Proactive Caching for Spatial Queries in Mobile Environments

Haibo Hu[1] Jianliang Xu[2] Wing Sing Wong[1] Baihua Zheng[3] Dik Lun Lee[1] Wang-Chien Lee[4]
[1]Hong Kong University of Science and Technology, {haibo,egwws,dlee}@cs.ust.hk
[2]Hong Kong Baptist University, xujl@comp.hkbu.edu.hk
[3]Singapore Management University, bhzheng@smu.edu.sg
[4]Penn State University, wlee@cse.psu.edu

Abstract

Semantic caching enables mobile clients to answer spatial queries locally by storing the query descriptions together with the results. However, it supports only a limited number of query types, and sharing results among these types is difficult. To address these issues, we propose a proactive caching model which caches the result objects as well as the index that supports these objects as the results. The cached index enables the objects to be reused for all common types of queries. We also propose an adaptive scheme to cache such an index, which further optimizes the query response time for the best user experience. Simulation results show that proactive caching achieves a significant performance gain over page caching and semantic caching in mobile environments where wireless bandwidth and battery are precious resources.

1 Introduction

In mobile environments, a user's common practice to know about his/her proximity area is to issue spatial queries. These queries share more common results than they do in traditional environments because the user moves continuously and thus the areas he/she queries exhibit high spatial locality. On the other hand, battery and wireless bandwidth are precious resources of mobile clients. Given these two facts, caching results at the client side is an attractive technique for spatial query processing [18].

Semantic caching [7, 16] has been proposed to answer spatial queries at the mobile client side [14, 15, 20]. It maintains both the results and semantic descriptions of some previous queries. Before a new query is submitted to the server, it is trimmed against the cached queries. Only the trimmed part of the query, i.e., the part that does not overlap any cached queries is submitted. The following example illustrates this idea.

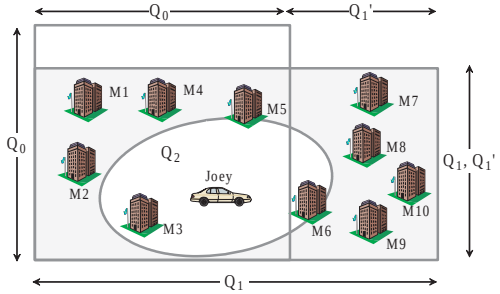


Figure 1. The Semantic Caching Example

Example 1.1 *Suppose Joey is driving and looking for a motel in the neighborhood. He issues a range query Q_0 and the server returns $M_1 \sim M_5$ (see Figure 1). As he does not find a favorite one among them, he issues another query Q_1 with a wider range. Since the query windows of Q_0 and Q_1 overlap, the cached results of Q_0 (i.e., $M_1 \sim M_5$) are returned immediately and the client only needs to submit Q_1', i.e., $Q_1 - Q_0$.*

Semantic caching not only saves the wireless bandwidth but also reduces the query response time. Nevertheless, we have observed several problems with semantic caching.

First of all, semantic caching captures the semantics of the queries only, but not the semantics of the cached objects. In other words, the granularity of cache reuse is at the query level. As such, a new query can be answered only by the queries of the same type, which makes it difficult to share the cached objects among various query types. Example 1.2 illustrates such a drawback.

Example 1.2 *Suppose in Example 1.1, after Q_0 Joey issues a 3-nearest-neighbor (3NN) query Q_2 instead of Q_1. Semantic caching cannot trim a 3NN query from a range query Q_0. The client has to submit the complete Q_2, although the result objects M_3 and M_5 are already cached as Q_0's results. They are partial results of Q_2 and should have been returned to the user immediately. The retransmission*

of these objects not only wastes the wireless bandwidth, but also prolongs the response time.

Furthermore, the types of spatial queries supported by semantic caching are rather limited. Only simple range query [15] and nearest neighbor (NN) query [20] have been studied. It is difficult to support complex queries such as k-nearest-neighbor (kNN) and spatial join queries. Semantic caching also entails complicated cache management. When a new query to be cached overlaps some cached query, a decision has to be made for whether to coalesce these two queries or to trim either of them. In addition, the organization of a semantic cache is plain. Many operations on the cache, e.g., query processing or cache replacement, require a sequential scan in the cache. This drawback is more remarkable as the cache size gets larger.

With the above drawbacks of semantic caching in mind, we propose in this paper an innovative caching model, called *proactive caching*. It caches the result objects as well as the index that supports the objects as the query results. The cached index helps determine if a cached object is a result of subsequent queries. In this way, the granularity of cache reuse is at the object level. The cached index can be B+-tree nodes in the relational database domain. For spatial queries, the index consists of the R-tree nodes in the object's vicinity. In this sense, the caching model prepares the index in advance of subsequent queries. This is where the name "proactive" comes from. Example 1.3 shows how proactive caching improves performance over semantic caching.

Example 1.3 *Suppose proactive caching is adopted in Example 1.2 and the whole area shown in Figure 1 is indexed by one R-tree node. After Q_0, this node and all its ancestors in the R-tree are cached according to proactive caching. The next query Q_2 can utilize the proactively cached index nodes and be processed locally by an NN search algorithm. Thus, M_3 and M_5 are returned immediately. Only M_6 needs to be transmitted from the server.*

Our contributions in this paper are listed as follows:

- We propose a proactive caching model which addresses the drawbacks of semantic caching. In particular, the proposed cache captures the semantics of cached query results at the object level (c.f., semantic caching at the query level), thus enabling sharing of cached objects among different query types.
- We develop a generic spatial query processing technique on proactive caching to reuse the cached results among various types of queries.
- We propose an adaptive proactive caching scheme to further optimize the query response time for the best user experience.
- We perform extensive experiments which show that the proactive caching model works better than semantic caching in resource-constrained mobile environments.

The rest of the paper is organized as follows. The next section reviews the existing caching models. Section 3 introduces proactive caching and presents its query processing scheme. Section 4 proposes the caching cost model and an adaptive scheme for caching the index. We study the cache organization and replacement schemes in Section 5. The performance is analyzed by simulation results in Section 6. Finally, we conclude and show future directions of this work.

2 Related Work

Caching frequently accessed data at the client side not only improves the user's experience of the distributed system, but also alleviates the server's workload and enhances its scalability. The **page caching** model is widely used in RDBMS [9]. The cached items are typically disk pages or tuples, which can be looked up by their identifiers. OODBMS adopts a similar caching model, except that the cached items are autonomous objects [8]. Early mobile broadcasting systems originated from OODBMS and thus adopted the same page caching model [1]. The mobile caching model in unicast environments was first studied in [12], where a dynamic client data replication scheme was proposed. A datum is cached if the read operations are more frequent than the write operations for the last k requests. Chan et al. further explored this caching model in [6]. They proposed two caching granularities other than the entire object, namely, the attribute level and the hybrid level of object and attribute. Thus, the spatial and temporal locality of the client request is more finely exploited.

Since no query information is stored, page caching can only support equi-select queries on the objects' keys. The **semantic caching** model proposed in [7] organizes the cache in the granularity of a query. The client maintains both the semantic descriptions and the associated results of some previous queries in the cache. A new query from the user can be totally or partially answered by the cached queries. In case of being partially answered, the query is trimmed from the cached ones and a remainder query is sent to the server. The original semantic caching [7] considered only selection queries. Ren et al. extended the work to support selection-projection queries and formally defined the semantic model of a query and its memory organization [16]. Adapting semantic caching to mobile environments was first attempted by Lee et al. [14], where cache partitioning schemes were proposed to reduce the granularity of a cached item. Thus, the cache is more adaptive to the changes in the query pattern. Ren and Dunham elaborated the cache replacement scheme for location-dependent queries [15]. Their *FAR* policy chooses the region (they call queries as "regions") farthest away from the user's current location as the victim for replacement.

Zheng and Lee presented a semantic caching scheme for NN queries in mobile environments [20]. Their idea is to return a minimum validity time of the query result so that the same result can answer any subsequent NN query within this time window. This idea is further explored by Zhang et al. [19], where the validity of the query result is described by "influence objects". Xu et al. investigated cache management for the validity information of cached query results [18]. However, in all these semantic caching schemes, the cached queries can help answer subsequent queries of the same type only.

The term of "proactive cache" was also used by Cao in [5]. He studied caching in broadcasting environments where objects are simply accessed through their keys. His proactive caching scheme is to prefetch objects on the broadcast channel based on their access rates, update frequencies, and sizes. Since no queries are involved, his work is completely different from ours.

3 Proactive Caching and Query Processing

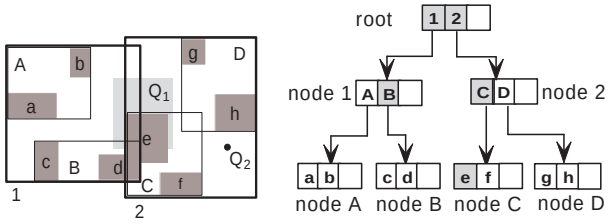
In this section, we first review the R-tree index and spatial query processing. Then, we introduce the overall proactive caching architecture. Finally, we present the generic spatial query processing algorithm under this architecture.

3.1 Preliminaries: R-tree and Spatial Queries

The predominant access method for a spatial database is the R-tree and its variations [2, 10]. The R-tree was extended from the B-tree for multidimensional data. It is a balanced tree along which the data objects are partitioned. Each tree node contains a subset of the objects, and its child nodes further partition this subset into subsets. A child node corresponds to an entry (MBR, p) in its parent node, where MBR is the minimum bounding rectangle of the object subset and p is the physical address of this child node. For leaf nodes, p points to an actual data object. Figure 2(a) illustrates the placement of data objects $a, b, ..., h$ and Figure 2(b) shows the corresponding R-tree, where $root$, 1 and 2 are the intermediate tree nodes and $A \sim D$ are the leaf nodes.

Spatial queries, such as range, kNN, and join queries, can be processed on the R-tree index. The range query Q_1 shown in light gray in Figure 2(a) is processed as follows. From the root node, we visit its child nodes whose MBRs overlap Q_1 (shown in gray in Figure 2(b)). The process continues recursively until it reaches the leaf nodes and returns the data objects that overlap Q_1. In this example, object e is returned when leaf node C is visited.

For kNN queries, the best known algorithm is *best-first* search (BFS) [11]. It uses a priority queue $\mathcal{H}$ to store to-be-explored entries whose corresponding nodes may contain nearest neighbors. The entries in $\mathcal{H}$ are sorted by their MBRs' minimum distances to the query point. BFS works by popping up the top entry from $\mathcal{H}$, pushing its child entries into $\mathcal{H}$, and then repeating the process all over. When a leaf entry, i.e., an entry of a leaf node, is popped, the corresponding object is returned as a nearest neighbor. The algorithm terminates if k objects have been returned. As an example, BFS processes the 1NN query Q_2 in Figure 2(a) as follows. It first pushes $root$ to $\mathcal{H}$, and then pops up entry 2, followed by pushing C and D. Next, it pops up D and pushes g and h. Finally, h is popped as a nearest neighbor and the algorithm stops here.



(a) Objects Placement (b) Corresponding R-tree

Figure 2. An Example of R-tree

A spatial join on datasets R and S is to find object pairs $\langle a, b\rangle$ ($a \in R, b \in S$) that satisfy a spatial predicate, such as "mutual distance less than a threshold" (i.e., distance join) and "a intersects b" (i.e., intersection join). To process such joins on the R-tree index, the RJ algorithm [3] is the most commonly used. It visits the two R-trees for R and S simultaneously. Starting from the root entry pair, it processes an entry pair $\langle r, s\rangle$ by recursively calling itself to process the pairs of r and s's child entries until it reaches leaf entries and then the corresponding object pairs are returned.[1] As an example, the intersection self-join (i.e., $R = S$) in Figure 2(b) is processed as follows. Starting from the root entry pair $\langle 1, 2\rangle$, RJ calls itself to process $\langle B, C\rangle$ (but not $\langle A, C\rangle$ because A does not intersect C, neither do $\langle B, D\rangle$ and $\langle A, D\rangle$), which finds out eligible pair $\langle d, e\rangle$. Huang and Jing improved the RJ algorithm and proposed Breadth-First R-tree Join (BFRJ) in [13]. BFRJ removes the recursive call by maintaining an *intermediate join index* (IJI), which contains all the $\langle r, s\rangle$ entry pairs to be processed. When BFRJ processes a pair $\langle r, s\rangle$ from the IJI, an in-memory join between r's child entries and s's child entries is performed. The eligible pairs of child entries for the join predicate are put into the IJI if they are not leaf entry pairs, and are returned as results if they are. Thus, the IJI resembles the priority queue $\mathcal{H}$ in the BFS algorithm since both of them contain the entries or entry pairs to be explored.

[1] To simplify the presentation, we assume the two R-trees have the same heights.

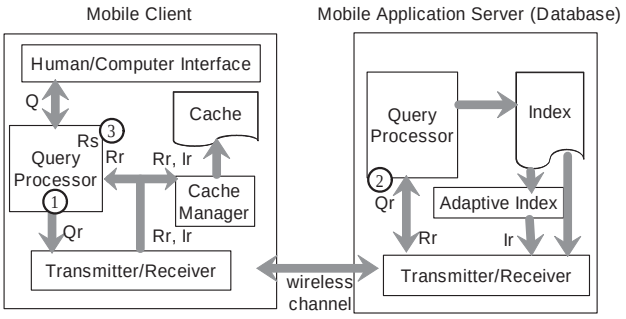


Figure 3. The Architecture of Proactive Caching

3.2 Proactive Caching Architecture

Given a spatial query Q, a proactive cache stores both Q's result objects (denoted as set R) and the R-tree index nodes (denoted as set I) which support the fact that R contains the results of Q and only Q. By this definition, the simplest form of I is the set of accessed R-tree nodes when Q is processed. An adaptive scheme to choose I will be presented in Section 4.

Figure 3 depicts the architecture for proactive caching, where the arrows show the processing flow for Q. The processing is divided into three stages (labelled as ①②③). At the first stage, the client-side query processor executes Q based on the cached index. The partial results are called **saved objects** and denoted by R_s. If $R_s \neq R$ (see Section 3.3 for how to determine this), a **remainder query** Q_r is submitted to the server. Otherwise, the processing terminates without contacting the server. At the second stage, the server-side query processor evaluates Q_r and sends back both the result objects R_r and the supporting index I_r. At the third stage, the client-side query processor returns $R = R_s \cup R_r$ and the cache manager inserts R_r and I_r into the cache. It is noteworthy that semantic caching shares the same processing flow, but Q_r is simply a truncation of the original Q and $I_r = Q_r$ (i.e., no more information other than the query description is stored in the cache).

3.3 Processing Queries with Proactive Caching

To show how the client-side processor executes Q and constructs Q_r, we first generalize the processing algorithms for various queries described in Section 3.1. In general, any spatial query on the R-tree is processed by descending the tree from the root and recursively exploring the child nodes that may contain eligible objects. During the exploration, a priority queue $\mathcal{H}$ is used to store the entries to be explored.[2] More specifically, Q is processed by: (1) pushing the root entry into $\mathcal{H}$; (2) popping up the top entry from $\mathcal{H}$ and pushing eligible child entries (with respect to the query Q) into $\mathcal{H}$ (if the eligible entry is a leaf entry, the corresponding object is returned as a result); (3) repeating (2) until $\mathcal{H}$ is empty or a termination condition specific to the query type of Q is satisfied. For example, the kNN processing terminates if k objects have been returned.

In proactive caching, when the client-side processor executes Q as described above, the corresponding index node of a popped intermediate entry or the corresponding object of a popped leaf entry may miss from the cache. In this case, the entry (called a *missing entry*) is pushed back to $\mathcal{H}$ and the next entry is popped from $\mathcal{H}$.[3] If at some moment all entries in $\mathcal{H}$ are missing entries but the processing has not yet terminated, a remainder query Q_r is constructed and submitted to the server. The content of Q_r contains Q and $\mathcal{H}$. In other words, the execution state of Q is handed over to the server, which resumes the processing based on the missing entries in $\mathcal{H}$. The pseudo-code in Algorithm 1 describes the complete procedure of processing Q. Note that for spatial joins, n is a pair of entries rather than a single entry.

Algorithm 1 Proactive Query Processing

Input: Q: the query
$\mathcal{C}$: the proactive cache
Output: R: the result set for Q
Procedure:

```
1:  build empty priority queue H;
2:  put root entry to H;
3:  while termination condition is false AND H has non-
    missing entries do
4:      pop entry n from H;
5:      if n is missing from C then
6:          set n as missing;
7:          put n back to H;
8:      else if n is a leaf entry representing a cached object
        then
9:          insert n into R_s;
10:     else
11:         for each of n's child entries u do
12:             if u satisfies Q then
13:                 put u into H;
14: if termination condition is false OR H has missing leaf
    entries then
15:     Q_r = {Q, H};
16:     submit Q_r to the server and wait for R_r and I_r;
17:     insert R_r and I_r into C;
18:     return R = R_s ∪ R_r.
```

A unique issue for kNN query is that, a leaf entry should

[2] For join queries, it stores pairs of entries.

[3] For spatial joins, an entry pair is a missing entry pair if either entry in the pair is a missing entry.

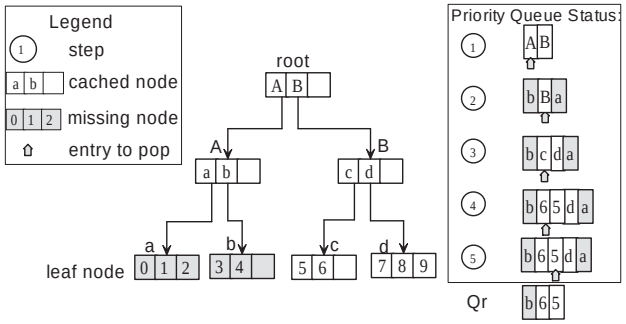


Figure 4. An Example of kNN Search in Proactive Caching

be returned as a result only if there is no missing non-leaf entry prior to it in $\mathcal{H}$. Let m and n denote the number of returned objects so far and the number of missing leaf entries in $\mathcal{H}$. The termination condition for a kNN query changes from $m = k$ (for normal query processing) to $m + n = k$ (for query processing with proactive caching). Furthermore, the remainder query changes to a $(k - m)$-NN search with $\mathcal{H}$. Let's see the following example:

Example 3.1 *A 2NN query accesses the root and pushes entries A and B into $\mathcal{H}$ (see Figure 4). When A is popped, b and a are pushed into $\mathcal{H}$. Since leaf node b is missing from the cache, it stays in $\mathcal{H}$ and then B is popped (step ②). When leaf entry 6 is popped at step ④, since entry b is a missing non-leaf entry prior to 6 in $\mathcal{H}$, 6 is not returned, neither is entry 5. The client processing terminates at step 5, when $m + n = 0 + 2 = 2$ (entries 6 and 5). A remainder 2NN query Q_r is sent. To make Q_r concise, we further prune those entries after the current k^{th} leaf entry in $\mathcal{H}$ because none of these entries contain objects that are closer to the query point than the k^{th} leaf entry. In this example, entry 5 is the 2^{nd} leaf entry, so entries d and a are pruned.*

4 Adaptive Proactive Caching

As discussed in Section 3.2, the simplest form of the supporting index I_r for the remainder query Q_r is the set of accessed R-tree nodes when Q_r is processed at the server side. However, caching the exact copy of each node is not a prerequisite: an R-tree node corresponds to a page on the disk, but the cache is memory-based rather than disk-based. Furthermore, it is also not optimal in performance because an entry far away from an object might not help support this object to be the result of future queries. And this is further dependent on the type of the query: to support an object as the result, a range query does not need any other index entry (other than the one containing the object), whereas a kNN query needs some surrounding entries. In this section, we propose to adapt the content of cached index nodes to the queries to optimize the query response time. The object is to strike a balance between the overhead and the effectiveness of the cached index so that the memory of the cache is fully exploited. In what follows, we first derive the cost model of query response time, then introduce the notion of *compact form* R-tree node, and finally propose the adaptive indexing algorithm using this form.

4.1 Cost Model

As the user receives the result objects individually, the **query response time** ($resp(Q)$) is defined as the average response time of each byte of the results, which is the elapsed time from Q being issued till this byte being returned to the user. This is a fairer metric than the response time of the last returned byte, since in practice the user often wants to access the results as early as possible. On the other hand, in mobile computing the wireless communication always dominates in the access delay and the power consumption cost, outweighing CPU. Under this assumption, those bytes of the objects in R_s have a negligible response time because they are locally cached. For each byte of the objects in R_r, the response time is the transmission time for Q_r plus that for the already transmitted bytes of the objects in R_r.[4] As such, $resp(Q)$ is given by:

$$resp(Q) = \frac{|R_r|(T_{Q_r} + \frac{1}{2}|R_r| \cdot T_d)}{|R|}, \qquad (1)$$

where the operator $|\,|$ denotes the size of a dataset in bytes, T_{Q_r} is the wireless communication delay to submit Q_r to the server and T_d is the delay to download a byte. Since T_{Q_r} is relatively small,[5] minimizing $resp(Q)$ is equivalent to minimizing $\frac{|R_r|}{|R|}$, or maximizing

$$hit_c = \frac{|R_s|}{|R|} = 1 - \frac{|R_r|}{|R|}.$$

The hit_c is called the **cache hit rate**. In the literature, there is another notion, **byte hit rate** (hit_b) which is the probability of a requested byte being in the cache. Let C denote the set of cached objects. We can represent hit_b as:

$$hit_b = \frac{|R \cap C|}{|R|}.$$

If we further assume a uniform access pattern and an independent reference model, then we have

$$hit_b = \frac{size_c}{\sum_{i \in DS} size_i},$$

[4] For simplicity, the fixed transmission overhead is ignored as it does not affect the analysis of minimizing $resp(Q)$.

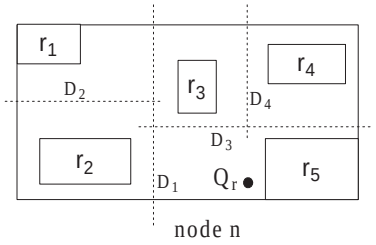
[5] In Section 6, we show that the size of Q_r is generally one or two orders of magnitude smaller than that of R_r.

where $size_c$ is the cache size for storing objects and $size_i$ is the size of each object i in the dataset DS. In semantic caching or proactive caching, the two notions hit_c and hit_b are different as $R \cap C \subseteq R_s$, i.e., even if an object is actually a result and is cached, it is possible that it is not returned as a saved object if no enough index is cached to support this fact. This is called a **false miss** for the cache. As such, the cache hit rate can be derived from byte hit rate as:

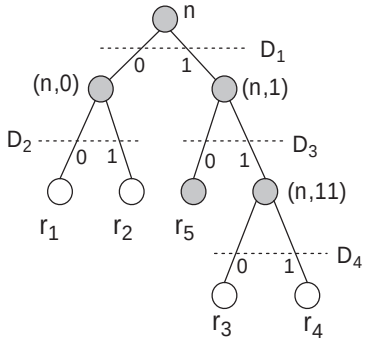
$$hit_c = hit_b \cdot [1 - P(o \notin R_s | o \in R \cap C)]$$
$$= \frac{size_c}{\sum_i size_i}[1 - P(o \notin R_s | o \in R \cap C)]. \quad (2)$$

The conditional probability $P(o \notin R_s | o \in R \cap C)$ in Equation (2) is defined as the **false miss rate** (fmr). It indicates to what degree the cache supports the cached objects to answer subsequent queries. fmr depends not only on the caching models (e.g., semantic caching or proactive caching) but also on the types and the parameters of the queries. This is because different queries need different ranges and precisions of the information around an object o to confirm o as a result. For a range query, only o's location information is needed. For a kNN query issued at point p, the range of the confirming information is the circle centered at p with radius $Distance(p, o)$. And the required precision of this information is just to show whether or not there are less than k objects in the circle. For a distance spatial join with threshold d, the range is the circle centered at o with radius d. But the required precision is specified on the other dataset, which is to show if there is any object of this dataset within this circle. Obviously, given the same supporting index for the cached objects, the queries which require large-ranged and highly-precise confirming information lead to high fmr, and vice versa.

In order to minimize hit_c in Equation (2), the page, semantic, and proactive caching models have different tradeoffs between $size_c$ and fmr. Page caching provides no auxiliary knowledge for the cached objects and, hence, $fmr = 1$ and $hit_c = 0$. But its $size_c$ is equal to the actual cache size. Semantic caching provides the knowledge by caching the query descriptions. But such knowledge is available to the same type of queries only. Thus, it is not efficient in lowering the overall fmr. Proactive caching provides the auxiliary knowledge for all types of queries by caching the index nodes. But it does so without knowing if these cached nodes are useful for subsequent queries. Obviously, if the subsequent queries require only short-ranged and imprecise confirming information around the objects, caching the complete index nodes overly supports them and wastes the cache space. Therefore, in the following two subsections, we propose an *adaptive proactive caching scheme* with the objective of striking a balance between fmr and $size_c$. The basic idea is to represent far-



(a) Spatial Placement of n



(b) The Binary Partition Tree

Figure 5. Binary Partition Tree of R-tree Node

away entries coarser than those entries close to the cached objects. The degree of the coarseness is adaptive to the current fmr so that the cached index just keeps necessary supporting information.

4.2 The Compact Form of R-tree Node

We define the **full form** of an R-tree node n as n itself with all its entries. Since not every entry in n is accessed by Q_r, the non-accessed entries can be combined into "super entries" to reduce the size of n. Such a coarse representation is called a **compact form** of n with respect to Q_r, denoted as $CF(n, Q_r)$.

To efficiently compute the compact form of n, we build a binary partition tree of all the entries in n by recursively partitioning a set of entries into two subsets until the set has only one entry. The partitioning uses the R-tree node splitting algorithm to assure minimal overlap between the MBRs of the two subsets of entries. Figure 5 illustrates the partition tree for an R-tree node n which contains entries $r_1 \sim r_5$. D_1 through D_4 show the recursive splitting. Each intermediate node in this tree is regarded as a super entry of n, and is designated by a unique id $(n, code)$, where $code$ is formed by concatenating the binary digit 0/1 along the path from the root to this intermediate node.

When Q_r is processed, n or a super entry $(n, code)$ is popped from the priority queue $\mathcal{H}$. Only its two children in the partition tree, i.e., $(n, code \oplus 0)$ and $(n, code \oplus 1)$

($\oplus$ is the string concatenation operator) are considered as child entries. For example, to process the NN query Q_r in Figure 5(a) on n, we first push n's two children, i.e., $(n, 0)$ and $(n, 1)$ to $\mathcal{H}$. Then, $(n, 1)$ is popped, and r_5 and $(n, 11)$ are pushed into $\mathcal{H}$. Finally, popping up r_5 terminates the processing. Only the grey nodes in Figure 5(b) are accessed by Q_r. The compact form of n with respect to Q_r comprises the leaf entry r_5 and all the grey nodes whose child nodes are not in grey, i.e., $CF(n, Q_r) = \{(n, 0), (n, 11), r_5\}$. Compared with the full form of 5 entries, i.e., $\{r_1, r_2, r_3, r_4, r_5\}$, the compact form saves storage space by 40%.

The compact form is efficient to compute because the computation is embedded in the query processing. Let N denote the number of entries in n. The number of intermediate nodes in the binary partition tree is $N-1$. The original query processing accesses all the N entries; the new algorithm in the worst case accesses all the $N-1$ intermediate nodes and all the N entries, which doubles the processing time. Nonetheless, the average processing time is expected to be much shorter since, in most cases only a small portion of the intermediate nodes and entries are accessed, as will be demonstrated in the experiments (Section 6.4). Building the binary partition tree for each R-tree node is an offline and one-time operation. Each tree has $N-1$ entries plus $2*(N-1)$ pointers. Therefore, the additional space required to store the binary partition trees for all the R-tree nodes is no more than two times that of the R-tree index itself.

4.3 Adaptive Proactive Caching

The full form has the most entry information around each result object, whereas the compact form (called **normal compact form** hereinafter) only provides sufficient entry information to support result objects for query Q_r, thus leading to a higher fmr but a larger $size_c$. To obtain a balance between these two metrics, we first generalize the two forms into the **d^+-level compact form**. It is obtained by replacing each entry in the compact form with its d level descendant nodes or the entries whichever come first in the binary partition tree. For example, in Figure 5(b), the 1^+-level compact form of a normal form $\{(n, 0), (n, 1)\}$ is $\{r_1, r_2, r_5, (n, 11)\}$. In general, a d^+-level compact form is approximately 2^d times finer than the normal compact form. Suppose the height of the binary partition tree is h, the 0^+-level compact form is the normal compact form and the h^+-level compact form is the full form. Therefore, the choice of d determines how fine and how proactively the cache stores the supporting entry information around the cached objects.

The remaining problem is to determine the value of d so that the cached index sufficiently but not overly supports the cached objects for subsequent queries. We adapt d to the client's current fmr, which indicates not only how much confirming information the recent queries need, but also how well the current d provides such information. The adaptive scheme works as follows. The client periodically submits its recent fmr to the server. If the value is higher than the last recorded fmr by s percent (s is the sensitivity parameter), it means that the recent queries require finer entry information around the cached objects; so the value of d for this client is increased by 1. On the contrary, if it is lower than last fmr by s percent, d is decreased by 1. Otherwise, d remains its last value.

5 Proactive Cache Management

The cache manager views both index nodes and objects as **items** which support to answer queries locally. The manager is responsible for choosing victims from these items for cache replacement. In this section, we derive an optimal replacement scheme $GRD3$ for proactive caching.

Let M denote the size of cache $\mathcal{C}$, $size(i)$ the size of item i, and $benefit(i)$ the benefit of caching i. The cache replacement problem is, given the incoming items of size M', to find a victim set $\mathcal{C}' \subset \mathcal{C}$ such that $\sum_{i \in \mathcal{C}-\mathcal{C}'} size(i) \leq M - M'$ and $\sum_{i \in \mathcal{C}-\mathcal{C}'} benefit(i)$ is maximized. This problem is the *0/1 Knapsack* problem, except that for cache replacement, the victims are removed from, instead of filled into the knapsack. The greedy algorithm (denoted as GRD1), which removes the worst items in terms of $benefit(i)/size(i)$ from $\mathcal{C}$, has been proved to be a 2-approximation algorithm.

5.1 The Constrained Knapsack Problem

By the definition of proactive caching, if a cached index node is removed, all its descendants including the index nodes and the objects are no longer accessible and, thus, should also be removed. Therefore, the proactive caching replacement problem is a **constrained knapsack problem** with the constraint that if item i is removed, all its descendants must be removed altogether.

From Section 4.1, the benefit of a cached item i is the response time saved from retransmitting it on the wireless channel for subsequent queries, i.e., $benefit = prob(i) \times size(i) \times T_d$, where $prob(i)$ is the access probability of item i. As T_d is a constant for every item, we omit it for simplicity. Hence, $benefit = prob(i) \times size(i)$. Since removing i presumes removing all i's descendants from the cache, $benefit(i)$ must count for i as well as i's descendants. That is,

$$benefit(i) = \sum_{j \in D(i)} prob(j) \times size(j) + prob(i) \times size(i),$$

where $D(i)$ denotes the set of descendants of i. The metric $benefit(i)/size(i)$, by which GRD1 picks victims, now has a physical meaning of **expected bitwise response time saving** ($EBRS(i)$),

$$EBRS(i) = \frac{\sum_{j \in D(i)} prob(j)size(j) + prob(i)size(i)}{\sum_{j \in D(i)} size(j) + size(i)}.$$

$EBRS(i)$ has the following two features:

Corollary 5.1 *Let $Ch(i)$ denote the set of child items of i. If $Ch(i) = \emptyset$, i is called a* **leaf item** *and EBRS(i) = prob(i).*

Corollary 5.2 *The recursive definition of $EBRS(i)$:*

$$EBRS(i) = \sum_{j \in Ch(i)} \frac{SIZE(j)}{SIZE(i)} EBRS(j) + \frac{size(i)}{SIZE(i)} prob(i). \quad (3)$$

Here $SIZE(i)$ is the total size of i and $D(i)$, i.e., $SIZE(i) = \sum_{j \in D(i)} size(j) + size(i)$. The recursive formula shows that, after the removal of an item together with its descendants, only the $EBRS$ of its ancestors are changed.

Therefore, the greedy algorithm, GRD2, for the constrained knapsack problem is the same as GRD1, except that: (1) the metric for finding the victims is $EBRS$; (2) after the removal of victim i, its ancestors' $EBRS$ values should be updated according to Equation (3).

There are two problems associated with GRD2: (1) the $EBRS$ update is costly because the derivation of $EBRS(i)$ is recursive according to Equation (3); (2) GRD2 loses the approximation bound with GRD1. In the following, we extend GRD2 to GRD3 which is proved to yield the same outcome as GRD2, but is more efficient and has an approximation bound of 2. To derive it, we need the following lemmas:

Lemma 5.3 *If item j is item i's descendant, $prob(i) \geq prob(j)$.*

Lemma 5.4 *$\forall$ item i, $\min_{j \in Ch(i)} EBRS(j) \leq EBRS(i) \leq prob(i)$.*

Proof: We prove the inequality by mathematical induction.
(1) For the leaf items, by Corollary 5.1, $EBRS(i) = prob(i)$, and since i has no children, the inequality holds.
(2) Suppose that for any item i whose deepest descendants is fewer than or equal to k ($k \geq 0$) levels away from i, $\min_{j \in Ch(i)} EBRS(j) \leq EBRS(i) \leq prob(i)$. Thus, for any item i whose deepest descendants is $k+1$ levels away, we have $EBRS(j) \leq prob(j)$. From Lemma 5.3, $prob(j) \leq prob(i)$. Therefore, $prob(i) \geq \max_{j \in Ch(j)} EBRS(j)$. On the other hand, according to Corollary 5.2,

$$EBRS(i) = \sum_{j \in Ch(i)} \frac{SIZE(j)}{SIZE(i)} EBRS(j) + \frac{size(i)}{SIZE(i)} prob(i)$$

In other words, $EBRS(i)$ is the weighted arithmetic average of all $EBRS(j)$ and $prob(i)$. Therefore, $EBRS(i)$ is between the minimum and the maximum of these values, which are $\min_{j \in Ch(i)} EBRS(j)$ and $prob(i)$, respectively. Therefore, the inequality holds for $k+1$.
From (1) and (2), $\forall$ item i, $\min_{j \in Ch(i)} EBRS(j) \leq EBRS(i) \leq prob(i)$. □

Lemma 5.4 shows that GRD2 chooses the victim only from the leaf items. And Corollary 5.1 shows that for any leaf item i, $EBRS(i) = prob(i)$. Thus, GRD2 is equivalent to picking the leaf items with the lowest $prob$ values. As such, it is unnecessary to maintain $EBRS$ anymore. We call the following enhanced algorithm **GRD3**.

Definition 5.1 *Algorithm GRD3: Efficient Replacement Algorithm for Proactive Caching*
(1) scan $\mathcal{C}$ and discard any i, s.t. $size(i) > M - M'$;
(2) put all leaf items in a priority queue G whose key is $prob$;
(3) pop item i from G and remove it from $\mathcal{C}$;
(4) if i is its parent p's last child, push p into G;
(5) if $\sum_{i \in \mathcal{C}} size(i) > M - M'$, goto (3);
(6) denote the most recently removed item i as B. If $prob(B) \times size(B) > \sum_{j \in \mathcal{C}} prob(j) \times size(j)$, remove everything remained in $\mathcal{C}$ and insert B back to $\mathcal{C}$.

GRD3 is much more efficient than GRD2 because it does not compute $SIZE$, $benefit$ and $EBRS$. Furthermore, the following theorem shows that GRD3 is a 2-approximation algorithm for the constrained knapsack problem.

Theorem 5.5 *GRD3 is a 2-approximation algorithm for the constrained knapsack problem.*

Proof: Let KP denote the original knapsack problem and CKP denote the constrained knapsack problem where items must be removed with their descendants. Let $OPT(\cdot)$ denote the optimal algorithm for a problem. We have,
(1) $GRD1(KP) \geq \frac{1}{2} OPT(KP)$, by the proof from the literature.
(2) $GRD3(CKP) = GRD1(KP)$, because Lemma 5.4 guarantees that if $GRD1(KP)$ is executed, it always chooses leaf items with the lowest $prob$ values, which is exactly the same as $GRD3(CKP)$.
(3) $OPT(KP) \geq OPT(CKP)$, because a non-constrained problem must have a better optimal solution than its constrained counterpart.
From (1)(2)(3), we have $GRD3(CKP) = GRD1(KP) \geq \frac{1}{2} OPT(KP) \geq \frac{1}{2} OPT(CKP)$. □

5.2 Implementation Issues for GRD3

To make GRD3 to work, each item is associated with some *metadata*. More specifically, the metadata includes

the following properties of the item: (1) physical address, (2) size, (3) time of insertion (in terms of the sequence id of the query when it is inserted), (4) number of hit queries, (5) parent item id, (6) number of cached children.

Metadata (5) provides a pointer to look up the parent item, which is missing from the common indexes such as the R-tree and the B^+-tree. Metadata (6) indicates whether an item is a leaf item. This value is updated when its child nodes are inserted and removed from the cache.

In practice, $prob(i)$ can be estimated by the ratio of metadata (4) to the total number of queries that i has lived through:

$$prob = \frac{\#hit\_queries}{T - time\_of\_insertion},$$

where T denotes the sequence id of the current query.

6 Performance Evaluation

6.1 Simulation Model

The simulation emulates a mobile client issuing spatial queries about its neighborhood. The client moves according to two mobility models: the *random waypoint* (RAN) and the *directed movement* (DIR). Under RAN [4], the client selects a random point as its destination and moves to it at a randomly chosen speed spd; upon arrival, it pauses for a random period and selects a new destination. DIR restricts the selection of the next destination so that the moving direction is roughly reserved. This is a better model for on-purpose movements [15].

The event of client issuing queries is modeled as a *Poisson* process. More specifically, the client waits for an exponentially distributed random period (called *thinking time*) to issue a new query after the current query completes. In each experiment run, the client issues 10,000 queries. We use two large-scale datasets, NE which contains 123,593 postal zones of New York, Philadelphia, and Boston of the United States, and RD which contains 594,103 railroads and roads in US, Canada, and Mexico [17]. The coordinate systems of both datasets are normalized to unit squares. The average object size $\overline{|o|}$ is $10KB$. The sizes of individual objects follow a Zipf distribution with the skewness parameter θ being 0.8. The sizes of the R*-tree indexes (with a page capacity of 4KB) for these two datasets are 3.8MB and 18.5MB, respectively. The query type is randomly selected from range, kNN, and join. The window of a range query is centered at client's current position with an average size $Area_{wnd} = 10^{-6}$. The join query is a distance self-join which returns pairs of objects whose distances are lower than a threshold $Dist_{join}$. The k for a kNN query is randomly chosen from 1 to K_{max}. The client has a 384Kbps wireless channel, which is the standard for a 3G network.

Parameter	Value	Parameter	Value
$\overline{spd}$	0.0001	$\overline{think\_time}$	50s
$Area_{wnd}$	10^{-6}	$Dist_{join}$	5×10^{-5}
K_{max}	5	bandwidth	384Kbps
$\|C\|$	0.1%∼5%	$\overline{\|o\|}$	10KB
θ	0.8	s	20%

Table 6.1. System Parameters Settings

The cache size is varied from 0.1% to 5% of the total dataset size with 1% as the default value.

The server is implemented on a Pentium 4 1GB PC running Win2000 Server and IIS 5.0, and the client is simulated on a Pentium 4 512MB PC running WinXP. The client and the server communicate through the SOAP/HTTP protocol. We set the system sensitivity parameter s for the adaptive proactive caching at 20%. Table 6.1 summarizes the parameter settings for the simulation study.

We implement the semantic caching for range and kNN queries according to [15, 20]. No semantic caching techniques are available for join queries. When the client receives a join query, it directly passes it to the server. To have a fair performance comparison, we choose the state-of-the-art cache replacement scheme for each of the three cache models: FAR for semantic caching [15], LRU for page caching, and $GRD3$ for adaptive proactive caching. The metrics for performance comparison are the query-wise *uplink bytes*, *downlink bytes*, *response time* (defined in Section 4.1), and the overall *cache hit rate* (hit_c) and *byte hit rate* (hit_b). The *uplink bytes* and *downlink bytes* metrics imply the query cost on wireless communication and power consumption, which are the major concerns in mobile environments. To simplify the notation, in the sequel, PAG represents page caching, SEM represents semantic caching, and $APRO$ represents adaptive proactive caching.

6.2 Overall Performance Comparison

Figure 6 shows the measured performance for the three caching models when the mobility model is DIR and $|C| = 1\%$ of the size of dataset NE.[6] For better legibility, the values from the three models for each metric are normalized to $[0, 1]$ and the maximum value is shown in the parenthesis following the metric label.

PAG always has the highest uplink bytes since it needs to submit the identifiers of all cached objects to the server. As a reward, it downloads the fewest bytes. However, since PAG does not store any supporting information for these objects, the cache hit rate is zero. As a result, the response time is rather poor. SEM downloads the highest

[6] Similar results were observed for the RD dataset; they are omitted in this paper due to space limitations.

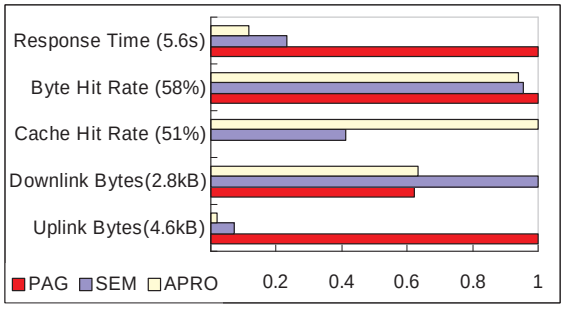
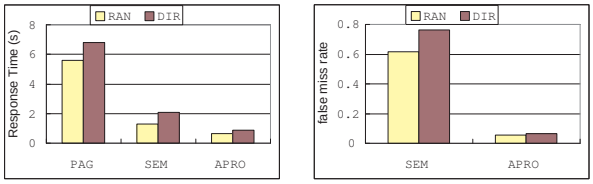


Figure 6. Overall Performance Comparison



(a) Response Time (b) False Miss Rate

Figure 7. Performance under Different Mobility Models

number of bytes, because range and kNN queries cannot share cached objects. The same reason explains why its cache hit rate is only one-third that of $APRO$. $APRO$ addresses this problem by sharing the cached objects among all query types. The result is a 51% cache hit rate, and yet the downlink bytes is just slightly larger than that of PAG. All these factors explain why $APRO$ achieves the shortest response time among the three cache models in Figure 6. It is noteworthy that the achievement sacrifices none or little in the uplink and downlink bytes metrics, which implies that $APRO$ is also efficient in terms of the power and bandwidth consumption.

We also compare them under different mobility models and show the results in Figure 7. The response time for DIR is larger than that for RAN in all cache models as shown in Figure 7(a). This is expected as RAN exhibits better query locality than DIR; so the benefit of caching is more prominent. An interesting observation is that for $APRO$, the response time increases very little when the mobility model changes to DIR. This is because $APRO$ proactively retrieves entry information for cached objects; hence, even if the client visits a new place, the cached index information may already cover this area in supportive of the cached objects. Figure 7(b) justifies this explanation: the false miss rate of $APRO$ almost remains the same under the two mobility models.

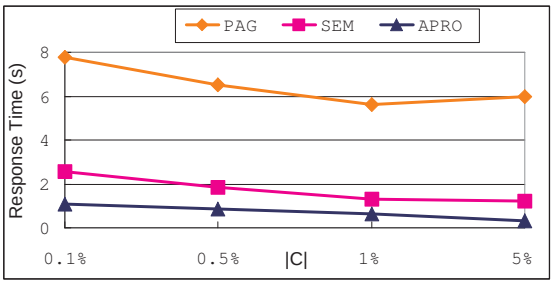


Figure 8. Performance under Different Cache Sizes

6.3 Impact of Cache Sizes and Replacement Schemes

In this set of experiments, we measure the response time under four cache size ($|C|$) settings: 0.1%, 0.5%, 1%, and 5%. The mobility model is RAN. From Figure 8, we observe that when $|C|$ increases from 1% to 5%, PAG's response time even increases. This is because its uplink bytes is proportional to $|C|$ and the performance gain of caching no longer compensates the increased uplink overhead for $|C| > 1\%$. For SEM, the response time also saturates when $|C| > 1\%$. This is due to the fact that its performance gain is separate for each query type, and when $|C| > 1\%$, caching range queries and caching kNN queries have both reached their performance limits. $APRO$ overcomes this drawback by sharing the cached index nodes and objects among all query types. Therefore, it still achieves notable performance gain after $|C| > 1\%$.

To further justify the $APRO$ caching model, we also show in Figure 9 the client CPU processing time per query under various cache sizes. It is measured by subtracting the network transmission time from the whole processing time for each query. Therefore, it includes the time cost for all necessary CPU operations such as query processing and cache maintenance. $APRO$ is more expensive than PAG and SEM in most cases, since it partially processes the queries, especially the spatial joins which are CPU-intensive. But its sensitivity to cache size is much lower, thanks to the cached index structure. In other words, $APRO$ does not need to search sequentially in the cache, which is the case for both PAG and SEM. Thus, it is expected to outperform PAG and SEM for larger cache sizes. On the other hand, the figure also justifies our assumption that the response time is predominantly incurred on wireless communication.[7]

In the next set of experiments, we evaluate $APRO$ under various replacement schemes, namely LRU, MRU, FAR

[7]The actual mobile CPU may work much slower than the CPU where the simulation is carried on, but the gap between the CPU time and the communication delay is still larger than one order of magnitude.

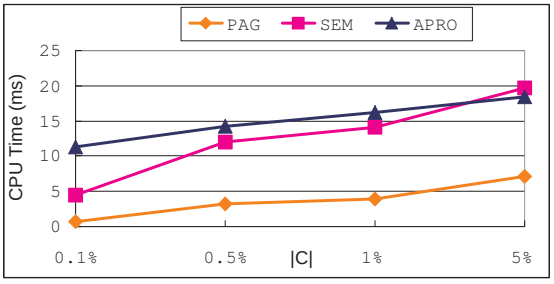


Figure 9. CPU Cost under Different Cache Sizes

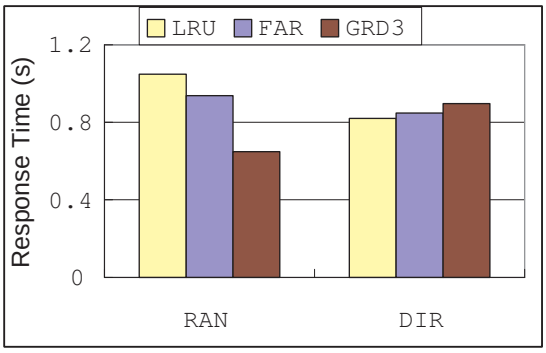


Figure 10. Performance under Different Cache Replacement Schemes

[15], and GRD3. Since MRU is always the worst of all, Figure 10 shows the rest three only. LRU outperforms FAR and $GRD3$ under the DIR mobility model. This is because under DIR, an object no longer accessed can be more quickly detected and removed by LRU. For $GRD3$, the object is dropped only when its $prob$ value becomes the lowest, which takes a longer time than LRU. For FAR, the object replacement depends on its distance to the user; thus the performance is less predictable. However, the result under the RAN model is just the opposite. Due to the random movement, LRU might erroneously remove the objects that are to be requested by the near future queries. Since FAR and $GRD3$ are both independent of the very recent access history, they perform much better. On the whole, $GRD3$ behaves more stable than the other two considering both mobility models, because the $prob$ value is accumulated and less vulnerable to the movement changes.

6.4 Effectiveness of Adaptive Proactive Caching

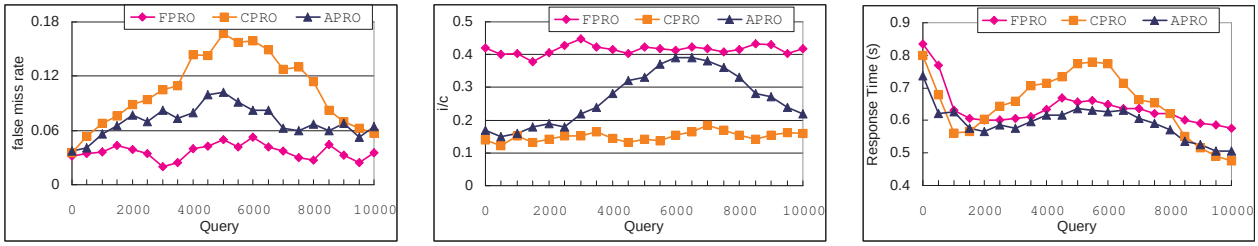
This set of experiments compares the performance of adaptive proactive caching ($APRO$) with its non-adaptive counterparts, i.e., caching full form index ($FPRO$) and normal compact form index ($CPRO$). We simulate the change of queries by exclusively issuing kNN queries and controlling the average k. The average k decreases gradually from 10 to 1 for the first 5,000 queries, and then increases gradually up to 10 for the second 5,000 queries. We measure the false miss rate, the index size in the cache (in terms of the ratio of index size to total cache size, i/c) and the response time for every 500 queries and plot them in Figures 11(a), 11(b), and 11(c), respectively. In order to highlight the performance change, we choose a small cache size 0.1% and the less predictable mobility model RAN.

As known from Section 4.1, the higher the k value, the less precise the confirming information is needed and, thus, the lower the fmr. From Figure 11(a), the false miss rate of $CPRO$ is the most vulnerable to the changes of k; actually its trend almost reflects the opposite trend of changing k. This is expected as $CPRO$ caches only the necessary entry information to support the query being processed. A high fmr may it lead to, $CPRO$ consumes the least cache size. $FPRO$, on the contrary, achieves the least and the most stable fmr, but it almost consumes half of the cache size. As a result, when k is large, $FPRO$ has the longest response time; and when k is small or moderate, $CPRO$ has the highest. $APRO$ maintains a steady fmr throughout the experiment, and it increases index cache size only when k is small, i.e., more precise entry information around the cached object is needed. Therefore, its response time is the best among the three almost all the time. However, it is noteworthy that in response to the changes of k, the adaptive scheme has certain degree of delay (see Figure 11(b)), since it takes time to fade out those old cached index nodes in order to decrease the index cache size. Therefore, $APRO$ is expected to work fine when both the cache size and the changing pace of the query impact on the fmr are small or moderate.

Regarding the price of $APRO$ over the nonadaptive schemes, the sizes of the binary partition trees for datasets NE and RD are 4.2MB and 23.7MB, respectively. The query processing time of the server is even reduced from 0.0081s (for $FPRO$) to 0.0067s (for $APRO$), which coincides with our analysis in Section 4.2 that the average CPU cost for the adaptive scheme is low because, in most cases only a small portion of the intermediate nodes and entries in a binary partition tree are accessed.

7 Conclusion and Future Work

In this paper, we propose the proactive caching model for spatial queries in mobile environments. Proactive caching captures the semantics of queries by caching the index that is responsible for answering them. The merits of proactive caching over semantic caching are two-folded: (1) the granularity of cache reuse is at the object level, finer than that at the query level; (2) the cached objects can be easily shared



(a) False Miss Rate (b) Index Size in Cache (c) Response Time

Figure 11. Performance Time Series under Adaptive and Nonadaptive Schemes

among common types of spatial queries. Furthermore, we propose the adaptive proactive caching which further optimizes the query response time. Empirical results show that proactive caching achieves a significant performance gain over page caching and semantic caching in terms of various performance metrics.

For the future work, we plan to investigate the impact of server updates on proactive caching and devise efficient cache invalidation schemes. We also plan to extend proactive caching so that the cached index is shared not only among various types of queries on the same client, but also among various clients in the neighborhood. Since these clients exhibit high query locality, such cache collaboration is beneficial in terms of cache reuse and bandwidth saving. This is particularly useful in a mobile ad-hoc network, where the bandwidth of local connections is much broader and cheaper than that of remote connections.

References

[1] D. Barbara. Mobile computing and databases – a survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):108–117, 1999.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *ACM SIGMOD Conference, Atlantic City, NJ*, pages 322–331, 1990.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the ACM SIGMOD Conference*, pages 237–246, 1993.

[4] J. Broch, D. A. Maltz, D. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the ACM/IEEE MobiCom Conference*, pages 85–97, 1998.

[5] G. Cao. Proactive power-aware cache management for mobile computing systems. *IEEE Transactions on Computers*, 51(6):608–621, 2002.

[6] B. Y. Chan, A. Si, and H. V. Leong. Cache management for mobile databases: Design and evaluation. In *Proceedings of the ICDE Conference*, pages 54–63, 1998.

[7] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the VLDB Conference*, pages 330–341, 1996.

[8] D. Dewitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. In *Proceedings of the VLDB Conference*, pages 107–121, 1990.

[9] M. Fanklin, M. Carey, and M. Livny. Global memory management in client-server dbms architectures. In *Proceedings of the VLDB Conference*, pages 596–609, 1992.

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference, Boston, Massachusetts*, pages 47–57, 1984.

[11] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, 1999.

[12] Y. Huang, P. Sistla, and O. Wolfson. Data replication for mobile computers. In *Proceedings of the SIGMOD Conference*, pages 13–24, 1994.

[13] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *Proceedings of the 23rd VLDB Conference*, pages 396–405, 1997.

[14] K. C. K. Lee, H. V. Leong, and A. Si. Semantic query caching in a mobile environment. *ACM SIGMOBILE Mobile Computing Communication Review*, 3(2):28–36, 1999.

[15] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the MOBICOM Conference*, pages 210–221, 2000.

[16] Q. Ren, M. H. Dunham, and V. Kumar. Semantic caching and query processing. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):192–210, 2003.

[17] The R-tree Portal. Low-dimensional real datasets. http://www.rtreeportal.org/spatial.html.

[18] J. Xu, X. Tang, and D. L. Lee. Performance analysis of location-dependent cache invalidation schemes for mobile environments. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):474–488, 2003.

[19] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *Proceedings of the ACM SIGMOD Conference*, pages 443–454, 2003.

[20] B. Zheng and D. L. Lee. Semantic caching in location-dependent query processing. In *Proceedings of the 7th International Symposium on Spatial and Temporal Databases*, pages 97–116, 2001.