vdash} + \gamma \cdot \frac{\sum_{i=1}^{l} n_i \cdot f_r(\Delta_i)}{n \cdot f_r(\epsilon_\vdash)} \leq z \cdot (1+\gamma) \\ \qquad \forall_{j \in [1..k]},\ \tau_\vdash \leq P_i \leq \tau_\dashv,\ \forall_{i \in [1..l]},\ \epsilon_\vdash \leq \Delta_i \leq \epsilon_\dashv \end{array}}$$

Note that this is a non-linear program, since the constraints have $1/P_j$ terms and are not linear. We now describe MQLS – a fast, greedy algorithm for setting the re-evaluation periods and inaccuracy thresholds to solve the above stated QoS-aware load shedding problem.

6.2 The MQLS Algorithm

The basic principle of the MQLS algorithm is to start with the ideal case of $\forall_j, P_j = \tau_\vdash$ and $\forall_i, \Delta_i = \epsilon_\vdash$ and incrementally reduce the load to z times that of the ideal case by repetitively increasing the re-evaluation period or the inaccuracy threshold that gives the smallest quality loss per unit cost reduction. The algorithm is greedy in nature, since it takes the minimum quality loss per cost step. Concretely, we partition the domain of re-evaluation periods and inaccuracy thresholds into β segments, such that we increase the P_j's and Δ_i's in increments of size $c_v = (\tau_\dashv - \tau_\vdash)/\beta$ and $c_u = (\epsilon_\dashv - \epsilon_\vdash)/\beta$, respectively. The MQLS algorithm maintains a min. heap that stores a *qlpc* (quality loss per unit cost[2]) value for each re-evaluation period and each inaccuracy threshold. The *qlpc* value of a re-evaluation period (or an inaccuracy threshold) gives the quality loss per unit cost for increasing it by c_v units (or c_u units). The *qlpc* value is denoted by S_j^v for query group C_j and S_i^u for region A_i. We have:

$$S_j^v = \sum_{q \in Q} f_c(\{q\}) \cdot \frac{\mathcal{V}_j^*(P_j + c_v) - \mathcal{V}_j^*(P_j)}{f_c(C_j) \cdot (\frac{1}{P_j + c_v} - \frac{1}{P_j})} \tag{9}$$

$$S_i^u = \gamma \cdot n \cdot f_r(\epsilon_\vdash) \cdot \frac{\mathcal{U}_i^*(\Delta_i + c_u) - \mathcal{U}_i^*(\Delta_i)}{n_i \cdot (f_r(\Delta_i + c_u) - f_r(\Delta_i))} \tag{10}$$

The nominators of the second components in the above equations represent the changes in the quality due to the increment, whereas the denominators represent the changes in the cost. Note that the first components of the above equations are used to normalize the costs in the denominators of the second components, so that S_j^v's and S_i^u's can be compared.

When the MQLS algorithm starts, the current load expenditure of the system, which is the sum of the load due to update and query load shedding appropriately weighted by γ, is above our load budget imposed by the throttle fraction z. The algorithm iteratively pops the topmost element of the min. heap and depending on whether we have a re-evaluation period or inaccuracy threshold makes the increment using either c_v or c_u. The *qlpc* value of the popped element is updated based on Equation 9 (or Equation 10) and is put back into the heap unless no further increments are possible. The algorithm runs until the load expenditure of the system is within the budget or all the re-evaluation periods and inaccuracy thresholds hit their maximum value. In the latter case the load cannot be shed to meet the processing constraint and random dropping of incoming updates as well as delay in query re-evaluations will unavoidably take place. The pseudo-code of MQLS is given in Algorithm 1.

The total number of greedy steps the algorithm can take is given by $\beta \cdot (l+k)$, which happens when all re-evaluation periods and inaccuracy thresholds have to be increased to their maximum values. Each greedy step takes $\mathcal{O}(\log(l+k))$ time, since the min. heap has $l+k$ elements and the heap operations used take logarithmic time on the heap size. The final time complexity of the MQLS algorithm directly follows as $\mathcal{O}(\beta \cdot (l+k) \cdot \log(l+k))$ and the space complexity as $\mathcal{O}(l+k)$.

6.3 Setting the Throttle Fraction

We set the throttle fraction adaptively based on feedback with regard to how well the system is performing in terms shedding the correct amount of load. When the throttle fraction z is larger than what it should be, the system will not be able to re-evaluate all queries at all of their re-evaluation points and/or will not able to admit all position updates into the system. Let σ_v represent the fraction of query load imposed by the set of re-evaluation periods that was actu-

[2]This is *qlpc* for a query group or for a region, and not for a query as it was first introduced in Section 4.2. The core concept is the same.

Algorithm 1: The MQLS Algorithm

```
Input: $z$: throttle fraction; $c_v$: period incr.; $c_u$: threshold incr.
Output: $P_j, j \in [1..k]$: periods; $\Delta_i, i \in [1..l]$: thresholds
GREEDYINCREMENT($z$, $c_v$, $c_u$)
(1)  $H$: empty, min heap of $S^v_j$'s and $S^u_i$'s
(2)  $V \leftarrow \sum_{q \in Q} f_c(\{q\})/\tau_{\vdash}$, $V_{\dashv} \leftarrow z \cdot V$ {query expend., budget}
(3)  $U \leftarrow n \cdot f_r(\epsilon_{\vdash})$, $U_{\dashv} \leftarrow z \cdot U$ {update expend., budget}
(4)  for $j = 1$ to $k$ {init. $S^v_j$'s, add to $H$}
(5)      $S^v_j \leftarrow \frac{\mathcal{V}^*_j(\tau_{\vdash}+c_v)-\mathcal{V}^*_j(\tau_{\vdash})}{f_c(C_j)\cdot\left(\frac{1}{\tau_{\vdash}+c_v}-\frac{1}{\tau_{\vdash}}\right)}$ {initial query qlpc}
(6)      $S^v_j \leftarrow S^v_j / V_{\dashv}$ {normalize}
(7)      $P_j \leftarrow \tau_{\vdash}$, $H$.INSERT($S^v_j$) {add query qlpc}
(8)  for $i = 1$ to $l$ {init. $S^u_j$'s, add to $H$}
(9)      $S^u_i \leftarrow \frac{\mathcal{U}^*_i(\epsilon_{\vdash}+c_u)-\mathcal{U}^*_i(\epsilon_{\vdash})}{n_i\cdot(f_r(\epsilon_{\vdash}+c_u)-f_r(\epsilon_{\vdash}))}$ {initial update gain}
(10)     $S^u_i \leftarrow \gamma^{-1} \cdot S^u_i / U_{\dashv}$ {normalize}
(11)     $\Delta_i \leftarrow \epsilon_{\vdash}$, $H$.INSERT($S^u_i$) {add update gain}
(12) repeat {start increment loop}
(13)     $S \leftarrow H$.POPMAX() {next $P_j$ or $\Delta_i$ to incr.}
(14)     if $S$ is for a period, $S = S^v_j$
(15)         $V \leftarrow V - \frac{f_c(C_j)}{P_j} + \frac{f_c(C_j)}{P_j+c_v}$ {query expend.}
(16)         $P_j \leftarrow P_j + c_v$ {increment $P_j$}
(17)         if $P_j \leq \tau_{\dashv}$ {further incr. possible}
(18)             $S^v_j \leftarrow \frac{\mathcal{V}^*_j(P_j+c_v)-\mathcal{V}^*_j(P_j)}{f_c(C_j)\cdot\left(\frac{1}{P_j+c_v}-\frac{1}{P_j}\right)}$ {new query qlpc}
(19)             $S^v_j \leftarrow S^v_j / V_{\dashv}$ {normalize}
(20)             $H$.INSERT($S^v_j$) {insert the query qlpc}
(21)     else if $S$ is for a threshold, $S = S^u_i$
(22)         $U \leftarrow U - n_i \cdot f_r(\Delta_i) + n_i \cdot f_r(\Delta_i + c_u)$ {update expend.}
(23)         $\Delta_i \leftarrow \Delta_i + c_u$ {increment $\Delta_i$}
(24)         if $\Delta_i \leq \epsilon_{\dashv}$ {further incr. possible}
(25)             $S^u_i \leftarrow \frac{\mathcal{U}^*_i(\Delta_i+c_u)-\mathcal{U}^*_i(\Delta_i)}{n_i\cdot(f_r(\Delta_i+c_u)-f_r(\Delta_i))}$ {new update qlpc}
(26)             $S^u_i \leftarrow \gamma^{-1} \cdot S^u_i / U_{\dashv}$ {normalize}
(27)             $H$.INSERT($S^u_i$) {insert the update qlpc}
(28) until $V + \gamma \cdot U \leq V_{\dashv} + \gamma \cdot U_{\dashv}$ {budget reached}
             or $H$.SIZE() $= 0$ {all period and thresholds maxed}
```

ally handled with respect to query processing. This can be calculated by observing the number of query re-evaluations performed and skipped during the last adaptation period, appropriately weighted by query costs. Similarly, let σ_u represent the fraction of update load imposed by the set of inaccuracy thresholds that was actually handled with respect to update processing. This can be calculated by observing the number of updates admitted and dropped since the last adaptation period. Once σ_v and σ_u are computed, we can capture the performance of the system in handling the amount of load imposed by the current throttle fraction z as follows:

$$\phi = \frac{z_v \cdot \sigma_v + \gamma \cdot z_u \cdot \sigma_u}{z \cdot (1 + \gamma)} \tag{11}$$

The denominator of Equation 11 is the amount of load the system was supposed to handle and the nominator is the actual amount of load that was handled. In order to take into account the cases where z is lower than what it should ideally be, we also consider the utilization of the system, μ. When we have an overshot z the utilization of the system will be 1, whereas it would be less that 1 when we have an undershot z since the system would be idle at times not processing any queries or updates. As a result we adjust z as follows for the two cases:

$$z \leftarrow \begin{cases} z \cdot \phi & \mu = 1 \\ min(1, z/\mu) & \mu < 1 \end{cases} \tag{12}$$

This concludes our description of the MobiQual system.

7. EXPERIMENTAL EVALUATION

In this section we compare the performance of the MobiQual system to a number of other alternatives. These include:

– **Query-only load shedding** refers to QoS-aware differentiated load shedding with respect to re-evaluation periods only (see Section 4) and uses a fixed inaccuracy threshold of $\epsilon_{\vdash}$.

– **Update-only load shedding** refers to QoS-aware differentiated load shedding with respect to inaccuracy thresholds only (see Section 5) and can be seen as the QoS-aware extension of the Lira approach [4]. Thus we name it as Lira$^+$.

– **Single Δ-P** refers to combined QoS-aware query and update load shedding, but without query grouping (QLBC algorithm from Section 4.3) and space partitioning (extended GRIDREDUCE algorithm from Section 5.2). It represents a special case of the MobiQual system with $k = l = 1$.

We evaluate the MobiQual system using four main evaluation metrics. These include:

i) The overall quality metric Ψ, as defined by Equation 4.

ii) The mean period delay D, which is defined as the average difference between the ideal case period $\tau_{\vdash}$ and the assigned period of queries, $\tau_q = P_j$ for $q \in C_j$. The mean period delay is formulated as:
$D = \frac{1}{m} \sum_{q \in Q} (\tau_q - \tau_{\vdash})$

iii) The mean position error R, which is defined as the average error in the positions of the mobile nodes within query results, relative to the error for the ideal case of $\forall_{i \in [1..l]}\, \Delta_i = \epsilon_{\vdash}$. It is formulated as:
$R = \frac{1}{m} \sum_{q \in Q} (\epsilon_q - \epsilon_{\vdash})$

iv) The running time of the adaptation step, which includes configuring a new set of re-evaluation periods and in-accuracy thresholds using the MQLS algorithm.

7.1 Experimental Setup

To create the mobile node movement trace used in the experiments, we used a real-world road map from the Chamblee region of the state of Georgia, USA. The trace covers a region of around 200km^2 and part of it is shown in Figure 4. We used real-world traffic volume data at the granularity of specific road types (such as expressway, arterial, collector), taken from Gruteser and Grunwald [7], to simulate cars going on roads. The trace contains around 15K mobile nodes. The default re-evaluation period range used for the experiments is $[\tau_{\vdash}, \tau_{\dashv}] = [1, 10]$ seconds, whereas the inaccuracy threshold range used is $[\Delta_{\vdash}, \Delta_{\dashv}] = [5, 100]$ meters. The number of regions used for partitioning is set as $l = 250$ (see [4]). The increments used by the MQLS algorithm are determined using $\beta = 100$, i.e., the maximum number of increments possible is 100 for each re-evaluation period and inaccuracy threshold. The queries used in the experiments are range queries. The query distribution is proportional to the object distribution. Inverse and random distributions

Figure 4: The road map used in the experiments, Chamblee, GA, USA

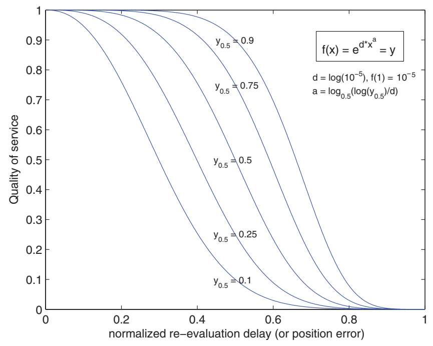

Figure 5: Example QoS functions, with different mid-point QoS values ($y_{0.5}$)

were also used, with similar results. The query side lengths were randomly chosen from the range $[0, 1000]$ meters.

A number of system and workload parameters were varied in the course of the experiments to understand their impact on the query result quality and running-time performance of the MobiQual system. These include the number of query groups used, i.e., the k parameter used by the QLBC algorithm (default value: 16), the number of queries to number of objects ratio (default value: 0.01), the emulated capacity of the system (default: $z = 0.5$), and the QoS functions specified by the queries. Figure 5 gives the general template of the QoS functions that were used for both V_q (freshness) and U_q (accuracy) components of a query q's QoS specification function S_q, whereas the α value that adjusts the relative importance of the freshness and accuracy components of quality were chosen at random from the range $[0, 1]$. The QoS functions were approximated by 10 linear segments and a parameter called *mid-point QoS threshold* was used to pick a random V_q or U_q component from the set of available functions, a subset of which is shown in Figure 5. Any given V_q or U_q is chosen by randomly picking a number, say $y_{0.5}$, between 0 and the mid-point quality threshold, and determining the QoS function whose value for the mid-point of its domain is equal to $y_{0.5}$ (and matching the template given in Figure 5).

7.2 Experimental Results

We divide the experimental results into three parts. The first part deals with the impact of the amount of load to be shed on the query result quality. The second part deals with the performance of MobiQual under different query loads and the impact of the number of query groups on the query result quality as well as on the time it takes to perform the adaptation step. The third part deals with the impact of the QoS specifications on the performance of MobiQual.

7.2.1 *Impact of the Throttle Fraction*

The graphs in Figure 6 plot the overall quality of the query results as a function of the throttle fraction (i.e., at different load shedding levels) for the competing approaches. At a given load shedding level, if $1 - z$ fraction of the load cannot be shed by a load shedding algorithm, then the QoS value is not plotted for that z value, and for smaller z values thereof. For instance, we observe from Figure 6 that, for the default settings, the query-only approach can only support load shedding for $z \geq 0.7$ and Lira$^+$ for $z \geq 0.5$, whereas MobiQual and Single Δ-P can support $z \geq 0.2$. MobiQual significantly outperforms update-only and query-only load shedding schemes, as it is observed from the rapidly declining QoS values of the latter two approaches with decreasing z. Furthermore, MobiQual outperforms Single Δ-P, for a wide range of z values. While shedding 60% ($z = 0.4$) of the load, MobiQual is able to keep the QoS around $\Psi = 0.9$, whereas this value is only around 0.75 for the Single Δ-P approach. Similarly, MobiQual manages to sustain a QoS value of $\Psi = 0.7$ for 70% load shedding, compared to a mere 0.4 for Single Δ-P. The two approaches both hit the $\Psi = 0$ boundary when MobiQual is forced to set all query re-evaluation periods and inaccuracy thresholds to their maximum value, at which point there is no difference between the two approaches. The superior performance of MobiQual compared to Single Δ-P illustrates the strength of the differentiated load shedding concept, whereas the poor performances of update-only and query-only load shedding attest to the importance of performing combined quary and update load shedding.

The graphs in Figures 7 and 8 plot the mean period delay and mean position error as a function of the throttle fraction for competing approaches, respectively. Note that the query-only approach has zero mean position error (as observed from Figure 8), whereas the update-only approach has zero mean period delay (as observed from Figure 7). However, since a good overall quality requires balancing freshness and accuracy, these two approaches do not provide good overall QoS as observed from Figure 6. The mean period delay of Single Δ-P stays slightly above that of MobiQual for $z > 0.3$. After this point Single Δ-P registers lower mean period delays. This is because further increasing the single re-evaluation period has diminishing benefit in terms of the qlpc metric, since Single Δ-P cannot provide differentiated load shedding. In contrast, the MobiQual approach can locate queries that can tolerate further staleness with less impact on the QoS value, due to the QLBC algorithm, and thus can increase the re-evaluation periods further for such queries. Even though this results in higher mean period delay compared to Single Δ-P, it translates into a higher overall QoS due to a better balance between query and update load shedding. It is observed from Figure 8 that MobiQual consistently outperforms Single Δ-P in terms of the mean position error. This is not because MobiQual sheds

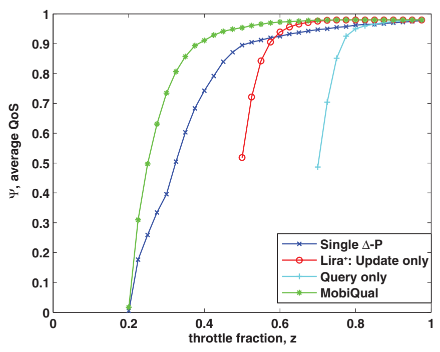

Figure 6: Overall result quality as a function of the throttle fraction

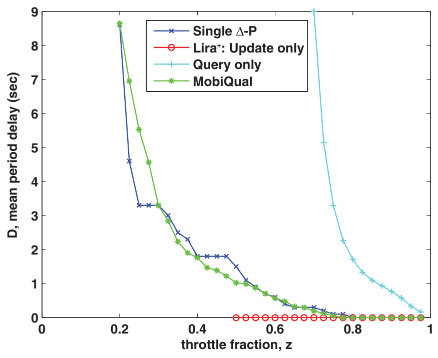

Figure 7: Mean period delay as a function of the throttle fraction

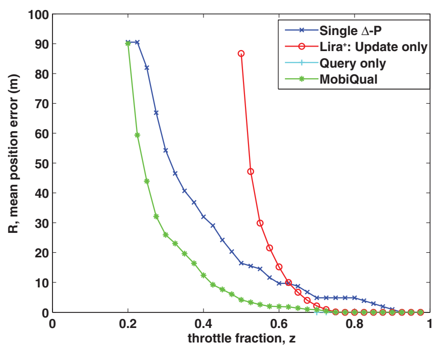

Figure 8: Mean position error as a function of the throttle fraction

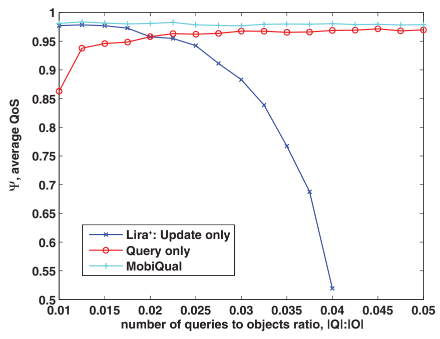

Figure 9: Overall result quality with changing query workload

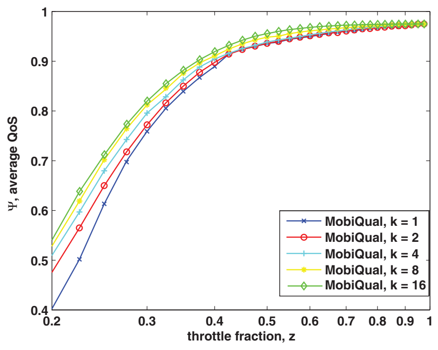

Figure 10: The impact of number of query groups on result quality

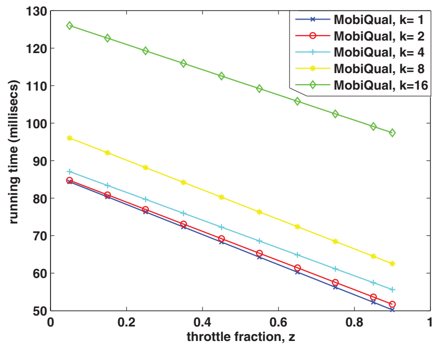

Figure 11: Adaptation time with different number of query groups

less update load, but it is because MobiQual sheds the update load from regions that has lesser impact on the query results, due to the QoS-aware partitioning algorithm it employs.

7.2.2 *Impact of the # of Queries and Query Groups*

The graphs in Figure 9 plot the overall QoS of the query results as a function the number of queries to number of mobile nodes ratio, for MobiQual vs. query-only and update-only (Lira$^+$) load shedding. The throttle fraction is set to 0.75 for this experiment. It is interesting to observe that as the number of queries increase, the update-only load shedding loses its advantage over query-only load shedding. This is because with increasing number of queries, the dominant cost becomes the query re-evaluation, since the full update load does not depend on the number of queries. This shows the importance of performing combined query and update load shedding, which is effective independent of the number of queries or the number of mobile nodes, as evidenced by the superior performance of MobiQual compared to query-only and update-only approaches with changing number of queries to number of mobile nodes ratio (see Figure 9).

An important parameter that impacts the performance of the MobiQual system is the number of query groups, k. As discussed in Section 2.1, in general the higher the number of query groups the more fine grained is the differentiated load shedding. The only limiting factor in increasing the value of k is the time it takes to execute the adaptation step, as the computational complexity of the MQLS algorithm is dependent on k. However, increasing the number of k has diminishing return in terms of the overall QoS, as shown by Figure 10, since the query groups become more and more homogeneous in terms of the QoS functions of the queries contained within. The graphs in Figure 10 plot the overall QoS as a function of the throttle fraction (x-axis is in logarithmic scale) for different k values. This experiment is run for 1000 continual queries. We clearly see from the figure that the gain in QoS when going from $k = 8$ to $k = 16$ is significantly lower than the gain in QoS when going from $k = 1$ to $k = 2$. This shows that having query groups smaller than 50-60 queries does not bring much gain in overall query result quality. Even though small query groups are unnecessary, the MQLS algorithm can support large k values with low overhead. Figure 11 shows that for $k = 16$ and $z = 0.5$ the adaptation step takes around 110 milliseconds. In a mobile CQ system, the change in the workload in terms of the number of CQs and mobile nodes is not spontaneous, and significant shifts in the workload is likely to happen within minutes. Thus the time it takes to run the adaptation step in order to configure the new set of re-evaluation periods and inaccuracy thresholds is really small compared to the adaptation period and results in a very lightweight load shedding scheme.

7.2.3 *Impact of the QoS Specifications*

The graphs in Figures 12 and 13 plot the overall query result quality as a function of the mid-point QoS threshold used for the freshness component of the QoS specifications and the accuracy component of it, respectively. Decreasing

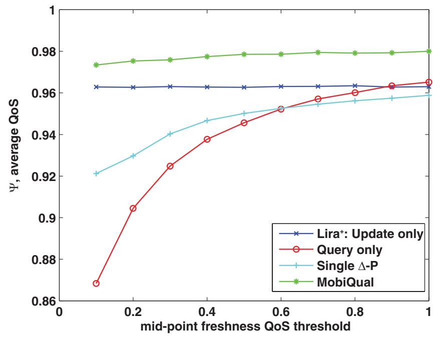

Figure 12: Query result quality for varying freshness QoS specs.

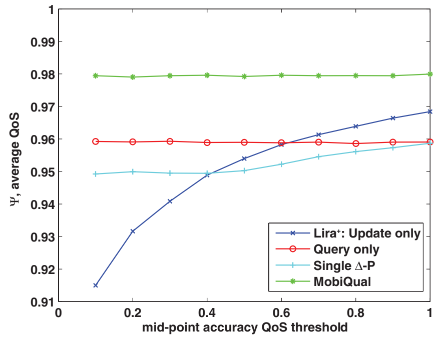

Figure 13: Query result quality for varying accuracy QoS specs.

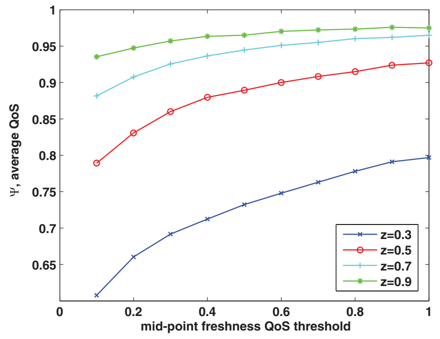

Figure 14: Result quality under changing z & freshness QoS specs.

values along the x-axis represent QoS specifications with increasingly stringent freshness components for Figure 12 and increasingly stringent accuracy components for Figure 13. A high throttle fraction value of 0.75 was used to make sure that all competing approaches can shed the required fraction of the load. Note that update-only load shedding (Lira$^+$) is indifferent to the freshness components of the QoS specifications, whereas query-only load shedding is indifferent to the accuracy components. As a result, the lines for update-only and query-only load shedding are flat in Figures 12 and 13, respectively. We observe from Figure 12 that MobiQual is very robust to changes in the freshness components of the QoS specifications and shows a smaller decrease in overall QoS with increasing intolerance to staleness in QoS specifications, compared to alternative approaches. It provides up to 12% better QoS compared to query-only load shedding and 5% better compared to Single Δ-P. These values are valid for shedding 25% percent of the load. The improvement provided by MobiQual over the closest competitor reaches 80% when shedding 70% of the load, and for a mid-point freshness QoS threshold of 0.75 (see previous Figure 6). The results presented in Figure 13 for the mid-point accuracy QoS threshold are very similar in nature.

In Figures 14 and 15 we further study the sensitivities of the MobiQual system to changes in the QoS specifications of the queries. We do this by looking at the change in the overall QoS of the query results at different levels of load shedding, under changing values of the mid-point QoS thresholds. We observe from Figures 14 that even for a mid-point freshness QoS threshold of 0.1, which implies that $V_q(\frac{\tau_\vdash+\tau_\dashv}{2})$ is always less than 0.1 for a query q, MobiQual is able sustain an overall QoS value of > 0.78 for $z \geq 0.5$ (shedding at most half of the load). Similarly, for a mid-point accuracy QoS threshold of 0.1, which implies that $U_q(\frac{\epsilon_\vdash+\epsilon_\dashv}{2})$ is always less than 0.1 for a query q, MobiQual is able sustain an overall QoS value of > 0.85 for $z \geq 0.5$. The overall QoS sharply drops when shedding more than half of the load, and MobiQual becomes more sensitive to increasing intolerance to staleness and inaccuracy in QoS specifications of queries. This is clearly observed from the increasing gap between the QoS lines in Figures 14 and 15, and their increasing slope with decreasing values of the throttle fraction. Yet, even for shedding 70% of the load at the most stringent configurations of the freshness and accuracy components of the QoS specifications, MobiQual is able to provide an impressive QoS value of > 0.6.

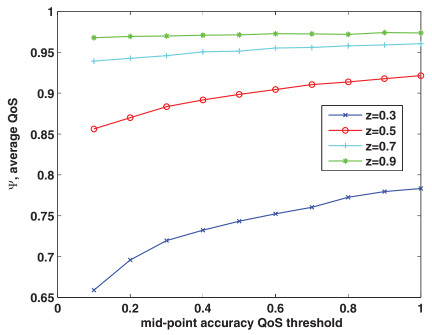

Figure 15: Result quality under changing z and accuracy QoS specs.

8. RELATED WORK

Previous work on mobile CQ systems have focused on four major themes with respect to scalability and performance. These are: i) indexing schemes to process position updates more efficiently [16, 10, 11, 20]; ii) query processing techniques to evaluate continual queries more efficiently [12, 5, 15, 19]; motion modeling techniques to reduce the number of position updates received from the mobile nodes, while keeping the position accuracy high [17, 2]; and iv) distributed mobile CQ systems that achieve scalability by performing query-aware update filtering on the mobile node side to receive updates that only relate to the current set of queries installed in the system [1, 9, 3]. Most of these works, with the exception of the works listed under item iv, are mostly orthogonal to our work and can be incorporated into MobiQual easily. For instance, MobiQual can use a TPR-tree [16] as its underlying index structure on the server side, can make use of advanced motion modeling techniques [17] on the mobile node side, and can employ incremental query processing techniques [5] for query re-evaluation. Unlike the set of works listed under item iv, MobiQual receives updates from all the nodes, so that ad-hoc and historical queries can also be supported. However, MobiQual prefers to shed position updates from regions that have minimal impact on the currently installed queries, thus achieving best of both worlds.

To the best of our knowledge, none of the previous works in the field of mobile CQ systems has addressed the problem of QoS-aware query management. MobiQual addresses this issue by introducing a novel load shedding framework. Note that mobile node movement is not discrete, but continuous. As a result, zero staleness and inaccuracy in the query results is impossible to achieve with finite resources. Thus, a solution is required to adjust the balance between the update processing and query re-evaluation components in mobile CQ systems. Moreover, this balance is dependent on the tolerance of the individual queries to staleness and inaccuracy in the query results. Prior works on mobile CQ systems not only have overlooked the QoS aspect of the problem, but also either have not address how frequent the position updates should be received from the mobile nodes or have not specified how frequent query results should be updated by re-evaluating the queries. However, as we show in this paper, an integrated, QoS-aware approach is essential for achieving high quality query results.

In our previous work [4], we introduced the Lira system for performing update load shedding in mobile CQ systems. However, the work in Lira not only misses the consideration of the query load shedding and its impact on update load shedding, but also ignores any QoS-aware mechanisms as a part of the load shedding framework. Thus the Lira results are limited. In contract, MobiQual provides a general QoS-aware load shedding framework that effectively combine query load shedding with update loading shedding to maximize the freshness and accuracy of the query results. The MobiQual space partitioning algorithm is built on top of the GridReduce algorithm in Lira. Our experiments showed that even a QoS-enhanced version of Lira which uses only QoS-aware update load shedding is significantly inferior to MobiQual.

9. CONCLUSION

In this paper we have presented MobiQual, a load shedding system aimed at providing high quality query results in mobile continual query systems. MobiQual has three unique properties. First, it uses per-query QoS specifications that characterize the tolerance of queries to staleness and inaccuracy in the query results, in order to maximize the overall QoS of the system. Second, it effectively combines query load shedding and update load shedding within the same framework, through the use of differentiated load shedding concept. Finally, the load shedding mechanisms used by MobiQual are lightweight, enabling quick adaption to changes in the workload, in terms of the number of queries, number of mobile nodes or their changing movement patterns. Through a detailed experimental study, we have shown that the MobiQual system significantly outperforms approaches that are based on query-only or update-only load shedding, as well as approaches that do combined query and update load shedding but lack the differentiated load shedding elements of the MobiQual solution, in particular the query grouping and space partitioning mechanisms.

10. ADDITIONAL AUTHORS

11. REFERENCES

[1] Y. Cai and K. A. Hua. Real-time processing of range-monitoring queries in heterogeneous mobile databases. *IEEE Trans. on Mobile Computing*, 5(7):931–942, 2006.

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

[3] B. Gedik and L. Liu. Mobieyes: A distributed location monitoring service using moving location queries. *IEEE Trans. on Mobile Computing*, 5(10):1384–1402, 2006.

[4] B. Gedik, L. Liu, K.-L. Wu, and P. S. Yu. Lira: Lightweight, region-aware load shedding in mobile CQ systems. In *IEEE ICDE*, 2007.

[5] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Processing moving queries over moving objects using motion adaptive indexes. *IEEE Trans. on Knowledge and Data Engineering*, 18(5):651–668, 2006.

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

[7] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *ACM MobiSys*, 2003.

[8] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, August 2000.

[9] H. Hu, J. Xu, and D. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *ACM SIGMOD*, 2005.

[10] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *VLDB*, 2004.

[11] 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 *VLDB*, 2003.

[12] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *ACM SIGMOD*, 2004.

[13] NextBus web page. http://www.nextbus.com/, January 2004.

[14] S. Pandey, K. Dhamdhere, and C. Olston. WIC: A general-purpose algorithm for monitoring web information sources. In *VLDB*, 2004.

[15] 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 Trans. on Computers*, 51(10):1124–1140, 2002.

[16] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD*, 2000.

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

[18] K.-L. Wu, S.-K. Chen, and P. S. Yu. Efficient processing of continual range queries for location-aware services. *Information Systems Frontiers*, 7:435–448, 2005.

[19] K.-L. Wu, S.-K. Chen, and P. S. Yu. Incremental processing of continual range queries over moving objects. *IEEE Trans. on Knowledge and Data Engineering*, 18(11):1560–1575, 2006.

[20] X. Xiong and W. G. Aref. R-trees with update memos. In *IEEE ICDE*, 2006.