Collaborative Caching for Spatial Queries in Mobile P2P Networks

Qijun Zhu #1, Dik Lun Lee #2, Wang-Chien Lee *3

Computer Science and Engineering, Hong Kong Univ. of Science and Technology, Hong Kong

[1]qijunzhu@cse.ust.hk

[2]dlee@cse.ust.hk

*Computer Science and Engineering, The Pennsylvania State University, USA

[3]wlee@cse.psu.edu

***Abstract*—We propose a novel collaborative caching framework to support spatial query processing in Mobile Peer-to-Peer Networks (MP2PNs). To maximize cache sharing among clients, each client caches not only data objects but also parts of the index structure built on the spatial objects. Thus, we call the proposed method *structure-embedded collaborative caching (SECC)*. By introducing a novel index structure called Signature Augment Tree (SAT), we address two crucial issues in SECC. First, we propose a cost-efficient collaborative query processing method in MP2PNs, including peer selection and result merge from multiple peers. Second, we develop a novel collaborative cache replacement policy which maximizes cache effectiveness by considering not only the peer itself but also its neighbors. We implement two SECC schemes, namely, the periodical and adaptive SAT-based schemes, with different SAT maintenance policies. Simulation results show that our SECC schemes significantly outperform other collaborative caching methods which are based on existing spatial caching schemes in a number of metrics, including traffic volume, query latency and power consumption.**

I. Introduction

Traditional spatial query processing for mobile applications is typically implemented based on the *client-server model*, in which mobile clients issue spatial queries such as range or k nearest neighbors (kNN) queries over a wireless network to a server hosting a spatial database. This paper studies collaborative caching for spatial query processing on Mobile Peer-to-Peer Networks (MP2PNs). Clients in a MP2PN communicate with each other directly or indirectly via intermediate clients using short-range wireless communication (e.g., WiFi). Clients can access the backend spatial database via access points (APs) connected to the Internet. When a client is out of range of the APs, its queries will be routed through the MP2PN to one of the APs and then to the spatial server for processing. This routing scheme facilitates *collaborative caching*, which utilizes the caches of the mobile clients to improve the efficiency of query processing,

Under the context of location-aware mobile applications, mobile clients are location-aware and are mostly interested in spatial objects in their surroundings. Thus, spatial queries issued from the mobile clients are expected to exhibit high spatial locality. That is, the results of a spatial query are very likely available on mobile clients near the query client, making collaborative caching techniques particularly promising for spatial queries in MP2PNs. In this paper, we develop a collaborative caching framework for supporting spatial queries in MP2PNs.

Collaborative caching techniques have been studied previously for Internet and World Wide Web [1]. Unfortunately, these techniques cannot be directly employed in wireless networks, which have dynamic topology and limited resources [2]. Recently, researchers have started to look into cache model and replacement issues in collaborative caching techniques for wireless networks [3], [4], [5], [6]. However, their proposed techniques targeted at simple data requests and hence cannot be applied to spatial queries. With the high locality of spatial queries and data, we envisage the advantage of adopting collaborative caching techniques for the support of heterogeneous spatial query processing in MP2PNs. To the best of our knowledge, this is the first research work towards this vision. The following example illustrates the applications and technical challenges of our research.

Example *1:* (**Park Information**) Consider a large national park, where infostations (or servers) are deployed to provide information about the park such as restaurants and camp sites to tourists. Since the number of infostations is small compared to the size of the park, the wireless coverage of the infostations is not able to cover the entire park. Tourists with mobile clients can form an MP2PN so that spatial queries from remote tourists can still reach the infostations via the MP2PN. This scenario is depicted in Fig. 1. Suppose user U_1 had issued a two nearest neighbor (2NN) query Q_{11} to find two closest point-of-interests (POIs) and acquired E_3 and E_4 from the spatial database. Later, at a different location, he issues another 2NN query Q_{12}, for which E_2 and E_3 are the answer. Since E_3 and E_4 from the earlier query Q_{11} have been cached, he can obtain the partial result E_3 from the local cache and issue a *remainder query* to find the missing answer E_2 from the server, which requires a two-hop connection (i.e., $U_1 - U_3 - Server$), as shown in Fig. 1. Alternatively, U_1 may find nearby peers (called *helpers*) to resolve the remainder query. For example, if U_2 had cached E_2 for an earlier range query Q_{21}, U_1 may obtain E_2 from U_2, which only requires a one-hop connection. □

The above example illustrates a number of challenges to collaborative caching techniques supporting spatial queries in MP2PNs. First, what is the operation model for employing collaborative caching to answer complex spatial queries in MP2PNs? Second, what information should be cached to facilitate efficient collaborative caching and processing of complex spatial queries? Third, how does a query client seek help from helpers and the server (i.e., who will be involved in query processing)? Finally, how do mobile clients replace

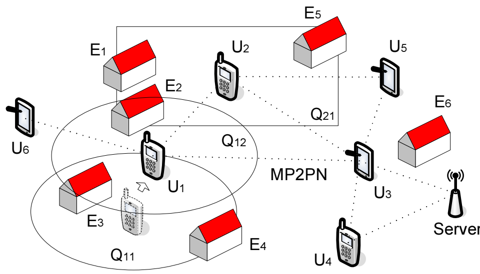


Fig. 1. The Collaborative Caching Example.

cached data when their caches become full?

Responding to these challenges, we propose a collaborative caching framework, called *structure-embedded collaborative caching (SECC)*, to realize our vision. SECC operates in a paradigm similar to *semantic caching* [7], [8], which caches both the search results and semantic descriptions of the queries. When a new query is issued, only the part of the query, called the *remainder query*, that does not overlap with any cached query, needs to be processed on the server. However, traditional semantic caching techniques are limited to the same query types. To efficiently support heterogeneous spatial queries, our SECC framework caches the spatial objects as well as partial index structures built on those objects, collectively called the *cache structure* of the client. Moreover, SECC aims to i) answer the remainder queries from nearby helpers, thus facing the challenge of identifying helpers in the MP2PN, and ii) maximize cache effectiveness which considers not only the query client itself but also its neighbors. To address these issues, we propose a *Signature Augment Tree* (SAT) to encode the information regarding which peers have cached a certain object or index node and address the problem of how a client discovers the cache structures of its peers. Accordingly, we develop collaborative query processing algorithms for two classical spatial queries, i.e., range and kNN queries, based on the SECC framework. As shown in later sections, two critical issues must be considered in the process, including i) helper identification which is conducted with efficient support of SAT, and ii) result merge, which merges the partial results from different helpers. Finally, we study the cache replacement problem in SECC and develop a collaborative cache replacement scheme to support SECC.

We evaluate the performance of SECC using simulation experiments. The results show that our SECC schemes significantly outperform other collaborative caching methods employing traditional spatial caching schemes in a number of metrics, including MP2PN traffic size, query latency, power consumption, and the number of peer connections. Moreover, SAT-based SECC schemes achieve better performance in query processing than non-SAT schemes. Finally, the results also verify the effectiveness of our collaborative cache replacement method.

The main contributions of this paper are four-fold:

- Propose the Structure-Embedded Collaborative Caching (SECC) framework, which exploits the locality of spatial queries to support heterogeneous spatial query processing by utilizing the caches of collaborative peers.
- Design a novel Signature Augment Tree (SAT) to cost-efficiently record the cache structures of other peers in MP2PNs, which acts as a basis of SECC. Two implementations of SECC are developed to explore different SAT maintenance policies.
- Provide a cost-efficient collaborative query processing method based on SAT, including effective algorithms for peer selection and result merge in a collaborative environment.
- Introduce a collaborative cache replacement scheme based on SAT, especially designed for SECC to maximize the benefits obtainable from the caches in MP2PNs.

The rest of this paper is organized as follows. Section II reviews the related work. Section III gives the background on the techniques and metrics adopted in this paper and formalizes the query processing steps in SECC. Section IV introduces the structure of SAT. Sections V and VI detail the algorithms for collaborative query processing in MP2PN and cache replacement, respectively. Section VII evaluates the performance of SECC. Finally, Section VIII concludes the paper and discusses future directions.

II. Related Work

Caching techniques have been widely adopted in computing systems for efficient data access. Under the client-server model, caching of frequently accessed data on the client reduces not only query latency but also the server workload. Item-based caching, which caches disk pages or tuples, has been adopted in operating systems and database management systems [9]. In mobile computing, similar ideas have been used in wireless data broadcast systems [10]. A caching technique for unicast and the associated scheme for dynamic client data replication have been proposed in [11].

An item-based caching scheme is limited to simple "equality queries" on page id or object key. It does not support proximity queries that are required in advanced applications. To overcome this problem, Dar et al. proposed a semantic caching technique [12] that caches items in accordance with a query type. The client maintains the semantic description of the query scope and the resulting data objects in its cache, which allows the derivation of a remainder query for retrieving data objects not available in the cache. This work was further extended in [13] with a formal semantic caching model, based on which efficient query processing strategies were investigated. For location-based query processing, Zheng and Lee proposed a semantic caching technique for NN queries [7] by keeping the validity time of the previous query results to help answer the subsequent NN queries. This idea was improved in [8] by defining validity regions for two common spatial query types, namely, NN and range queries, and developing efficient algorithms for computing validity regions.

Proactive Cache [14] and *Complementary Cache* [15] are caching schemes that allow cached objects to answer different types of queries (e.g., both window and kNN queries) by keeping track of information beyond the cached objects, whereas a traditional semantic caching method is designed to support one type of queries only (e.g., only window queries). Both proactive and complementary caches considered a mobile client-server environment, but SECC operates on the peer-to-peer collaborative scenario that is much more complicated than the simple client-server model.

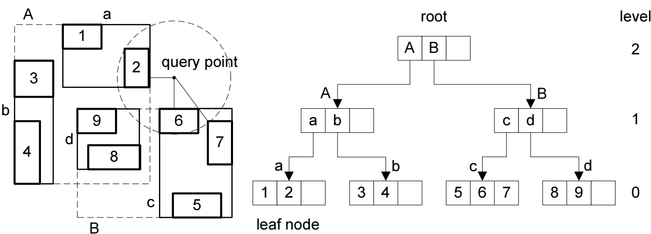


Fig. 2. An example of R-tree.

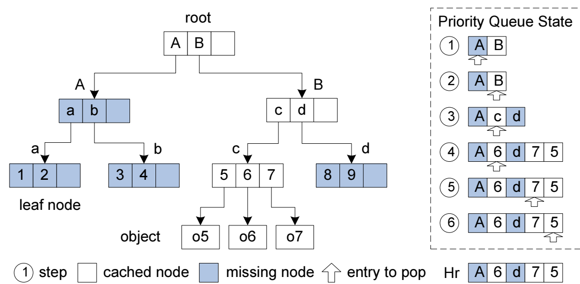


Fig. 3. 3NN query processing in proactive caching.

Collaborative caching has been explored to improve Web performance. An important piece of work is the Internet cache protocol (ICP) [16], which supports communication between caching proxies by message exchange. In the context of ad hoc wireless networks, [17] addressed the problem of optimal cache placement in static ad hoc wireless networks and proposed a greedy algorithm to minimize the weighted sum of energy expenditure and access delay. Hara proposed several replica allocation methods to increase data accessibility [18] and collaborative caching of broadcast data [5] in mobile ad hoc wireless networks. The hybrid cache [3] and cluster-based collaborative caching [6] schemes were developed to further improve performance. Sailhan and Issarny [19] studied query evaluation in a Internet-based mobile ad hoc network (IMANET) and devised a cooperative caching scheme to increase data accessibility via P2P communication among mobile devices when the devices were out of range from the fixed network infrastructure.

In general, existing collaborative caching techniques only facilitate fetching of simple data items, but our work aims at spatial queries, which are more complex and ubiquitous in mobile applications.

III. Structure-Embedded Collaborative Caching (SECC)

A. Preliminaries

Most spatial databases use R-tree or its variants [20] as an access method to answer spatial queries (e.g., range, kNN, and spatial join queries). Fig. 2 shows nine objects in a two-dimensional space and how they are aggregated into bounding boxes recursively to build the corresponding R-tree. In general, a query is processed by traversing the R-tree from the root to explore the child nodes for eligible objects. During the process, a priority queue H is used to maintain the entries to be explored. A generic evaluation procedure for a query Q can be summarized as follows. (1) Push the root entries into H; (2) Pop the top entry from H; (3) For a leaf entry, check the corresponding object against the query and return it as a result object if the query is satisfied; if the entry is not a leaf, check the eligible child entries for Q and push them into H; (4) Repeat (2) and (3) until H is empty or a termination condition of Q is satisfied. In the rest of this paper, we will adopt this generic model of processing spatial queries on R-tree in SECC and use the example in Fig. 2 as a running example to illustrate the ideas.

For R-tree-based spatial databases, proactive caching [14] improves query performance by allowing mobile clients to answer part of a query locally using the data objects and index nodes in the clients' caches. The local processing of a query in a client is similar to generic query procedure, except that the priority queue H may contain intermediate index entries corresponding to some uncached index nodes or objects. They are called *missing entries*. Fig. 3 describes the procedure for client u_Q to process a $3NN$ query Q at the query point shown in Fig. 2 with proactive caching. After query processing, if Q is not solved, a *remainder query* Q_r consisting of Q and the updated priority queue H_r will be generated.

B. System Framework

The main goal of SECC is to facilitate collaborative caching and processing of heterogeneous spatial queries among clients in a mobile peer-to-peer network. To achieve this goal, the clients cache both the result objects and the supporting index nodes as in proactive caching. Query processing on SECC consists of the following phases.

1) The **local processing** phase evaluates the received query Q against the local cache of the query client using the generic evaluation procedure as described above.
2) If Q cannot be resolved in the local processing phase, as indicated by the existence of missing entries in the priority queue H_r, the **collaborative processing** phase is invoked for the remainder query Q_r when the query client u_Q is out of the ranges of the servers (otherwise, directly go to phase 3). It includes two steps, i) u_Q efficiently identifies helper peers according to the locality property and merges the partial results as well as the supporting index nodes from them in the MP2PN. Thus, a further result can be achieved. ii) If the query is still unsolved, the remainder query Q_r' will be routed hop-by-hop to one of the servers via a routing scheme. Note that each peer along the routing path will process the query against its local cache and then forward the updated remainder query. If the query is solved in the middle, the routing terminates and the result as well as the supporting index nodes are routed back to u_Q.
3) In the **server processing** phase, the remainder query Q_r'' which arrives at the server will be processed to complete the answer.
4) Finally, in the **cache update** phase, the query client will cache the result objects and supporting index nodes collected from the helper peers and the server.

The two crucial issues for SECC, namely, collaborative query processing and cache update, are discussed based on a specially designed index structure signature augment tree (SAT) in the rest of this paper. For clarity, this paper only considers one type of spatial objects and common spatial queries such as kNN and window queries, although our method

can be easily extended to handle multiple types of spatial objects (e.g., hotels, restaurants, gas stations, etc.) and other query types such as spatial joins. However, we leave out the details because of space limitation.

IV. Signature Augment Tree

As described earlier, SECC supports two important functions: 1) spatial query processing; and 2) cache replacement, both in a "collaborative" fashion. In other words, a peer in SECC not only evaluates spatial queries with assistance from helpers in its neighborhood but also replaces spatial objects in its cache by taking into account what are available in the caches of neighbor peers. To perform these functions effectively and efficiently, some knowledge about the cache content in its neighbor peers obviously is beneficial. To meet this need, we propose the *Signature Augment Tree* (SAT), which along with its cache content provides a view of the overall spatial object space. As discussed in Section III, a peer in SECC actually caches a partial R-tree which is part of the original R-tree at the server. Thus, aggregating the partial trees of peers in the neighborhood can provide a more detailed partial R-tree with a rich collection of data objects and index nodes. In this section, we first describe our design and construction of the SAT and then discuss the maintenance issues.

A. Design of SAT

SAT is designed out of two considerations. First, it is not cost-efficient to directly collect objects and index nodes from other peers when it is uncertain if they are useful in future, because it may waste much energy to transmit unnecessary spatial data. Further, we want to keep the most valuable objects and index nodes considering that the cache size is usually much smaller than the dataset. Second, the size of the index in the spatial database cannot be ignored. Thus, SAT records both objects and index nodes of other peers and targets at achieving the largest benefit from collaborative caching with small transmission and storage cost.

Simply speaking, SAT aims to capture a view of the neighbors' cache content by providing information about which peers cached which index nodes or objects. Consider a client u. Let C_u denote the set of index nodes and objects in u's cache and L_u be the list of peers maintained by u. Client u maintains two types of information in its SAT: (a) the set of peers that contain data corresponding to index nodes or objects cached in u, and (b) the set of peers that contain data corresponding to *missing entries* in u. As shown in later sections, type (a) information is to be used for collaborative cache replacement and type (b) information for collaborative query processing. Moreover, both of them are important for SAT updates.

Basically, for each index node or object n in C_u, client u maintains a peer list corresponding to n, denoted by $L_{u,n}$, i.e., $L_{u,n} = \{u' | u' \in L_u \land n \in C_{u'}\}$. Meanwhile, for each missing entry e in C_u, u maintains a peer tree corresponding to e, denoted by $L_{u,e}$. This peer tree captures the peer lists corresponding to e's descendants which are cached by peers in L_u. Let $nid_{n'}$ denote the ID of a descendant node n' of e. $L_{u,e}$ is comprised of $\{(nid_{n'}, L_{u,n'}) | n'$ is a descendant node of $e\}$. Thus, SAT can be constructed by associating the peer

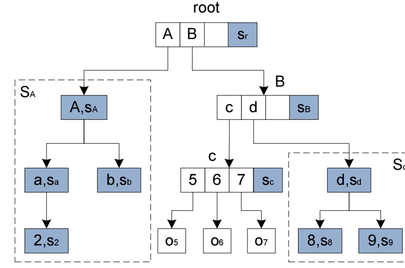


Fig. 4. An example of Signature Augment Tree (SAT), in which dark nodes denote the signatures.

list $L_{u,n}$ with each index node/object $n \in C_u$ and associating the peer tree $L_{u,e}$ with each missing entry $e \in C_u$

To realize the SAT, we use a bit string, called a *signature* (denoted as s_n[1]), to encode $L_{u,n}$, the peer list corresponding to n. If the k-th bit in s_n is set, the k-th peer in L_u is a member of $L_{u,n}$. Similarly, $L_{u,e}$ is encoded as a tree of *signature nodes*, each of which contains the ID and the signature for a descendent node, This tree is thus called a *signature subtree* (denoted as S_e). Finally, we associate the signatures and signature subtrees with the cached nodes and missing entries in C_u. As a result, the extended index tree is the *Signature Augment Tree* (SAT). Fig. 4 shows a possible SAT for the query client u_Q in the running example. As shown, the missing entry A is associated with a signature tree S_A and the index node B is associated with a signature s_B. Notice that, while u_Q maintains a partial R-tree in its own cache, it has the knowledge of an enriched partial R-tree consisting of the other partial R-trees cached in nearby peers. Consequently, u_Q is able to achieve near optimal peer selection through collaboration with other peers.

B. Maintenance of SAT

Every client must maintain its own SAT. Since the content of SAT not only involves its owner but also the peers being kept track of, several factors need to be considered in SAT maintenance, e.g., which peers are tracked in SAT, when to update the SAT, how to do it, etc. Since an SAT maintenance event involves exchange of metadata about cached content among peers, efficient representation of the exchanged information is critical. In SECC, we propose to serialize the structures of cached partial R-trees in peers to facilitate exchanges occurring on a wireless channel. To generate the serial representation, a peer, based on the depth-first traversing order of its cached R-tree structure, sequentially generates a stream T of tuples $t = (nid, pid)$, where nid is the ID of a node in the cache, and pid is the node ID of its parent. Suppose a client u_A, which has cached $root$, A, a, b, o_2, is kept track of by u_Q in Fig. 2. Then, u_A generates a tuple stream $T_A = \{(root, -1), (A, root), (a, A), (2, a), (b, A)\}$. In this way, the cache structure of a peer can be serialized into a tuple stream with extremely small size, which incurs trivial transmission cost in MP2PNs.

Another issue is to decide the tracked peers whose cache content contribute to formation of the SAT. Since there are many peers in the network, we restrict those tracked peers to *1-hop neighbors*, i.e., those who are within the direct

[1]We omit u in s_n since the SAT is specifically for u.

Algorithm 1 SAT Update

Input: tuple stream T, peer position i, initial SAT r
Output: updated SAT r

SignUpdate(tuple stream T, position i, SAT r)
1: purify r;
2: let a global pointer p point to the first tuple of T;
3: let n point to the root of r;
4: *SignUpdateNode*(p, i, r);
5: **return** r;

SignUpdateNode(pointer p, position i, SAT node n)
6: **assert** $nid_p = nid_n$;
7: **if** n has no signature **then**
8:    create a signature $s_n = 0$ for n;
9: **end if**
10: set the ith bit of s_n to be 1;
11: p++;
12: **while** $pid_p = nid_n$ **do**
13:    **if** n is a signature node **then**
14:       **if** a child n' of n has ID nid_p **then**
15:          *SignUpdateNode*(p, i, n');
16:       **else**
17:          create a signature node n' with nid_p and $s_{n'} = 0$;
18:          let n' be the child of n;
19:          *SignUpdateNode*(p, i, n');
20:       **end if**
21:    **else**
22:       find the entry e in n which corresponds to nid_p;
23:       **if** e points to a node (or signature node) n' **then**
24:          *SignUpdateNode*(p, i, n');
25:       **else**
26:          create a signature node n' with nid_p and $s_{n'} = 0$;
27:          let e point to n';
28:          *SignUpdateNode*(p, i, n');
29:       **end if**
30:    **end if**
31: **end while**

communication range. As such, no relay is needed for exchange of the R-tree streams. This decision comes from two observations. First, the peers farther away from the query client are less likely to contain the query results. Second, flooding the exchange streams in multiple hops to far-away peers incur much higher costs in terms of bandwidth, energy consumption, communication collision, etc.

Next, we consider the issue of how to update the SAT at a client u. Upon reception of an exchange stream T from v, u will update the signatures in its own SAT r. The basic idea is to recursively traverse r and update the signatures. Specifically, when a tuple of T corresponds to node n that does not exist in r, a signature node with the form (nid_n, s_n) will be generated. Let v's position in L_u is i. Algorithm 1 depicts the SAT update procedure, which takes the incoming tuple stream T, v's position i and SAT r as input and outputs the updated SAT r. Initially. r is purified by clearing the bits of the signatures which correspond to v (Step 1). Then, r is traversed recursively to update the signatures (Steps 2-5). For any visited node n, we claim that the current tuple p of T corresponds to n (Step 6). The signature of n is updated by

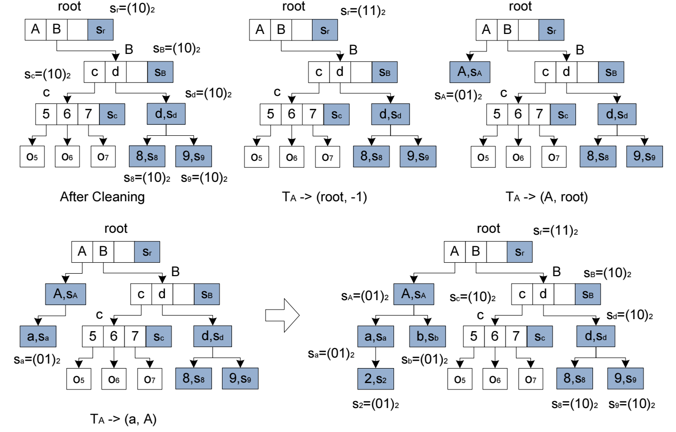


Fig. 5. An example of SAT update in client u_Q, $L_{u_Q} = \{u_A, u_B\}$. The signatures for peer u_B (caching $root$, B, c, d, o_8, o_9) have been generated.

setting the ith bit to be 1 (Steps 7-10). Then, move p to the next tuple of T (Step 11). If p is a child of n, then consider two cases: i) if n is a signature node, find the child n' of n which has ID nid_p or create a signature node n' for p, and then, recursively update SAT n' (Step 13-20); ii) if n is an index node or object, find the corresponding entry e (Step 22). If e points to a node n', then recursively update SAT n' (Steps 23-24); otherwise, create a signature node n' for p and recursively update SAT n' (Steps 25-28). These actions are repeated until n is not the parent of p. The update procedure for the running example is illustrated in Fig. 5.

Finally, regarding the timing of invoking SAT updates, we explore two different SAT maintenance policies in this paper.
1. Periodical SAT maintenance. Under this policy, each peer broadcasts the tuple stream periodically (e.g., every 10 seconds) to its 1-hop neighbors. Accordingly, the receivers update their tracked peer lists and SATs. We call the interval between broadcasts the *sampling time*. To guarantee the correctness of the SAT, a client u sets L_u as the peers from which the tuple streams are received in the last round. In particular, the tuple streams can be combined with periodic beacons [21], in which case L_u always equals the 1-hop neighbors B_u^1 of u.
2. Adaptive SAT maintenance. Under this policy, a peer broadcasts the tuple stream to its 1-hop neighbors adaptively, e.g., when updates on the cache occur or the number of new 1-hop neighbors exceeds a threshold. Each time when client u receives a tuple stream from client v, v is added into L_u if v does not exist in L_u and the signatures of the SAT is updated. Therefore, L_u may also contain peers which were 1-hop neighbors of u. Specifically, when the size of L_u reaches a the preset maximal size of L_u, the oldest peer of L_u will be selected as a victim to be removed. Hence, we can keep the space overhead of the signatures in SAT small.

V. Collaborative Query Processing

In SECC, collaborative query processing is invoked when a spatial query Q cannot be resolved locally with the cache of the query client u_Q. Hence, the remainder query resulted from local processing is delivered to helper peers for further processing. Two critical issues facing collaborative query processing are: (i) to efficiently identify helper peers which maintain the maximal number of index nodes and objects

useful for evaluating the remainder query, nd (ii) to properly merge the partial results and supporting index nodes returned from the helper peers to generate the query answer. Formally, let Q_r be the remainder query and C_{Q_r} be the set of index nodes and objects necessary for resolving Q_r. Consider a peer set B_{u_Q} consists of peers reachable in any number of hops by the query client u_Q. Therefore, the problem of collaborative query processing for Q_r can be formulated as building a query plan based on a subset of peers $B \subseteq B_{u_Q}$ with minimal processing cost such that $\forall B' \subseteq B_{u_Q}$ $(B' \neq B)$, $((\cup_{u' \in B'} C_{u'}) \cap C_{Q_r}) \subseteq ((\cup_{u' \in B} C_{u'}) \cap C_{Q_r})$.

From the above description, we can see that the cost of collaborative query processing mainly depends on three factors: 1) the topological relations among the helper peers and query client in the MP2PN, 2) the distribution of index nodes and objects among helper peers that could be used for evaluating the remainder query, and 3) the query plan. That is, the schedule of query processing among helper peers and the merge plan for the partial results and index nodes.

In this paper, we adopt a greedy method for collaborative query processing that takes into account all three factors discussed above. In this method, helper peers are selected in a localized and distributed fashion. The core idea is to let collaborative query processing proceed in a number of rounds. At round i, decisions are made independently from the $(i-1)$-hop neighbors to select helper peers from the i-hop neighbors for query processing. Moreover, when the processing results are returned from helper peers in round i, an efficient result merge procedure is performed to generate results for return to the query peer. Fig. 6 illustrates the process of collaborative query processing. Specifically, round i consists of two phases:

1. Peer selection. In this phase, the updated remainder query in round i denoted by Q_r^i is propagated from query client u_Q to all $(i-1)$-hop neighbors. Upon reception of Q_r^i, an $(i-1)$-hop neighbor u of u_Q first processes Q_r^i with its local cache. If Q_r^i is solved, u immediately returns the result and supporting index nodes to u_Q and the whole collaborative query processing ends; otherwise, u finds helper peers based on its SAT from its 1-hop neighbors B_u^1 for the updated remainder query $Q_r^{i,u}$ and forward $Q_r^{i,u}$ to them for processing. For example, in Fig. 6, u_A, u_B, u_C, and u_D independently determine the helper peers from their neighbors at round 2. Considering that the helper peers in $(i-1)$-hop neighbors have been selected in the last round, u usually can not process Q_r^i further. Therefore, it mainly plays the role of a decider of helper peers instead of a helper peer in this round. The details about *1-hop peer selection* are given in Section V-A. Obviously, if 1-hop peer selection guarantees that the selected peers can contribute to the remainder queries, then the selected peers must be i-hop neighbors of u_Q, since all the peers within $i-1$ hops have processed the query.

2. Result merge. In this phase, the partial results and index nodes generated by the i-hop helper peers and $(i-1)$-hop neighbors (if they process the remainder query further) will be merged along with the path automatically formed during query propagation in phase 1 (i.e., indicated by the solid arrows in Fig. 6). In particular, each intermediate peer of the tree will merge the results and index nodes received and then return the improved result and index nodes (if the query

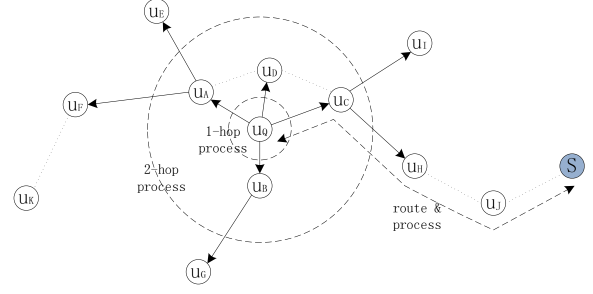


Fig. 6. SAT-based collaborative query processing. The white nodes denote the peers and the dark node denotes the server.

is solved, the intermediate peer will immediately returns the result and supporting index nodes to u_Q). As shown in Fig. 6, this merge process may happen in u_A, u_C, and u_Q at round 2. The details about result merge are given in Section V-B. Note that index merge simply removes the index nodes which proves to be useless for the query (e.g., for a kNN query, the index nodes with farther distance than the kth nearest object currently found are certainly useless). Finally, at u_Q, the whole collaborative query processing ends if one of the following situations happens, 1) The query is solved; 2) No helper peers can be selected in the i-hop neighbors; 3) The round number i achieves the maximum round number λ or $h_S - 1$, where h_S denotes the routing length to the nearest server. Otherwise, it will enter the next round of query processing.

Based on the independent SATs maintained by $(i-1)$-hop neighbors, the effectiveness of peer selection at round i may degrade as i grows because more peers with the same contribution to the query may be selected. Moreover, the peers farther away from the query client are less likely to contain the query results, we set a maximum round number λ to guarantee the quality of collaborative query processing. We call the collaborative query processing with maximum λ rounds the *λ-hop collaborative query processing*.

A. 1-hop Peer Selection

1-hop peer selection aims at finding proper peers within one hop of peer u for the remainder query Q_r. Based on the problem formulated earlier, we have to identify the set of index nodes and objects necessary for resolving Q_r, denoted as C_{Q_r}, to make proper decision. However, it usually incurs significant processing and transmission overheads due to the complexity of spatial query processing. Hence, we devise an efficient approximation of C_{Q_r}, which does not miss any node that is necessary for evaluating the remainder query. Since the set of index subtrees $\{C_e\}$ (including index nodes and objects) rooted at the missing entries $\{e\}$ in the remainder query Q_r can guarantee a complete answer of the remainder query, we have $C_{Q_r} \subseteq \cup_{e \in Q_r} C_e$. Thus, we can use $\cup_{e \in Q_r} C_e$ to approximate C_{Q_r} and apply the heuristic function $(\cup_{e \in Q_r} C_e) \cap (\cup_{u \in B} C_u)$ to measure how much a peer set B may contribute to the remainder query.

Specifically, with the SAT of u (denoted as r_u), we only consider the peers of $L'_u = B_u^1 \cap L_u$. Assuming that peers that have poor network and workload connections have been screened out, the selection of peers for answering a remainder query is purely based on the cache structures of the peers. Note that in SECC when an index node is cached, all its ancestors

are also cached. Thus, the issue is to find minimal number of peers B that cache all the leaf nodes of the helpful subtrees contributed by L'_u, i.e., $(\cup_{e \in Q_r} C_e) \cap (\cup_{u' \in L'_u} C_{u'})$. Let S_{Q_r} denote the set of the signatures in r_u which correspond to those leaf nodes. Then, the requirement can be described as,

$$\forall s \in S_{Q_r}, \exists u' \in B, s^{pos_{u'}} = 1,$$

where s^j denotes the jth bit of signature s and $pos_{u'}$ the position of peer u' in L_u. This is a set cover problem, for which a greedy algorithm can achieve an approximation ratio of $ln|S_{Q_r}|$ [22]. To illustrate this process, recall our running example in Fig. 3, in which the remainder query $Q_r = (Q, \{A, 6, d, 7, 5\})$, the missing entries are A and d, and hence S_{Q_r} contains the leaf signatures descended from A and d. Suppose $L_u = \{u_A, u_B, u_C, u_D\}$ equals B_u^1 and $S_{Q_r} = \{s_2 = 1001, s_b = 0001, s_8 = 1010, s_9 = 0110\}$. The greedy peer selection procedure will identify $\{u_A, u_B\}$ as the minimal subset covering S_A and S_d and hence the selected peers.

B. Result Merge

Suppose N partial results $\{R_i\}$ are collected by an intermediate peer u for the remainder query Q_r, where $R_i = (H_i, O_i)$ is obtained from cache C_i (the union of the partial R-trees of the helper peers which generates R_i), H_i is the priority queue containing the result entries and O_i is the result objects available in C_i. Result merge aims at generating an improved result $R = (H, O)$ by merging the information of $\{R_i\}$, which guarantees that H can be expanded to contain the final result queue. Obviously, if R is equal to the result generated from $C = \cup_{0 \le i \le N} C_i$, where $C_0 = C_u$, it is a merged result close to the final result with respect to currently known cached nodes. Thus, we call it the *optimal merged result*.

Algorithm 2 General Result Merge

Input: N partial results $\{R_i = (H_i, O_i)\}$
Output: merged result $R = (H, O)$

```
    ResultMerge(Result {R_i})
1:  let H = ∪_{1≤i≤N} H_i;
2:  for ∀e ∈ H do
3:      if ∃H_i(∄e' ∈ H_i, e' ∈ A(e) or e = e') then
4:          remove e from H;
5:      end if
6:  end for
7:  prune H according to the query Q
8:  build O according to H;
9:  return (H, O);
```

Since the result object set O is determined by the priority queue H, we focus on the merge of priority queues. If the ancestor-descendant relationship among the entries of any two priority queues (denoted as the *necessary ancestor-descendant relationship information*) is given, we are able to get the optimal merged result. Let $A(e)$ denote the set of entry e's ancestors in the R-tree. Algorithm 2 depicts the steps for merging a set of results when the necessary ancestor-descendant relationship information is available. Algorithm 2 guarantees the priority order when H is generated. In Step 7, the entries which do not satisfy the query are pruned. For example, kNN queries require pruning, while window queries do not result in invalid entries and hence do not require pruning. Finally, in Step 8, only the objects from the received results which currently satisfy the query are maintained. Based on Algorithm 2, we have Theorem 1 (the detailed proof is omitted due to space limitation).

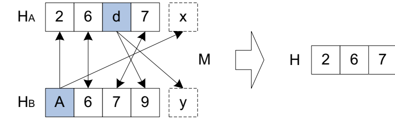


Fig. 7. An example of exact result merge.

Theorem 1: The merged result generated by Algorithm 2 is the optimal merged result. Let H be the merge queue in Algorithm 2, and H' the final queue generated from $C = \cup_{0 \le i \le N} C_i$. Then we have $H = H'$.

According to Theorem 1, we can obtain the optimal merged result if we have the necessary ancestor-descendant relations. This is called *exact result merge*. In the running example, consider the remainder query $Q_r = (Q, \{A, 6, d, 7, 5\})$ after local query processing. In round 1 of collaborative query processing, suppose u_Q issues Q_r to both u_A and u_B, then u_Q will receive the partial results $R_A = (\{2, 6, d, 7\}, \{o_2\})$ and $R_B = (\{A, 6, 7, 9\}, \{o_9\})$ from u_A and u_B (see Fig. 5), respectively. Fig. 7 shows the result merge by applying Algorithm 2 to the running example, in which x and y denote, respectively, the pruned entry sets of H_A and H_B. As shown in the figure, there exists a mapping M which represents the ancestor-descendant relationship among the entries. That is, each directed edge in M denotes the fact that the starting entry is an ancestor of the ending entry.

If the necessary ancestor-descendant relations are not given, the optimal merged result can not be surely achieved. We provide an example here. Consider the case that query client u_Q only contains node *root*, client u_A contains *root*, A, B, a, b, and c, and client u_B contains *root*, A, B, c, and d. We can still get the same H_A and H_B as shown in the running example. Fig. 8 shows the result merge without the necessary ancestor-descendant relations, which is denoted as *approximate result merge*. G includes all possible ancestor-descendant relations among the entries. For example, the possible ancestor-descendant relations can be derived from the containment relationship among the entries (i.e., each containment relation can be regarded as a candidate ancestor-descendant relation), although they are different from the actual mapping M. Now we can infer two possible mappings M_1 and M_2 from G, which lead two possible merged results $H = \{2, 6, 7\}$ and $H = \{2, 6, d, 7\}$. In this case, we cannot decide which is the optimal merged result.

Based on the above observation, we obtain Theorem 2:

Theorem 2: The optimal merged result can be achieved iff the necessary ancestor-descendant relations are available.

Based on Theorem 2, we can either acquire the necessary ancestor-descendant relations to achieve exact result merge or pursue approximate result merge (e.g., based on the containment information of the results). Since the supporting indexes can infer the necessary ancestor-descendant relations, SECC can obtain the optimal merged result when both the partial

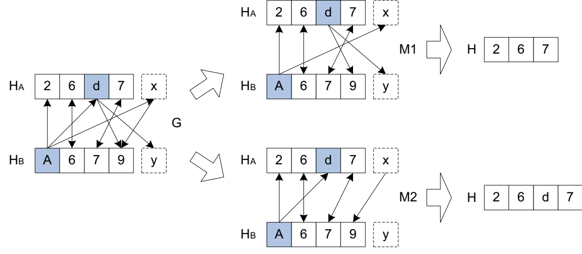


Fig. 8. An example of approximate result merge.

results and the supporting indexes are available.

VI. COLLABORATIVE CACHE REPLACEMENT (CCR)

If the cache is full when new data is admitted, cache replacement occurs. The general cache replacement problem can be formalized as follows. Let m denote the size of cache C, $size(n)$ the size of node n, and $b(n)$ the benefit of the cached node n, which can be an index node or an object. Given an incoming node of size m', cache replacement aims at finding a subset $C' \subset C$ to be replaced such that $\sum_{n \in C-C'} size(n) \leq m - m'$ and $\sum_{n \in C-C'} b(n)$ is maximized. Basically, it is a $0/1$ Knapsack problem. A greedy algorithm which removes the worst nodes (including objects) with minimal $b(n)/size(n)$ proves to be an effective method (for the unbounded knapsack problem, it is the 2-approximation algorithm).

SECC requires a collaborative cache replacement scheme, since the replacement decision made by a client not only relies on the client itself but also its neighboring clients because a required item may be got from the neighbors as well. In this paper, due to the fact that a client has the highest priority on its own resources and clients are independent from each other, we devise a locally optimal cache replacement algorithm, which takes into consideration the cache structures of the nearby peers (i.e., 1-hop neighbors). Let C'' denote the incoming nodes. Formally, the local optimization problem can be described as,

$$P : Maximize \sum_{n \in C \cup C''} b(n) \times x_n$$

$$s.t. \sum_{n \in C \cup C''} size(n) \times x_n \leq m,\ x_n \in \{0, 1\}$$

Note that in traditional cache replacement methods, the incoming nodes will always replace some existing cached nodes, but our method considers the overall benefit of a cache and does not distinguish the incoming nodes from the previously cached nodes. As we will show later, with a greedy algorithm, this generalized form generates the same replacement policy as a traditional replacement policy in the client-server model. However, in SECC, the situation is quite different since the caches in nearby peers have great impacts on the solution.

We define the benefit of a cached node as the cost of retransmitting the node via the wireless channel when the node is requested by a future query. Let $p(n)$ denote the access probability of cached node n. It can be estimated according to the following formula:

$$p(n) = \left(\frac{1}{T - last_access_time(n) + 1}\right)^{\alpha} \quad (1)$$

where $last_access_time(n)$ is the time when node n was last accessed and T is the time of the current query, which are represented logically by the sequence ids of the respective queries, and α ($\alpha > 0$) a factor determined by the properties of the query client (e.g., the speed and the query pattern). For the client-server model in which the communication cost is constant, the benefit, denoted by $b_S(n)$, can be measured by:

$$b_S(n) = p(n) size(n) c_S, \quad (2)$$

where c_S denotes the transmission cost of a unit data from the server. Define the unit benefit $\Delta b(n) = b(n)/size(n)$. We have $\Delta b_S(n) = p(n)c_S$. For incoming nodes, since $p(n) = 1$, $\Delta b_S(n) = c_S$, which is no less than any previously cached node. With a greedy algorithm, the generalized form of cache replacement generates the same victims as the traditional method.

In SECC, the benefit of a cached node is due to savings on the cost of accessing the server via a multi-hop routing (denoted as $b_S(n)$) and on the MP2PN (denoted as $b_M(n)$). $b_S(n)$ is the same as in Eqn. 2. $b_M(n)$, however, is more complicated, because retrieving the missing nodes in MP2PN may lead to duplicates when multiple peers are selected. Due to the fact that 1-hop neighbors have more stable connections to the query peer and play a more important role in the collaborative query processing (because of the locality of the spatial queries) compared to other peers, we only consider them to maximize the savings of the collaborative caches.

Let $N(n)$ denote the number of 1-hop neighbors selected to retrieve n in peer selection. Then,

$$N(n) \leq ones(s_n),$$

where s_n is the signature maintained by the query peer for node n and $ones(s_n)$ denotes the number of ones in s_n. Let β ($0 < \beta < 1$) be the probability that a requested neighbor cannot return node n (e.g., because the neighbor cannot be accessed or its cache has been changed). Therefore, β depends on the properties of the local networks, e.g., variation of client connections and cache updates. If node n is not cached, retrieving n on MP2PN incurs an overall data size $size_M(n)$:

$$size_M(n) = size(n) \sum_{i=1}^{N(n)} i C_i^{N(n)} (1-\beta)^i \beta^{N(n)-i}, \quad (3)$$

where $C_i^{N(n)}$ is the number of combinations of selecting i elements from a set of $N(n)$ elements and $C_i^{N(n)}(1-\beta)^i\beta^{N(n)-i}$ denotes the probability of returning i nodes from the $N(n)$ collaborative neighbors. Eqn. 3 can be simplified to:

$$size_M(n) = (1-\beta)N(n)size(n) \quad (4)$$

Based on this, $b_M(n)$ can be computed as:

$$b_M(n) = p(n) size_M(n) c_M, \quad (5)$$

where c_M denotes the transmitting cost of a unit data from 1-hop neighbors in the MP2PN. Considering the probability $\beta^{N(n)}$ that none of requested neighbors returns node n when $s_n \neq 0$, the expected benefit $b(n)$ can be represented as:

$$b(n) = \begin{cases} b_S(n), & s_n = 0 \\ \beta^{N(n)} b_S(n) + b_M(n), & s_n \neq 0 \end{cases}$$

Using $ones(s_n)$ to estimate $N(n)$, the unit benefit $\Delta b(n)$ can be computed as:

$$\Delta b(n) = \begin{cases} p(n)c_S, & s_n = 0 \\ p(n)(\beta^{ones(s_n)}c_S + (1-\beta)ones(s_n)c_M), & s_n \neq 0 \end{cases}$$

From the above formula, we can observe that although an incoming node n may have high access probability $p(n)$, it does not necessitate a high unit benefit $\Delta b(n)$.

In this paper, we use the connection length in the MP2PN to reflect the benefit of a cached node. Let h_S be the connection length from the query peer to the nearest server in the MP2PN (assume that it is provided by the routing scheme). Therefore, $c_S = h_S$ and $c_M = 1$. Thus, $\Delta b(n)$ can be rewritten as:

$$\Delta b(n) = \begin{cases} p(n)h_S, & s_n = 0 \\ p(n)(\beta^{ones(s_n)}h_S + (1-\beta)ones(s_n)), & s_n \neq 0 \end{cases}$$

Moreover, since in SECC if the cached node n is removed all of its cached descendant nodes are removed, our cache replacement strategy compares the value $\Delta b(n)$ for each cached leaf node (including incoming objects and supporting index nodes) and removes those with the smallest $\Delta b(n)$ to free up memory space. The detailed algorithm is described in Algorithm 3, where $\delta = h_S$. In our implementation, we treat α and β as empirical values and use the $last_access_time$ associated with each cached node to estimate the access probability $p(n)$ based on Eqn. 1.

Algorithm 3 Cache Replacement

Input: SAT cache C, cache size m, incoming nodes C''
Output: new SAT cache C
CacheReplacement(SAT cache C, Nodes C'')
1: keep a priority queues H whose key is $\Delta b(n)$;
2: **for** each leaf node n in $C \cup C''$ **do**
3:   **if** $s_n = 0$ **then**
4:     $\Delta b(n) = p(n)\delta$;
5:   **else**
6:     $\Delta b(n) = p(n)(\beta^{ones(s_n)}\delta + (1-\beta)ones(s_n))$;
7:   **end if**
8:   put n into H;
9: **end for**
10: **while** $\sum_{n \in C \cup C''} size(n) > m$ **do**
11:   pop n from H;
12:   remove node n from $C \cup C''$;
13:   **if** n is its parent n_p's last child **then**
14:     create signature subtree S based on (nid_n, s_n) and $\bigcup_{e \in n} S_e$;
15:     attach S to the corresponding entry of n_p;
16:     **if** $s_{n_p} = 0$ **then**
17:       $\Delta b(n_p) = p(n_p)\delta$;
18:     **else**
19:       $\Delta b(n_p) = p(n_p)(\beta^{ones(s_{n_p})}\delta + (1-\beta)ones(s_n))$;
20:     **end if**
21:     push n_p into H;
22:   **end if**
23: **end while**
24: **return** C;

TABLE I
SYSTEM PARAMETER SETTINGS

Parameter	Value	Parameter	Value
Client #	200	Object #	2000
Server #	5	Trans. rate	$100kbps$
Pause time	$0 \sim 10s$	Trans. range	$200m$
Speed	$1 \sim 2m/s$	Trans. overhead	$32B$
Tuple size	$4B$	Avg. think time	$50s$
Cache size	$2 \sim 5\%$	Page size	$1KB$
K_{max}	5	$Area_{wnd}$	$10000m^2$

VII. PERFORMANCE EVALUATION

A. Simulation Model

We built an event-driven simulator to evaluate the performance of SECC. It simulates a Wi-Fi-based MP2PN and captures the states of the simulated nodes (including the clients and the servers). We assume that the servers are powerful enough to satisfy concurrent query requests from the clients.

The simulated mobile environment is composed of 5 fixed servers and 200 mobile clients, randomly scattered in an area of $2000m \times 2000m$. All mobile clients move within this area and follow the random waypoint mobility model. Query arrival is modeled as a Poisson process. That is, after a client completes its current query, it waits for an exponentially distributed random period, called $thinking\ time$, before it issues a new query. We implemented two basic types of spatial queries, namely, range query and kNN query, which were randomly picked for execution in the experiments. The window of a range query is centered at the client's current position with average size $Area_{wnd}$. The value of k for a kNN query is randomly chosen from 1 to K_{max}. The dataset contains 2000 objects randomly distributed in the area. The average object size $\overline{|o|}$ is $3.05KB$. The sizes of the objects follow a Zipf's distribution with skewness parameter $\theta = 0.8$. The page capacity is set to $1K$ for the R-tree indexes. The cache size for each client is the same, with a default value of 2% of the total dataset size. Important parameter are summarized in Table I.

We implemented SECC on the proposed periodical SAT update and adaptive SAT update schemes, which are denoted as SECC-PS and SECC-AS, respectively. The default sampling time of SECC-PS is set to $30s$ and the threshold for the new neighbors in SECC-AS is set to half of the 1-hop neighbors. We also implemented three other collaborative caching schemes for comparison:

- Item-based collaborative caching (CC-IC). CC-IC is based on the item-based caching scheme [10], which simply submits the query and the identifiers of all cached objects to the server and the server returns the missing objects for the query. In CC-IC, when the query and the identifiers are routed through a MP2PN to the server, each peer along the path will add new identifiers according to its local cache. Then, the server only returns the result objects which are missed in all of the peers along the path. For other result objects which are missing to the query peer, they must exist in some peers along the path and thus will be routed back from the nearest one to the query peer.
- Proactive caching based collaborative caching (CC-PC). In CC-PC, the peers in the MP2PN adopt the proactive

caching scheme [14] and the query peer directly sends the remainder query to the server. During the routing, the remainder query will be further processed at each intermediate peer based on the local cache (as step 2 of the collaborative processing in SECC). If the query is not solved during routing, the server will received it and process it for the final result.

- SECC with λ-hop flooding (SECC-FL). In SECC-FL, a query client simply floods the remainder query to its λ-hop neighbors before sending it to the server. At each intermediate peer, the remainder query is updated with the local cache and forwarded further. Finally, result merge is performed as in the SAT-based SECC schemes. As a special form, SECC-FA denotes the SECC-FL scheme with $\lambda = +\infty$. Thus, in SECC-FA, the remainder query is flooded to all the peers within $h_S - 1$ hops (h_S denotes the routing length to the nearest server).

For the SECC schemes, we set $\lambda = 3$ as the default value. To ensure a fair comparison, we choose the state-of-the-art cache replacement schemes for each caching model: LRU for item-based caching, GRD3 for proactive caching and SECC-FL, and CCR ($\alpha = 2$ and $\beta = 0.8$) for SAT-based SECC schemes.

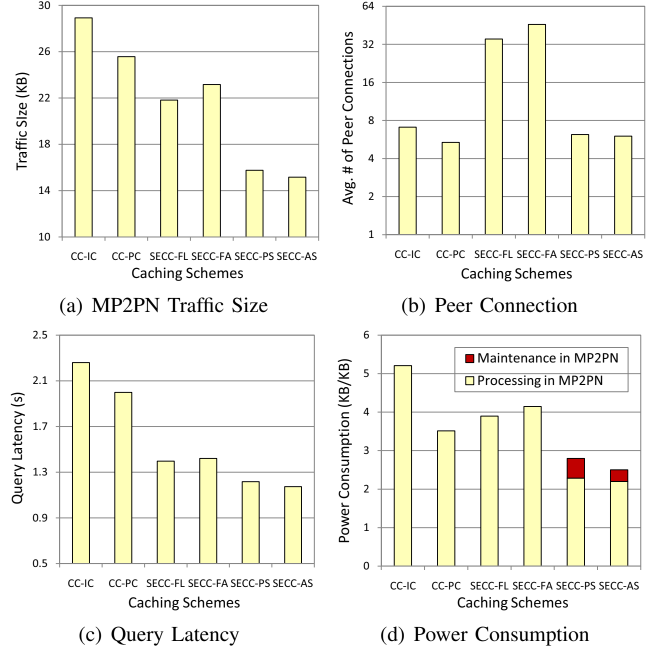
The metrics for performance comparison reflect five aspects of each scheme: a) MP2PN traffic size, b) number of peer connections, c) query latency, d) power consumption, and e) cache hit rate. All the metrics except power consumption are measured in the query processing time. However, power consumption is measured in both processing and maintaining time (e.g., signature updates). Particularly, we use the average transmission size (including in and out) for obtaining $1KB$ result data to represent the power consumption. Therefore, the unit of power consumption is KB/KB. Cache hit rate (denoted as CHR) is defined as

$$hit_c = \frac{|R_c|}{R} \tag{6}$$

where R_c denotes the partial result objects obtained from the caches and R the whole result objects. It reflects how many results can be obtained from the caches (i.e., local cache or MP2PN caches).

B. Overall Performance Comparison

Fig. 9 shows the performance comparison among different caching schemes. The plots are obtained from the average query performance of all 200 peers in $2000s$ of simulation time. In terms of MP2PN traffic size, Fig. 9(a) shows the average data transmission volume for a query in each scheme. The SECC schemes, relying on helps from the nearby peers, can greatly reduce the traffic in MP2PN. For example, the MP2PN traffic size in SECC-AS is only 50% that of CC-IC and 60% that of CC-PC. Besides, the SAT-based SECC schemes (i.e., SECC-PS and SECC-AS) transmit 30% less data than non-SAT SECC schemes (i.e., SECC-FL and SECC-FA). Fig. 9(b) gives a comparison of the average number of peer connections. SECC-FL and SECC-FA incur much more peer connections than other schemes, which means the trivial flooding method causes much overhead and is not suitable for collaborative spatial query processing. On the contrary, SECC-AS and SECC-PS, by utilizing SAT, achieve more effective



(a) MP2PN Traffic Size (b) Peer Connection (c) Query Latency (d) Power Consumption

Fig. 9. Overall performance comparison.

TABLE II
CACHE HIT RATE

Rate	CC-IC	CC-PC	SECC-FL/-FA	SECC-PS/-AS
hit_{local}	0	0.414	0.414 / 0.414	0.416 / 0.416
hit_{MP2PN}	0	0.068	0.392 / 0.396	0.349 / 0.363
hit_{server}	1	0.518	0.194 / 0.19	0.235 / 0.221

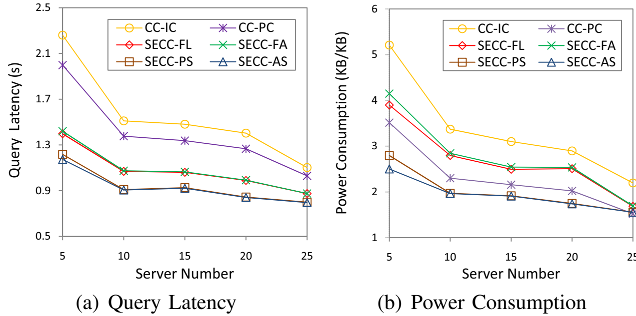
peer selection and thus requires much less connections, which is even less than that of CC-IC.

We conduct experiments to evaluate the query latency (as shown in Fig. 9(c)). Our experimental result shows that the transmission cost dominates the peer processing cost, i.e., the CPU time is negligible. Thus, we only show the latency delay caused by accesses to MP2PN. The SECC schemes significantly outperform CC-IC and CC-PC by at least 40%. In particular, the SAT-based schemes achieve shorter latency than SECC-FL by around 15%. For power consumption as shown in Fig. 9(d), we can see that the SAT-based SECC schemes incur much less energy cost, i.e., 50% that of CC-IC or 70% that of IC-PC, while the non-SAT SECC schemes perform worse than CC-PC. The maintenance cost in MP2PN indicates the delivering cost for SAT maintenance. We can see that by spending small energy on SAT maintenance, SECC-AS and SECC-PS gain much more benefit in query processing. Compared to SECC-PS, SECC-AS costs less energy but achieves better query latency. It means most of the periodical stream exchanges are unnecessary. Considering that the non-SAT SECC schemes incur much more peer connections per query, the cost of connections and radio collisions among peers may further degrade their performance. Therefore, the SAT-based SECC schemes are preferred in practice. In the experiment, the storage overhead of the signatures is 0.2% for SECC-PS and 0.1% for SECC-AS. The average size of the tuple stream is $0.32KB$, 0.26% of the cache size, which only costs about $2.5ms$ for transmission.

Table II summarizes the cache hit ratios at different levels among the caching schemes. As we can see, the query pro-

TABLE III
AVERAGE ROUTING LENGTH TO SERVER

Server #	5	10	15	20	25
Avg. length to server	3.54	2.51	2.38	2.29	1.79



(a) Query Latency (b) Power Consumption

Fig. 10. Performance comparison under various server numbers.

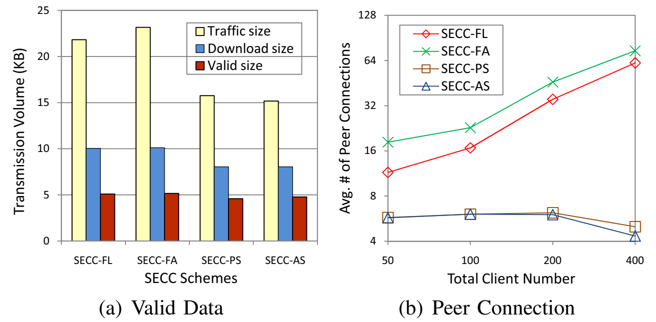
cessing for the SECC schemes is concentrated on the local cache and the collaborative caches in MP2PN. The cache hit rates of the local cache and MP2PN for them reach 0.8. In other words, 80% of the result objects can be retrieved from a client's own cache or the peer caches before a remainder query is submitted to the server. This value is almost twice that of CC-PC. For the special cases when the servers are not reachable from the query peer, the SECC schemes may still provide the results or partial results. For example, the SECC schemes in the experiments provide 35% of the results when the servers are not available.

C. Impact of Server Number

In the experiments, we use server number to control the average routing length to the nearest server. Table III summarizes the relations between them. Obviously, a smaller number of servers usually lead to larger average routing length to server. The performance of the caching schemes under various number of servers is presented in Fig. 10. For query latency (Fig. 10(a)), the SECC schemes always outperform CC-IC and CC-PC under different server numbers. The gap between their performance becomes more significant when the server number decreases. It means when the average routing length to server is big, finding the nearby helpers in MP2PN is more valuable. Similar findings can also be found in power consumption (as shown in Fig. 10(b)).

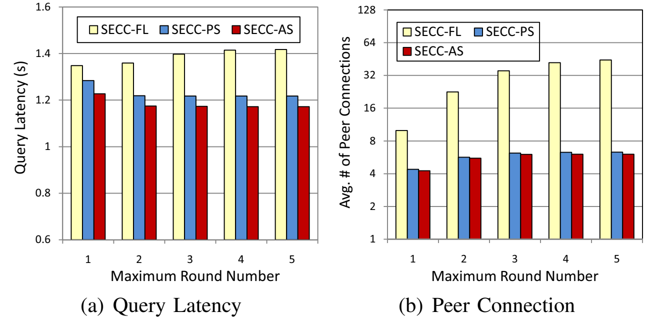
D. Effectiveness of Collaborative Query Processing

This section examines the performance of the collaborative query processing in different SECC schemes. Fig. 11(a) shows the detailed traffic volumes in MP2PN, where valid size denotes how many of the downloaded objects are the result objects of a query. For SECC-FL, the download size is much smaller than the traffic size, which verifies the effectiveness of the result merge scheme. The SAT-based schemes cause much less traffics in MP2PN and their valid sizes are quite close to SECC-FL. It means that the SAT-based schemes can effectively determine the peers which contain the largest number of index nodes and objects required for evaluating the remainder query. Although SECC-FA gets the largest number of valid result objects due to its largest probing area, the communication overhead is extremely high. This is because the clients far away from the querying point are less likely



(a) Valid Data (b) Peer Connection

Fig. 11. Performance of collaborative query processing for the SECC schemes.



(a) Query Latency (b) Peer Connection

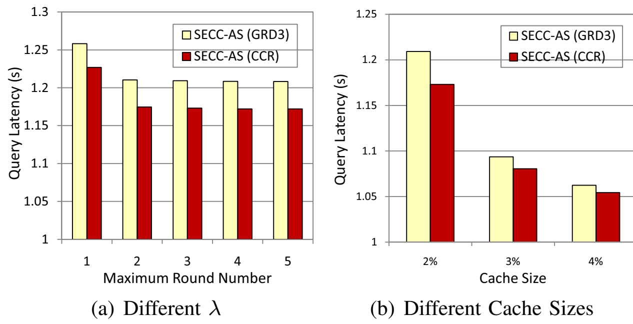
Fig. 12. Performance comparison among the SECC schemes under various maximum round number λ.

to have cached the result objects. In particular, the neighbors within 2 hops have cached nearly 80% of the total valid result objects in all peers. Moreover, Fig. 11(b) summarizes the average number of peer connections during collaborative query processing with respect to different client densities. The SAT-based SECC schemes always incur significantly less connections than SECC-FL and SECC-FA. Moreover, when the clients are dense enough (e.g., client number = 400 in the experiments) that all the helpers can be found in one hop, then less peer connections are required. Similar results can also be found for other metrics like query latency and power consumption. Thus, our SAT-based collaborative query processing is proved to be cost-efficient.

To explore the impact of the maximum round number λ, Fig. 12 shows the performance comparison among the SECC schemes under various λ. The performance of the SECC schemes do not always improve as more peers are allowed to access. As shown in Fig. 12(a), after some point, the query latency increases (e.g., for SECC-FL) or remains almost the same (e.g., for SECC-PS and SECC-AS). Similar results can also be found for power consumption (the Fig. is not provided because of space limitation). It is because the peers far away from the query client are less likely to cache the necessary nodes for the query. Moreover, from Fig. 12(b), we can see that the increment of the number of peer connections in the SAT-based SECC schemes is significantly smaller than that of SECC-FL, which means our collaborative query processing can effectively avoid the negative effect of the distributed decision making for peer selection.

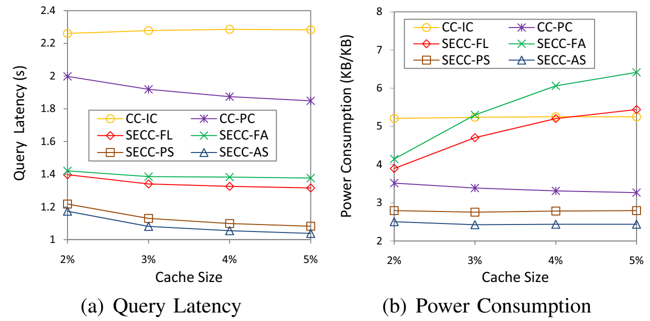
E. Impact of Cache Sizes and Replacement Schemes

In this section, we investigate the effectiveness of the collaborative cache replacement method. For comparison, we also implement some variations of the SAT-based schemes, namely,



(a) Different λ (b) Different Cache Sizes

Fig. 13. Effectiveness of CCR compared to GRD3.



(a) Query Latency (b) Power Consumption

Fig. 14. Performance comparison under various cache sizes.

SECC-PS(GRD3) and SECC-AS(GRD3), in which GRD3 (the state-of-the-art method) is adopted as the cache replacement algorithm. Fig. 13 shows the performance comparison between SECC-AS(GRD3) and the original SECC-AS which adopts the collaborative cache replacement algorithm (denoted as SECC-AS(CCR)). Generally, SECC-AS(CCR) outperforms SECC-AS(GRD3) in different settings. Since CCR targets at local optimization, the improvements of CCR are similar for different λ, as shown in Fig. 13(a). Moreover, Fig. 13(b) shows the improvement of CCR increases as less cache is available. Since the query processing relies more on the caches in MP2PN for smaller cache sizes, the cache replacement scheme plays a more important role in the SECC schemes. Similar results can also be observed for SECC-PS.

We further compare the performance of different caching schemes with four cache size settings (see Fig. 14): 2%, 3%, 4%, and 5%. In general, the SAT-based SECC schemes outperform other schemes under each setting in terms of query latency (Fig. 14(a)) and power consumption (Fig. 14(b)). As the cache size of each client increases, the performance of the SAT-based SECC schemes with $\lambda = 3$ improves accordingly. On the contrary, the non-SAT SECC schemes outperform CC-IC and CC-PC in query latency but get worse performance after some points in power consumption. It means that, when effective helper selection schemes are not available, a larger cache size yields more redundant transmissions, thus degrading the performance of SECC.

VIII. Conclusion

In this paper, we proposed the Structure-Embedded Collaborative Caching (SECC) method, which supports spatial query processing in a MP2PN by utilizing the caches of a set of collaborative peers and, when complete answer cannot be obtained from the peers, sends the remainder query to the server for processing through the MP2PN. To facilitate SECC, we designed a novel index structure, called Signature Augment Tree (SAT), which can cost-efficiently record both the index nodes and objects in the nearby peers. Based on SAT, we developed a cost-efficient collaborative query processing scheme and an effective algorithm for cache replacement in a collaborative environment. Two implementations of SECC were developed to explore different SAT maintenance policies. Extensive experiments have been performed to evaluate the performance of SECC. For future research, we will investigate problems associated with updates on the databases, e.g., cache invalidation and update propagation on SECC.

Acknowledgment

Qijun Zhu and Dik Lun Lee were supported by grant 615707 from Hong Kong Research Grant Council. Wang-Chien Lee was supported in part by National Science Foundation grants CNS-0626709 and IIS-0534343.

References

[1] M. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," in *Proceedings of the USENIX OSDI*, 1994.

[2] H. Shen, S. K. Das, M. Kumar, and Z. Wang, "Cooperative caching with optimal radius in hybrid wireless networks," in *NETWORKING*, 2004.

[3] L. Yin and G. Cao, "Supporting cooperative caching in ad hoc networks," in *IEEE INFOCOM*, 2004.

[4] W. H. O. Lau, M. Kumar, and S. Venkatesh, "A cooperative cache architecture in support of caching multimedia objects in manets," in *Proceedings of the ACM international workshop on Wireless mobile multimedia*, 2002.

[5] T. Hara, "Cooperative caching by mobile clients in push-based information systems," in *Proceedings of the CIKM*, 2002.

[6] N. Chand, R. C. Joshi, and M. Misra, "Supporting cooperative caching in mobile ad hoc networks using clusters," *International Journal of Ad Hoc and Ubiquitous Computing.*, 2007.

[7] B. Zheng and D. L. Lee, "Semantic caching in location-dependent query processing," in *Proceedings of the International Symposium on Spatial and Temporal Databases*, 2001.

[8] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee, "Location-based spatial queries," in *Proceedings of the SIGMOD Conference*, 2003.

[9] M. Fanklin, M. Carey, and M. Livny, "Global memory management in client-server dbms architectures," in *VLDB*, 1992.

[10] S. Acharya, M. Franklin, S. Zdonik, and R. Alonso, "Broadcast disks: Data management for asymmetric communication environments," in *Proceedings of ACM SIGMOD Conference*, 1995.

[11] Y. Huang, P. Sistla, and O.Wolfson, "Data replication for mobile computers," in *Proceedings of the SIGMOD Conference*, 1994.

[12] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *VLDB*, 1996.

[13] Q. Ren, M. H. Dunham, and V. Kumar, "Semantic caching and query processing," *IEEE TKDE*, 2003.

[14] H. Hu, J. Xu, W. S. Wong, B. Zheng, D. L. Lee, and W.-C. Lee, "Proactive caching for spatial queries in mobile environments," in *Proceedings of the ICDE Conference*, 2005.

[15] K. C. K. Lee, W.-C. Lee, B. Zheng, and J. Xu, "Caching complementary space for location-based services," in *EDBT*, 2006.

[16] D. Wssels and K. Claffy, "Icp and the squid web cache," in *IEEE Journal on Selected Areas in Communication*, 1998.

[17] P. Nuggehalli, V. Srinivasan, and C.-F. Chiasserini, "Energy-efficient caching strategies in ad hoc wireless networks," in *Proceedings of the ACM MobiHoc*, 2003.

[18] T. Hara, "Effective replica allocation in ad hoc networks for improving data accessibility," in *IEEE INFOCOM*, 2001.

[19] F. Sailhan and V. Issarny, "Cooperative caching in ad hoc networks," in *MDM*, 2003.

[20] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of ACM SIGMOD Conference*, 1984.

[21] B. Karp and H. T. Kung, "Gpsr: Greedy perimeter stateless routing for wireless networks," in *MobiCom*, 2000.

[22] U. Feige, "A threshold of ln n for approximating set cover," *Journal of the ACM (JACM).*, 1998.