Supporting Complex Multi-dimensional Queries in P2P Systems

Bin Liu[†] Wang-Chien Lee[§] Dik Lun Lee[†]
[†]*Department of Computer Science*
Hong Kong University of Science and Technology
Clearwater Bay, Hong Kong
{liubin, dlee}@cs.ust.hk
[§]*Department of Computer Science and Engineering*
Pennsylvania State University
University Park, PA 16802
wlee@cse.psu.edu

Abstract

More and more applications require peer-to-peer (P2P) systems to support complex queries over multi-dimensional data. For example, a P2P auction network for real estate frequently needs to answer queries such as "select five available buildings closest to the airport". Such queries are not efficiently supported in current P2P systems. Towards an efficient and scalable P2P system capable of processing complex multi-dimensional queries, we first propose a comprehensive framework for sharing, indexing, and querying multi-dimensional data, where (i) peers with more computational power coordinate indexing and query processing, and (ii) other peers participate in part of the computation in order to achieve scalability and load-balance. Based on this framework, we propose Network-R-tree (NR-tree), a P2P adaptation of the dominant spatial index - R-tree. NR-tree, indexing spatial data at clustered peers, is capable of processing complex queries such as range queries and k-nearest neighbor queries. We propose query processing algorithms for range and k-nearest neighbor queries and experimentally prove the effectiveness of proposed techniques with real data.*

1. Introduction

Peer-to-peer (P2P) has become a pervasive paradigm of data exchange. As more applications and users share or access data through P2P networks, many current systems seem inadequate due to the inability to support multi-attribute or multi-dimensional queries efficiently. The first generation of P2P systems, namely file sharing applications such as Gnutella [23] or most recent BitTorrent [22], support only keyword lookups and mostly provide no load-balancing. The second generation, including Chord [17], CAN [13], Pastry [14], and Tapestry [27], are mainly *structured* P2P systems supporting basic key-based routing while providing load-balancing and logarithmic hop routing. They are not yet designed for complex queries (e.g., range query, *k*-nearest neighbor query) in multi-dimensional data, but their desirable features make them fruitfully used by many third generation systems, such as CAN-MC [15] (CAN) and pSearch [21] (CAN). Some recent systems support only range queries (e.g., SkipNet [8] and Mercury [2]) or similarity queries (e.g., SSW [11]) and not requiring a structured overlay, but they cannot support other complex query types.

1.1 Motivation

Complex queries, such as range queries and *k*-nearest neighbor queries, are becoming more important as voluminous data are shared and more applications use the P2P paradigm. For example, a P2P auction network [19] where peers store information on local real estate (geographical location, price, etc) needs to frequently deal with queries such as "find available buildings at most 10 km from the city center" or "select five available buildings closest to the airport". In order to answer such queries efficiently in P2P networks, we need to take into account their characteristics. Studies such as [18] have shown that peer exhibits strong heterogeneity in computational power, storage, and bandwidth (e.g., up to 3 orders of magnitude difference in bandwidth). This phenomenon implies that an efficient and realistic P2P system should take into account the non-uniformity in computational resources among peers. Towards this, super-peer networks [26] have emerged as a powerful balance where peers with more resources take an *active* role as local servers while resource-scarce peers can be *passively* served. Such systems have already shown their flexibility and adaptability with applications such as KaZaa [24] and continue to grow with great potential. Therefore, enabling super-peer systems for complex multi-dimensional queries is the focus of this paper.

1.2 Contributions

This paper contains three important contributions that efficiently support complex multi-dimensional queries in super-peer systems. First, we propose a comprehensive framework for sharing, indexing, and querying multi-dimensional data in super-peer networks. In particular, peers with more computational power (super-peers) act as local servers to which peers with relatively weaker capability (passive-peers) connect and thus forming a cluster. In order not to overload super-peers and balance the workload, part of the indexing and query processing task is

pushed to passive-peers, which makes our systems scalable and stable.

Second, we propose the Network-R-tree (abbreviated as NR-tree), a P2P adaptation of the most popular multi-dimensional index, R*-tree. The NR-tree is build in the same spirit as R*-tree and therefore it can easily accommodate existing R*-tree based algorithms with minor modification. The NR-tree is built in super-peers by indexing minimum bounding rectangles (MBRs) *summarizing* peer data in the same cluster at a coarse level. It is thus light-weight and easy to build and maintain, with full capability of an R*-tree. NR-trees facilitate the processing of complex multi-dimensional queries in P2P networks. We also devise techniques for maintaining NR-trees under frequent node departures in order to adapt to the dynamic nature of P2P networks.

The last contribution is a thorough evaluation of proposed system under various settings. We perform experiments with different network sizes, data distribution, node failures, etc. By testing various factors, we experimentally prove the effectiveness of the proposed system.

The rest of the paper is organized as follows. Section 2 surveys previous work, focusing on complex queries in P2P systems and the R*-tree with its related algorithms, due to their immediate relevance with the NR-tree. Section 3 overviews the system and section 4 discusses cluster formation mechanism. Section 5 introduces the NR-tree construction and maintenance, before we discuss algorithms for range query and k-nearest neighbor query in section 6. Section 7 verifies the effectiveness of proposed techniques with rigorous experiments and section 8 concludes the paper with future work.

2. Related work

In this section, we will introduce the most related research results in peer-to-peer networks, with the focus on processing multi-dimensional queries. Another topic, R*-tree and its query processing algorithms, is also pertinent and will be introduces in section 2.2.

2.1 Query processing in P2P systems

Ever since the popularity of peer-to-peer file sharing applications (e.g., Gnutella), research community has put significant amount of effort on extending P2P networks or building new P2P systems to support more than just keyword search. The first significant step towards this goal is structured P2P networks, such as Chord [17], CAN [13], Pastry [14], and Tapestry [27]. They offer efficient lookup routing, load-balancing, resilience, and scalability by building self-organizing overlays to store and locate resources in P2P systems. These systems are not *directly* suitable for complex multidimensional queries (e.g., range query, nearest neighbor query) since they often use hashing for locating resources. Nevertheless, their desirable features make them often foundations of successive P2P systems. Among those, notably pSearch [[21], employs CAN for organizing the overlay network. It uses Vector Space Model (VSM) and Latent Semantic Indexing (LSI) to generate a semantic space and organize contents around their semantics. pSearch is still designed for structured networks due to the underlying CAN, which makes it not suitable for the P2P system we focus on.

Significant amount of work has been put on range queries in P2P systems. With techniques such as replicated B-trees [10], space filling curves [25] and skip lists (SkipNet [8]), range queries can be efficiently processed. By leveraging the properties of small-world networks, Mercury [2] can route range queries efficiently and Semantic Small World (SSW) [11] can process similarity queries. However, these systems have not yet considered other types of multi-dimensional queries such as *k*-nearest neighbor queries.

Very recently, more progress has been made towards supporting complex multi-dimensional queries. Demirbas and Ferhatosmanoglu propose a tree-based index for sensor networks [5], where sensors collectively build an index tree and process various types of spatial queries. However, their solution is specific to sensor-networks, and the performance is not clear since no experiments are presented. A P2PR-tree is proposed in [12], which is an extension of the R*-tree in P2P settings. However, the most important issue, maintenance of a dynamic R-tree in unstructured P2P networks is left unaddressed. The maintenance cost is expected to be considerably high, since every maintenance operation has to be communicated among peers with messages in order to finish. Zhang et al. propose a skip graph [1] based structure in [28], which is a multi-dimensional extension of SkipNet [8]. Most recently, a quad-tree-based technique is proposed by Tanin and Harwood [19] capable of supporting range queries, which, however, builds on Chord and makes it only suitable for structured P2P networks.

2.2 The R*-tree based query processing

Among various multi-dimensional access methods [6], the R*-tree [3], a variation of the original R-tree [7], has been widely accepted by industry and research community. The R*-tree is a multi-dimensional extension of a B+-tree. Figure 2.1 illustrates an R*-tree indexing a set of points {*a*, *b*, *c*,...} assuming a capacity of three entries per node. Points close in space (e.g., *e*, *f*, *g*) are clustered in the same leaf node (E_6) represented as a minimum bounding rectangle (MBR). Nodes are then recursively grouped together following the same principle until the top level, which consists of a single root.

R*-trees can efficiently answer various types of multi-dimensional queries, especially *range query*. Given a *query window q* (shaded region in Figure 2.1), a range query retrieves all objects inside or intersecting *q*. Range queries can be processed using the original algorithm from [7]. Processing starts from the root level of the R*-tree. For any entry whose MBR intersects the query window, its sub-tree is recursively explored. If a leaf entry is encountered, all objects whose bounding rectangle intersects the query window are examined. Entries not intersecting the query window (e.g., E_3) are not examined.

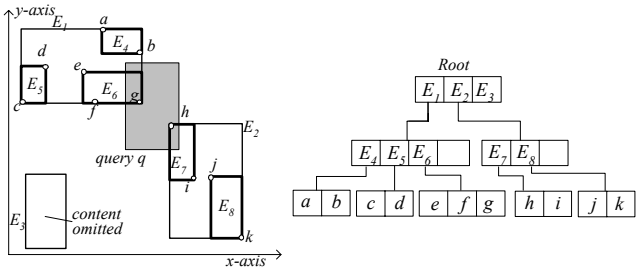


Figure 2.1: R-tree example

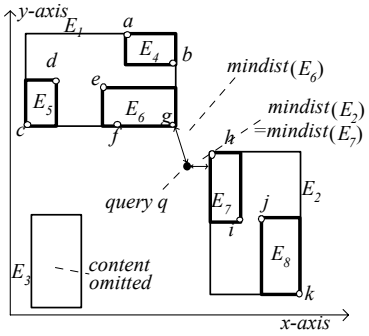


Figure 2.2: *k*NN example

Another important type of queries that R*-trees efficiently support is *k*-nearest neighbor (*k*NN) query. Given a query point *q* and parameter $k(\geq 1)$, a *k*NN query retrieves the top *k* objects (if possible) with the shortest distance to *q*. Roussopoulos et al. [16] propose a branch-and-bound algorithm (enhanced by Cheung and Fu [4]) that traverses the R*-tree in a depth-first manner. Specifically, starting from the root, entries are sorted according to their minimal distance (*mindist*) to query point *q*, and sub-tree of entries with smaller *mindist* are explored first. This process repeats recursively until the leaf level is reached and the first *k* candidates are discovered. Assume that mindist from farthest candidate is *nn_mindist*. During backtracking to upper levels, only entries with *mindist* smaller than *nn_mindist* are visited. We will illustrate this algorithm with an example in Figure 2.2. Assume we intend to find the 2-nearest neighbor of point *q*. Starting form the root level of the R*-tree (same tree as in Figure 2.1), the *mindist* from *q* to all entries in the same level are computed. Entry E_2 is visited first since it has the smallest *mindist*, and it is recursively traversed. Leaf entry E_7 is visited next and point *h* and *i* are candidates of nearest neighbors. The current *nn_mindist* is the distance from *i* to *q*. The process now backtracks the R*-tree, and E_8 is not visited since it has a larger *mindist* than *nn_mindist*. The next node to be visited is E_1, and in a similar fashion object g is found and g replaces *i* since it is closer to *q*. The current *nn_mindist* is adjusted to be the distance from *g* to *q*. Other nodes/entries are not visited according to *mindist* metric, and the results are point *g* and *h*. Hjaltason and Samet [9] propose an improved algorithm which traverses the R*-tree in a best-first manner using a heap.

3. System overview

This section provides an overview of the proposed network index structure, before we proceed with details in subsequent sections. We use a super-peer network model [26] as shown in Figure 3.1. In this model, among a *cluster* of nearby clients, some *super-peer*s (e.g., S_1 and S_2) act as local servers, to which other *passive-peers* (e.g., C_1 and C_2) are connected. A super-peer is computationally more powerful and has more network bandwidth, comparing with passive-peers. Thus super-peers can play the more important role of local servers of passive-peers, which submit queries to their connected super-peer and receive results. This model is realistic and versatile, since it takes into account the heterogeneity of computational capabilities and network bandwidth among peers. Yang and Garcia-Molina thoroughly studied the pros and cons of super-peer networks and devised several rules of thumbs for their design [26].

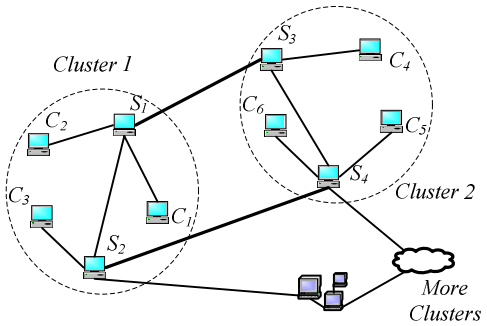


Figure 3.1: A super-peer network

We also assume that peers have reasonable computational capabilities. In particular, peers should be capable of communicating with each other, building multidimensional indexes, and processing queries on their own data. As the R*-tree is the dominant index for multi-dimensional data (implemented in *Oracle*, *IBM Informix* and *DB2*, etc), we assume that each peer indexes its own data using an R*-tree, which is built before it joins the P2P network.

According to analysis of [26], redundancy in super-peers is necessary for two reasons: (i) in case one super-peer fails

others can still serve the cluster, and (ii) super-peers share the load with each other. Therefore we have more than one super-peer in a cluster, though a passive-peer connects to only one super-peer at the same time. Clusters are formed by grouping peers holding semantically close data, where the closeness is measured by the centroid of data. By recursively dividing the data space, each cluster takes one *zone* in the semantic space. When a new peer contacts any super-peer for joining the network, it can be routed to the cluster responsible for the *zone* where the data centroid of the peer falls into. Inter-cluster routing mechanism is similar to that of CAN (details omitted to save space; note that we do no use any hashing). Inside each cluster, super-peers index data shared by all peers using a special R*-tree, Network-R-tree (abbreviated as NR-tree). The NR-tree is an R*-tree structure, but the actual data indexed still reside in all peers in a cluster (except data shared by the super-peer). As such, the NR-tree possesses the query processing power of an R*-tree but is much more space-efficient. When building this index, in order to save space and processing power for super-peers, not all data in peers are inserted into NR-trees and migrated to super-peers. Instead, only a small number of MBRs summarizing peer data are inserted (illustrated in section 4). The NR-tree provides some knowledge of data in the cluster to a super-peer. Upon receiving a query (range query or *k*-nearest neighbor query), a super-peer forwards the query to the corresponding cluster. For example, a range query is forwarded to the cluster responsible for the *zone* that the centroid of query range falls into. The corresponding cluster processes the query by checking local data and expanding to neighboring clusters. Those clusters will process the query and return results to the original peer.

The above discussion serves as a high-level description, omitting, however, several key issues: (1) cluster formation and maintenance, (2) exact index structure within each cluster, and (3) query processing techniques. In subsequent sections, we will give detailed treatment for these issues.

4. Cluster formation and maintenance

A *cluster* is a group of peers formed by super-peers and their respective passive-peers, where super-peers act as local servers. Forming clusters that host geographically close data is the foundation of our system.

Without loss of generality, for any multi-dimensional data, we can assume that there is a *universe* range in each dimension (e.g., [0, 10k]). We can always properly scale it if there are data out of range in any dimension. By considering the centroid of its data, we can position each peer uniquely in the *universe*. In this sequel, the *position* of a peer means the centroid of its shared data, and we use the centroid to stand for the peer in the *universe*.

As the system starts up, there is only one cluster. When the total number of peers exceeds the capacity of a cluster, the universe is split to two equal *zones* and each contains a new cluster. This split process is same as that of a CAN. In each zone new super-peers are elected according to their computational power. To illustrate, consider the example in figure 4.1, where we assume that cluster capacity is four and it is in 2D. Initially there are only four peers and P_1 is a super-peer (part *a*; for simplicity, we only show one super-peer in each cluster). When P_5 joins, the space is divided into two zones, and in the right zone P_4 becomes a super-peer (part *b*). When new peers join, they are assigned to the zone in which their centroid of data falls. A zone may be further split when more peers join (part *c*).

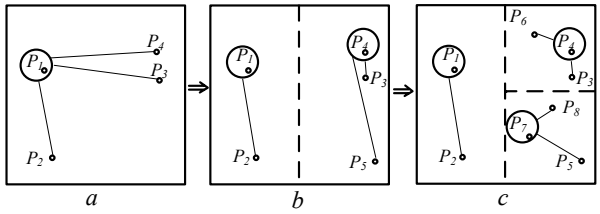


Figure 4.1: Space division

The cluster formation mechanism forces peers close in data universe to fall in the same or close clusters, and it significantly improves the locality of data in the same cluster. With the same routing mechanism as CAN, for a *d*-dimensional space partitioned into *n* equal zones (clusters), the average routing path is of length $(d/4)(n^{1/d})$ and individual cluster maintains $2d$ inter-cluster neighbors [13].

5. Index construction and maintenance

Inside each cluster, super-peers organize the indexing and query processing with cooperation of passive-peers. In order not to overload super-peers and balance the workload, part of the indexing and query processing task is pushed to passive-peers. In this section, we will discuss building an NR-tree index *inside* a cluster (section 5.1) and maintenance of index structure when peers join/depart (section 5.2).

5.1 Building NR-tree

In addition to an R*-tree *precisely* indexing its own data, each super-peer maintains an NR-tree, *coarsely* indexing peer data as well as its own data. The R*-tree of a super-peer plays the same role as those in passive-peers. NR-tree, however, takes a different role by indexing special MBRs summarizing data in each passive peer and super-peers. These special MBRs are formed by further grouping

existing MBRs in R*-trees of individual peers. We will illustrate the process of tree construction using a running example. Consider the example in Figure 5.1 (for comparison purpose, we still use the same data points as in Figure 2.1). Assume three peers (S_1, P_2, and P_3) in the cluster, and S_1 is a super-peer. We assume one super-peer for convenience, but the example can easily extend to the case of two super-peers. S_1 contains points $\{a, \ldots, g\}$ and P_2 has points $\{h, i, j, k\}$. They both index their data using an R*-tree (Figure 5.2). Assume that originally only S_1 is in the network, and P_2 contacts S_1 to join. S_1 should inform P_2 the number of MBRs that can be accommodated (denoted as m). A large m, for example, 2 in this case, allows passive-peers to send more MBRs to the super-peer but introduces more space overhead. Assume m to be 1, and P_2 should regroup current MBRs in its R*-tree to form a larger MBR. As there is only one entry in the root level (E_2), it is sent to S_1. In the mean time, S_1 also needs to perform the same task with its own R*-tree. S_1 selects MBR of entry E_1. Similarly, assume there is anther MBR from peer P_3. With three MBRs, S_1 can start building its NR-tree as shown on the right of Figure 5.1 (only the shaded region). The shaded root level is the NR-tree stored at S_1, and actually data reside in all peers. Each entry in the NR-tree has an additional bit marking whether the data is locally stored or at other peers. If there are multiple super-peers in the cluster, same operations as in S_1 are carried out for all super-peers in order to achieve replication of the NR-tree.

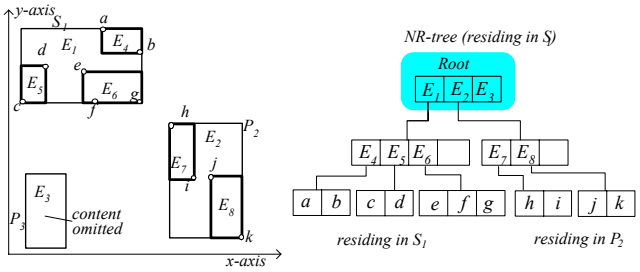


Figure 5.1: NR-tree example

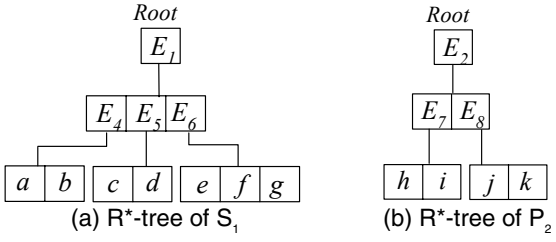


Figure 5.2: R*-tree in S_1 and P_2

In practical scenarios, the number of MBRs from each peer (m) is negotiated among super-peers inside a cluster. For general values of m, in order to form m MBRs from its data, a peer can perform a k-means clustering in all objects. We can also leverage existing R*-trees at peers to perform clustering more efficiently.

When MBRs are inserted to an NR-tree, if there is an overflow during the insertion process, the overflow handling mechanism of the R*-tree is automatically carried out. Thus we can maintain a NR-tree at the super-peer easily with minor modification of existing R*-tree algorithms. The total cost incurred when a peer joins is the same as insertion of m rectangular objects, which is very cheap for small values of m.

5.2 Maintenance of NR-tree

The maintenance of NR-tree is triggered by departure of peers. When a passive-peer leaves, we just need to mark this fact in the NR-trees of super-peers. When a super-peer departs, a new super-peer needs to be elected to take its place.

When a passive-peer departs, the NR-tree should be updated to reflect the fact that its shared data are no longer available. All MBRs from the disconnected peer should be deleted from the NR-tree. When a super-peer departs, a new peer is elected from passive-peers by existing peers based on the computational capabilities and bandwidth. The new super-peer fetches a copy of NR-tree from any existing super-peer, turn all existing *local* bits to false, delete entries from the disconnected super-peer, and mark the entries corresponding to its own data as *local*. If there is only one super-peer at the moment and it fails, the cluster is destroyed and peers rejoin the network. We expect this case to be very rare since we enforce redundancy in super-peers. Change of super-peers may also happen when a peer with superior computational power joins the cluster, which will replace the weakest present super-peer.

6. Query processing algorithms

Query processing is coordinated by super-peers, and part of the processing is pushed to passive-peers as well. When a passive-peer intends to issue a query, it sends the query to its super-peer. For example, in the case of a range query, the passive-peer sends the query rectangle to its super-peer. In the case of a *kNN* query, query point and the desired number of nearest neighbors (k) are sent. A super-peer, upon receiving a query from its passive-peer, forwards the query to the corresponding cluster. In particular, the query is forwarded to the cluster responsible for the zone where the centroid of query range (for range query) or query point (for kNN query) lies. We denote the super-peer in this cluster that receives the query as the *primary* super-peer for this query. The *primary* super-peer starts processing the query on behalf of the issuing peer. It first checks it own NR-tree for results, and it also forwards the query to its

neighbors in the *zone* with a time-to-live (TTL) denoting the length of search path. Other clusters process the query in a similar fashion, and they decrease TTL by one before they forward the query further. If TTL is zero, the query is not forwarded any more.

As peer failures are common to P2P systems, super-peers may fail during processing of queries. In order to make our system more robust, we set up a mechanism in super-peers such that if one fails, others can continue to process queries it was responsible for. In particular, when a super-peer receives a query, it randomly selects another super-peer in the same cluster (if any) as a backup and forwards the query together with the address of query-issuer. After a preset period, if the backup super-peer does not receive an acknowledgement from the initial one, it can process the query. Only if both peers fail simultaneously the query fails to be processed. This mechanism can be adjusted for networks of different dynamics by tuning the number of backup super-peers.

The distributed index structures proposed in this paper can efficiently support various types of multi-dimensional queries (e.g., range queries and *k*-nearest neighbor queries). The query processing power roots from (i) the NR-tree which indexes local data inside a cluster and (ii) the cluster formation mechanism. With NR-tree actually being an R*-tree (except that data are not locally stored), it can process any query that R*-trees are capable of processing. In this section, we will discuss two types of queries of fundamental interest, namely, range queries and *kNN* queries.

6.1 Range query processing

Range queries are collaboratively answered by super-peers and passive-peers. When a super-peer receives a range query from its passive-peer, it routes the query to the cluster responsible for the *zone* where the centroid of query range falls in. In this way, the query is routed to its *primary* super-peer, which checks its NR-tree and finds passive-peers with data intersecting the query region. These peers are then passed with the query and address of the peer that initialized the query, and they process the query with their own R*-trees and return results individually. In the mean time, the super-peer also forwards the query to its neighboring clusters since they might contain possible results. The query is forwarded together with a TTL, denoting how long it can be further forwarded. The query is processed in the same fashion in other clusters.

To illustrate the process of range query processing, consider the example in Figure 6.1 and Figure 6.2 (we mainly focus on the algorithm in the *primary* super-peer as others are similar). Figure 6.1 shows peers in the cluster S_1 (we use the name of the super-peer to denote a cluster), and Figure 6.2 shows the clusters around S_1. Assume that a passive-peer P_i in some cluster submits a range query q (shown as a shaded rectangle) and it is initially routed to super-peer S_1. Thus S_1 is the *primary* super-peer for this query. On the super-peer side, S_1 checks its NR-tree and it indicates that q intersects with MBR E_2 from passive-peer P_2, as well as a *local* MBR E_1. S_1 immediately forwards the query q as well as the address of P_i to P_2, which will continue to process the query. Note that this forwarding is done after traversal of NR-tree is finished. In this manner, a passive peer is not passed with the same query multiple times. S_1 also forwards the same information to its neighboring clusters (S_2, .., S_5), together with a TTL counter. S_2 is a *subsequent* super-peer for this query (same for other super-peers contacted). On the passive-peer side, P_2 processes q using a standard range query processing algorithm [7], retrieves qualified result (point h), and pass it to query issuing peer P_i. Another super-peer, S_2, processes q in a similar fashion as S_1, while the only difference is that S_2 will decrease TTL by 1 if it forwards q to some other clusters.

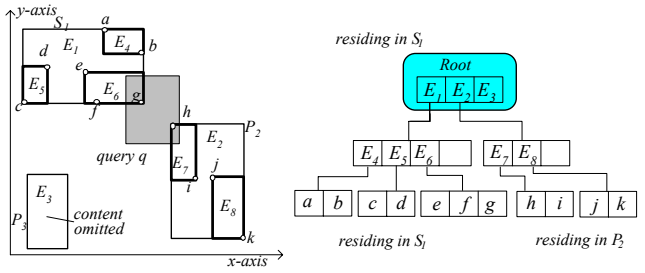


Figure 6.1: Example of a range query

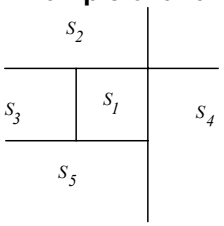


Figure 6.2: Clusters around S_1

The procedure of range query processing at the *primary* super-peer is shown in Figure 6.3, and it can be readily modified for *subsequent* super-peers by adding lines for checking TTL expiration and decrement. Large values of TTL will increase the number of results, but increase system workload as well. According to the rule of thumbs in [26], TTL should be minimized, and we will experimentally determine the best values for TLL.

Algorithm range_query
Input ***q***: range query rectangle, ***addr***: address of the peer that issues ***q***
1. check NR-tree with ***q*** and retrive all MBRs in leaf level intersecting ***q***
2. for each MBR ***mbr*** retreived in step 1
3. if ***mbr*** is local
4. check local R*-tree and return results to ***addr***

5. else //***mbr*** is non-local
6. forward ***q*** and ***addr*** to the peer containing ***mbr***
7. forward ***q*** and ***addr*** to neighboring clusters

End range_query

Figure 6.3: Range query processing at *primary* super-peer

6.2 *k*-nearest neighbor query processing

Processing of *k*-nearest neighbor is also finished by super-peers and passive-peers together. When a super-peer receives a *k*NN query from its passive-peer, it forwards the query to the cluster responsible for the *zone* where the query point falls in (thus reaching the *primary* super-peer). At the *primary* super-peer, we take depth-first traversal algorithm over its best-first counterpart since it requires less space. Note that leaf level of the NR-tree contains no real data, as data reside either in passive-peers or indexed by another R*-tree in the super-peer. Our algorithm starts from the root level of NR-tree, sorting entries by their *mindist* to query point, and recursively traverses sub-tree of entries with smallest *mindist*. When a leaf entry in *NR-tree* (which not really contains any data) is visited, the corresponding peer that contains the data is passed with the query and the current *k*-th smallest *mindist*. This enables the peer to continue processing the query itself. When a passive-peer receives a *k*NN query, it traverses its own R*-tree with the *mindist* it received as the pruning metric. Nodes/entries with larger *mindist* are not examined. The passive-peer returns at most *k* objects to the super-peer together with their distance to the query point. Results from passive-peers are then combined by the *initial* super-peer to produce the top-*k* nearest neighbor candidates. Denote the largest *mindist* value as *mindist_k*. In the mean time, the *primary* super-peer forwards the query to neighboring clusters with the current *mindist_k*. Subsequent super-peers can use *mindist_k* for pruning. Similar as range queries, a TTL is set for each *k*NN query, and final results are combined at the *initial* super-peer before sending to the query-issuing peer. The TTL timer ensures that the query will not excessively span in the network and results can be timely returned.

To illustrate the query processing procedures, consider an exemplary *k*NN query in Figure 6.4, which uses the same setting as Figure 6.1 and Figure 6.2. Suppose passive-peer P_i in some cluster issues a 2-NN query and it is routed to super-peer S_1, and the query point is q. S_1 is the *primary* super-peer for this query. S_1 checks its NR-tree and decides to visit E_2 first since it has the smallest *mindist*. However, E_2 corresponds to data residing in peer P_2. So the query and the current *mindist_k* (infinity, since there is no candidate found) are passed to P_2. P_2 processes the *k*NN query using the algorithm we described in section 2.2, and reports to S_1 point h and i together with their *mindist* to q. S_1 adjusts *mindist_k* to be the 2nd smallest distance from current candidates before deciding whether to visit E_2. Since *mindist* of E_2 is smaller than that of point i, sub-tree of E_2 has to be visited. The query is passed to the R*-tree at S_1, and point g is found to replace i as a candidate. Now S_1 can safely prune entry E_3 with the *mindist* metric. In the same time, S_1 forwards the query and the current *mindist_k* to S_2, which then performs the exact algorithm except: (i) the initial *mindist* is set to *mindist_k* for pruning and (ii) TTL will be further decreased before the query is forwarded. S_2 will return results to S_1 afterwards. Algorithm procedures are shown in Figure 6.5 (for the primary super-peer), and they can be modified for subsequent super-peers by initializing *mindist_k* to the value passed with the query.

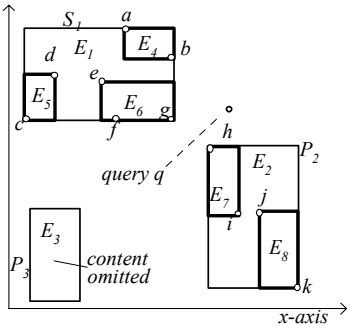


Figure 6.4: Example of a kNN query

Algorithm *k*NN_query

Input *q*: query point, ***k***: number of nearest neighbors, ***addr***: address of the query-issuing peer

Variables: ***mindist_k,*** the largest mindist among nearest neighbor candidates, initially set to infinity, $\boldsymbol{DIST_N}$, *mindist* of node ***N*** to ***q***

1. for each node ***N*** in the NR-tree
2. if the current node ***N*** is the leaf level of NR-tree
3. if $\boldsymbol{DIST_N < mindist\_k}$
pass ***q***, ***mindist_k***, and ***addr*** to the peer hosting data in ***N***
5. else
6. Sort entries in ***N*** and recursively ascending their sub-tree
7. forward ***q***, ***mindist_k***, and ***addr*** to neighboring clusters

End *k*NN_query

Figure 6.5: *k*NN query processing at *primary* super-peer

7. Experimental evaluation

In this section, we demonstrate the effectiveness of the proposed techniques with extensive experiments.

7.1 Simulation setup

Initially there is only one peer in the network, and new peers keep joining until the network reaches a certain size (N). Each peer hosts data randomly drawn from real data, and the centroid of data in each peer follows uniform or Zipf distribution. In particular, we use a real data set, CA [20], containing 1314k geographical locations (X, Y coordinates) in California, Los Angeles. The values are all normalized to range [0, 10k]. To collect statistics, we randomly (based on certain ratios) inject a mixture of operations (peer join, departure, and search) into the network. The proportion of peer join and departure is kept roughly the same to maintain a stable size of the network. On average, each peer issues 100 queries during the time it is online. Initially we separately experiment range queries and kNN queries, and then we test a mixture of queries where each type of queries takes half. For range queries, we generate square regions with side length q_L uniformly distributed in space. For kNN queries, query points are uniformly distributed in space and k varies from 1 to 5. Additional parameters are in Table 1, and their default values are underlined.

Table 1

	Descriptions	Values
N	Number of peers in the network	256, <u>1024</u>, 4096, 16k
P	Number of passive-peers in a cluster	1, 3, <u>9</u>, 27, 81
S	Number of super-peers in a cluster	1, 2, <u>3</u>, 4, 5
M	Number of MBRs from a peer	1, 5, <u>10</u>, 15, 20
C	NR-tree and R*-tree node capacity	5, <u>10</u>, 15, 20
n	Number of data objects per peer	500, 7000, <u>1000</u>, 1300, 1600
γ	Join/departure percentage	0%~50%, <u>20%</u>
q_L	Side length of range query	200~1000, <u>600</u>

7.2 Scalability and stability

In this section, we evaluate proposed system with various numbers of peers and peer data distributions. The size of network decides the total number of clusters and inter-cluster routing cost, and peer data distribution determines distribution of centroid of peer data. As clusters are formed primarily based on the centroid of peer data, it is expected that the distribution of centroids will affect cluster formation and thus search and maintenance cost. Search cost is measured as the average number of peers (passive peer or super-peer) visited over a mixture of range and kNN queries. Maintenance cost is measured by two parts: (i) the average number of inter-cluster links a super-peer has to maintain, and (ii) the average number of NR-tree node accesses incurred by peer join/departure in each super-peer. The first part is taken into account because a peer needs to be routed to the right cluster before it can join the network, and the second part is from the maintenance cost of the NR-tree. In order to take statistics, 10N node joins/departures are injected into the network for each value of N (256, 1024, 4096, and 16k), which means that on average every peer joins and departs five times. The results are shown in Figure 7.1. Part (a) shows the trend of number of visited peers per query, part (b) shows the average number of inter-cluster links, and part (c) shows the average node accesses. Two conclusions can be made from the results: (i) Search and maintenance cost grows gracefully with total number of peers, and (ii) the system performs slightly worse under Zipf distribution but the difference is insignificant. Search cost increases due to elongated path between clusters (average routing path in CAN is proportional to $G^{1/2}$, where G is the total number of clusters). The average number of inter-cluster links is relatively stable (approximately 2*d, where d is dimensionality) since each super-peer only has to maintain links to its four neighboring clusters. The average number of node access is also quite stable, since the workload is actually directly linked to the total number peers in a cluster, which is stable during experiments. The slight increase in node accesses is due to irregularity of cluster sizes. In sum, skewed distribution of centroids leads to uneven zones in clusters and elongates routing paths. However, the difference is insignificant due to the stability of CAN. For simplicity, in the following experiments we only use Zipf distribution.

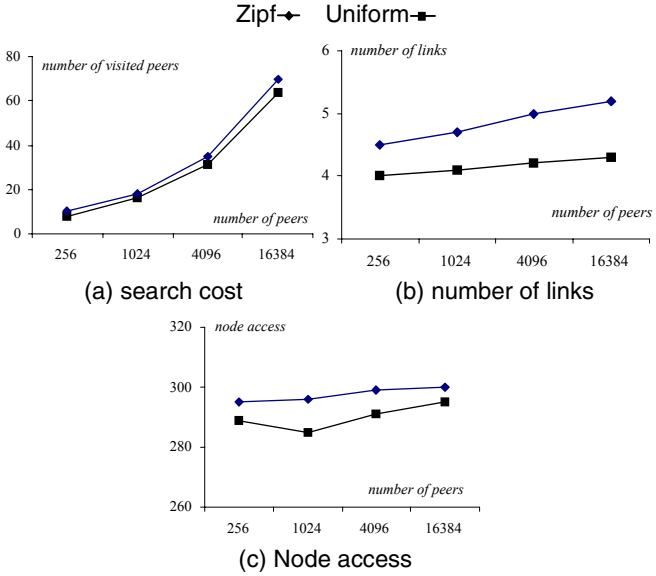
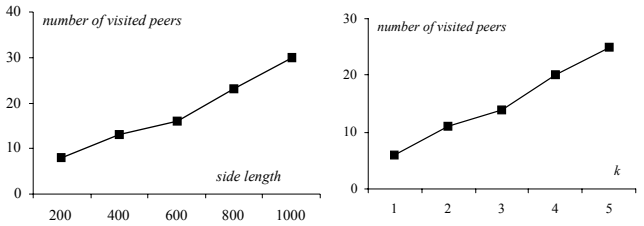


Figure 7.1: Cost vs. data distribution

7.3 Query processing cost and query size

Queries affect processing cost and time. In particular, a larger side length of range query (q_L) implies more results to retrieve and hence more nodes to visit. Similar implication applies to value of k of kNN queries. Figure 7.2a shows that the number of visited peers grows as the

size of range query increases, similar trend exhibits in figure 7.2b for *k*NN query.

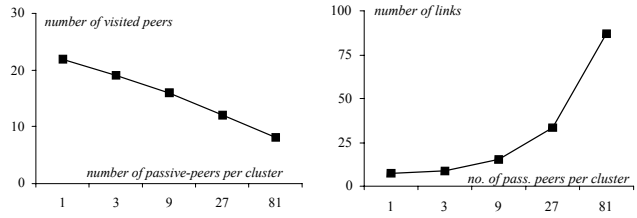


(a) range query (b) *k*NN query

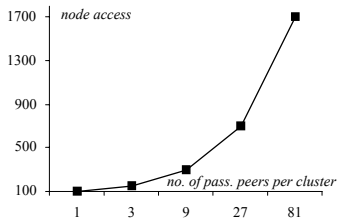
Figure 7.2: Cost vs. query size

7.4 Cluster size and super-peer redundancy

Cluster size is important since it is related to number of clusters and workload of super-peers. A larger number of peers in a cluster mean smaller number of clusters, and thus cheaper inter-cluster routing cost. However, this also implies that super-peers have to handle more operations (peer join/departure, query, and routing). These operations, if shared by more super-peers in a cluster, will be less a burden to individuals. In this section, we measure the maintenance cost by two parts, node access incurred by peer joins/departures and the *total* number of links maintained by a super-peer. We fix the network size to be 1024, and injected 100k node joins/departures interleaved with queries into the network as we take statistics.



(a) search cost (b) number of links



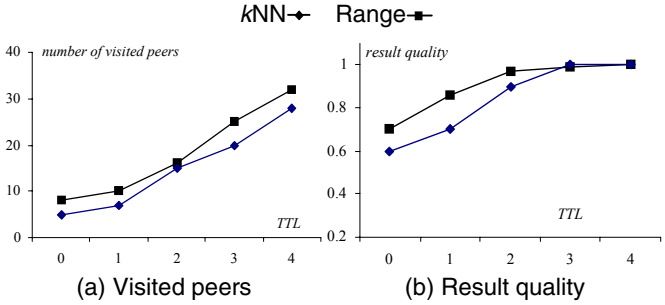
(c) Node access

Figure 7.3: Cost vs. cluster size

Results are shown in Figure 7.3, where in each cluster there are three super-peers. Part (a) shows that search becomes more efficient as clusters grow larger, which is because of shortened inter-cluster routing path and larger NR-trees indexing more data. The negative side of large cluster is reflected in part (b) and (c), where each super-peer has to maintain more links and perform more maintenance operations. As for multiple super-peers, they can share the work load of query processing evenly, but they will slightly increase the amount of maintenance work (as this is quite easy to conceive, the results are not presented). As a conclusion, as long as super-peers are not overloaded, it is desirable to have more peers in a cluster; if they are overloaded, it is good to have more super-peers to share the load.

7.5 Result quality and cost

Our system is capable of providing complete answers as well as approximate answers to a query with reduced cost. This is achieved by adjusting the value of TTL for a query, since TTL controls the length of search path in terms of clusters. This section demonstrates the relationship between the quality of results and search costs, where quality of a query is defined as: $Q = |S_{real} \cap S_{approx.}| / |S_{real}|$ (percentage of correctly retrieved results). Note that in this set of experiments, we use 10 for *k* in order to measure the quality of *k*NN results. Figure 7.4a shows that the number of visited nodes increases with the value of TTL. This is reasonable since the more cluster to visit, the more likely to find peers containing related data. Figure 7.4b shows the result quality increases with TTL, and both range queries can *k*NN queries can be answered with high quality answers with rather small values of TTL.



(a) Visited peers (b) Result quality

Figure 7.4: Cost vs. cluster size

7.6 Tolerance of node failures

Peer failures in P2P networks happen frequently. Our system has a backup mechanism for queries (section 6) and thus it is very robust under super-peer failures. If passive-peer fails and it contains the only copy of data required by a query, results of the query will be affected. In this set of experiments, we vary the number of backup super-peers for each query, and observe the percentage of queries that fail (assuming 5 super-peers in each cluster). With different peer failure rates (20% and 40%), we measure the probability that a query fails (dropped due to failed super-peers). As we can observe from Figure 7.5, the percentage of query failure is extremely low with 2 backup peers even

under 40% of super-peer failure rate. This verifies the effectiveness of proposed backup mechanism in case of peer failures.

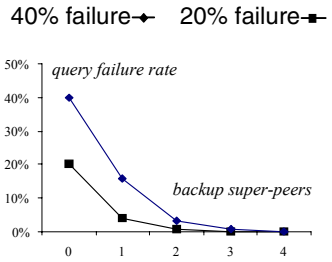


Figure 7.5: Cost vs. cluster size

8. Conclusion and future work

Current P2P systems can not efficiently support complex queries over multi-dimensional data, which seriously limits their practical value. In this paper, we solve this problem in the framework of super-peer networks. We first propose a framework for sharing, indexing, and query processing multi-dimensional data where peers with stronger computational power serve as local servers. We then propose the NR-tree, a P2P adaptation of R*-tree, for indexing and querying multi-dimensional data in a P2P framework. Our system is capable of answering various types of queries with complete or approximate answers, and its efficiency, stability, versatility, and scalability are demonstrated with rigorous experiments. For future work, we plan to devise cost models in order to facilitate optimization in the system.

Acknowledgement

Wang-Chien Lee was supported in part by US National Science Foundation grant IIS-0328881. This work was also supported by a grant from the Research Grant Council, Hong Kong SAR, China (HKUST 6179/03E).

References

[1] Aspnes, J., Shah, G. Skip Graphs. *SODA*, 2003.

[2] Bharambe, A. R., Agrawal, M. Mercury: Supporting Scalable Multi-Attribute Range Queries. *SIGCOMM*, 2004.

[3] Beckmann, N., Kriegel, H. P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[4] Chueng, K., Fu., W. Enhanced Nearest Neighbour Search on the R-tree. *SIGMOD Record*, 27(3), 1998.

[5] Demirbas., M., Ferhatosmanoglu, H. Peer-to-Peer Spatial Queries in Sensor Networks. *P2P*, 2003.

[6] Gaede, V., Günther, O. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[7] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD*, 1984.

[8] Harvey, N., et al. SkipNet: A Scalable Overlay Network with Practical Locality Properties. *USITS*, 2003.

[9] Samet, H., Hjaltason, G. Distance Browsing in Spatial Databases. *ACM TODS*, 1999.

[10] Johnson, T., Krishna, P. Lazy Updates for Distributed Search Structures. *SIGMOD*, 1993.

[11] Li, M., Lee. W., Sivasubramaniam, A. Semantic Small World: An Overlay Network for Peer-to-Peer Search. *ICNP*, 2004.

[12] Mondal, A., Yilifu, Kitsuregawa, M. P2PR-tree: An R-tree-based Spatial Index for Peer-to-Peer Environments. *EDBT Workshop*, 2004.

[13] Ratnasamy, S., et al. A Scalable Content-Addressable Network. *SIGCOMM*, 2001.

[14] Rowstron, A., Druschel, P. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-peer Systems. *MIDDLE-WARE*, 2001.

[15] Ratnasamy, S., Handley, M., Karp, R., Schenker, S. Application-level Multicast Using Content-Addressable Networks. *NGC*, 2001.

[16] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. *SIGMOD*, 1995.

[17] Stoica, I., et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *SIGCOMM*, 2001.

[18] Saroiu, S., Gummadi, P., Gribble, S. A Measurement Study of Peer-to-Peer File Sharing Systems. 2002.

[19] Tanin, E., Harwood, A. A Distributed Quadtree Index for Peer-to-Peer Settings. *ICDE*, 2005.

[20] http://www.census.gov/geo/www/tiger/

[21] Tang, C., Xu, Z., Dwarkadas, S. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. *SIGCOMM*, 2003.

[22] BitTorrent. http://bittorrent.com

[23] Gnutella. http://www.gnutella.com

[24] KaZaa. http://www.kazaa.com

[25] Andrzejak, A., Xu, Z. Scalable, Efficient Range Queries for Grid Information Services. *P2P*, 2002.

[26] Yang, B., Garcia-Molina, H. Designing a Super-Peer Network. *ICDE*, 2003.

[27] Zhao, B. Y., et al. Tapestry: A Resilient Global-Scale Overlay for Service Deploymenta Super-Peer Network. *IEEE JSAC*, 2004.

[28] Zhang, C., Krishnamurthy, A., Wang, R. SkipIndex: Towards a Scalable Peer-to-Peer Index Service for High Dimensional Data. *Princeton Univ. Tech. Report*, 2004.