Caching Complementary Space for Location-Based Services

Ken C.K. Lee[1], Wang-Chien Lee[1], Baihua Zheng[2], and Jianliang Xu[3]

[1] Pennsylvania State University, University Park, USA
{cklee, wlee}@cse.psu.edu
[2] Singapore Management University, Singapore
bhzheng@smu.edu.sg
[3] Hong Kong Baptist University, Hong Kong
xujl@comp.hkbu.edu.hk

Abstract. In this paper, we propose a novel client-side, multi-granularity caching scheme, called "*Complementary Space Caching*" (CS caching), for location-based services in mobile environments. Different from conventional data caching schemes that only cache a portion of dataset, CS caching maintains a global view of the whole dataset. Different portions of this view are cached in varied granularity based on the probabilities of being accessed in the future queries. The data objects with very high access probabilities are cached in the finest granularity, i.e., the data objects themselves. The data objects which are less likely to be accessed in the near future are abstracted and logically cached in the form of complementary regions (CRs) in a coarse granularity. CS caching naturally supports all types of location-based queries. In this paper, we explore several design and system issues of CS caching, including cache memory allocation between objects and CRs, and CR coalescence. We develop algorithms for location-based queries and a cache replacement mechanism. Through an extensive performance evaluation, we show that CS caching is superior to existing caching schemes for location-based services.

1 Introduction

Due to the rapid advances in wireless and positioning technologies, location-based services (LBSs) [1] have emerged as one of the killer applications for mobile computing. To improve the access efficiency and alleviate the contention of limited wireless bandwidth in mobile environments, data caching techniques are particularly important for LBSs.

Conventional caching techniques cache a portion of a database in units of tuples or pages. Due to the lack of data semantics, clients cannot be sure whether the cached data alone can sufficiently satisfy some complex queries, forcing them to submit requests to the server even if the answers are completely available in the cache. Semantic caching addresses this problem by caching query results along with their corresponding queries (which serve as the semantic descriptions of the cached query results) [2, 3, 4]. Thus, a query and its result form a semantic

Y. Ioannidis et al. (Eds.): EDBT 2006, LNCS 3896, pp. 1020–1038, 2006.

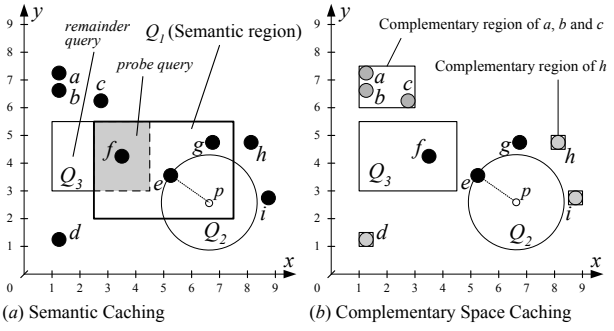

Fig. 1. Semantic caching and Complementary space caching

region. By consulting cached semantic regions, a new query can be decomposed into a *probe query* which can be answered locally by the cache and a *remainder query* which is only answerable by the server. If a query is fully covered by cached semantic regions, no contact with the server is needed.

However, the representation of semantic regions is highly query-dependent. If a query is of different type from the queries captured by semantic regions, the cached data objects cannot be reused. Besides, because clients' knowledge about data objects is constrained by cached semantic regions, the clients are unable to determine whether there are objects beyond the cached semantic regions. Therefore if a query is partially covered by semantic regions, remainder queries (i.e., uncovered portions of the query) must be formed and submitted to the server to retrieve possibly missing objects. The following example (as shown in Fig. 1(a)) illustrates the above described deficiencies.

Example 1. Suppose that a database server contains 9 objects, namely, 'a' through 'i'. A client with an empty cache submits a window query, Q_1, to the server. A result set of three objects $\{f, e, g\}$ is returned and cached along with the query window as a semantic region. Later, a nearest neighbor (NN) query, Q_2 (with a query point p), is issued. Due to incompatibility between window and NN queries, the cached semantic region cannot answer the query even though the result (i.e., e) is in the cache. Consequently, Q_2 is submitted to server and e is retrieved again. Later, the client issues another window query, Q_3, which is partially covered by the semantic region. Thus, a remainder query is submitted to the server even though this query actually retrieves no object. ■

These deficiencies are due to the lack of a *global view* of data in the cache. For a cache designed to support various kinds of queries, it is desirable to maintain certain auxiliary location information that provides a global view of all data objects in the database. Motivated by this observation, we propose a novel multi-granularity data caching scheme for mobile clients called *Complementary Space Caching* (CS caching). The CS caching distinguishes itself from other caching schemes by having a global view of the whole dataset. Different portions of this cached view have varied granularity based on the probabilities of corresponding

data objects to be accessed in the future queries. The data objects with very high access probabilities are cached in the finest granularity, i.e., the actual data objects. Those data objects less likely to be accessed in the near future are not physically cached, rather are abstracted and logically cached in the form of *complementary regions (CRs)* in a coarse granularity. In our design, CRs present auxiliary location information regarding to *missing objects*, i.e., those objects in the server but not kept in the cache. This auxiliary information can facilitate the local processing of various location-dependent queries and alleviate unnecessary queries to the server. Fig. 1(b) shows that same scenario as Example 1 except that CS caching is adopted (where black dots are objects and rectangle boxes are CRs). Since the result of Q_1 is cached, object e in the cache can answer Q_2 because no other object or CR is closer than e to p. Q_3 finds only object f inside the query window so no additional objects are needed from the server.

Due to limited cache memory, there is a trade-off between keeping objects and keeping CRs in the cache. Storing more data objects in the cache can potentially provide a higher cache hit rate but reduce the precision of the auxiliary location information in CRs (because more CRs need to be merged and stored in a coarse granularity in order to make rooms for objects). This could lead to more *false misses*, i.e., a query finds a CR for potentially answer objects but no objects are returned. On the other hand, taking more cache memory to maintain fine-granularity CRs will reduce the number of data objects cached physically and thus reduce the cache hits. The design of CS caching strives to optimize the cache memory allocation for physical data objects and CRs in order to achieve a high cache hit rate and a low false miss rate (which leads to the excellent performance in terms of response time and bandwidth consumption). In this paper, we explore several design and system issues of CS caching. We develop and implement algorithms for processing window, range, and k nearest neighbor queries based on CS caching, and develop a very efficient cache replacement mechanism for cache maintenance. Through comprehensive experiments based on simulation, we validate our proposal and show that CS caching is superior to existing caching schemes for location-based services.

The rest of this paper is organized as follows. Section 2 gives an overview of the CS caching model and reviews related work. Section 3 describes the query processing in CS caching. Section 4 discusses CR coalescence, an important technique for reducing transmission overhead. Section 5 describes the cache management. Section 6 reports the simulation result. At last, Section 7 states our future directions.

2 Complementary Space Caching

In this section, we first briefly review the R-tree and the notion of minimum bounding boxes that we adopt to represent complementary regions. We then describe the CS caching model and discuss some relevant research.

2.1 Preliminaries

In many spatial databases, objects are very often indexed using R-tree [5] or its variants for its efficiency and wide acceptance. In R-tree, objects close in space are clustered in a leaf node represented as a *minimum bounding box (MBB)*. Nodes are then recursively grouped together following the same principle until the top level, which consists of a single root. To process a query, a search algorithm starts traversal at the root and recursively visits index nodes and objects. By simply examining MBBs of index nodes or objects (e.g., by checking the intersection of the query and an MBB for window queries or the *mindist* between a query point and an MBB for kNN queries [6]), whether the enclosed objects are candidates of the query can be quickly determined. If an MBB does not satisfy the query requirement, the corresponding subtree (i.e., the enclosed group of objects) can be safely discarded from further investigation. The query traversal ends when all objects are retrieved and all irrelevant subtrees are pruned.

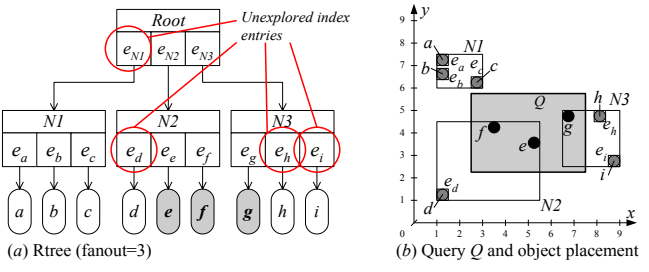

Fig. 2. R-tree example

Fig. 2(a) depicts an R-tree (with a fanout of 3) that has 3 leaf nodes, labeled $N1$, $N2$ and $N3$, and 9 objects labeled 'a' through 'i'. The corresponding object placements are shown in Fig. 2(b). Suppose a window query Q is evaluated. It first traverses the root and skips its child entry e_{N1} whose MBB does not overlap with the query. Next, $N2$ is explored. Entry e_d is not explored since it is outside Q. Then objects e and f are collected. Similarly for $N3$, object g is retrieved while e_h and e_i are not explored. Finally, objects e, f and g are collected as the query result while the unexplored entries are $\{e_{N1}, e_d, e_h, e_i\}$.

It can be observed from the above index traversal that the unexplored entries exactly represent the complement set of objects to queried objects. In other words, both queried objects and those unexplored entries' MBBs cover the entire data space. If such information is available in mobile clients, a new query, Q', of any type can be supported and resolved at the client side by simply examining the previous queried objects and by checking whether the areas covered by MBBs need to be further explored (i.e., by sending requests to the server). This observation inspires the ideas we proposed for the design of CS caching.

2.2 Complementary Space Caching Model

We assume that the server is stateless and a point-to-point communication channel is established between the server and a client. We also assume that the database is indexed by R-tree. To simplify our discussion, we assume all updates occur in the server and the update is infrequent. The issue of cache coherence is out of scope of this study and will be the extension of this work.

Formally, we consider a database at the server composed of a set of objects, $\mathcal{O}$. All object locations (x-,y-coordinates) constitute a bounded geographical space, $\mathcal{S}$. Data objects $O \subseteq \mathcal{O}$ residing in a subspace $S \subseteq \mathcal{S}$ can be determined by a function, m, i.e. $m(S) \rightarrow O$[1]. Conversely, given a set of objects O, the corresponding (minimal) subspace S can also be determined.

The CS cache C is defined as (O, R), where O is a set of cached objects and R is a set of subspaces that is *complementary regions* (CRs). The CRs are presented in a form of MBBs[2]. Initially, a client cache is empty and is initialized as $(\emptyset, \{\mathcal{S}\})$. After the first query is processed by the server, queried objects along with MBBs of unexplored entries are returned to the client and are cached. There is obviously an overhead for maintaining the MBBs in the client cache, but it is justifiable for the following reasons: (1) collection of unexplored entries and their MBBs via R-tree based query processing at the server is almost effortless; (2) individual MBBs are compact in size and thus do not consume a lot of bandwidth and cache memory; (3) the number of unexplored entries (and MBBs) is reasonably small since most of irrelevant data objects are pruned at high-level branches of the R-tree due to its nice clustering property; (4) It is only a one-time cost, which will be amortized over subsequent queries; and (5) as shown previously, keeping MBBs in the cache can effectively avoid sending unnecessary queries to the server. As to be discussed in Section 6, our evaluation demonstrates that the performance gain outweighs the overhead cost.

With both objects and CRs kept in the cache to preserve a global view of the dataset, query processing and cache management in CS caching behave differently from the conventional ones. Fig. 3 that continues the running example in Fig. 2(b) gives the overview of query processing and cache replacement. Suppose after Q, the cache content of a client becomes $(\{e, f, g\}, \{r_{N1}, r_h, r_i, r_d\})$ (where r_x is the MBB of e_x). Suppose the client moves and issues a query that covers r_{N1} (see Fig. 3(a)). The client explores r_{N1} by querying the server. Then an object c together with two MBBs, r_a and r_b, that are part of r_{N1} are received. Both are in a finer granularity and they represent other areas of current client interest. In that sense, query processing resembles as a *zooming-in* action that brings more details about queried area into the cache. Memory should be reclaimed to accommodate new coming objects and other finer CRs if the cache is full. Suppose an object g is chosen to be removed from the cache (see Fig. 3(b)). First, r_g, a CR for object g is introduced in the g's position (so that the global view is preserved) and then g is physically deleted. Further, if more free space is demanded, r_g, r_h and r_i, three closely located CRs, are coalesced into a single

[1] This function is logically supported by the database.

[2] If no ambiguity caused, we would use CR and MBB interchangeably.

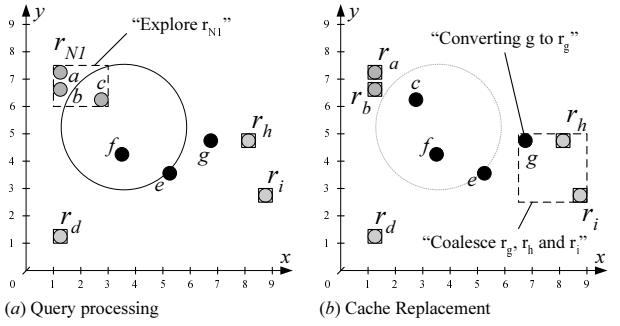

Fig. 3. Overview of query processing and cache replacement with CS caching

CR that is cached in a coarser granularity. This cache replacement is analogous to a *zooming-out* action that removes the details of an area that is currently not interested by the client.

In order for query processing and cache replacement to maintain a global view of the dataset, CS caching reinforces the following integrity requirements:

Requirement 1 Full dataset coverage. *At any time, the union of cached objects, O, and missing objects captured by the set of CRs, R, must equal* $\mathcal{O}$*, formally* $(\cup_{r \in R} m(r)) \cup O = \mathcal{O}$. ■

This integrity requirement assures that every missing object ($\notin O$) is captured by one of the CRs, $r \in R$. Based on R, a client can determine whether there are potentially missing objects for a query.

Requirement 2 No CR-object overlap. *A CR should not cover any cached object, formally* $\forall r \in R, m(r) \cap O = \emptyset$. ■

Requirement 3 No full CR-CR containment. *No CR is contained in another CR, formally,* $\forall r_i \in R, r_j \in R, i \neq j, m(r_i) \nsubseteq m(r_j)$. ■

The second requirement aims at reducing false misses. The third requirement eliminates redundant CRs. A CR is redundant if it is already covered by another CR and thus is safe to remove.

2.3 Related Work

As discussed earlier, semantic caching [2, 3, 4] is query-dependent and provides limited knowledge about the cached subspace. By fixed space partitioning, chunk-based caching [7] partitions the semantic space into *chunk*s, independent of query and object distribution. Every window query is mapped into a set of chunks. Query fetches chunks from the server if they are not in the cache, regardless of whether the chunks have objects or not. Without keeping an entire view of semantic space, chunk-based caching cannot support various kind of queries.

Similar to CS caching, proactive caching [8] supports a number of different types of spatial queries. It is important to note that these two caching schemes

are conceptually and functionally different. Proactive caching tightly adheres to R-tree, yet CS caching does not. Proactive caching maintains traversed index paths (a portion of index) and a set of objects below the cached index paths in the cache. The cached partial index enables a client to execute query processing algorithms as the server does. If a query needs to find any missing index nodes or objects, the query and all intermediate execution states are shipped to the server for remaining execution. The cached index path in proactive caching is the only means to access underlying objects so implicitly the index nodes are granted higher priority than objects to cache. This has an impact on cached hit rate because cached index nodes alone (without beneath objects) cannot make query locally answered. Besides, excessive bandwidth is taken to transmit index structures which in fact can be reconstructed using objects and CRs as shown in CS caching. Without the necessity to conform to the R-tree at the server, CRs can be flexibly coalesced and partitioned for optimizing the cache performance.

3 Location-Based Query Processing

With the global view maintained in the cache, a location-based query of any kind can be answered by reusing cached objects and exploring some involved CRs for missing objects from the server. Generally speaking, query processing with CS caching is a three-step procedure:

1. *Cache probing:* Qualified objects in the cache are collected as a tentative query result and CRs that could contribute to the query result are identified. The cache probing varies with different types of queries and will be discussed shortly in Section 3.1.
2. *Remainder query processing:* If no CR is identified for the query meaning the query is fully covered by the cache, the query processing terminates here. Otherwise, the missing objects in the identified CRs need to be requested from (and possibly checked by) the server. This will be discussed in Section 3.2.
3. *Cache maintenance:* After the remainder query is answered, the newly returned data objects and CRs are admitted to the cache. This invokes the cache maintenance operations such as cache replacement and CR coalescence that will be discussed in Section 5.

3.1 Cache Probing

In the following, we informally describe the cache probing for some typical location-based queries such as window, range, and k nearest neighbors (kNN) queries[3]. They are incompatible in nature but can be processed in a similar fashion using CS caching. The outputs of cache probing are cached objects in the answer set and CRs to be explored in the server via remainder query processing.

[3] A range query is specified by a query point and a radius.

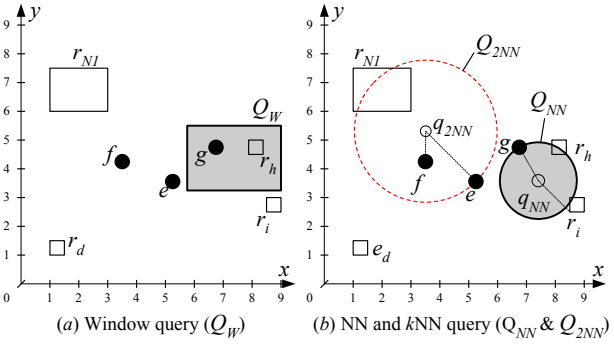

Fig. 4. Examples of lookup mechanism for various types of queries

Cache probing for a window (or range) query is pretty straightforward. The client scans the cached objects and CRs to return those overlapped by the query window (or range). Figure 4(a) shows an example of a window query, Q_W. In this example, cached object, g, and CR, r_h, are identified. Without an explicit search range, cache probe for an NN query expands a search range from a query point outwards until one object is touched. Then, all CRs within the search range are identified for further exploring. The extension to handle kNN query is straightforward by extending the search range to first k covered objects. Fig. 4(b), the client finds objects f and e, the two closest objects to q_{2NN} of the query Q_{2NN}. The CR r_{N1} overlapped with the vicinity circle across e is identified for further exploring.

3.2 Remainder Query Processing

A remainder query is submitted to the server to retrieve missing objects if some CRs are identified for a query. Besides, refined CRs (i.e., MBBs of those entries inside submitted CRs but not explored by a query) may be returned. One of the primary issues in processing remainder queries is "how to express the query" which has a major impact on the processing cost (in terms of response time and bandwidth overhead) and the quality of cached information. We examine two possible approaches: 1) CRs only, and 2) Query+CRs.

The first approach is to submit only the identified CRs treated as window queries in the server. As shown in Fig. 5(a), a query, Q, overlaps with three objects, e, f, g and three CRs, r_{N1}, r_h, r_i. The remainder query in this approach is expressed as (r_{N1}, r_h, r_i). Because r_{N1} covers a large area outside Q's range, some extra objects may be returned to the client. Even worse, they would not be used at all eventually. This approach consumes minimal uplink bandwidth.

The second approach is to submit the original query along with the identified CRs. The CRs are used as filters for processing of Q in the server. When the R-tree index is traversed, only the branches overlapped with the CRs are further explored. The MBBs unexplored by Q and intersecting with the CRs are also returned (as refined CRs in the original CRs) along with qualified data objects to the client. As shown in Fig. 5(a), a remainder query in this approach is

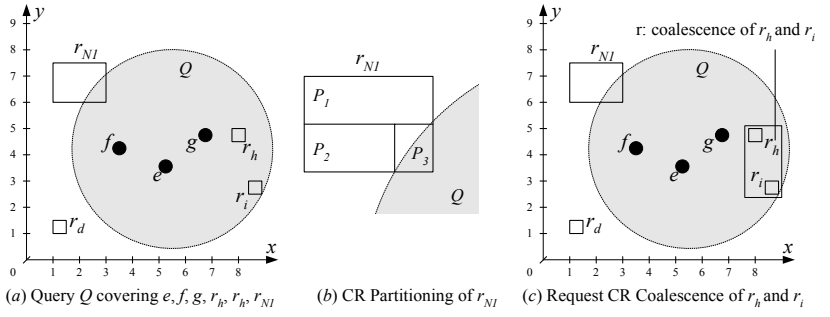

Fig. 5. Remainder query

expressed as $Q+(r_{N1}, r_h, r_i)$. With this approach, the downlink cost is expected to be reduced because a precise set of required objects and a smaller number of CRs are downloaded.

Delivering a large number of fine-granularity CRs back to the clients may incur an excessive downlink overhead (and the additional energy consumption of clients). To address this issue, a client can partition CRs if they are only partially covered by a query during remainder query preparation. An example is depicted in Fig. 5(a), CR r_{N1} is partially covered by a query Q and thus can be partitioned into three parts, namely, P_1, P_2 and P_3 (as shown in Fig. 5(b)). The portion, P_3, enclosing the overlap between r_{N1} and Q, is taken to formulate a remainder query. Thus, r_{N1} is removed while P_1 and P_2 are retained as CRs in the cache. This partitioning may result in some savings of download overhead because the number of refined CRs in P_3 is expected to be smaller than those in r_{N1}. However, a low precision CR will be resulted like P_1 and P_3 having not exact bounding box of enclosed objects. It is also probable that the partitioned CRs have no missing object inside. Examining them definitely causes false misses.

On the other hand, a large number of CRs could be covered by a query. Submitting all those individual CRs to the server incurs a high upload cost. Thus, CRs can be merged into a few coarse CRs. This merging of CRs is called *CR coalescence*. As shown in Fig. 5(c), r_h and r_i are coalesced to a coarse CR r. r_{N1} and r can be submitted instead, i.e., the remainder query is $Q+(r, r_{N1})$. However, the newly formed CR may overlap with some answer objects already found in the cache. For example, further coalescence of r and r_{N1} may form a larger CR, r', that overlaps with cached objects e, f, and g. Using r' as request CR will redundantly fetch these already cached objects.

To tackle this problem, we supplement IDs of overlapped cached objects in the remainder query as a result filter that removes the objects already cached from the downlink. Then, the new remainder query becomes $Q+(r')+(\{e, f, g\})$ and is sent to the server. It raises a question if an expression $Q+(r')+(\{e, f, g\})$ saves more uplink bandwidth than $Q+(r, r_{N1})$ or other expressions else. This is an optimization issue in CR coalescence, which will be discussed in Section 4. Finally, for NN or kNN query processing, instead of exploring all potential CRs

covered by the conservative vicinity circle as described above, we can take an incremental approach to explore the identified CRs one by one until the answer set is obtained.

4 CR Coalescence

CRs are essential information to transmit between the client and the server. An efficient CR coalescence algorithm is needed to condense those overly fine CRs to save bandwidth. In this section, we first devise a generic CR coalescence algorithm, based on which two specializations for coalescing CRs in request messages and in reply messages are derived.

4.1 Generic CR Coalescence Algorithm

Given a set of CRs, R, a coalescence algorithm selects n subsets of R, that is $R'_1, \cdots, R'_n$ ($R'_i \subseteq R$), to coalesce. Each R'_i is replaced by a newly formed CR called *coalescing CR*, denoted by $r_{R'_i}$, that is an MBB of all original CRs in R'_i. Hence, the coalescence operation on R can be described as $(R - \cup_i R'_i) \cup (\cup_i \{r_{R'_i}\})$.

The key issue here is to determine the optimal subsets of CRs to be coalesced. In order to tackle this selection problem, we first formulate a cost model. Every CR, r, bears a cost, $c(r)$, which measures the performance loss due to missing objects. The definition of cost function varies with the operation scenario (to be detailed later in this section). In general, the larger is the region, the higher is the cost, and the more is the potential performance loss. Therefore, after coalescing R'_i's ($i = 1, 2, \cdots, n$), the cost increase is:

$$\textit{total cost increase} = \sum_{1 \le i \le n} \Big(c(r_{R'_i}) - \sum_{r \in R'_i} c(r) \Big), \tag{1}$$

but the number of CRs is reduced by:

$$\textit{total CR saving} = \big(\sum_{1 \le i \le n} |R'_i| \big) - n \tag{2}$$

Given an expected CR saving, the optimal selection algorithm should minimize the cost increase. Here, we propose a greedy algorithm to choose CRs to coalesce until an application-dependent termination condition is met. The algorithm is outlined in Fig. 6. At each step, it selects the best pair of CRs to merge. The "best" means the least cost increase after coalescing the pair of CRs. A priority queue is used to keep track of the possible CR pairs. Initially, we determine the best counterpart for each CR in R and coalesce the best pair of CRs. After coalescence, the original CRs r_i and r_j are replaced with the coalescing CR, $r_{\{r_i, r_j\}}$; the CR saving and cost increase are 1 and $c(r_{\{r_i, r_j\}}) - c(r_i) - c(r_j)$, respectively. Based on $r_{\{r_i, r_j\}}$, a new candidate pair is inserted to the queue. The algorithm continues until the termination condition is satisfied. The termination condition can be specified by limiting the number of CRs coalesced so

```
Algorithm. GenericCRCoalescence(R: a set of CRs)
Input/output: R: a set of CRs;
Local:        Q: priority queue;
Begin
1    foreach r ∈ R do
2      find r's best counterpart CR, r' from R − {r};
3      push (r, r', anticipated cost increase) into Q;
4    while (termination condition is not satisfied) do
5      pop (r_i, r_j, anticipated cost increase) from Q;
6      r ← coalesce(r_a, r_b);
7      replace r_i and r_j with r in R;
8      find r's best partner CR, r' from R − {r};
9      push (r, r', anticipated cost increase) into Q;
10   output R;
End.
```

Fig. 6. Generic CR coalescence algorithm

that the remaining number of CRs can be controlled or by setting a threshold on cost increase metric that guarantees the CR fineness. In the following two subsections, we shall derive specific coalescence techniques for coalescing CRs in requests (request CRs) and CRs in replies (reply CRs).

4.2 Client Request CR Coalescence

In Section 3.2, we briefly discussed the issue of request CR coalescence and raised the question about what CRs should be coalesced. Here, we address this problem with our generic coalescence algorithm described above. Let r be a CR. We set the cost of r, $c(r)$, as the number of objects covered by r. As the size of remainder query is our main concern in coalescing request CRs, we aim at maximizing the overhead reduction specified below:

$$\begin{aligned} \textit{overhead reduction} &= \textit{total CR saving} \times \textit{CR size} - \\ &\quad \textit{total cost increase} \times \textit{object ID size}. \end{aligned} \tag{3}$$

This expression considers the volume saved by CR coalescence (CR saving) and the overhead of including additional object IDs (cost increase). Reconsider the situation in Fig. 5(c), r_h and r_i can be coalesced to form a coalescing CR, r, with 1 CR saved and no object included, i.e., $c(r) = 0$. Further, coalescing r and r_{N1} into r' has 1 more CR saved but covers three objects, i.e., $c(r') = 3$. As a CR and an object ID respectively take 16 bytes and 4 bytes, the total overhead reduction for taking r' and $\{e, f, g\}$ is $32 - 12 = 20$ and that for taking r and r_{N1} is 16. Thus, both r' and $\{e, f, g\}$ are used to express the remainder query.

4.3 Server Reply CR Coalescence

Very often, portions of CRs, submitted as remainder queries, might not be fully explored for answering queries in the Query + CRs approach. Thus, refined CRs

are returned to the client along with the answer objects. To save the downlink cost, the server reply CRs can be coalesced. The optimization should consider reducing the number of CRs while retaining the quality of CRs such that a low false miss rate can be achieved. We associate CR quality with some quantitative metrics by defining cost function $c(r)$ for a CR r based on different heuristics:

- **Area**. Generally, the larger the area of a CR, the more likely the CR provides a higher false miss rate since it may include more empty regions in which no objects exist. Therefore, $c(r)$ is set to the area of r, $area(r)$. In this case, we expect a smaller average size of coalescing CRs.
- **Distance**. With spatial access locality, the closer is the CR located to the user location, the more likely is the CR to be accessed in the near future. It is thus important to have a fine granularity for those nearby CRs. Hence, we model $c(r)$ as the inverse of its distance to the user, i.e., $1/dist(r)$. In this case, we expect to coalesce farther CRs.
- **Area By Distance**. Area and distance are two orthogonal factors and they can be used in setting the cost $c(r)$, i.e., $area(r)/dist(r)$.

Server reply CR coalescence can save download cost but it also haunts the CR quality. To balance the transmission overhead saving and the quality of coalesced CRs, we limit the CR saving. In our implementation, we set a threshold that is the percentage of the total number of CRs before coalescence. When the number of remained CRs falls below the threshold, the coalescence terminates. Note that server reply CR coalescence has an additional constraint in coalescing CRs. If a coalesced CR contains some returning objects, the corresponding coalescence is prohibited because the resultant CR is highly possible to give a false miss if the client issues the same queries later (see Requirement 2 in Section 2.2).

5 Cache Management

As CS caching keeps both objects and CRs to preserve the global view of a dataset, its cache management is totally different from conventional ones that cache homogeneous caching units such as data objects. In this section, we discuss the CS cache organization and two cache space allocation strategies, followed by description of the cache CR coalescence and the cache replacement algorithm.

5.1 Cache Organization

The cache memory is structured as a table. Each table entry is of equal size and large enough to accommodate either one object or a collection of CRs. A table entry assigned to maintain CRs (called *CR entry*) keeps at most n CRs and one *coalescing CR*, which is an MBB enclosing all the CRs within this entry with n the capacity of a CR entry. Each stored CR has a timestamp about the latest access time. The coalescing CR facilitates fast CR lookup and CR coalescence in the cache, serving cache replacement.

The admission of an object is straightforward, i.e., finding a vacant entry to accommodate the object. The admission of CRs is handled in a way similar to R-tree insertion. A CR entry is chosen to store the admitted CRs if the expansion of its coalescing CR after insertion is the smallest among all candidate CR entries. If a CR entry overflows after insertion, all CRs (except the coalescing CR) are migrated to other CR entries with free space. If the space is insufficient, the collection of CRs in a CR entry are split into two groups and each group is placed into two CR entries. Deletion removes a CR from an entry. To maintain high occupancy, an occupancy threshold is set[4]. An underflowed entry (i.e., its occupancy below the threshold) is removed and all its CRs (except the coalescing CR) are re-inserted to other CR entries.

We propose two possible space allocation strategies, namely *static allocation* and *dynamic allocation*. For static allocation, cache memory is split into two portions with each dedicated to caching objects or CRs. Dynamic allocation has no fixed portions and treats objects and CRs in the same way to exploit higher flexibility in space utilization.

5.2 Cache CR Coalescence

Cache CR coalescence replaces a set of fine CRs with a bounding CR in a coarser granularity to release cache space. The efficiency of CR coalescence is crucial to the performance when cache replacement occurs frequently. Therefore, instead of using the generic algorithm described in Section 4, we adopt a pre-clustering technique that groups CRs in the same CR entry into their corresponding coalescing CR.

The pre-clustering of CRs is performed when CRs are admitted to the cache (as described in Section 5.1). We use minimal expansion of coalescing CRs as the criteria to determine which CR entry a new CR can be inserted into. Since the coalescing CR in a CR entry readily represents the result of coalescing all CRs in the entry, we can quickly perform CR coalescence to release a CR entry by looking up the coalescing CRs only.

5.3 Cache Replacement

Cache replacement in CS caching is responsible not only for fitting objects and CRs in the cache but also for balancing the granularity of different portions of the global view (in terms of objects and CRs) maintained in the cache. An object removal is performed as converting the object to a CR. A CR removal implies coalescence of a set of CRs. However, to make cache replacement efficient, usually all CRs in a victim CR entry will be removed by inserting its coalescing CR into another CR entry. In the following, we discuss the replacement algorithms corresponding to both static allocation and dynamic allocation.

Static allocation. Replacement starts in the object portion. If the object portion is full, victim objects are removed by transforming them into CRs, which

[4] Our simulation uses $n/2$ where n is the capacity of a CR entry.

are put to the CR portion. If a CR portion is full, victim CR entries are chosen to remove and their coalescing CRs are re-inserted to the CR portion. The victim selection (i.e. cache replacement policy) is based on LRU and FAR [9] heuristics. For FAR heuristic, distance is measured between the current client position and the anticipated CR (either resulted from object deletion or CR coalescence).

Dynamic allocation. Both object replacement and CR coalescence can make room for new objects and CRs. Cache replacement policy for both operations is crucial for ensuring the overall cache performance. In order to prioritize object replacement and CR coalescence which are totally different in nature, we use a replacement score based on the expected reloading cost of objects or CRs. Let $size_o$ and $size_r$ denote the object size and the CR size respectively, and ρ denote the access probability of an entity (either an object or a CR). In this work, we consider access probability based on LRU and FAR. The communication cost of reloading an object from the server is $\rho \times size_o$ and that of reloading CRs is $\rho \times m \times size_r$, where m is the number of CRs involved in the CR coalescence. Taking the reloading cost as a replacement score, we describe our cache replacement operation as follows. We maintain a priority queue of existing table entries. The queue always returns one entry with the least reloading cost. If a table entry is retrieved from the queue, it is freed to accommodate the new object and the newly formed CR (resulted from conversion of object or CR coalescence) is inserted back to an appropriate CR entry. Similarly, CRs downloaded from the server are inserted to CR entries. It may be the case that a CR entry overflows and additional entry space is required. Then, an additional entry space is reclaimed as that for a new incoming object. It might be possible that the newly formed CR entry whose reloading cost is less than those in the queue. In this case, CR coalescence is immediately performed and the coalescing CR is re-inserted to the cache.

6 Performance Evaluation

We conduct a performance evaluation on our proposal based on a simulation developed in C++. In the simulation, there are only one client and one server communicating via a point-to-point wireless channel with bandwidth of 384Kbps, the typical capacity of 3G network. The server maintains a synthetic dataset with 100,000 point objects uniformly distributed in a unit square of $[1, 1]$ and indexed with a R*tree [10] which has a node page size of 1Kbytes and its maximum fanout is 50. The size of each object is ranged from 64, 128, 256 to 512 bytes. The client has 128 Kbyte cache memory. In the experiments, client movement patterns are generated based on two well known mobility models, Manhattan Grid model and Random Waypoint, using BonnMotion [11]. For Manhattan Grid model, we set the mean speed to $1 \times 10^{-3}/sec$ and standard deviation to $0.2 \times 10^{-3}/sec$. For Random Waypoint model, we set the speed ranging between $0.5 \times 10^{-3}/sec$ to $1.5 \times 10^{-3}/sec$. The maximum think times (i.e., time duration that the client remains stationary during moving path change) for both models are set to 60 seconds. Meanwhile, we generate a query workload with query

inter-arrival time following the exponential distribution with mean varying from 10 to 30 at step of 10 (seconds). We assume that the client issues queries along her journey. We examine three types of queries: 1) range queries (with radius of 5×10^{-3} to 10×10^{-3}), 2) window queries (with window size of $(10 \times 10^{-3})^2$ to $(20 \times 10^{-3})^2$), and 3) kNN (with $k \in [10, 20]$). Each type of queries has the same weight in our experiments. The simulation runs for 10,000 seconds.

The performance metrics used in our evaluation include *response time*, *bandwidth consumption*, *cache hit ratio* and *answerability* while the answerability measures how many queries can be completely answered by the client cache without the server help. In addition to our proposed CS caching scheme, we implement *Chunk-based caching* [7], *Semantic caching* [2, 12], and *Proactive caching* [8] for comparison. Note that the chunk-based caching only supports window queries. Semantic caching supports window and kNN queries by caching two types of semantic regions. However, each type of semantic regions can only support queries of the same type. Proactive caching keeping a portion of R-tree index can support all queries we considered. When a client receives a query that is not supported, it requests the server to process it and results of these queries are not cached.

6.1 Evaluation 1. Performance of Caching Schemes

We first examine the performance of different caching schemes, namely, Semantic, Chunk-based, Proactive and CSC in terms of *response time*, *bandwidth* (both upload and download) and *cache hit ratio*. For CSC, remainder queries are expressed as Query + CRs, with CR partitioning and both request and reply CR coalescence. The Manhattan Gird model is adopted. The result is depicted in Fig. 7 (where the cache size and object size are fixed at 128KByte and 256 bytes, respectively).

From the plots, we can see that CSC outperforms the rest in all metrics for its effectiveness in supporting different queries, the efficient use of cache memory, and the low overhead in data transmission. Semantic is the weakest among all because it maintains two types of semantic regions that may result in an overlap of cached objects, in turn degrading the effective use of cache (as indicated by its low cache hit ratio). Chunk-based performs better since chunks contain extra objects that can be used to answer later window queries, thus outweighing some loss in processing kNN and range queries. Proactive performs worse than Chunk because the cached partial server index reduces the availability of cache memory for data objects. This evaluation validates CSC for location-based services.

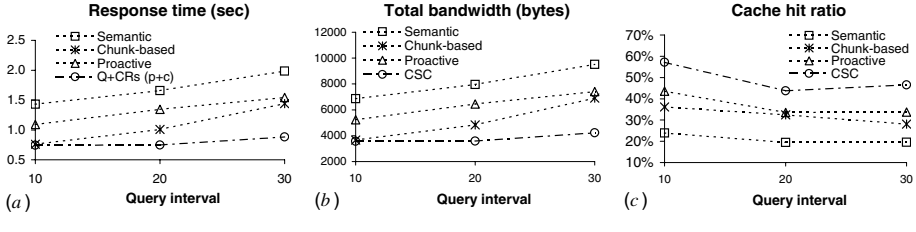

Fig. 7. Performance of caching schemes on query interval

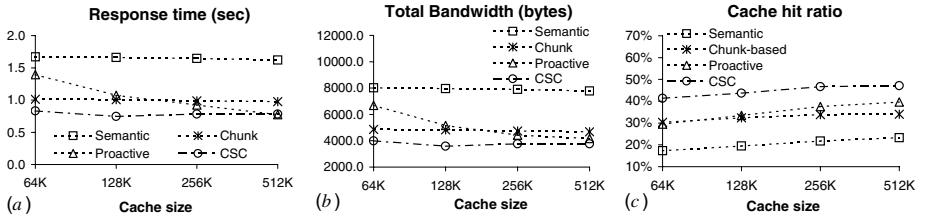

Fig. 8. Performance of caching schemes on cache size

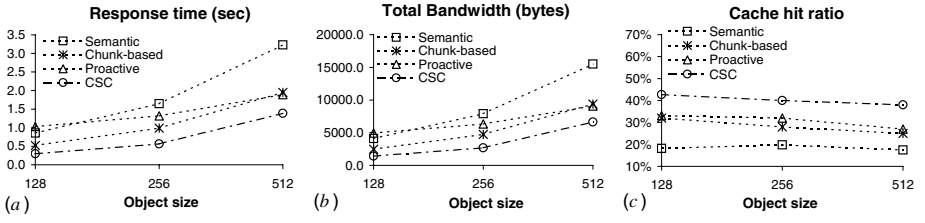

Fig. 9. Performance of caching schemes on object size

In addition, we study the impact of cache size on caching schemes. In Fig. 8 (where the object size is fixed at 256 bytes and the mean query inter-arrival time is 20 seconds), the response time of Semantic and Chunk-based is shown to be more or less invariant since they use cache space to maintain the query results they support. In effect, they may not fully utilize the space. For CSC, response time is a bit higher when cache size is 64K. For Proactive, response time drops when more space is available to store the index nodes.

In Fig. 9 (where the cache size is 128K and mean query inter-arrival time is 20 seconds), all caching schemes show that the larger the object size, the longer the expected response time. As the object size is increased, the cache hit is reduced accordingly because less objects are cached. The download cost, a major time consuming component, also increases when larger objects are experimented. For the same reasons discussed in above two settings, CSC is shown superior to others.

6.2 Evaluation 2. Performance of Remainder Query Expressions

Here we evaluate three different forms of remainder queries, i.e., original query plus CRs (denoted as Q+CRs), Q+CRs with partitioning (denoted as Q+CRs (p)), and Q+CRs with both partitioning and client request CR coalescence (denoted as Q+CRs (p+c)). These three forms of remainder queries have the same cache hit (so the plot is not shown to save space). Also, we have evaluated the remainder query with CRs only but its performance is much worse. The plot is not shown for space saving. The difference in their performance is due to the compression and improved precision of CRs. Using partitioning (i.e., Q+CRs (p) and Q+CRs (p+c)), the client avoids downloading extra CRs (see Fig. 10(c)). The response

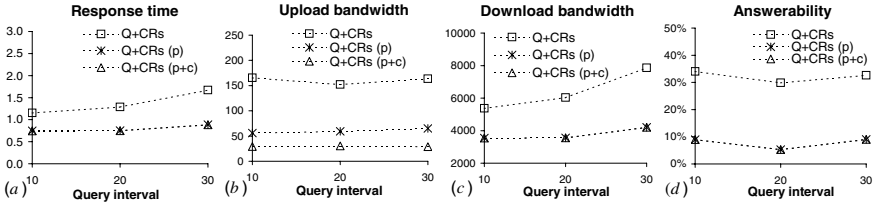

Fig. 10. Performance of using all remainder query expressions

time is also shortened in Fig. 10(a). However, the answerability is much lower than the basic Q+CRs because the CRs partitioned by the client are less precise (see Fig. 10(d)). Both of Q+CRs (p) and Q+CRs (p+c) can only answer 10% of queries without the server help while the Q+CRs needs to do that for 33% of time. Finally, the CR coalescence Q+CRs (p+c) is shown to be very effective in reducing the uplink bandwidth (see Fig. 10(b)). It saves almost 50% uplink bandwidth compared with Q+CRs (p). This saving is important because mobile clients consume more energy in sending packets than receiving packets.

6.3 Evaluation 3. Performance of Server Reply CR Coalescence

We study three heuristics, Area, Distance, and Area By Distance, used in coalescing CRs in server replies (see Fig. 11). We assume that remainder queries are sent in form of Q+CRs. We vary the percentage of CRs coalesced (where 0% means no coalescence) to observe its impact on response time and total bandwidth. Area is generally not a good heuristic because CRs close to the query are often of smaller area. Forming coarse CRs with those close CRs will degrade the cache performance since those close CRs are likely to be accessed. However, Area By Distance can provide very good performance (even better than Distance). Balancing on client location and CR size renders an appropriate granularity for returned CRs.

6.4 Evaluation 4. Performance of Cache Management

Finally, we study the two space allocation strategies, i.e., static allocation (Static) and dynamic allocation (Dynamic). For Static, we allocate x percent of cache storage for CRs. As shown in Fig. 12(a), Dynamic generally performs the best in term of response time. For Static, we can see that increasing cache space for CRs from 10% to 20% improves the response time but not when it is increased 25% as reflected by their corresponding cache hit ratios (shown in Fig. 12(a)). The more cache space allocated to CRs, the less cache space is available for objects, so the cache hit ratio drops when the CR portion of cache expands. Though Static 10% and Static 20% by allocating less space to CRs have higher cache hit ratios than Dynamic, they hold overly coarse CRs and result in a high false miss rate, which in turn increases the response time. For cache management, we also tested cache replacement using FAR and LRU policies upon different moving model. FAR generally outperforms LRU. The results are not shown due to limited space.

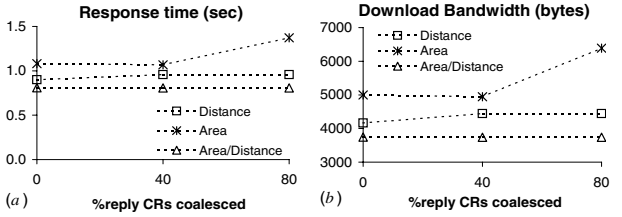

Fig. 11. Performance of heuristics in server reply CR coalescence

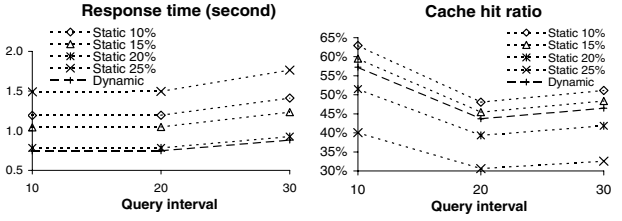

Fig. 12. Performance of cache allocation strategies

7 Future Works

As for the next steps of this research, we plan to study the issues of cache coherence caused by updates. We also plan to perform a more extensive performance evaluation to bring out more insights. Finally, we plan to prototype the system and to perform the feasibility test in a realistic mobile computing environment.

Acknowledgements

In this research, Wang-Chien Lee and Ken C.K. Lee were supported in part by US National Science Foundation grant IIS-0328881. Baihua Zheng's work was partially supported by the Office of Research, Singapore Management University. Jianliang Xu's work was partially supported by grants from the Research Grants Council, HKSAR, China (Project Nos. HKBU 2115/05E and FRG/04-05/II-26).

References

1. Schiller, J.H., Voisard, A., eds.: Location-Based Services. Morgan Kaufmann (2004)
2. Dar, S., Franklin, M.J., Jónsson, B.T., Srivastava, D., Tan, M.: Semantic data caching and replacement. In: Proc. of 22th International Conference on Very Large Data Bases (VLDB), Bombay, India, Sep 3-6. (1996) 330–341
3. Lee, K.C., Leong, H.V., Si, A.: Semantic query caching in a mobile environment. ACM Mobile Computing and Communication Review (MC^2R) **3** (1999) 28–36
4. Ren, Q., Dunham, M.H., Kumar, V.: Semantic Caching and Query Processing. IEEE Trans. on Knowledge and Data Engineering (TKDE) **15** (2003) 192–210

5. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: Proc. of the ACM SIGMOD International Conference on Management of Data, Boston, MA, Jun 18-21. (1984) 47–57
6. Roussopoulos, N., Kelly, S., Vincent, F.: Nearest Neighbor Queries. In: Proc. of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, USA, May 22-25. (1995) 71–79
7. Deshpande, P.M., Ramasamy, K., Shukla, A., Naughton, J.F.: Caching Multidimensional Queries Using Chunks. In: Proc. of the ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA, Jun 9-12. (1998) 259–270
8. Hu, H., Xu, J., Wong, W.S., Zheng, B., Lee, D.L., Lee, W.C.: Proactive Caching for Spatial Queries in Mobile Environments. In: Proc. of the 21st International Conference on Data Engineering (ICDE), Tokyo, Japan, Apr 5-8. (2005) 403–414
9. Ren, Q., Dunham, M.H.: Using Semantic Caching to Manage Location Dependent Data in Mobile Computing. In: Proc. of the International Conference on Mobile Computing and Networking (Mobicom), Boston, MA, USA, Aug 6-11. (2000) 210–221
10. Backmann, N., Kriegel, H.P., Schneider, R., Seegar, B.: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In: Proc. of the ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25. (1990) 322–331
11. BonnMotion: A mobility scenario generation and analysis tool. (website: http://web.informatik.uni-bonn.de/IV/Mitarbeiter/dewaal/BonnMotion/)
12. Zheng, B., Lee, W.C., Lee, D.L.: On Semantic Caching and Query Scheduling for Mobile Nearest-Neighbor Search. Wireless Networks **10** (2004) 653–664