ISSN:0975-9646

Load Flaking on Continuous Monitoring of Spatial Queries

Narendra Babu.Pamula,G.Bargavi

Prakasam Engineering College, Kandukuru
Prakasam Dist, Andhrapradesh,India

Abstract— **Wireless data broadcast is a promising technique for information dissemination that leverages the computational capabilities of the mobile devices, in order to enhance the scalability of the system. Under this environment, the data are continuously broadcast by the server, interleaved with some indexing information for query processing. Clients may then tune in the broadcast channel and process their queries locally without contacting the server. In location-based, mobile continual query (CQ) systems, two key measures of quality-of-service (QoS) are: freshness and accuracy. In continuous monitoring an air indexing framework that (i) outperforms the existing (i.e., snapshot) techniques in terms of energy consumption, while achieving low access latency, and (ii) constitutes the first method supporting efficient processing of continuous spatial queries over moving objects. So to achieve freshness, the CQ server must perform frequent query revaluations. To attain accuracy, the CQ server must receive and process frequent position updates from the mobile nodes. In this paper, we formulate this problem as a load flaking one, and develop MobiQual—a QoS-aware approach to performing both update load flaking and query load flaking. The design of MobiQual highlights three important features like Per-query QoS specification, 3) Low cost adaptation: MobiQual dynamically adapts, with a minimal overhead, to changing load conditions and available resources. Load flaking, 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 flaking.**

Keywords-**Query processing, load flaking, Spatial databases, query processing, location based services, wireless data broadcast, air indexes.**

1.INTRODUCTION

Mobile devices with computational, storage, and wireless communication capabilities (such as PDAs) are becoming increasingly popular. At the same time, the technology behind positioning systems is constantly evolving, enabling the integration of low cost GPS devices in any portable unit. Consequently, new mobile computing applications are expected to emerge, allowing users to issue location-dependent queries in a ubiquitous manner. Consider, for instance, a user (mobile client) in an unfamiliar city, who would like to know the 10 closest restaurants. This is an instance of a *k nearest neighbor* (*k*NN) query, where the query point is the current location of the client and the set of data objects contains the city restaurants. Alternatively, the user may ask for all restaurants located within a certain distance, i.e., within 200 meters. This is an instance of a *range* query. *continuous monitoring* of multiple queries over arbitrarily moving objects. In this setting, there is a central server that monitors the locations of both objects and queries. The task of the server is to report and continuously update the query results as the clients and the objects move. As an example, consider that the data objects are vacant cabs and the clients are pedestrians that wish to know their *k* closest free taxis until they hire one. As the reverse case, the queries may correspond to vacant cabs, and each free taxi driver wishes to be continuously informed about his/her *k* closest pedestrians. Several monitoring methods have been proposed, covering both range and *k*NN queries. Some of these methods assume that objects issue updates whenever they move, while others consider that data objects have some computational capabilities, so that they inform the server only when their movement influences some query. In the aforementioned methods, the processing load at the server side increases with the number of queries. In applications involving numerous clients, the server may be overwhelmed by their queries or take prohibitively long time to answer them. Continual query (CQ) systems have been proposed to handle long running location monitoring tasks in a scalable manner 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 reevaluation. 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 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 reevaluation. It is dependent on the frequency of query reevaluations performed at the server. As mobile nodes continue to move, there are further deviations in mobile node positions after the last query reevaluation. However, such post query reevaluation deviations are not attributed to inaccuracy.

To obtain fresher query results, the CQ server must reevaluate the continual queries more frequently, requiring more computing resources. Similarly, to attain more accurate query results, the CQ server must receive and process position updates from the mobile nodes at a higher rate, demanding communication as well as computing resources. However, it is almost impossible for a mobile CQ system to achieve 100 percent 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 flaking framework for mobile CQ systems. MobiQual is capable of providing high-quality query results by dynamically determining the appropriate amount of update load flaking (discarding certain location update messages) and query load flaking (skipping some query reevaluations) to be performed according to the application-level QoS specifications of the queries. An obvious advantage of combining query load flaking and update load flaking within the same framework is to empower MobiQual with differentiated load flaking capability, that is, configuring query reevaluation 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 flaking and query load flaking 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 reconfiguring 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 works has exploited the potential of performing load flaking to maximize the application-level freshness and accuracy of mobile queries. In contrast to the existing work on scalable query processing and indexing techniques, MobiQual provides a QoS-aware framework for performing both update load flaking and query load flaking, 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 flaking and location query load flaking can significantly outperform the approaches that are based on query-only or update-only load flaking and 2) MobiQual provides higher quality guarantees compared to the approaches that lack the supports of QoS awareness and differentiated load flaking. A preliminary version of the MobiQual framework was described. In the current paper, we have substantially expanded the MobiQual framework by providing 1) a complete description of QoS-aware update load flaking in which includes the GRIDREDUCE algorithm for performing space partitioning (Section 6.2); 2) several additional sets of experiments in Section 9, evaluating a MobiQual-Light scheme that focuses on update load flaking; and 3) a revised performance comparison of MobilQual with various schemes.

2 RELATED WORK

Wireless Broadcasting and Air Indexes

The transmission schedule in a wireless broadcast system consists of a series of *broadcast cycles*. Within each cycle the data are organized into a number of index and data *buckets*. A bucket (which has a constant size) corresponds to the smallest logical unit of information, similar to the page concept in conventional storage systems. A single bucket may be carried into multiple network *packets* (i.e., the basic unit of information that is transmitted over the air). However, they are typically assumed to be of the same size (i.e., one bucket equals one packet).

The most common data organization method is the (1*;m*) *interleaving scheme* as shown in Figure 2. The data objects are divided into *m* distinct segments, and each data segment in the transmission schedule is preceded by a complete version of the index. In this way, the access latency for a client is minimized, since it may access the index (and start the query processing) immediately after the completion of the current data segment. also introduces an alternative distributed index that reduces the degree of replication in order to further improve the performance. Specifically, instead of the entire index being replicated prior to each data segment, only the index that corresponds to the subsequent segment is included (i.e., replication occurs at the upper levels of the index tree).

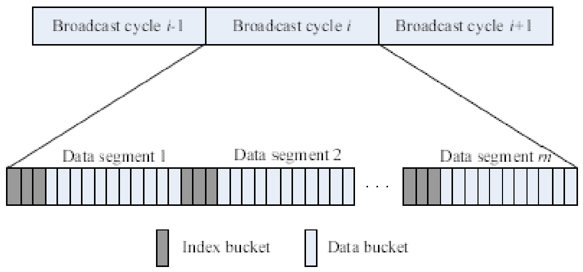

Fig 1:Interleave Schemas

The main motivation behind air indexes is to minimize the power consumption at the mobile client. Although in a broadcast environment the uplink transmissions are avoided, receiving all the downlink packets from the server is not energy efficient. For instance, the Cabletron 802.11 network card (wireless LAN) was found to consume 1400 mW in the *transmit*, 1000 mW in the *receive*, and 130 mW in the *sleep* mode Therefore, it is imperative that the client switches to the sleep mode (i.e., turns off the receiver) whenever the transmitted packets do not contain any useful information. Based on the data organization technique of Figure 1, the query processing at the mobile client is performed as follows: (i) the client tunes in the broadcast channel when the query is issued, and goes to sleep until the next index segment arrives, (ii) the client traverses the index and determines when the data objects qualifying its query will be broadcast, and (iii) the client goes to sleep and returns to the receive mode only to retrieve the corresponding data objects. To measure the efficiency of an indexing method, two performance metrics have been considered in the literature: (i) *tuning time*, i.e., the total time that the client stays in the receiving mode to process the query, and (ii) *access latency*, i.e., the total time elapsed from the moment the query is issued until the moment that all the corresponding objects are retrieved. In other words, the tuning time is a measure of the power consumption at the mobile client, while the access latency reflects the user-perceived quality of service.

Previous works on mobile CQ systems have focused on roughly five major categories with respect to scalability and performance. They are as follows:

1. Indexing schemes to process position updates more efficiently
2. Query processing techniques to evaluate continual queries more efficiently
3. Motion modeling techniques to reduce the number of position updates received from the mobile nodes, while keeping the position accuracy high
4. Load flaking approaches that achieve scalability on the server side by only processing specially defined significant updates
5. 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

The majority of these works, with the exception of the works listed under category 5, 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 for query reevaluation. Unlike the set of works listed under category 5, 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 4 are, to some extent, similar to MobiQual, in terms of flaking load in position updates. However, they use different techniques for load flaking. More importantly, they do not consider query load flaking.

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 flaking 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 reevaluation 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 addressed how frequent the position updates should be received from the mobile nodes or have not specified how frequent query results should be updated by reevaluating the queries. However, as we show in this paper, an integrated, QoS-aware approach is essential for achieving high-quality query results.

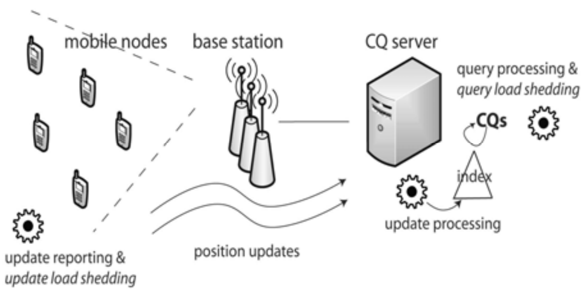

Fig 2: mobile CQ systems

3 Design Overview

3.1 Load flaking 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 Fig. 1) and periodically evaluates the installed continual queries (such as continual range or nearest neighbour queries) over the last known positions of the mobile nodes.[3] Since the mobile node positions change continuously, motion modelling 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 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 shed-ding 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 flaking 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 timevarying processing demands of a mobile CQ system entail that update and query load flaking 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.

3.2 The MobiQual Approach

The MobiQual system aims at performing dynamic load flaking 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 flaking among different queries, but also how to find a balance between query load flaking and update load flaking. The main idea is to apply differentiated load flaking to adjust the accuracy and freshness of queries. Namely, load flaking on position updates and query re-evaluations is done in such a way that the freshness and accuracy of queries are non uniformly impacted.

From the perspective of update load flaking, 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 flaking 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.

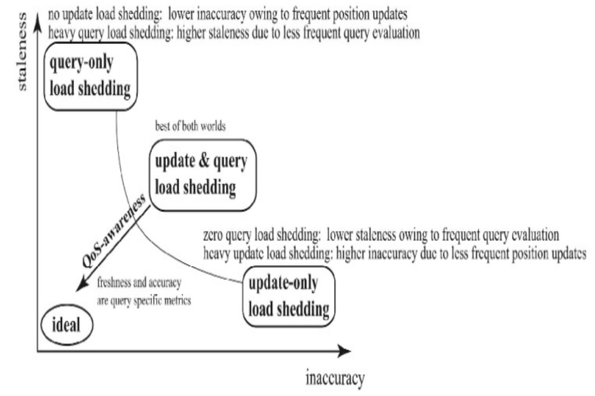
staleness
no update load shedding: lower inaccuracy owing to frequent position updates
heavy query load shedding: higher staleness due to less frequent query evaluation
query-only load shedding
best of both worlds
update & query load shedding
QoS-awareness
zero query load shedding: lower staleness owing 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. 3. QoS-aware update load flaking and QoS-aware query load flaking.

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 flaking: We use inaccuracy thresholds from motion modelling as control knobs to adjust the amount of update load flaking to be performed, where the same amount of increase in inaccu-racy thresholds for different geographical regions brings differing amounts of load reduction and QoS degradation with respect to accuracy. We refer to the load flaking 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 flaking.

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

MobiQual dynamically maintains a throttle fraction, which defines the amount of load that should be retained. It performs both update load flaking and query load flaking to control the load of the system according to this throttle fraction, while maximizing the overall quality of the query results. As illustrated in Fig. 2, MobiQual not only strikes a balance between freshness and accuracy by employing both query and update load flaking, 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.

3.3 Notation and Fundamentals

The set of continual queries installed in the system is denoted by Q. For each query q 2 Q, there is an associated QoS specification S_q. The QoS function takes a value in (0,1)where 1 represents perfect quality in terms of freshness and position error, and 0 represents the worst. □$_q$ and □$_q$ are used to denote the degree of staleness and inaccuracy in the query results, respectively. □$_q$ corresponds to the query re-evaluation period for q, whereas □$_q$ corresponds to the average of the inaccuracy thresholds used in motion modelling for the mobile nodes within the query result of q. At any given time, the result of query q can be at most □$_q$ 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 □$_q$ meters, on average. The mobile CQ system supports a minimum staleness value of □· and a minimum position error of □·. For any query q, we have S_q(T├ , €├)=1. Similarly, we introduce a maximum staleness value, denoted by □$_a$, and a maximum position error, denoted by S_a. The staleness in the query results cannot exceed the maximum threshold value of □$_a$ at which point the results are assumed to be useless. Also the position error is bounded by □$_a$. In summary 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 revaluation 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; use the same inaccuracy threshold α_i. A query q_u whose result lies completely within region A_i . For queries whose results contain mobile nodes from different regions, α_u is given by a weighted average of α_i values of the involved regions.

A key question for query load flaking 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.

3.4 *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 flaking 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. 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 10, we experimentally evaluate the benefit/cost trade-off in setting k and l to show that with lightweight adaptation, we can achieve high-quality query results.

3.5 *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 recomputed 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 reevaluation 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:the throttle fraction which defines the amount of load that can be retained relative to the load of providing perfect quality the setting of reevaluation periods and the setting of inaccuracy thresholds. 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 4). We then present the formulation of the QoS-aware query load flaking problem and present the quality-loss-based clustering (QLBC) algorithm for clustering the queries into k groups (Section 5). Then we formalize the QoS-aware update load flaking problem and provide a brief description of the QoS-aware space partitioning algorithm for dividing the geographical space of interest into l regions (Section 6). Finally, we present the formulation of the problem of combining query load flaking with update load flaking, and present the minimum quality loss per cost step (MQLS) algorithm for performing the adaptation step (Section 7).

4 AGGREGATING THE QOS FUNCTIONS

The aim of QoS aggregation is to associate an aggregate function for each query group C_j, and an aggregate function for each region A_i, such that the overall QoS of the system, denoted by □, is maximized. We define

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

where m is the total number of queries denotes the QoS specification for query q and can be defined as follows:

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

In other words, S_q(T ├ , € ├)is a linear combination of the freshness QoS function V_q (Tq) and the accuracy one $U_q(€_q)$. The parameter $\alpha_q \in [0, 1]$ called freshness weight, is used to adjust the relative importance of the two components, freshness and accuracy. V_q (Tq) and $U_q(€_q)$are nonincreasing positive functions, where V_q (T ├)=1 and U_q(€ ├)=1.

Since the query groups are no overlapping, we have the following:

$$\mathcal{V}^{*}_{j}(P_j) = \sum_{q\in C_j} \alpha_q \cdot \mathcal{V}_q(P_j).$$

5.PARTITIONING THE SPACE WITH GRIDREDUCE

The goal of the GRIDREDUCE space partitioning algorithm is to partition the geographical space of interest into l flaking regions such that this partitioning produces query results of higher accuracy.

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 nonuniform partitioning of the space can be constructed. The second stage follows a top-down process and creates the final set of l flaking 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.

5.1 *The Statistics Grid*

The statistics grid is an α x α 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 □ □ □ grid is called an (α,l)partitioning.

5.2 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, no 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 $O(\alpha^2)$ time and consumes $O(\alpha^2)$ space.

5.3 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 four child nodes in the quad-tree. This process continues until we reach l visited tree nodes (corresponding to l flaking 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 four subregions 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 flaking region, that is, t's region. Formally, we have

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

Let $E_p[t]$ be the average result inaccuracy if we had four flaking regions that correspond to the regions of t's child nodes t_i; i € [1::4]. Using n[t] to denote the number of mobile nodes in the region of tree node t, we have

$$E_p[t] \leftarrow min_{\{\Delta_{t_i}\}} \sum_{i=1}^{4} \mathcal{U}_{t_i}^*(\Delta_{t_i})$$

$$\text{s.t.} \sum_{i=1}^{4} n[t_i] \cdot f_r(\Delta_{t_i}) \le z \cdot n[t] \cdot f_r(\Delta_\vdash).$$

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 flaking regions. Concretely, computation of E[t] requires to solve for node t with l ¼ 1 and computation of E_p [t], requires to solve for the four 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 O(l log l) time and consumes O(l) space, bringing the combined time complexity to O(l . log l + α^2) and space complexity to O(α^2+l).

6.PROBLEM FORMALIZATION

The objective of the combined load flaking problem is to maximize the overall quality Ψ=1/~~m~~(Ψ_v + ψ_u). 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_i} 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 th~~resholds {Δ}. 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 onstraint as follows:

$$z_v + z_u \cdot \gamma \le z \cdot (1+\gamma).$$

The parameter γ in (11) represents the cost of performing update processing with the setting of $\forall_i$, Δ_i= $\in_\vdash$ compared to the cost of performing query re-evaluation with the setting of $\forall_j$; P_j =$T_\vdash$. In other words, for the ideal case, the query re-evaluation costs 1 unit, whereas the update processing costs γ ∈ (0,∞) units. Note that γ is not a system-specified parameter and is learned adaptively as follow: Let U be the observed cost of update processing and V be the observed cost of query re-evaluation during the last that the workload does not significantly change within the time frame of the adaptation period. Recall that the load flaking parameters are configured after each adaptation period, thus yielding new values for z_u and z_v (by way of changing (P_js andΔ_is). Thus, the combined load flaking problem is formalized as follows:

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

Note that this is a nonlinear 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 flaking problem.

The MQLS Algorithm

The basic principle of the MQLS algorithm is to start with the ideal case of γ_j, P_j = $T_\vdash$ and $\forall_I$, Δ_i =$\in_\vdash$ and incrementally reduce the load to z times that of the ideal case by repetitively increasing the re-evaluation period or the inaccuracy thresh-old 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-evaluatio periods and inaccuracy thresholds into □ segments such that we increase the P_js and Δ_is in increments of size C_v=(T├ - T├)/β and c_u= (∈├ - ∈├) respectively. The MQLS algorithm maintains a min. heap that stores a quality loss per unit cost[6] (qlpc) 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 . The qlpc value is denoted by Svj for query group Cj and Sui for flaking region Ai. We have

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

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

The numerators of the second components in the above quations 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^vs and S_i^us 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 flaking appropriately weighted by □, is above our load budget imposed by the throttle fraction z. The algorithm iteratively pops the topmost element of the min. heap and depending on whether we have a reevaluation period or inaccuracy threshold makes the increment using either c_v or c_u. The qlpc value of the popped element is updated 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 total number of greedy steps the algorithm can take is given byβ. (l + k) which happens when all re-evaluation periods and inaccuracy thresholds have to be increased to their maximum values. Each greedy step takes 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 O (β. (l + k). log (l +k))and the space complexity as O(l +k).

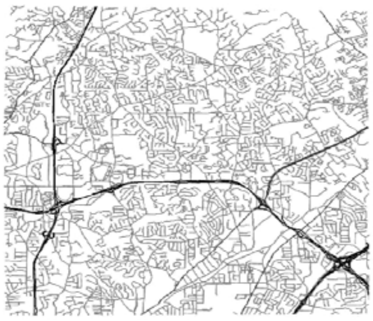

Fig. 4. The road map used in the experiments, Chamblee, Georgia, US.

The pseudocode of MQLS is given in Algorithm 1.

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)    $S_j^v \leftarrow \frac{\mathcal{V}_j^*(\tau_{\vdash}+c_v)-\mathcal{V}_j^*(\tau_{\vdash})}{f_c(C_j)\cdot(\frac{1}{\tau_{\vdash}+c_v}-\frac{1}{\tau_{\vdash}})}$ {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 $P_j \leq \tau_{\dashv}$ {further incr. possible}
(18)       $S_j^v \leftarrow \frac{\mathcal{V}_j^*(P_j+c_v)-\mathcal{V}_j^*(P_j)}{f_c(C_j)\cdot(\frac{1}{P_j+c_v}-\frac{1}{P_j})}$ {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 $\Delta_i \leq \epsilon_{\dashv}$ {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 $V + \gamma \cdot U \leq V_{\dashv} + \gamma \cdot U_{\dashv}$ {budget reached}
           or $H$. SIZE ()= 0 {all period and thresholds
           maxed}
```

6.2 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 flaking 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 revaluation points and/ or will not be able to admit all position updates into the system. Let $□_v$ 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 □$_u$ 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 □$_v$ and □$_u$ 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:

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

The denominator of (14) is the amount of load the system was supposed to handle (recall right-hand side and the numerator is the actual amount of load that was handled (left-hand side 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 than 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

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

7. EXPERIMENTAL EVALUATION

We evaluate MobiQual in two parts. First, we evaluate MobiQual without query load flaking and with no user-defined QoS specifications[7].

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 $\mathcal{V}_j^*(P_j) = 1$.

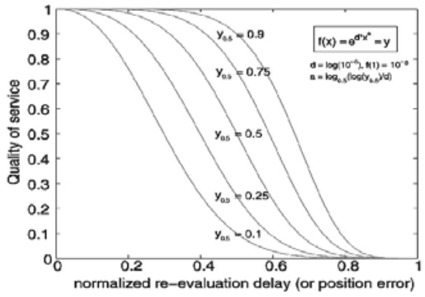

Fig. 5. Example QoS functions, with different midpoint QoS values ($y_{0:5}$).

The motivation behind this mode, named MobiQual-Light, is the fact that update load flaking 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 flaking 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 flaking capabilities as well as accuracy and freshness-based QoS support. The latter study illustrates the drastic improve-ments that could be achieved by minimally modifying the query engine to integrate query load flaking and QoS support.

8.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 flaking regions based on (l, α) partitioning. Instead, it uses equal-sized flaking regions based on an l-partitioning.

8.1 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 flaking and the second set of metrics deals with the cost of performing load flaking.

8.1.1 Query-Result Accuracy

Mean Containment Error, denoted by E_{rr}^C, 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 ∊ Q under load flaking, and $R^*(q)$ denote the correct result set under Then,

$\forall_{i \in [1..l]} \ \Delta_i = \Delta_{\vdash}$

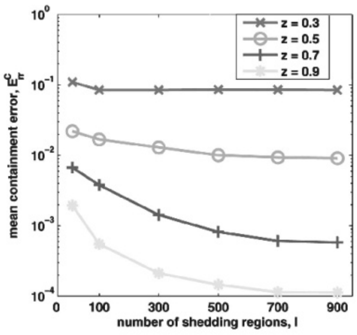

Fig. 6. Containment error of MobiQual-Light versus number of flaking regions

Mean Position Error, denoted by E_{rr}^P, 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 flaking and $p^*(o)$ denote the correct position of o under $\forall_{i \subset [1..l]} \ \Delta_i = \Delta_{\vdash}$.

We have

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

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 $C_{ov}^{C} = D^{C}_{ev} / E_{rr}^{C}$. These two metrics can also be extended to the position error.

8.1.2 Cost of Load Flaking

To evaluate the cost incurred by load flaking, we measure 1) the time it takes to execute the adaptation step that involves running the THROTLOOP, GRIDREDUCE, and MQLS algorithms and 2) the number of flaking regions that should be known by a mobile node, on average. The former metric measures the cost of load flaking from the perspective of the server, whereas the latter measures it from the perspective of the mobile node as well as the wireless network.

8.2 Experimental Results

We present the set of experimental results in two groups. The first group of results is on the query-result accuracy and highlights the superiority of MobiQual-Light compared to competing approaches for flaking position update load. The second group of results is on the additional cost brought by the MobiQual-Light load shedder, and shows that the overhead is minimal.

8.2.1 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.

Impact of the throttle fraction. The graphs in Figs. 6 and 7 plot the mean position error E_{rr}^{P} and mean containment error E_{rr}^{C} 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

8.2.2 Cost of Flaking Load

The cost of load flaking consists of 1) configuring the parameters of MobiQual-Light on the server side, which includes setting the throttle fraction, flaking regions, and update throttlers, 2) broadcasting the subset of flaking regions and update throttlers that correspond to the coverage area of each base station, and 3) installing the new set of flaking regions and update throttlers on the mobile node side.

Server-side cost. The graphs in Fig. 10 plot the time it takes to execute the THROTLOOP, GRIDREDUCE, and MQLS algorithms as a function of the number of flaking regions l, for different numbers of cells (α^2) for the statistics grid. For the default parameters of l =250 and α= 128, the config-uration of MobiQual-Light takes around 40 msecs. This will enable frequent adaptation, even though for most applica-tions that involve monitoring cars or pedestrians, it is unlikely that the update load will fluctuate with a period less than tens of minutes Messaging cost.

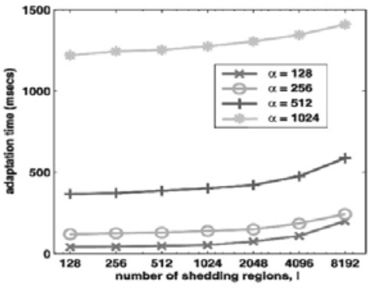

Fig. 7 Server-side cost of configuring MobiQual-Light

TABLE 2 Number of Flaking Regions per Base Station

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

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

Table 2 shows the average number of flaking 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) This nature of the base stations matches perfectly with MobiQual's space partitioning scheme, since the number of partitions is usually larger for dense areas and the small base station coverage areas help decreasing the average number of flaking regions known to a mobile node. Following this logic, we have used a node-density-dependent base station placement scheme and found that, on average, each node, and thus, each base station, should know around 41 flaking regions. Assuming a flaking region (which is square in shape) is represented by three 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 flaking regions and update throttlers is around 41 . (3 + 1) . 4 bytes = 656 bytes, on average. To assess the messaging cost of MobiQual, compare this number to 1,472 bytes, which is the maximum payload available to a UDP packet over Ethernet with a typical MTU of 1,500 bytes. When MobiQual reconfigures the load flaking para-meters, the new information is installed on all mobile nodes by using an average of one wireless broadcast packet per base station.

Mobile node side cost. Since the total number of flaking 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 X 5 grid index on the mobile node side, the flaking 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.

9. MOBIQUAL: EXPERIMENTAL EVALUATION

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

. Query-only load flaking: QoS-aware differentiated load flaking with respect to re-evaluation periods only (see Section 5) and uses a fixed inaccuracy threshold of $\epsilon_\vdash$.

. Update-only load flaking: QoS-aware differentiated load flaking with respect to inaccuracy thresholds only (see Section 6) 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 flaking, but without query grouping (QLBC algorithm from Section 5.3) and space partitioning (extended GRIDREDUCE algorithm from Section 6.2). It represents a special case of the MobiQual system with k= l=1

9.1 Evaluation Metrics

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

1. The overall quality metric $_3$, as defined by (5).
2. .The mean period delay D, which is defined as the average difference between the ideal case period $T_\vdash$ and the assigned period of queries $T_q=P_j$ for $q \in C_j$ The mean period delay is formulated as

$$D = \frac{1}{m}\sum_{q \in Q}(\tau_q - \tau_\vdash).$$

3. 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 $\forall_{i \in [1..l]}\ \Delta_i = \in_\vdash$ It is formulated as

$$R = \frac{1}{m}\sum_{q \in Q}(\epsilon_q - \epsilon_\vdash).$$

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

CONCLUSIONS

In this paper, we have presented MobiQual, a load Flaking 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 Flaking and update load Flaking within the same framework, through the use of differentiated load Flaking concept. Finally, the load Flaking 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 Flaking, as well as approaches that do combined query and update load Flaking but lack the differentiated load Flaking elements of the MobiQual solution, in particular, the query grouping and space partitioning mechanisms.

In this paper, we 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 . 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.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.

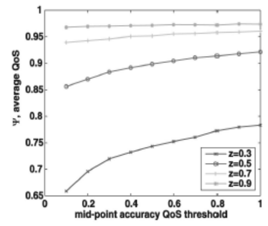

Fig: Result quality under changing z and accuracy QOS specs

An overestimated value for these parameters 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 that 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.

ACKNOWLEDGMENT

The satisfaction that accompanies the successful completion of any task would be incomplete with out the mention of people who made it possible and whose encouragement and guidance has been a source of inspiration throughout the paper .I feel elated to extend our floral gratitude to Prakasam Engineering College, Department of CSE Staff Members there encouragement all the way of during analysis of this paper there insinuations and critisms are the key behind the successful completion of the paper

REFERENCES

[1] NextBus, http://www.nextbus.com/, Jan. 2004.

[2] Google RideFinder Home Page, http://labs.google.com/ ridefinder, Feb. 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," Proc. ACM Int'l Conf. Management of Data, 2000.

[5] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in adhoc wireless networks. In *MOBICOM*, 2001.

[6] M.-S. Chen, P. S. Yu, and K.-L. Wu. Indexed sequential data broadcasting in wireless mobile computing. In *ICDCS*, 1997.

[7] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, 2004.

[8] B. Gedik, A. Singh, and L. Liu. Energy efficient exact kNN search in wireless broadcast environments. In *GIS*, 2004.