QACHE: Query Caching in Location-Based Services

Hui Ding[1], Aravind Yalamanchi[2], Ravi Kothuri[2], Siva Ravada[2], Peter Scheuermann[1]

[1] Department of EECS, Northwestern University, hdi117
email: peters@ece.northwestern.edu
[2] Oracle USA; email: [aravind.yalamanchi]@oracle.com

Abstract

Many emerging applications of location-based services continuously monitor a set of moving objects and answer queries pertaining to their locations. Query processing in such services is critical to ensure high performance of the system. Observing that one predominant cost in query processing is the frequent accesses to the database, in this paper we describe how to reduce the number of moving object to database server round-trips by caching query information on the application server tier. We propose a novel-caching framework, named QACHE, which stores and organizes spatially-relevant queries for selected moving objects. QACHE leverages the spatial indices and other algorithms in the database server for organizing and refreshing relevant cache entries within a configurable area of interest, referred to as the cache-footprint, around a moving object. QACHE contains appropriate refresh policies and prefetching algorithms for efficient cache-based evaluation of queries on moving objects. In experiments comparing QACHE to other proposed mechanisms, QACHE achieves a significant reduction (from 63% to $99%) in database roundtrips thereby improving the throughput of an LBS system.

Key words: location-based services, query processing caching

1 Introduction

Location-based services (LBS) [10] typically operate in a three-tier architecture: a central database server that stores past and current locations of all moving objects, applications that register to the database server their queries that are pertaining to the moving objects locations, and a set of moving objects that continuously change their locations (as shown in Fig. 1). As moving objects report their changing locations periodically, new answers are delivered to the applications when certain criteria are met. These queries on moving objects may contain predicates on the spatial locations as well as any other non-spatial attributes associated with the moving objects.

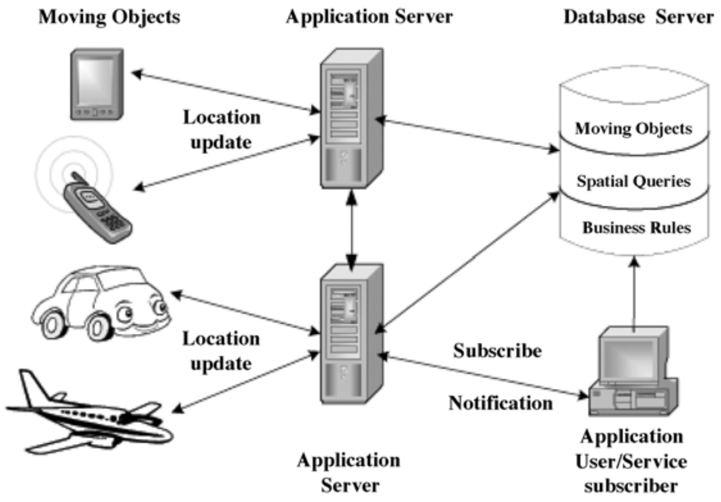

Fig. 1. Location-Based Services

Consider the following motivational scenario: a LBS system for local restaurant promotion sends appropriate restaurant information to nearby tourists. A registered restaurant specifies an area around its location using a spatial predicate (e.g., within-distance operator in commercial spatial databases: see [5] for more details) and restricts promotions only to tourists (identified by checking for “area_code != restaurant_area_code”) who are interested in its specific type of food (specified by predicate “user_food_interest == Chinese”). [12] describes how to specify such queries in Oracle database. Upon location updates of all mobile users, the LBS system must quickly decide whether one (or more) user matches all query criteria of a registered restaurant so that the promotion message can be sent before he/she travels out of the target area.

A critical problem about answering such queries in LBS is that any delay of the query response may result in an obsolete answer, due to the dynamic nature of the moving objects (in our example, tourists). This requires highly efficient query evaluation. On the other hand, while moving objects frequently report their location updates to the database server, many of the updates do not result in any new query answer. Take the above scenario as an example, the service system receives location updates from all tourists once every minute; it is too expensive to evaluate all location updates against the query criteria of all registered restaurants in the database server. Yet it is not necessary to do so because each query includes both spatial criteria and non-spatial criteria [8, 12] and an answer update should be delivered *only if both criteria* are met, e.g., location updates of tourists preferring Indian cuisine need not be evaluated even if they are in the area of Chinatown; likewise, location updates of tourists that are too far away from Chinatown need not be evaluated even if they do like Chinese food. In summary, query evaluation against irrelevant updates should be avoided as much as possible to reduce database burden and average response time.

To improve the performance of LBS on the delivery of in-time query answers, we focus on reducing query evaluation cost by minimizing the number of database accesses and the amount of computation required during evaluation. One effective technique toward this goal is to cache relevant data for fast answer delivery. In a three-tier LBS system, caching can be achieved on any of the three tiers.

- **On the mobile devices of end users:** queries are assumed to be issued by mobile users asking about its vicinity; when a user issues a query, the received answers are stored and used for answering future queries since spatial queries issued by the same mobile user usually exhibit high spatial locality. Unfortunately, this approach can only be used to cache objects that are static. Moreover, it highly relies on the tight processing and storage ability of the mobile devices and thus is not widely applicable.

- **On the database server:** most frequently referenced data and most frequently executed query plans can be cached by the database server to improve the performance of query processing. However, this approach increases burden on the already heavily loaded database server with large volume of incoming location updates [13].

- **On the middle-tier, i.e., application server:** relevant data items can be stored in the application server that serves as an external cache. When location updates are received, the application server can frequently use the cached data to process the updates and respond to the application users efficiently; location updates that cannot be evaluated are forwarded to the database for further processing.

In this paper, we adopt the third approach because it has the following advantages: (1) caching on the application server does not rely on the limited processing and storage ability of end users and it does not impose additional burden on the database server; (2) the application server can effectively cache data coming from heterogeneous sources to a single application; (3) the application server can provide caching for each moving object and this granularity is usually desirable in LBS, because a moving object may frequently be monitored for a series of events; and (4) the application server can filter out many of the updates that will not result in any new query answer and thus avoid unnecessary database accesses.

We present QACHE, a dynamic query-caching framework on the application server in LBS. This framework builds and improves on existing research solutions based on safe distance [7]. The main goal of QACHE is to improve the system performance in spatial query monitoring. To achieve this goal, QACHE identifies the most relevant spatial queries for the moving objects (in the sense that the upcoming location updates may result in new answers to these queries), and cache information of these queries in the application server. QACHE has the following characteristics:

- The items cached are not the moving objects but are the pending spatial queries pertaining to the moving objects. Since moving objects update their locations frequently, caching their locations would involve frequent cache replacement and update, and introduce significant overhead. In contrast, pending spatial queries are relatively stable[1] and should be cached to improve query response time.
- The granularity of the cache is per moving object, i.e., session-wise. The cache entry for a moving object stores queries that are interested in the moving object and are close to its current location. In addition, different sessions can share queries in the cache to minimize the storage requirement.

[1] The pending spatial queries may also change due to insertion or deletion, or modification to the query patterns etc. However, these changes occur much less frequently than the location updates.

- For a given moving object, only those queries that match the non-spatial (static) predicates can be cached in the cache entry.
- The queries cached are carefully organized to support efficient access for query answer update. In the cases where database access is necessary after cache access, the number of disk accesses can still be reduced by using the information stored in the cache.
- Our cache is dynamically updated as moving objects change their locations, so that queries that become farther away from a moving object are removed from the cache to make space for queries that get in the vicinity of that object.
- We propose the concept of *cache-footprint* for a cache entry, which is configured in terms of the minimum time interval between consecutive updates of the cache entries. This is represented as a distance $D_{\max}$ from the location of the moving object based on its known maximum velocity ($D_{\max} = refresh\_interval \times \max\_velocity$). For a fixed size of the cache entry, QACHE employs a two-pronged approach of storing the closest queries in *true detail* and the rest of the queries in cache-footprint region as approximations. The queries in true detail provide exact answers for a moving object whereas the approximated query regions reduce the false-positives. This two-level filtering improves the cache-effectiveness thereby increasing the throughput of the LBS system.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 describes the main components of QACHE and Section 4 elaborates on its implementation details. Section 5 describes our experimental evaluation results. Finally, Section 6 concludes the paper.

2 Related Work

Various techniques have been proposed to efficiently process spatial queries in LBS. The main approaches can be categorized as follows: (1) reducing the amount of computation when location updates are received by grouping pending queries using grid or similar indexing structures and conducting spatial join between moving objects and pending queries [6]; (2) reducing the number of queries performed by introducing safe distance/region for moving objects [7]; and (3) reducing the number of disk access by building a query index for all pending queries [7]. Unfortunately, the above techniques either focus on optimizing the performance within the database and hence fail to make use of the processing and storage power provided by the middle-tier, or have certain constraints on realistic applications. For example, many of the frequent location updates from the moving objects will not generate any new query answer and it is thus unnecessary to evaluate the pending queries against these updates.

Caching has been extensively studied in the area of operating systems, web information retrieval and content delivery networks. For example, the middle-tier data caching products developed by Oracle [3] was designed to prevent the database from being a bottleneck in content delivery networks. The main idea is to cache data outside of the database server to reduce database access load. QACHE differs from the traditional caches in that the items cached are queries instead of data. More importantly, the cached data is carefully organized for efficient access to minimize overhead. Recently, caching has also been applied to the area of mobile computing. The prevailing approach is to cache received answers at the client side for answering future queries. A Furthest Away Replacement (FAR) cache replacement policy was proposed in [9] where the victim is the answering object furthest away from the moving object's current location. Proactive caching for spatial queries [4] extends the caching granularity to per query object level. The compact R-tree presented in this work facilitates query processing when the cached item cannot answer the query. We use a similar approach in QACHE that treats both the database (query) index and the query data as objects for caching and manage them together to reduce the cache miss penalty.

3 Overview of QACHE

In this section we first briefly state the assumptions held when building the QACHE framework and provide an overview of the architecture and main components of QACHE. We then describe how QACHE handles location updates and maintains correct query answers.

3.1 Assumptions

The basic assumptions of QACHE are as follows:

1. Moving objects have the ability to determine their current location through GPS device. They also have the ability to communicate with the server periodically to report their location updates.
2. The only constraint on the motion of moving objects is that they are subject to a maximum speed.

3. All moving objects report their location updates to the server synchronously. Please note that this assumption simplifies our simulation and performance analysis, but is not necessary for QACHE to function correctly.
4. The queries stored in the database are indexed using spatial indices, such as the R-tree [1, 5].

3.2 System Architecture

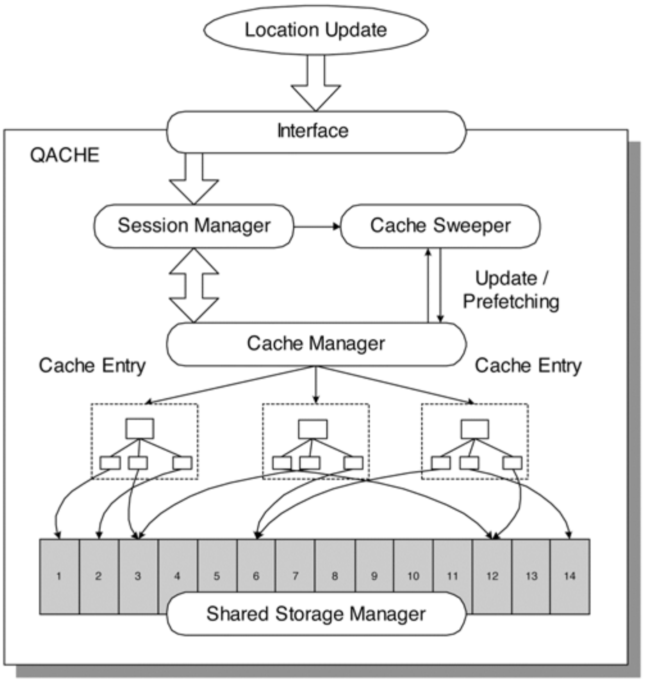

Fig. 2. System Architecture

As illustrated in Figure 2, QACHE has five main components: an interface that accepts location updates from moving objects, a *session manager* that manages the safe distance for each connected session (moving object), a *cache manager* that manages the cached contents for selected sessions, a *shared storage manager* that actually stores the spatial queries loaded from the database, and *a cache sweeper* that evicts invalid entries and prefetches new entries into the cache.

- **Session manager:** The session manager maintains a *look-up directory* that keeps, for each moving object, current location, a reference location called the base location, the safe distance from the reference location which is defined as the distance to the closest query region [7] and meta information about the corresponding cache-entry (such as cache-footprint if relevant). This look-up directory is indexed for efficient access. For objects that do not match any non-spatial predicates in any query the safe distance is set to infinity. The session manager could also maintain a location-logger that records all location updates and has them flush back to the database periodically.
- **Cache manager:** For each selected moving object, its cache entry consists of all relevant query regions in true detail/approximation. The cache manager manages such entries for moving objects whose safe distance does not exceed the cache-footprint. Due to memory constraints, the cache manager may create cache entries only for a subset of such moving objects based on their probability of being relevant to a query as described in Section 4.
- **Shared storage manager:** While the cache manager maintains cache entries for selected moving objects on a per session basis, it does not store the actual cached queries. Instead, all cached queries are managed by the shared storage manager to avoid duplication and thus save memory space. This is because a single query may be interested in multiple moving objects and hence may be cached more than once in QACHE. When a cache entry is accessed from the cache manager, a pointer is provided to visit the shared storage manager where the actual query is stored.
- **Cache sweeper:** The purpose of the cache sweeper is to refresh cache entries, evict invalidated cache entries and prefetch new entries that are not currently in the cache manager. Cache sweeper may refresh a cache entry for a moving object as it approaches boundary of the cache-footprint (refer to Section 4.2) and prefetch those prospective queries into QACHE. The refreshed/prefetched cache entry will center on the latest location of the moving object, i.e., within $D_{\max}$ distance from the latest location. Note that although prefetching introduces extra accesses to the database server, the operation is performed asynchronously thus the disk access is not on the critical path for query evaluation. Instead, when the prefetched queries do need to be evaluated against the next location update, no database access is necessary because those queries are already in QACHE thanks to prefetching. The cache sweeper can be implemented as background process that operates cooperatively with the cache manager.

3.3 Processing Location Updates

Figure 3 illustrated how QACHE handles location updates. When a location update from a moving object is received, the session manager first examines its look-up directory and checks whether the moving object is a new session. If so, the moving object is registered to the session manager, and the location and maximum speed of this moving object are used to query the database server for query evaluation and safe distance calculation. The safe distance calculated is then inserted into the look-up directory for future updates. If the calculated safe distance is less than the cache-footprint for the moving object, the corresponding cache-entry is created and inserted into the cache manager. On the other hand, if the location update is from an existing session, the session manager first examines its lookup directory and checks whether the moving object is still in its safe distance. If so, nothing needs to be done. Otherwise, the corresponding cache-entry is accessed to decide if this moving object has entered any query region. Note the cache entry has query regions in true detail or in approximate form.

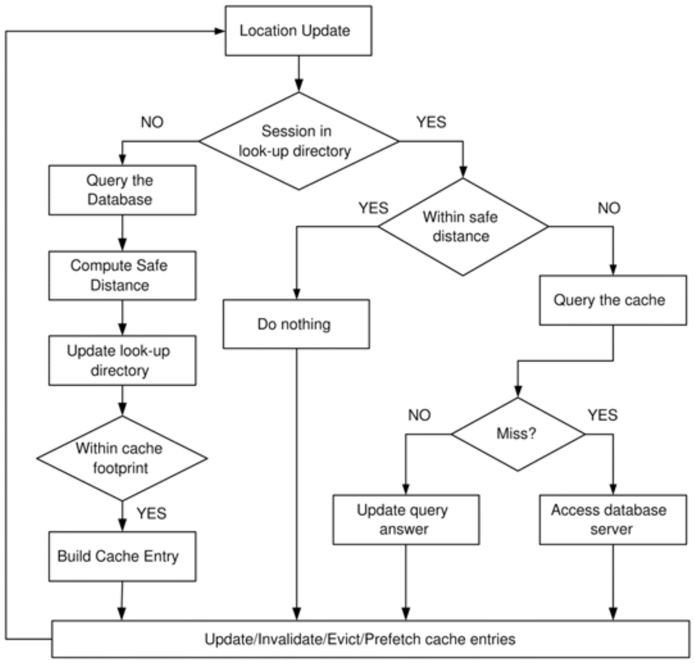

Fig. 3. Handling Location Updates in QACHE

For all true-detail query regions that the moving object matches, the query results are propagated to the application. For the matching approximate query regions, additional processing is performed in the database tier. This database processing is also required when a cache entry is missing (due to memory constraints, or invalidation by the cache sweeper).

In summary, when a new query is registered to the system, it is initially stored in the database and evaluated against all moving objects in the look-up directory of the session manager. A cache entry may be created, or an old cache entry may be replaced by the cache sweeper.

4 Design and Implementation of QACHE

This section elaborates on the design and implementation of three key components of QACHE, i.e., session manager, cache manager, shared storage manager. We describe: (1) how session manager maintains the safe distance for each moving object; (2) how cache manager selects moving objects and maintains a cache entry for each selected object to support efficient evaluation on location updates; and (3) how cached items are managed by shared storage manager and shared across selected moving objects to avoid duplication.

4.1 Maintaining the Safe Distance

The safe distance is the minimum distance within which a moving object will not enter any query region. Location updates of a moving object that are not beyond the safe distance need not be evaluated against any query, which indicates that the safe distance can serve as a filter in query processing.

When a moving object first registers to the application server, an initial safe distance is calculated for it by performing a nearest neighbor search on queries from the database server; the safe distance is then stored in the look-up directory of session manager. When a cache entry is created for this moving object, depending on the cache replacement policy such as LRU, the new safe distance must be recalculated and updated by the cache sweeper that mediate between the session manager and the database.

4.2 Building a cache entry

For each moving object, its corresponding cache entry (if presents) stores selected queries that are interested in the object. The selection of queries is decided by: the *QACHE refresh period* (QRP), i.e., the time interval between two consecutive cache updates, the maximum speed of the moving object $V_{\max}$ and the cache entry size B, i.e., the maximum number of items that can be stored in each cache entry.

QACHE attempts to cache queries within the cache-footprint of the moving object. Cache-footprint is described by a maximum distance $D_{\max}$ (see Eq. 1):

$$D_{\max} = V_{\max} \times QRP \tag{1}$$

Ideally, any query within distance $D_{\max}$ to the moving object should be cached since the moving object is very likely to enter the query region before the next cache refreshing. However, if the number of such queries exceeds the maximum size B of each cache entry, QACHE can't possibly cache all queries in full detail and has to aggregate some of them. Based on our assumption 4 in Section 3.1, queries are indexed using an R-tree in the database and hence the internal nodes of the R-tree can be used as an approximation of query aggregation.

As a consequence, each cache entry with a capacity of B stores two categories of items: (1) query regions that are stored in true detail: any moving object that satisfies such query regions is a *true-positive* match. A hit on this cached item indicates the moving object is a query answer; (2) query regions that are stored using *approximations*: any moving object that satisfies any such query approximations could be a *false-positive*. Additional processing needs to be done for such queries in the database. Moving objects not intersecting either category of regions is a *true-negative* and no further processing is required. This multi-category-based filtering serves as the backbone for the performance of QACHE in pending query evaluation.

To efficiently process location updates, QACHE organizes the cached items of each cache entry using an in-memory R-tree, i.e., the content of each cache entry is the internal nodes of the R-tree, while the actual cached items are managed by the share storage manager (please refer to Section 4.3). The algorithm used in QACHE for the construction of a cache entry is presented below. The algorithm starts by descending the query-index tree in the database from root and recursively explores child nodes that may contain eligible objects. A *priority queue* stores all nodes that are within distance $D_{\max}$ of the moving object. When a node is met, its children are enqueued; when a query object is met, it is added to a *query list* given that the non-spatial criteria of the query are also satisfied. This process terminates when the total size of the priority queue and the query list reaches the cache entry capacity B, or when the priority queue becomes empty. The query list stores all queries that are explicitly cached and the priority queue stores all cached nodes that aggregate the rest of eligible queries.

Algorithm 1 Create a cache entry

```
Input: Query-index tree in the database R, location $\{x,y\}$, maximum speed $V_{max}$, $QRP$,cache
    entry size $B$
Output: Cache tree $R$
 1: Priority queue Q $\Leftarrow \emptyset$, query list L $\Leftarrow \emptyset$
 2: $D_{max} \Leftarrow V_{max} \times QRP$
 3: Enqueue($root$(R))
 4: while Q not empty AND $(Q.size() + L.size()) < B$ do
 5:    $e \Leftarrow dequeue(\mathbf{Q})$
 6:    if $e$ is a leaf node of R then
 7:       c $\Leftarrow$ e's closest child to {x, y}
 8:       $e \Leftarrow e - c$, // remove the child from the node
 9:       if $distance(c, \{x,y\}) \leq D_{max}$ AND Expression($c$) evaluates to 'true' then
10:          Enqueue($c$)
11:       end if
12:       if $e$ not empty AND $distance(e, \{x,y\}) \leq D_{max}$ then
13:          Enqueue($e$)
14:       end if
15:    else if $e$ is an internal node of R then
16:       c $\Leftarrow$ e's closest child to {x, y}
17:       $e \Leftarrow e - c$, // remove the child from the node
18:       if $distance(c, \{x,y\}) \leq D_{max}$ then
19:          Enqueue($c$)
20:       end if
21:       if $e$ not empty AND $distance(e, \{x,y\}) \leq D_{max}$ then
22:          Enqueue($e$)
23:       end if
24:    else {// $e$ is a qualifying query object}
25:       Add $e$ to LC
26:    end if
27: end while
28: Create R-tree $R$ from objects in $Q$ and L
29: return $R$
```

For example, in Figure 4, O is the current location of a moving object for which a cache entry is to be constructed. I is the root of the database R-tree with three children: I_1, I_2, and I_3. The circle illustrates the region that is within distance $D_{\max}$ to the moving object; queries that intersect this region should be explicitly or implicitly cached. Suppose that the cache entry size B is set to five. I is first dequeued, it's three children are then examined. Only I_1 and I_2 are enqueued because they are within $D_{\max}$ (Step 2). I_1 is then dequeued and its three children are added to the query list (Step 3, 4, 5). So far, four items are cached: Q_1, Q_2, Q_3 in the query list and I_2 in the priority queue. Subsequently I_2 is dequeued; its closest child Q_5 is added to the query list, while $Q_4$and $Q_6$are re-aggregated to a new node which is put back to the priority queue (Step 6). At this time we have exactly five items in total: Q_1, Q_2, Q_3 and Q_5 in the query list and $Q_4 + Q_6$ in the priority queue. These five items are then used to build a in-memory R-tree for the cache entry

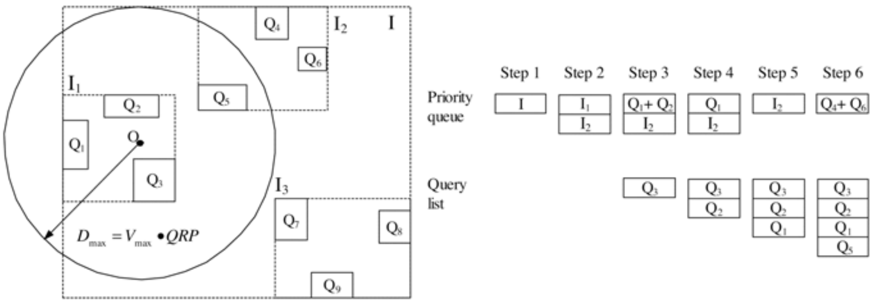

Fig. 4. Example: Create a Cache Entry

4.3 Sharing Cache Contents Among Sessions

One novelty of QACHE is its session-wise granularity. When a location update is received, it need not be evaluated against all queries in the cache because queries that are interested in this particular moving object are already selected into its own cache entry. Unlike conventional approach, this prevents the non-spatial predicate of a query to be evaluated every time: the non-spatial predicate is evaluated exactly once when the cache entry is create, while the spatial predicate may be evaluated on every subsequent location update.

However, this session-wise granularity has its own deficiency: potential waste of memory space. A query may be interested in multiple moving objects, and hence may be cached in multiple cache entries. To solve this problem, we implemented a shared storage manager that actually holds the data cached in memory. Each cache entry only stores pointers to the corresponding slots in the shared storage manager. This guarantees that only one copy of each query/node is kept in memory at any time.

The shared storage manager is implemented as a hash table that is a tuple of index, data, and a reference counter. When a query or an intermediate node is selected for caching, only its index is stored in the cache entry. The actual data, i.e., the geometry of a query or the minimum bounding box (MBB) of an intermediate node, will be stored in an entry in the storage manager based on the index. During a query evaluation, the storage manager identifies the location of the data using a hash function and the ID of the query/node as a hash key. When the storage manager receives a request for a data insert, it first checks whether the data already exists. If so, the storage manager increases the reference counter by one; otherwise, a new entry is created. When a cache entry is evicted, all queries/nodes cached will have their reference counter decreased by one. When a counter becomes zero, the actual data can be safely removed from the shared storage manager.

5 Performance Evaluation

We have built a simulation environment for QACHE with the *Java* programming language. We compare QACHE with two other approaches: (1) the naive approach where location updates are directly sent to the database server and evaluated every time; (2) the *safe distance approach* (SD) where only safe distance is used to reduce number of query evaluation. We examined the number of disk accesses to the database (R-tree) as well as storage requirement of each approach. With the experimental data, we also analyzed the processing time of different approaches to demonstrate the efficiency of QACHE.

5.1 Simulation Setup

Using our own data generator modified from the GSTD tool [11], a data set is generated that simulates a mobile environment where N objects moves following the *Random Waypoint Model* [2], a well accepted model in the mobile computing community. Each object starts at a randomly selected location in the region of [0...1, 0...1], moves for a period randomly generated between [0, *QRP*] at a speed randomly selected between [0, *QRP*], and sends its new location to the application server at time *QRP*; after this the same process repeats. When an object hits the boundary, its moving direction is adjusted to guarantee constant number of moving objects in the simulation space. The query workload contains 1000 queries that are evenly distributed in the simulation space; currently only static range queries are considered.

In our simulation, new location updates from all *N* objects are collected at the same time and processed before the next round of location updates arrives. Our simulation processes 5000 rounds of location updates. All experiments were performed on a 3.0 GHz Pentium 4, 1 GB memory workstation running Windows XP SP2.

5.2 Disk Access and Memory Requirement

We conducted three sets of experiments where the number of moving objects ($N_{m.o.}$) grows from 1000 to 10000. In each set, we varied the number of cache entries ($N_{c.e.}$) from 5% to 20% of ($N_{m.o.}$). The cache entry capacity B, i.e., the number of cached items in each entry, is set to 10. A fixed number of queries (1000) are organized in the database server as an R-tree, the size of which is 640KB excluding the non-spatial predicates. For the three approaches (in short, naive, SD, and QACHE), we collected the expected number of disk page accesses ($E(dpa)$) to the database index R-tree on every round of location updates. We also recorded the memory requirement and the cache hit ratio when applicable. The performance of QACHE and the other two approaches are presented in Table 1.

Table 1. Disk access and memory requirement of the three different approaches

$N_{m.o.}$		1000			5000			10000		
$N_{c.e.}$		50	100	200	250	500	1000	500	1000	2000
$E(dpa)$	naive	3626	3626	3626	18255	18255	18255	36191	36191	36191
	SD	374	374	374	2051	2051	2051	4009	4009	4009
	QACHE	137	49	2	754	193	7	1310	353	15
Cache hit ratio	naive	-	-	-	-	-	-	-	-	-
	SD	-	-	-	-	-	-	-	-	-
	QACHE	56%	85%	99%	54%	88%	99%	56%	89%	99%
Memory requirement (Byte)	naive	-	-	-	-	-	-	-	-	-
	SD	4000	4000	4000	20000	20000	20000	40000	40000	40000
	QACHE	11283	18035	24200	52074	73773	93805	93850	124740	150021

Compared to the safe distance approach, QACHE reduces by $E(dpa)$ at least 63%. In each set of experiments, $E(dpa)$ for the other two approaches remains constant for a given number of moving objects, but decreases significantly for QACHE when the number of cache entries is decreased. When $N_{c.e.}$ is 20% of $N_{m.o.}$, the expected disk page accesses is almost negligible. This is because almost all query evaluation can be completed by QACHE and only a few disk page accesses are generated from false-positive hits in the cache.

Another major observation from Table 1 is that QACHE is scalable in terms of memory storage requirement. We recorded the total number of bytes required by the look-up directory, cache manager and the shared storage manager; the results indicate that the total memory requirement does not grow in proportion to the number of moving objects. Moreover, considering the total size of query R-tree in the database, QACHE is highly efficient in utilizing memory space and providing a high hit ratio.

5.3 Processing Time

While the number of disk accesses is an important criteria when evaluating the effectiveness of QACHE, a quantitative analysis is necessary to decide the exact performance improvement. In this section we demonstrate the overall speed up that QACHE can achieve in query evaluation over the naive approach and the safe distance approach. In our analysis, the following terms are frequently used: (1) disk page access time T_{disk}; (2) memory access time T_{mem}; (3) query evaluation time T_{eval}; and (4) the height of the query R-tree in the database H_Q. For simplicity, we assume that an access to the query R-tree in the disk reads $0.75xH_Q$ disk pages. We also assume that the cache entry R-tree has a fan out of 2, thus the in-memory cache R-tree has a height of $\log_2 B$. The average response time to a location update can be calculated as follows:

- Naive approach:

$$T_{naive} = 0.75 \times H_Q \times (T_{eval} + T_{disk}) \tag{2}$$

- Safe distance approach: assuming that in each round of location updates, 10% are beyond the safe distance so that database accesses are required, the average response time is:

$$T_{sd} = T_{mem} + 0.1 \times 0.75 \times H_Q \times (T_{eval} + T_{disk}) \tag{3}$$

- QACHE: assuming that $N_{c.e.}$ is 20% of $N_{m.o.}$, then only 0.2% of the location updates will result in database access (see Table 1), the average response time is:

$$T_{qache} = T_{mem} + 0.75 \times \log_2 B \times (T_{eval} + T_{disk}) + \\ 0.002 \times 0.75 \times H_Q \times (T_{eval} + T_{disk}) \tag{4}$$

Based on a reasonable estimation of the relative parameters presented in Table 2, QACHE achieves a 498 times speed up over the naive approach and a 50 times speed up over the safe distance approach.

Table 2. Estimations of the required time for each operation

$T_{mem}(ns)$	$T_{eval}(ns)$	$T_{disk}(ns)$	H_Q	B
100	100	5000000	10	10

6 Conclusions

We have described and evaluated QACHE, a novel query caching framework for LBS systems. By caching spatial queries for appropriate moving objects on the application tier, a significant amount of database accesses can be eliminated, resulting in a dramatic performance improvement of LBS. We examined several important implementation issues and proposed effective solutions to them for QACHE to be deployed in real LBS systems. We compared QACHE with existing solutions based only on safe distance. Our simulation results indicate that with the cache capacity 20% of total number of moving objects, and the memory requirement ranging from 3% to 20% of the query R-tree size in database (depending on the number of moving objects), QACHE is capable of eliminating 99% of the disk accesses. On real LBS systems, this memory requirement is totally affordable. Further more, our quantitative analysis shows that QACHE achieves a 50 times speed up over the safe distance approach and a 498 times speed up over the naive approach where all location updates are directly processed in the database.

References

1. Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In: SIGMOD Conf, pp 322–331
2. Broch J, Maltz DA, Johnson DB, Hu Y-C, Jetcheva J (1998) A performance comparison of multi-hop wireless ad hoc network routing protocols. Mobile Computing and Networking:85–97
3. Greenwald R, Stackowiak R, Stern J (2001) Oracle Essentials. O'Reilly &Associates Inc., CA
4. Hu H, Xu J, Wong WS, Zheng B, Lee DL, Lee WC (2005) Proactive caching for spatial queries in mobile environments. In: ICDE, pp 403–414
5. Kanth KVR, Ravada S, Sharma J, Banerjee J (1999) Indexing medium-dimensionality data in oracle. In: SIGMOD Conf, pp 521–522
6. Mokbel MF, Xiong X, Aref WG (2004) SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In: SIGMOD Conf, pp 623–634
7. Prabhakar S, Xia Y, Kalashnikov DV, Aref WG, Hambrusch SE (2002) Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. IEEE Trans Computers 51(10):1124–1140
8. Kothuri R, Beinat EGA (2004) Pro Oracle Spatial. Apress
9. Ren Q, Dunham MH (2000) Using semantic caching to manage location dependent data in mobile computing. In: MOBICOM:210–221
10. Schiller J, Voisard A (2004) Location-Based Services. Morgan Kaufmann Publishers, CA
11. Theodoridis Y, Silva JRO, Nascimento MA (1999) On the generation of spatiotemporal datasets. In: SSD, pp 147–164
12. Yalamanchi A, Kanth Kothuri VR, Ravada S (2005) Spatial Expressions and Rules for Location-based Services in Oracle. IEEE Data Eng Bull 28(3): 27–34
13. Yalamanchi A, Srinivasan J, Gawlick D (2003) Managing expressions as data in relational database systems. In: CIDR