Load Shedding in Mobile Systems with MobiQual

Buğra Gedik, *Member, IEEE*, Kun-Lung Wu, *Fellow, IEEE*,
Ling Liu, *Senior Member, IEEE*, Philip S. Yu, *Fellow, IEEE*

***Abstract*—In location-based, mobile continual query (CQ) systems, two key measures of quality of service (QoS) are: *freshness* and *accuracy*. To achieve freshness, the CQ server must perform frequent query re-evaluations. To attain accuracy, the CQ server must receive and process frequent position updates from the mobile nodes. However, it is often difficult to obtain fresh and accurate CQ results simultaneously, due to (a) limited resources in computing and communication and (b) fast-changing load conditions caused by continuous mobile node movement. Hence, a key challenge for a mobile CQ system is: How do we achieve the highest possible quality of the CQ results, in both freshness and accuracy, with currently available resources? In this paper, we formulate this problem as a *load shedding* one, and develop *MobiQual* – a QoS-aware approach to performing both update load shedding and query load shedding. The design of MobiQual highlights three important features. (1) *Differentiated load shedding*: We apply different amounts of query load shedding and update load shedding to different groups of queries and mobile nodes, respectively. (2) *Per-query QoS specification*: Individualized QoS specifications are used to maximize the overall freshness and accuracy of the query results. (3) *Low-cost adaptation*: MobiQual dynamically adapts, with a minimal overhead, to changing load conditions and available resources. We conduct a set of comprehensive experiments to evaluate the effectiveness of MobiQual. The results show that, through a careful combination of update and query load shedding, the MobiQual approach leads to much higher freshness and accuracy in the query results in all cases, compared to existing approaches that lack the QoS-awareness properties of MobiQual, as well as the solutions that perform query-only or update-only load shedding.**

***Index Terms*—Location-based systems, Query processing, Load Shedding**

I. Introduction

Today, we are experiencing a world where we can stay connected while on-the-go because there are (1) myriads of affordable mobile devices and (2) ever increasing accessibility of wireless communications for these devices. Combined with the availability of low-cost positioning devices, like GPS, this has created a new class of applications 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 [1], Google ride finder [2]), fleet management, mobile games, and battlefield coordination.

A key challenge for LBSs is: How to design a scalable location monitoring system capable of handling a large number of mobile nodes and processing complex queries over their positions? Although several mobile continual query (CQ) systems have been proposed to handle long-running location monitoring tasks in a scalable manner [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], the focus of these works is primarily on efficient indexing and query processing techniques, not on the accuracy or freshness of the query results.

Accuracy (inaccuracy) is measured based on the amount of mobile node position errors found in the query results *at the time of* query re-evaluation. This accuracy measure is strongly tied to the frequency of position updates received from the mobile nodes. Although one can also use a higher level concept to measure accuracy, such as the amount of containment errors found in the query results[1], including both false positives (inclusion errors) and false negatives (exclusion errors), we argue that using position update errors for accuracy measure will provide a higher level of precision. This is primarily because by utilizing the amount of node position errors as the accuracy measure, one can easily bound the inaccuracy by a threshold-based position reporting scheme [3], [14]. Note that certain applications have higher tolerance to inaccuracy in position updates, such as region-based traffic density monitoring; whereas certain others require higher accuracy, such as path-based location tracking.

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 post-query-re-evaluation deviations are not attributed to inaccuracy. Hence, freshness can be seen as a metric capturing these deviations. It is important to note that higher freshness does not necessarily imply higher accuracy and vice versa. 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 [15]. Note that certain applications have higher tolerance to staleness in query results, such as monitoring slowly changing temporal aggregates like traffic rates; whereas certain others require higher freshness, such as location based alerts and triggers.

Accuracy and freshness are not completely independent of each other in the sense that both reflect the uncertainty about the precise locations of the moving objects at a given time. However, we use accuracy and freshness separately to capture the impact of such uncertainty from two independent perspectives. Accuracy is used to capture the effect of uncertainty due to location-update frequency on the mobile devices, while freshness is used to capture the effect of uncertainty due to query re-evaluation frequency on the server.

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

• *B. Gedik and K-L. Wu are with the IBM T.J. Watson Research Center, 19 Skyline Dr., Hawthorne, NY 10532. E-mail: {bgedik,klwu,psyu}@us.ibm.com.*
• *L. Liu is with the College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332. E-mail: lingliu@cc.gatech.edu.*
• *P. S. Yu is with the Computer Science Department, University of Illinois, 851 S. Morgan Street, Chicago, IL 60607. E-mail: psyu@cs.uic.edu.*

[1]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 or can be approximated.

mobile nodes at 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 therefore is: How do we achieve the highest possible quality of the query results in both freshness and accuracy, in the presence of changing availability of resources and changing workloads of location updates and location queries?

In this paper, we present MobiQual – 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 (discarding certain location update messages) and query load shedding (skipping some query re-evaluations) to be performed according to the application-level QoS specifications of the queries. An obvious advantage of combining query load shedding and update load shedding within the same framework is to empower MobiQual with *differentiated load shedding* capability, that is configuring *query re-evaluation periods* and *update inaccuracy thresholds* for achieving high overall QoS with respect to both freshness and accuracy.

Another salient feature of MobiQual design is its ability to perform dynamic update load shedding and query load shedding according to changing workload characteristics and resource constraints, and its ability to reduce or avoid severe performance degradation in query result quality under such conditions. MobiQual employs *query grouping* and *space partitioning* techniques to reduce the adaptation time required for re-configuring the system in response to high system dynamics, such as the number of queries, the number of mobile nodes, and the evolving movement patterns. To the best of our knowledge, none of the existing work has exploited the potential of performing load shedding to maximize the application-level freshness and accuracy of mobile queries. In contrast to existing work on scalable query processing and indexing techniques, MobiQual provides a QoS-aware framework for performing both update load shedding and query load shedding, in order to provide highly accurate and fresh query results, even under limited resources or overload conditions. Moreover, as a complementary solution, MobiQual can easily take advantage of existing query processing and indexing techniques.

We have conducted detailed experimental studies on the effectiveness of MobiQual. Our results show that (1) a careful combination of location update load shedding and location query load shedding can significantly outperform the approaches that are based on query-only or update-only load shedding; and (2) MobiQual provides higher quality guarantees compared to the approaches that lack the supports of QoS awareness and differentiated load shedding.

A preliminary version of the MobiQual framework was described in [16]. In the current paper, we have substantially expanded the MobiQual framework by providing (i) a complete description of QoS-aware update load shedding in Section VI, which includes the GRIDREDUCE algorithm for performing space partitioning VI-B; (ii) several additional sets of experiments in Section IX, evaluating a MobiQual-Light scheme that focuses on update load shedding; and (iii) a revised performance comparison of MobilQual with various schemes in Section X.

II. Related Work

Previous work on mobile CQ systems have focused on roughly five major categories with respect to scalability and performance. These are: (a) indexing schemes to process position updates more efficiently [4], [6], [8], [7], [12]; (b) query processing techniques to evaluate continual queries more efficiently [9], [13], [5], [11], [17]; (c) motion modeling techniques to reduce the number of position updates received from the mobile nodes, while keeping the position accuracy high [3], [14]; (d) load shedding approaches that achieve scalability on the server side by only processing specially defined significant updates [18], [19]; and (e) 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 [20], [10], [21].

The majority of these works, with the exception of the works listed under category (e), are mostly orthogonal to our work. Some of them can be incorporated into MobiQual relatively easily. For instance, MobiQual can use a TPR-tree [4] as its underlying index structure on the server side, can make use of advanced motion modeling techniques [3] on the mobile node side, and can employ incremental query processing techniques [13] for query re-evaluation. Unlike the set of works listed under category (e), 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. Those in category (d) are, to some extent, similar to MobiQual, in terms of shedding load in position updates. However, they use different techniques for load shedding. More importantly, they do not consider query load shedding.

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.[2]

III. Design Overview

A. 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

[2]A preliminary version of this paper appeared in *IEEE ICDE* 2008 [16]. We greatly extended and revised the earlier version by, among others, adding Sections VI-B and IX. Moreover, almost all the experiments in Section X were redone to compare MobiQual with MobiQual-Light.

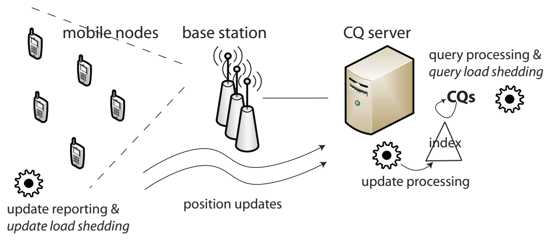

Fig. 1. Mobile CQ system and load shedding

continual range or nearest neighbor queries) over the last known positions of the mobile nodes.[3] Since the mobile node positions change continuously, motion modeling [3], [14] 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 [4], [8]. 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 by 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 current workload with the server's capacity, while meeting the accuracy and freshness requirements of queries.

B. 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 freshness and accuracy of queries are non-uniformly impacted.

[3]In certain CQ systems, a position update can be used to determine which user queries are affected, by running it against an index on queries – hence query re-evaluation is not required. However, such *query-indexing* approaches (like [5]) can easily take advantage of MobiQual. The query partitioning scheme applied in MobiQual can be used on the query index, such that not all location updates are taken through all the queries all the time.

From the perspective of update load shedding, we make two observations to show that non-uniform 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 employ *QoS-aware update load shedding*: We use inaccuracy thresholds from 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 *QoS-aware update load shedding*.

Similar to update load shedding, we make two observations regarding query load shedding to show that non-uniform 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 employ *QoS-aware query load shedding*: We use 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.

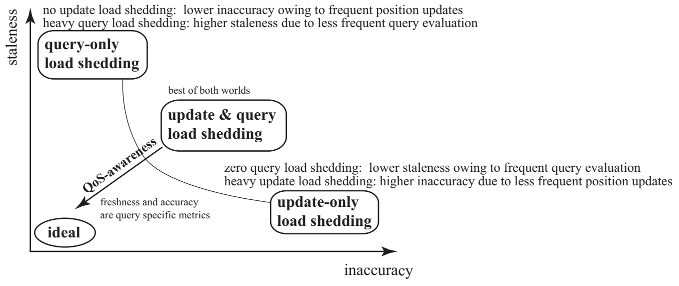
staleness
no update load shedding: lower inaccuracy owing to frequent position updates
heavy query load shedding: higher staleness due to less frequent query evaluation
query-only load shedding
best of both worlds
update & query load shedding
QoS-awareness
zero query load shedding: lower staleness owing 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

Fig. 2. QoS-aware update load shedding and QoS-aware query load shedding

$A_i, i \in [1..l]$	set of shedding regions, l of them
$C_j, j \in [1..k]$	set of query groups, k of them
Δ_i	inaccuracy threshold for nodes in A_i
P_j	re-evaluation period for queries in C_j
f_c	query evaluation cost function
f_r	position update cost function
z	throttle fraction
$q \in Q, m$	set of queries, m of them
$\mathcal{S}_q$	quality of service function for q
τ_q	staleness in query result of q
$\tau_\vdash, \tau_\dashv$	lower and upper staleness bounds
ϵ_q	inaccuracy in query result q
$\epsilon_\vdash, \epsilon_\dashv$	lower and upper inaccuracy bounds
Ψ, Ψ_u, Ψ_v	overall QoS, freshness QoS, accuracy QoS
$\mathcal{V}_q$	freshness component of $\mathcal{S}_q$
$\mathcal{U}_q$	accuracy component of $\mathcal{S}_q$
$\mathcal{V}^*_j$	aggregated freshness QoS for C_j
$\mathcal{U}^*_i$	aggregated accuracy QoS for A_i

TABLE I
MAJOR NOTATION USED IN THE PAPER

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 *QoS-aware 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.

C. Notation and Fundamentals

The set of continual queries installed in the system is denoted by Q. For each query $q \in Q$, there is 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, 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 [3], [22]. 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 ($i \in [1..l]$).

Similarly, 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 ($j \in [1..k]$).

Table I lists the major notation used throughout the paper, some of which are introduced in later sections.

D. 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 X, we experimentally evaluate the benefit/cost trade-off in setting k (see Figs. 15-16) and l (see Figs. 8-10) to show that with lightweight adaptation we can achieve high quality query results.

E. 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 IV). 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 V). 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 VI). 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 VII).

IV. 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:

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

where m is the total number of queries and $\mathcal{S}_q(\tau_q, \epsilon_q)$ denotes the QoS specification for query q and can be defined as follows:

$$\mathcal{S}_q(\tau_q, \epsilon_q) = \alpha_q \cdot \mathcal{V}_q(\tau_q) + (1 - \alpha_q) \cdot \mathcal{U}_q(\epsilon_q) \tag{2}$$

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 one $\mathcal{U}_q(\epsilon_q)$. 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:

$$\mathcal{V}_j^*(P_j) = \sum_{q\in C_j} \alpha_q \cdot \mathcal{V}_q(P_j) \tag{3}$$

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 an example of aggregating two piece-wise linear functions of 4 segments each.

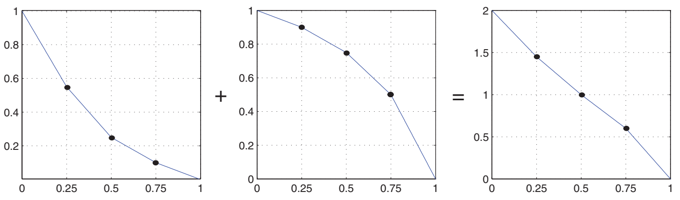

Fig. 3. Example of QoS function aggregation

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. Then, we have:

$$\mathcal{U}_i^*(\Delta_i) = \sum_{q\in Q,\ \text{s.t. } m_q(i)>0} (1 - \alpha_q) \cdot m_q(i) \cdot \mathcal{U}_q(\Delta_i) \tag{4}$$

The equality in Equation 4 holds when (a) $\mathcal{U}_q$'s are linear functions,[4] 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:

$$\Psi = \frac{1}{m} \left(\sum_{j=1}^{k} \mathcal{V}_j^*(P_j) + \sum_{i=1}^{l} \mathcal{U}_i^*(\Delta_i) \right) \tag{5}$$

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 regions, and similarly queries within 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, we end up creating a significant overhead for system optimization. This 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

[4]For Equation 4 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.

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 query load shedding and update load shedding separately in the next two sections, and then present our final solution for combining them.

V. 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.

A. *Formalization of the Problem*

The aim of the query load shedding problem is to maximize the first component of the overall quality from Equation 5, denoted by Ψ_v. Given k query groups, recalling that $\mathcal{V}^*_j(P_j)$ denotes 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:

$$\Psi_v = \sum_{j=1}^{k} \mathcal{V}^*_j(P_j) \tag{6}$$

Assume that the throttle fraction z is given, which defines the fraction of query load to keep. (The details for computation of z are described later in Section VII-C). 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 ($C_j, j \in [1..k]$), so as to maximize Ψ_v. This is performed by the QLBC algorithm described next.

B. *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:

$$G(q,z) = \frac{\alpha_q \cdot \frac{d(\mathcal{V}_q(\tau))}{d\tau}\Big|_{\tau=\tau_\vdash/z}}{f_c(\{q\}) \cdot \frac{d(1/\tau)}{d\tau}\Big|_{\tau=\tau_\vdash/z}} \tag{7}$$

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 7 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 7 can be simplified as follows, where $\mathcal{V}^a_q(i)$ denotes the slope of the ith ($i \in [1..\kappa]$) linear segment of $\mathcal{V}_q$:

$$G(q,z) = \frac{\alpha_q \cdot \mathcal{V}^a_q\left(\left\lceil \kappa \cdot \frac{\tau_\vdash/z - \tau_\vdash}{\tau_\dashv - \tau_\vdash}\right\rceil\right)}{-f_c(\{q\}) \cdot (z/\tau_\vdash)^2} \tag{8}$$

C. *Grouping Queries with* QLBC

MobiQual uses the quality loss per unit cost (*qlpc*) metric to define the similarity of queries, and the Euclidean distance function defined between two *qlpc* vectors (see Equation 9 below) for clustering queries into desirable query groups in terms of load shedding effectiveness. We call this algorithm the Q*uality* ***loss based clustering*** 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:

$$D(q_1,q_2) = \sum_{\iota\in([1..\kappa]-0.5)/\kappa} (G(q_1,\iota) - G(q_2,\iota))^2 \tag{9}$$

The QLBC algorithm uses k-means clustering [23] to form the final k set of query groups, based on Equation 9.

VI. 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.

A. Formalization of the Problem

The goal of the update load shedding problem is to maximize the second component of the overall quality from Equation 5, 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:

$$\Psi_u = \sum_{i=1}^{l} \mathcal{U}_i^*(\Delta_i) \tag{10}$$

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:

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

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 the GRIDREDUCE algorithm.

B. Partitioning the Space with GRIDREDUCE

The goal of the GRIDREDUCE 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.

1) Algorithm Overview: 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 QoS functions 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 gain in the query-result accuracy, called the *accuracy gain*, which is computed using the aggregated region statistics.

2) 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 space. For each grid cell $c_{i,j}$ the statistics grid stores the accuracy QoS function for that grid cell. The only data structure maintained over time by the MobiQual space partitioner is this grid. The partitioning generated by the GRIDREDUCE algorithm using an $\alpha \times \alpha$ grid is called an (α, l)-*partitioning*.

3) Stage I: Building the Region Hierarchy: In the first stage, 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 accuracy QoS functions associated with the grid cells for each node of the tree. The first stage of the algorithm takes $\mathcal{O}(\alpha^2)$ time and consumes $\mathcal{O}(\alpha^2)$ space.

4) Stage II: Drilling Down in the Hierarchy: In the second stage, we start with the root node of the tree, i.e., the entire space. At each step, we pick a visited tree node (initially only the root) and replace it with its 4 child nodes in the quad-tree. This process continues until we reach l visited tree nodes (corresponding to l shedding regions), assuming $l \mod 3 = 1$. The crux of this stage lies in how we choose a region to further partition during each step. For this purpose we maintain a max-heap of all visited tree nodes based on their accuracy gains, 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 $U[t]$ 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[5] :

$$E[t] \leftarrow min_{\Delta_t} \mathcal{U}_t^*(\Delta), \text{ s.t. } f_r(\Delta) \le z \cdot f_r(\Delta_\vdash)$$

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 $t_i, i \in [1..4]$. Using $n[t]$ to denote the number of mobile nodes in the region of tree node t, we have:

$$\begin{aligned} E_p[t] \quad &\leftarrow \quad min_{\{\Delta_{t_i}\}} \sum_{i=1}^{4} \mathcal{U}_{t_i}^*(\Delta_{t_i}) \\ \text{s.t.} \quad &\sum_{i=1}^{4} n[t_i] \cdot f_r(\Delta_{t_i}) \le z \cdot n[t] \cdot f_r(\Delta_\vdash) \end{aligned}$$

Then the difference $E[t] - E_p[t]$ gives us the accuracy gain $U[t]$.

The computation of $E[t]$ and $E_p[t]$, and thus the accuracy gain $U[t]$, requires solving the problem of inaccuracy threshold 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 4 child nodes of t with $l = 4$. 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 $\mathcal{O}(\alpha^2 + l)$.

VII. PUTTING IT ALL TOGETHER: 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

[5]We are abusing the notation here to represent the aggregated accuracy QoS function for the region of tree node t by $\mathcal{U}_t^*(\Delta)$

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 algorithm called THROTLOOP.

A. 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 5. 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:

$$z_v + z_u \cdot \gamma \le z \cdot (1 + \gamma) \tag{11}$$

The parameter γ in Equation 11 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:

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

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.

B. 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[6]) 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 shedding region A_i. We have:

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

```
MQLS($z$, $c_v$, $c_u$)
(1)  $H$: empty, min heap of $S_j^v$'s and $S_i^u$'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. $S_j^v$'s, add to $H$}
(5)      $S_j^v \leftarrow \frac{\mathcal{V}_j^*(\tau_{\vdash}+c_v) - \mathcal{V}_j^*(\tau_{\vdash})}{f_c(C_j) \cdot \left(\frac{1}{\tau_{\vdash}+c_v} - \frac{1}{\tau_{\vdash}}\right)}$ {initial query qlpc}
(6)      $S_j^v \leftarrow S_j^v / V_{\dashv}$ {normalize}
(7)      $P_j \leftarrow \tau_{\vdash}$, $H$.INSERT($S_j^v$) {add query qlpc}
(8)  for $i = 1$ to $l$ {init. $S_j^u$'s, add to $H$}
(9)      $S_i^u \leftarrow \frac{\mathcal{U}_i^*(\epsilon_{\vdash}+c_u) - \mathcal{U}_i^*(\epsilon_{\vdash})}{n_i \cdot (f_r(\epsilon_{\vdash}+c_u) - f_r(\epsilon_{\vdash}))}$ {initial update gain}
(10)     $S_i^u \leftarrow \gamma^{-1} \cdot S_i^u / U_{\dashv}$ {normalize}
(11)     $\Delta_i \leftarrow \epsilon_{\vdash}$, $H$.INSERT($S_i^u$) {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, $S = S_j^v$
(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 $P_j \le \tau_{\dashv}$ {further incr. possible}
(18)             $S_j^v \leftarrow \frac{\mathcal{V}_j^*(P_j+c_v) - \mathcal{V}_j^*(P_j)}{f_c(C_j) \cdot \left(\frac{1}{P_j+c_v} - \frac{1}{P_j}\right)}$ {new query qlpc}
(19)             $S_j^v \leftarrow S_j^v / V_{\dashv}$ {normalize}
(20)             $H$.INSERT($S_j^v$) {insert the query qlpc}
(21)     else if $S$ is for a threshold, $S = S_i^u$
(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 $\Delta_i \le \epsilon_{\dashv}$ {further incr. possible}
(25)             $S_i^u \leftarrow \frac{\mathcal{U}_i^*(\Delta_i+c_u) - \mathcal{U}_i^*(\Delta_i)}{n_i \cdot (f_r(\Delta_i+c_u) - f_r(\Delta_i))}$ {new update qlpc}
(26)             $S_i^u \leftarrow \gamma^{-1} \cdot S_i^u / U_{\dashv}$ {normalize}
(27)             $H$.INSERT($S_i^u$) {insert the update qlpc}
(28) until $V + \gamma \cdot U \le V_{\dashv} + \gamma \cdot U_{\dashv}$ {budget reached}
             or $H$.SIZE() $= 0$ {all period and thresholds maxed}
```

$$S_j^v = \sum_{q \in Q} f_c(\{q\}) \cdot \frac{\mathcal{V}_j^*(P_j + c_v) - \mathcal{V}_j^*(P_j)}{f_c(C_j) \cdot (\frac{1}{P_j + c_v} - \frac{1}{P_j})} \tag{12}$$

$$S_i^u = \gamma \cdot n \cdot f_r(\epsilon_{\vdash}) \cdot \frac{\mathcal{U}_i^*(\Delta_i + c_u) - \mathcal{U}_i^*(\Delta_i)}{n_i \cdot (f_r(\Delta_i + c_u) - f_r(\Delta_i))} \tag{13}$$

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, 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

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

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 12 (or Equation 13) 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 $\mathcal{O}(l+k)$.

C. Setting the Throttle Fraction with THROTLOOP

We set the throttle fraction adaptively based on feedback with regard to how well the system is performing in terms of shedding the correct amount of load, using the THROTLOOP algorithm. 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 be 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 actually 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:

$$\phi = \frac{z_v \cdot \sigma_v + \gamma \cdot z_u \cdot \sigma_u}{z \cdot (1+\gamma)} \qquad (14)$$

The denominator of Equation 14 is the amount of load the system was supposed to handle (recall right-hand side of Equation 11) and the nominator is the actual amount of load that was handled (left-hand side of Equation 11, adjusted by σ_v and σ_u). 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:

$$z \leftarrow \begin{cases} z \cdot \phi & \mu = 1 \\ min(1, z/\mu) & \mu < 1 \end{cases} \qquad (15)$$

This concludes our description of the MobiQual system.

VIII. EXPERIMENTAL EVALUATION

We evaluate MobiQual in two parts. First, we evaluate MobiQual without query load shedding and with no user defined QoS specifications [7]. The motivation behind this mode, named *MobiQual-Light*, is the fact that update load shedding aspect of MobiQual is completely transparent to the inner workings of the query engine. It can integrate cleanly and effortlessly with any mobile CQ engine that accepts position updates from mobile nodes to evaluate spatial CQs. The intelligent update load shedding capability by itself provides substantial improvement in overall query result accuracy and is a significant contribution of this work, and has wide applicability. Second, we evaluate MobiQual in its entirety, with update and query load shedding capabilities as well as accuracy and freshness-based QoS support. The latter study illustrates the drastic improvements that could be achieved by minimally modifying the query engine to integrate query load shedding and QoS support.

Fig. 4. The road map used in the experiments, Chamblee, GA, USA

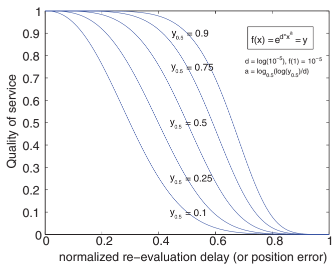

Fig. 5. Example QoS functions, with different mid-point QoS values ($y_{0.5}$)

A. 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 [24], 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 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 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), number of regions l used by the *GridReduce* algorithm (default value: 250), 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

[7]By setting $\mathcal{U}^*_i(\Delta_i) = 1 + (\Delta_{\vdash} - \sum_{q \in Q} m_q(i) \cdot \Delta_i)/(\Delta_{\dashv} - \Delta_{\vdash})$ and $\mathcal{V}^*_j(P_j) = 1$

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).

IX. MobiQual-Light: Experimental Evaluation

In this section we present experimental results on the effectiveness of the MobiQual-Light 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. We compare our MobiQual-Light 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 inaccuracy threshold settings are not performed.
– **Grid-Light**: A downgraded version of the MobiQual-Light 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.

A. Evaluation Metrics

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

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:

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

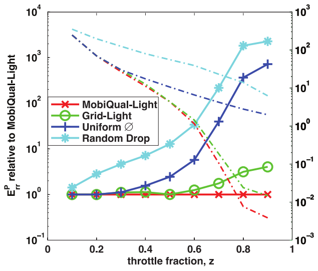

Fig. 6. Position Error vs. throttle fraction

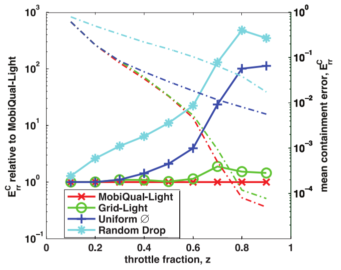

Fig. 7. Containment Error vs. throttle fraction

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

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 Mqls 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.

B. 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 MobiQual-Light compared to competing approaches for shedding position update load. The second group of results are on the additional cost brought by the MobiQual-Light load shedder, and show that the overhead is minimal.

1) Query-result Accuracy: We study the impact of several system and workload parameters on the query-result accuracy and the relative advantage of MobiQual-Light over competing approaches.

a) Impact of the Throttle Fraction: The graphs in Figures 6 and 7 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 MobiQual-Light 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 MobiQual-Light outperforms all other approaches throughout the entire throttle fraction range. Random Drop performs the worst, followed by Uniform Δ and Grid-Light. At $z = 0.75$, Random Drop has 300 times the mean position error of MobiQual-Light, Uniform Δ has 40 times that of MobiQual-Light, and Grid-Light has 2 times that of MobiQual-Light. At $z = 0.5$, Random Drop, Uniform Δ, and Grid-Light has 10, 2, and 1.08 times the E^P_{rr} of MobiQual-Light. 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 position 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 MobiQual-Light. This is because MobiQual-Light cuts the required fraction of position updates from the regions that do not contain any queries. Close to none error of MobiQual-Light near $z = 1$ boosts the relative error results for Random Drop and Uniform Δ.

b) Impact of the Number of Shedding Regions: The graphs in Figure 8 plot the relative mean containment error E^C_{rr} of Grid-Light with respect to MobiQual-Light 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 Grid-Light has up to 35% higher containment error in query results compared to MobiQual-Light. The improvement provided by MobiQual 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 MobiQual-Light at a better rate than Grid-Light, since MobiQual utilizes an intelligent space partitioning algorithm. However, when l gets too large the grid partitioning of Grid-Light achieves enough granularity to catch MobiQual-Light 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 containment error E^C_{rr} of MobiQual-Light 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 Figure 7. This conservative setting of l also results in a lightweight load shedding solution, as we illustrate later in this section.

2) Cost of Load Shedding: The cost of load shedding consists of i) configuring the parameters of MobiQual-Light 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.

a) Server Side Cost: The graphs in Figure 10 plot the time it takes to execute the THROTLOOP, GRIDREDUCE, and MQLS 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 MobiQual-Light takes around 40 msecs. This will enable frequent adaptation, even though for most applications

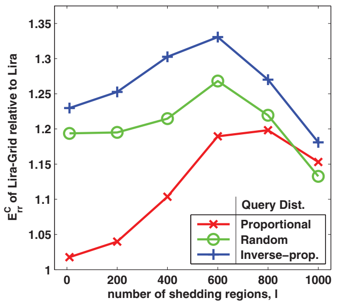

Fig. 8. E^C_{rr} of Grid-Light w.r.t. to MobiQual-Light vs. # of shedding regions

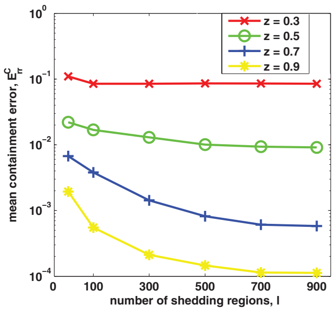

Fig. 9. Containment Error of MobiQual-Light vs. # of shedding regions

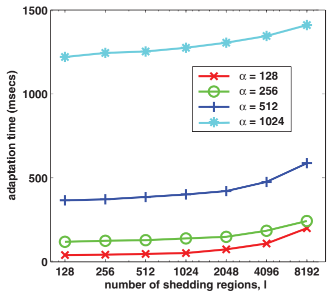

Fig. 10. Server side cost of configuring MobiQual-Light

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 MobiQual-Light 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 ($\approx$ 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 MobiQual-Light takes 500 msecs. This corresponds to $8 \cdot 10^{-4}$ fraction of a 10 minute adaptation period. These numbers show that MobiQual-Light is lightweight and introduces little overhead on the server side.

b) Messaging Cost: Table II 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) [25]. This nature of base stations match perfectly with MobiQual'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 MobiQual, compare this

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

TABLE II
NUMBER OF SHEDDING REGIONS PER BASE STATION

41 Δ_i's on average, takes $41 \cdot (3+1) \cdot 4$ bytes = 656 bytes. 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 MobiQual 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.

c) Mobile Node Side Cost: Since the total number of shedding regions known to a mobile node at any time is only around 41, MobiQual-Light 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. Since MobiQual does not incur additional mobile node side cost over MobiQual-Light, we conclude that MobiQual will work on computationally weak mobile nodes without any problem.

X. MOBIQUAL: EXPERIMENTAL EVALUATION

In this section we compare the performance of the MobiQual system in its entirety, with both update and query load shedding as well as per-query QoS specification support, to a number of other alternatives. These are:

- **Query-only load shedding**: QoS-aware differentiated load shedding with respect to re-evaluation periods only (see Section V) and uses a fixed inaccuracy threshold of $\epsilon_\vdash$.
- **Update-only load shedding**: QoS-aware differentiated load shedding with respect to inaccuracy thresholds only (see Section VI) and can be seen as the QoS-aware extension of MobiQual-Light. Thus we name it as MobiQual-Light$^+$.
- **Single Δ-P**: Combined QoS-aware query and update load shedding, but without query grouping (QLBC algorithm from Section V-C) and space partitioning (extended GRIDREDUCE algorithm from Section VI-B). It represents a special case of the MobiQual system with $k = l = 1$.

A. Evaluation Metrics

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

i) The overall quality metric Ψ, as defined by Equation 5.

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.

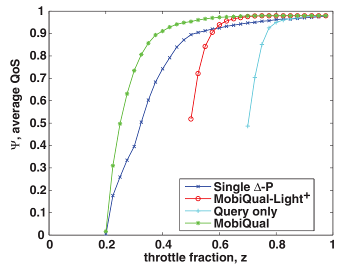

Fig. 11. Overall result quality as a function of the throttle fraction

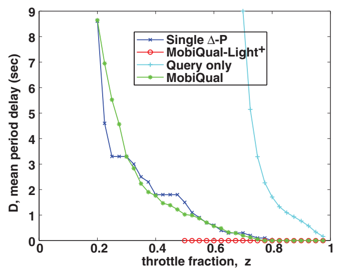

Fig. 12. Mean period delay as a function of the throttle fraction

B. 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.

1) Impact of the Throttle Fraction: The graphs in Figure 11 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 11 that, for the default settings, the query-only approach can only support load shedding for $z \geq 0.7$ and MobiQual-Light$^+$ 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. 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 query and update load shedding.

The graphs in Figures 12 and 13 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 13), whereas the update-only approach has zero mean period delay (as observed from Figure 12). However, since a good overall quality requires balancing freshness and accuracy, these two approaches do not provide good overall QoS as observed from Figure 11. The mean period delay of Single Δ-P stays slightly above that of MobiQual for $z > 0.3$. After

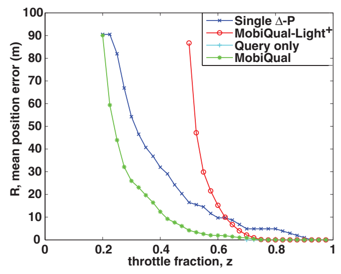

Fig. 13. Mean position error as a function of the throttle fraction

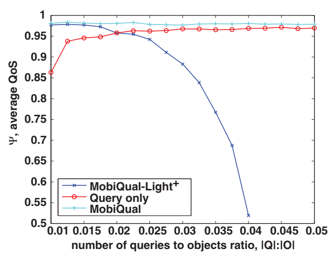

Fig. 14. Overall result quality with changing query workload

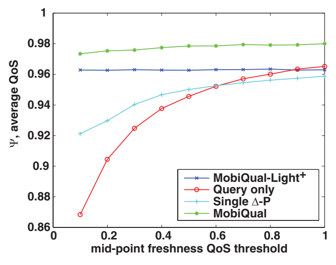

Fig. 17. Query result quality for varying freshness QoS specs.

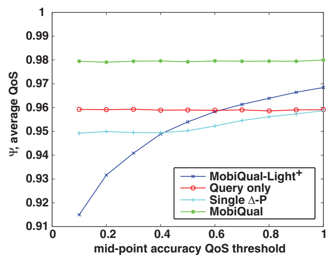

Fig. 18. Query result quality for varying accuracy QoS specs.

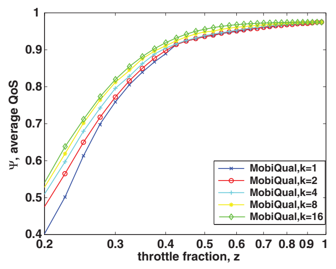

Fig. 15. The impact of number of query groups on result quality

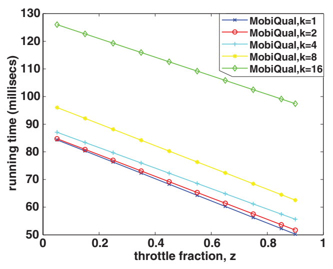

Fig. 16. Adaptation time with different number of query groups

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 13 that MobiQual consistently outperforms Single Δ-P in terms of the mean position error. This is not because MobiQual sheds 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.

2) Impact of the # of Queries and Query Groups: The graphs in Figure 14 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 (MobiQual-Light$^+$) 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 14).

An important parameter that impacts the performance of the MobiQual system is the number of query groups, k. As discussed in Section III-D, 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 15, since the query groups become more and more homogeneous in terms of the QoS functions of the queries contained within. The graphs in Figure 15 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 16 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 relatively small compared to the adaptation period, resulting in a very lightweight load shedding scheme.

3) Impact of the QoS Specifications: The graphs in Figures 17 and 18 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 values along the x-axis represent QoS specifications with increasingly stringent freshness components for Figure 17 and increasingly stringent accuracy components for Figure 18. 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 (MobiQual-Light$^+$) 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 17 and 18, respectively. We observe from Figure 17 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,

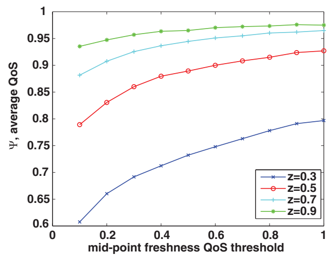

Fig. 19. Result quality under changing z & freshness QoS specs.

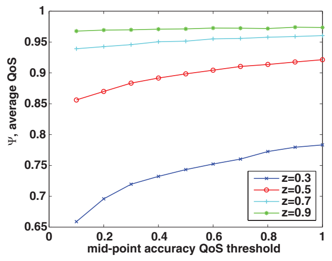

Fig. 20. Result quality under changing z and accuracy QoS specs.

and for a mid-point freshness QoS threshold of 0.75 (see previous Figure 11). The results presented in Figure 18 for the mid-point accuracy QoS threshold are very similar in nature.

In Figures 19 and 20 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 19 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 19 and 20, 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.

XI. Conclusions and Future Work

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.

There are several interesting issues for future work. (1) In this paper, we only considered range queries. However, MobiQual can be applied to kNN queries as well. There are various query processing approaches where kNN queries are first approximated by circular regions based on upper bounds on the kth distances [13]. Using such approximations, kNN queries can also take advantage of MobiQual. Supporting kNN queries may also require taking into consideration topology of the road network, as it is often more meaningful to define nearest neighbors in terms of the network distance rather than the euclidean distance. (2) MobiQual should be able to dynamically adjust the values of the l (number of shedding regions) and k (number of query groups) parameters as the workload changes. An overestimated value for these paramters means lost opportunity in terms of minimizing the cost of adaptation, whereas an underestimated value means lost opportunity in terms of maximizing the overall QoS. In this paper we have shown that the time it takes to run the adaptation step is relatively small compared to the adaptation period in most practical scenarios. This means relatively aggressive values for l and k could be used to optimize for QoS without worrying about the cost of adaptation. We leave it as a future work to adapt these parameters dynamically.

References

[1] “NextBus,” http://www.nextbus.com/, January 2004.
[2] “Google RideFinder home page,” http://labs.google.com/ridefinder, Febuary 2006.
[3] O. Wolfson, P. Sistla, S. Chamberlain, and Y. Yesha, “Updating and querying databases that track mobile units,” *Springer Distributed and Parallel Databases*, vol. 7, no. 3, pp. 257–387, 1999.
[4] 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.
[5] 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*, vol. 51, no. 10, pp. 1124–1140, 2002.
[6] 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.
[7] 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.
[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] 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.
[10] 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.
[11] K.-L. Wu, S.-K. Chen, and P. S. Yu, “Incremental processing of continual range queries over moving objects,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 11, pp. 1560–1575, 2006.
[12] X. Xiong and W. G. Aref, “R-trees with update memos,” in *IEEE International Conference on Data Engineering*, 2006.
[13] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu, “Processing moving queries over moving objects using motion adaptive indexes,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 5, pp. 651–668, 2006.
[14] 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*, vol. 17, no. 5, pp. 698–712, 2005.
[15] S. Pandey, K. Dhamdhere, and C. Olston, “WIC: A general-purpose algorithm for monitoring web information sources,” in *International Conference on Very Large Data Bases*, 2004.
[16] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu, “Mobiqual: QoS-aware load shedding in mobile CQ systems,” in *IEEE International Conference on Data Engineering*, 2008.
[17] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis, “Continuous nearest neighbor monitoring in road networks,” in *International Conference on Very Large Data Bases*, 2006.

[18] M. F. Mokbel and W. G. Aref, "SOLE: Scalable on-line execution of continuous queries on spatio-temporal data streams," *International Journal on Very Large Data Bases*, vol. 17, no. 5, pp. 971–995, 2008.

[19] R. V. Nehme and E. A. Rundensteiner, "ClusterSheddy: Load shedding using moving clusters over spatio-temporal data streams," in *Database Systems for Advanced Applications*, 2007.

[20] Y. Cai and K. A. Hua, "Real-time processing of range-monitoring queries in heterogeneous mobile databases," *IEEE Transactions on Mobile Computing*, vol. 5, no. 7, pp. 931–942, 2006.

[21] 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.

[22] B. Gedik, L. Liu, K.-L. Wu, and P. S. Yu, "Lira: Lightweight, region-aware load shedding in mobile CQ systems," in *IEEE International Conference on Data Engineering*, 2007.

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

[24] 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.

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