Indexing Distributed Complex Data for Complex Queries[*]

Egemen Tanin
Department of Computer Science and Software Engineering
University of Melbourne
egemen@cs.mu.oz.au

Aaron Harwood
Department of Computer Science and Software Engineering
University of Melbourne
aharwood@cs.mu.oz.au

Hanan Samet
Department of Computer Science
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland at College Park
hjs@cs.umd.edu

Abstract

Peer-to-peer networks are becoming a common form of online data exchange. Querying data, mostly files, using keywords on peer-to-peer networks is well-known. But users cannot perform many types of queries on complex data and on many of the attributes of the data on such networks other than mostly exact-match queries. We introduce a distributed hashing-based index for enabling more powerful accesses on complex data over peer-to-peer networks that we expect to be commonly deployed for digital government applications. Preliminary experiments show that our index scales well and we believe that it can be extended to obtain similar indices for many other data types for performing various complex queries, such as range queries.

1 Introduction

Data and interactions with data are becoming less dependent on centralized systems. For example, peer-to-peer (P2P) networks are becoming a common form of online data exchange. We believe that such applications, without a central service provider, will continue to form an attractive scalable medium of data exchange. Within the context of the Digital Government, databases can use many ideas from this domain for scalable exchange of data. For example, distributed database systems that are loosely connected to each other that host and manage large data sets, Geographic Information Systems (GIS) that can work over the Internet to connect to multiple hosts and visualize/manipulate complex data, and other similar applications can use the ideas from the P2P world to create efficient data exchange environments.

The bottleneck for current P2P applications is that the keyword-based searches over P2P networks do not provide the necessary functionality to perform many types of queries. First, one cannot search within the data (i.e., a file). Second, one cannot use many of the attributes of the data for the queries (i.e., except the file name). For a P2P application to yield the desired functionality, which is readily available on many centralized systems, it has to have the capability to facilitate queries, for example in a GIS, such as selecting all the available data from a given region on a US Census map. Various versions of such complex queries

[*]This work was supported in part by the US National Science Foundation under Grants EIA-99-01636, EIA-99-00268, IIS-00-86162, and EIA-00-91474.

can be generated without much effort and in combination with other attributes of the data. In particular, we focus on the spatial attributes throughout this paper but we believe our work can be applied to other data types as well.

For P2P networks, recently developed hashing-based distributed indices address the base problem of querying complete distributed data without a central authority [12, 15]. For example, one can provide the file name of a music file that is to be downloaded over a P2P network. Accessing such information is facilitated by the use of an index. These indices, although being quite scalable, do not support more complex queries such as rectangle intersections and, more generally, range queries, which are fundamental to many data management systems. Their mechanisms heavily rely on creating globally known mappings between the node addresses of a network and the data file names that are available in this network. To avoid all-to-all communications, the mapping functions should be known by all the peers of the network. In this paper, we introduce and test a distributed hashing-based index that facilitates responding to complex queries, specifically on spatial data, over P2P networks that is based on our recent proposal using quadtrees ([8]). Our first experiments showed that our index can work well under many circumstances.

The rest of this paper is organized as follows. Section 2 reviews the current state of the art in distributed hashing. Section 3 discusses P2P systems and recent related work. Section 4 introduces our distributed hashing-based index and related algorithms. Section 5 presents initial experimental findings using our index, while Section 6 contains concluding remarks.

2 Distributed Hash Tables

Hashing is increasingly being used for mapping and accessing distributed data over large networks. In this section we first explain consistent hash algorithms that are of particular importance for networked data access in general terms, and then show how these algorithms are important for distributed systems.

2.1 Consistent Hashing

A hash algorithm uses a hash function, H, that maps keys to locations (also known as buckets), $H : \mathcal{K} \rightarrow \mathcal{L}$, where $\mathcal{K}$ is the set of all keys and $\mathcal{L}$ is the set of all locations. A key is usually a unique identifier that represents the data object to be stored. For example, a file name is a key that represents a file and its contents. A hashing algorithm usually locates an object in constant time rather than, for example, $O(\log k)$ time in a binary search, where k is the number of keys in the hash table.

Current hash functions can typically take any arbitrarily long string as a key and provide a many-to-one mapping to the locations as depicted in Fig. 1. The figure shows L locations depicted as buckets

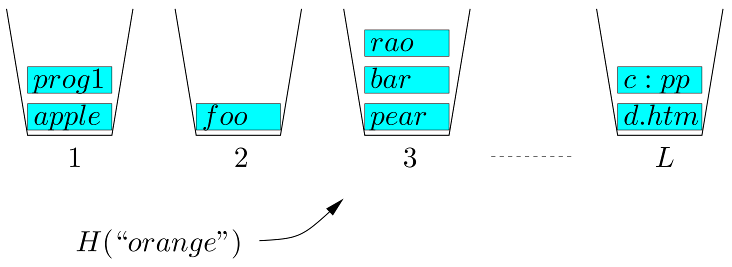
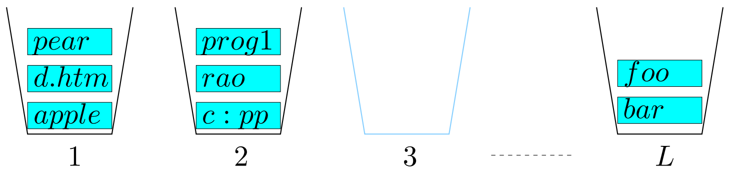


Figure 1: A basic hash function.

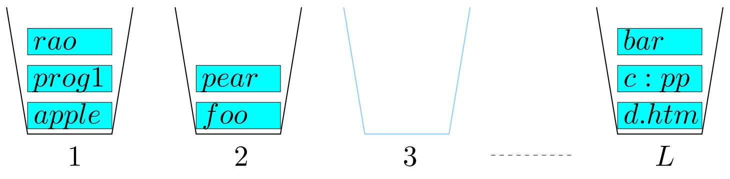
with data objects as shaded rectangles. The key for each object is a text string. When two or more objects hash to the same location, a secondary search operation is applied using, for example, an additional hash function, sequential search, binary search, etc. One possibility is to use the widely known SHA-1 (http://csrc.nist.gov/CryptoToolkit/tkhash.html) hashing function, which can hash arbitrary strings onto

$L = 2^{160}$ different locations. Using $L = 2^{160}$ locations means that it is highly unlikely for any two objects to hash to the same location (i.e., a collision) and thus secondary search is usually not necessary. It is important to note that a good hash algorithm will usually provide a distribution of keys over buckets that is close to uniformly random. As a consequence, similar keys do not hash to similar locations. For example the location $H(\text{“}apple\text{”})$ is independent of the location $H(\text{“}apples\text{”})$.

An important class of hash functions is one known as *consistent*. A consistent hash function minimizes the degree of remapping required when the set of locations changes. Basically, the set of locations will change when either: (i) an existing location is removed, or (ii) a new location becomes available. The concepts are explained using the illustrations in Fig. 2. In Fig. 2(a), an *inconsistent* hash function requires the



(a) An inconsistent hash function after remapping caused by the removal of location 3.



(b) A consistent hash function after remapping caused by the removal of location 3.

Figure 2: Examples of hashing (in)consistency.

objects from location 3 and all the other objects to be rehashed due to location 3 being removed. Comparing Fig. 2(a) and Fig. 1, we note that with inconsistent hashing, the change has relocated many of the existing data objects besides those in location 3. In Fig. 2(b) the removal of location 3 only leads to the relocation of the objects at location 3 – that is, consistent hashing reduces the effect of a change. Similarly when a new location is made available, only a small number (proportional to the ratio of the number of items to the number of locations) of objects need relocating. It is known that some hashing methods provide a good degree of consistency. This class of hashing methods is an important building block of efficient distributed systems that utilize distributed data and processing.

2.2 Distributed Hashing

A distributed hash algorithm uses a number of servers to maintain some number of locations. A server represents a computer on the Internet, e.g., a `http` server. In the simplest sense, if there are M servers and L locations, then each server should maintain roughly L/M locations. Although there are many methods for a distributed system to implement a distributed hash algorithm [11], we explain a method that has recently become widely known as the *Chord* method [15]. This method is particularly suitable for P2P applications due to its ability to handle high levels of dynamicity in a network. Our work builds upon the Chord method.

The Chord method maps both keys and servers to virtual locations, as depicted more generally in Fig. 3. The figure shows a hash algorithm using 2^t different locations, from 0 to $2^t - 1$. Each server has a unique

Internet Protocol (IP) address (and port number, which is not shown) that is used as a key. Each server is said to be the *owner* of the interval between it and the immediately preceding server (e.g., the interval owned by server 10.28.1.5 in Figure Fig. 3 corresponds to the entire region between server 128.56.32.5 and 10.28.1.5).

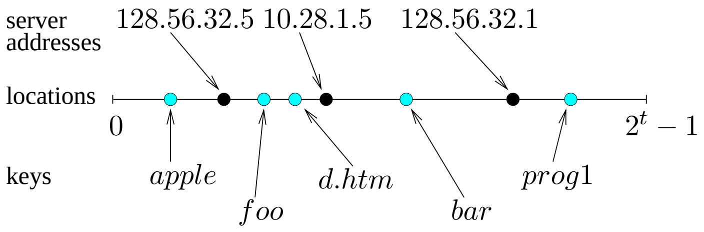


Figure 3: Hashing server addresses and object keys to the same name space.

The Chord method derives its name from the use of modulo arithmetic and the clarity of representation achieved by depicting the hash locations in a circle. Fig. 4 depicts the locations using a circle and also shows other details of the Chord method.

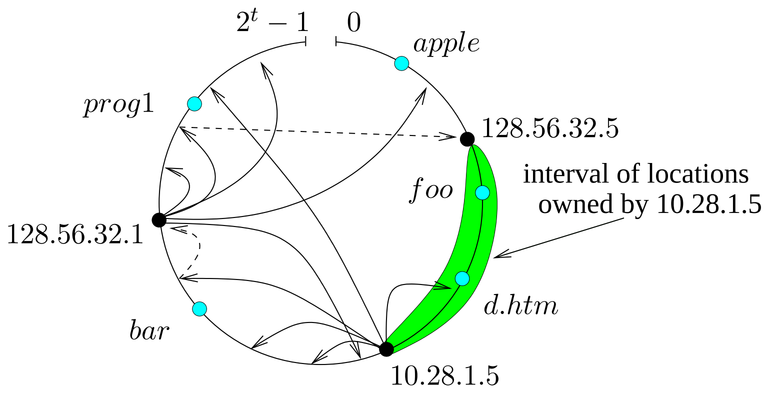


Figure 4: Details of the Chord method.

Each server in the Chord maintains a table of up to t other servers, where t is logarithmically proportional to the number of locations. Each entry in the table represents an interval that is larger than the interval of the previous entry (usually twice as large). In the example in Fig. 4, the entries in the table for two servers are shown as arrows from the servers. Each interval is associated with a successor server (the owner of the corresponding entry). These are depicted using directed broken lines although only a subset relevant to this example are actually displayed. The shaded area in the figure shows the name space for which server 10.28.1.5 is responsible.

Without giving an explicit algorithm, consider the case when server 10.28.1.5 is trying to locate the object with key *prog1*. Hence, we are looking for the server address who has *prog1* so that we can contact that server and get *prog1* itself. We will now use our table of servers to trace the location of *prog1*. The querying server 10.28.1.5 checks to see if he owns *prog1* and if not it scans through its table and finds the name space interval which the key *prog1* maps to. In this case *prog1* is in the interval defined by the third and fourth arrows, counting clockwise from 10.28.1.5, mapping to our third and fourth entries in the server table. We contact the server that is the successor server for the third arrow (to simplify the figure, only a subset of the broken arrows are shown). Note that using the fourth rather than the third arrow may miss a server that may exist between *prog1* and the fourth arrow. The request for *prog1* is now forwarded to server 128.56.32.1, getting us closer to the destination. Now, 128.56.32.1 checks whether it has *prog1* and if not (which is the case here as *prog1* is between the second and third interval defined by the arrows from server

$128.56.32.1$) repeats the process and hence forwards the request to server $128.56.32.5$ as the broken arrow from the end of the second interval is directed to it. This is the server that knows who actually has the *prog1* itself. Here we assume that objects are not relocated with keys when they were first hashed. In general, it can be proven that a request to locate an object will be forwarded $O(\log n)$ times with a high probability, where n is the total number of servers/peers in such an application of the Chord method.

For a dynamic system where many servers join and leave the environment, such as P2P systems, we need a consistent method to prevent remapping of large amounts of data. The Chord method is consistent. Any change in the system causes only a local change in the name space ownership. As depicted in Fig. 5, the addition (or removal) of a server requires relocating only the keys on one server. For example, in this figure, the addition of the new server will require the new server to start taking care of some of the keys that were owned by the old server $10.28.1.5$. In the case of a removal, e.g., assuming that our new server leaves the network after some time, then the departing server can give the keys back to the original (i.e., $10.28.1.5$) server without disturbing the rest of the system. In general, servers can be added and removed

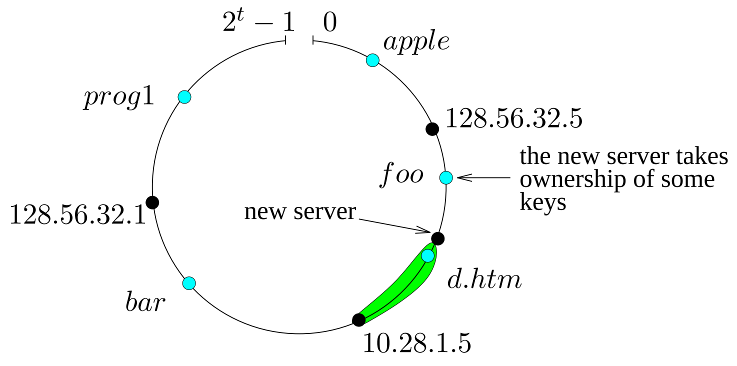


Figure 5: Inclusion of a new server.

from the Chord arbitrarily since the overhead is logarithmically proportional to the number of servers. Note that there are many more aspects to the Chord algorithm, which space does not permit us to explain, such as caching requests and objects for object locality and virtual servers for load balancing improvements [15].

3 Peer-to-Peer Networks

Historically, Napster [2] and Gnutella [1] were the first and most well-known of the P2P systems. Napster was a P2P system where the directory service is centralized on servers and users exchanged music files that they had on their disks. Hence, it is not commonly cited as a pure P2P system. Gnutella was a pure P2P file exchange system. Unfortunately, it suffered from scalability issues, i.e., floods of messages between peers to map the information in the system were required. It is commonly cited as an unstructured P2P system. Other systems followed these, each addressing a different flavor of sharing over the Internet. Multiple P2P storage systems have recently emerged. PAST [6], Freenet [5], and OceanStore [10] are some popular P2P storage systems. Some of these systems, like Freenet, have focused on anonymity while others, like PAST, have focused on persistence of storage. Also, other approaches, like SETI@Home [3] and distributed shells [16], made other resources, such as idle CPUs, work together over the Internet to solve large scale computational problems. Along with these systems, researchers have developed underlying routing and indexing utilities to address the core problems of these systems. These are designed for optimally finding and accessing an object in a decentralized dynamic distributed network over a wide-area. They are commonly described as structured P2P systems/utilities. Examples of some lower level utilities are CAN [12], Pastry [13], Chord [15], and Tapestry [17]. Multiple P2P systems are built on top of the lower level utilities. For example, PAST is built on the Pastry and OceanStore is built on the Tapestry routing and indexing services. All of these routing and indexing utilities are designed to help the nodes of the systems find complete objects. Our work differs from these systems by supplying a method to query the data, i.e., within an object, and using other attributes of the data rather than just its name. Very recently, researchers started to investigate methods to enable complex queries over P2P networks. Some of these database-oriented approaches [4, 7, 9] are larger scale frameworks that include performing general queries. Currently, many support only equality-based operations and our approach can easily fit into most of these systems to provide complex queries. However, none of the approaches consider the case of very complex data, i.e., spatial attributes and complex queries on this data (rectangle intersections), which is the focus of our work.

4 Indexing Distributed Complex Data for Complex Queries

4.1 Our Approach

In this paper we focus on complex data such as spatial data. Spatial data differs from conventional point data in that the objects also have extent — that is, often more than one location is associated with each object. For example, cities are spatial objects whose extent is defined by the space that they occupy. There are many ways to describe such objects ranging from their boundary, the locations that make up their interior, their minimum axis-aligned bounding box, etc. Thus we see that the spatial description is more than just a longitude and a latitude value, which is what is usually used if we are representing cities in a large scale map. A query can also be defined as a spatial object Q, and the result of the query is all the objects in the database that intersect with Q.

Unlike documents that are accessed by name (i.e., where the query is simply a file name), spatial objects and queries require intersection computations – the query is actually a search for objects that satisfy it. In $\mathbb{R}^2$ space, a query can be efficiently executed by first recursively subdividing the space into four congruent regions, i.e., finding the result using a quadtree. The objects are indexed into the quadtree using some predefined subdivision rule. Each query region can be mapped to a quadtree in a similar manner. This has the effect of exponentially reducing the average number of intersection calculations per query. For example, each descent to a deeper level of the tree reduces the number of possible resulting objects by a factor of four.

A distributed system requires assigning responsibility for regions of space to the peers in the system. If a peer is responsible for a region of space, then it is responsible for query computations that intersect that region of space. While it is possible to subdivide the underlying space into a regular grid (i.e., equal-sized cells), this trivial subdivision does not maintain the scalability of a quadtree, which subdivides the underlying space recursively into four quadrants.

In our implementation we make use of an MX-CIF quadtree (e.g., [14]). In the MX-CIF quadtree, each object is associated with the smallest quadtree block that contains the object in its entirety. Subdivision ceases whenever a quadtree block contains no objects. For each subdivision there is a center point, termed a *control point*, that the two subdivision lines of the underlying space intersect. We hash the control points so that responsibility for a quadtree-block is associated with a peer in the P2P system. For example, $H(\text{“}(5,2)\text{”})$ is the location of the point $(5,2)$ on the Chord. We allow the control points to be dynamically determined using a globally known function to recursively subdivide space (or to expand the space when the domain is not large enough to contain the data).

Fig. 6 depicts some control points and example hashings. Objects are inserted into the distributed hash table by mapping them into blocks and finding the control points to which they map, and hashing onto the Chord by using the coordinates of the control point as the key. A copy of the object can be stored at the control point to which it maps. Alternatively, a pointer to the original peer containing the object can be stored at the control point. Any peer in the P2P network can have a small database of objects, forming the larger system.

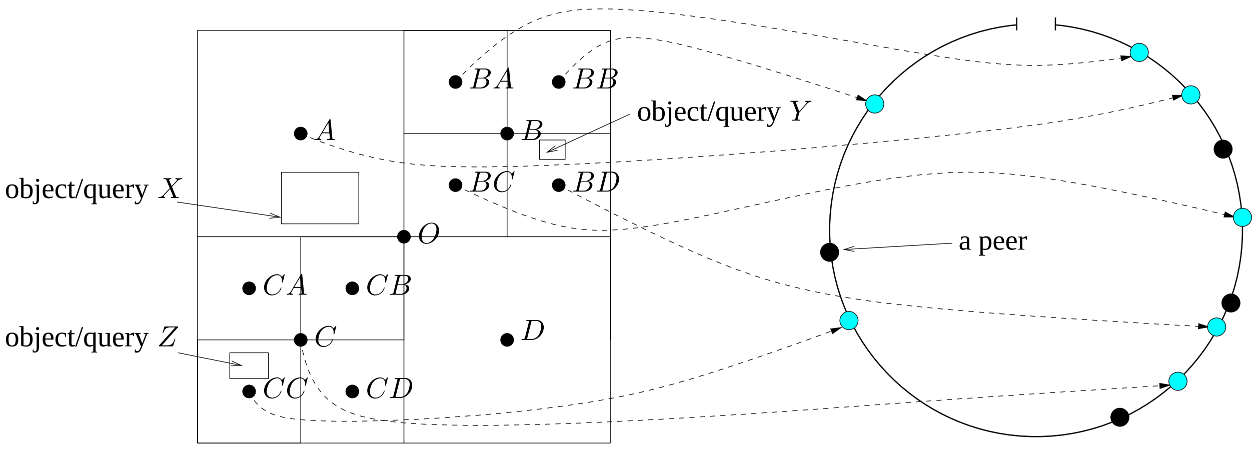


Figure 6: Spatial objects/queries, X, Y, Z, 3 levels of control points, and example hashings to the Chord, i.e., the coordinate values of a control point are used as the key and hashed onto the Chord. Dark dots are the peers that are currently in the system. Light dots are the control points hashed onto the Chord.

4.2 Spatial Algorithms

Given a query we have to visit the peers that maintain the control points which can contain intersecting objects with the query. We use flags to accommodate the traversal of our distributed quadtree. For example, we use *downward* flags to direct the query from the root level to the control points with which objects are associated. When inserting an object at a level i control point, a downward flag is marked at the parent level $j = i - 1$ control point to indicate the presence of an object at level i. The flags are propagated up through the quadtree until reaching a control point for which the downward flag already exists. The result is similar to a pyramid thereby enabling access from any level of the tree. Clearly, for a large number of objects, downward flags will most likely exist for the origin control point O as well as for the first few levels of control points. Note that we can cache the peer addresses that form a parent/child relationship to reduce the need for the full Chord traversal at every query. We can resort to the full Chord method when there is a cache miss, i.e., a peer dies in this relationship. So when we start our query we can look for the list of objects that fall into each quadrant block by finding which peer the control point of that block maps to using the Chord method and later follow the downward flags deeper into the tree for other levels. The query can start at any peer because using the Chord method it is always possible to find the root control point.

The existence of the root implies that there can be a bottleneck in query processing in the sense that each query has to start from the root. We overcome this bottleneck by adding two concepts. We use an *upward* flag to indicate that some objects are hashed at a level closer to the root level than the current level. For this case, an object insertion in a block with control point c can be followed by propagating upward flags to all descendant control points. In essence, these are control points for which objects have already been hashed. With the aid of the upward flags, a query can now proceed from any level of the MX-CIF quadtree to any other level. We can stipulate a desired *level of parallelism* and start the execution of queries at a level other than the level of the root. However, as described, the search method generally requires every query to propagate to the highest level for which control points have been hashed and in those blocks where objects are known to exist. While some cases lead to the query propagating down to the leaves of the quadtree, most cases will lead to queries propagating up to the root. To avoid the bottleneck caused by queries that propagate to the root, we stipulate that no object is hashed to control points at a level less than some globally constant value called the *fundamental level*. For example, we could stipulate that control points, O, A, B, C, and D in Fig. 6 may be forbidden to be used, which is using a fundamental level equal to 2. This means that no objects or query-objects map to the root of the MX-CIF quadtree and hence to just one peer in the network. Query parallelism is desirable and in general a high level of query parallelism is good. As we now can see, multiple intersection computations can occur when an object intersects a query-object in more than one occasion as a side effect of forcing objects into the lower levels of an MX-CIF quadtree than they are supposed to be. Eliminating this intersection multiplicity is a direction for further investigation.

Letting u be a control point, we let the structure that holds information about that control point be:

$$D(u) = (upward, downward, list)$$

if control point u exists (i.e., it has already been hashed). $D(u)$ is empty if control point u does not exist yet. The default values for a $D(u)$ are $(0, 0, empty)$, which are used when an algorithm implicitly creates a new point u. Here $upward$ and $downward$ are flags taking on values in $\{0, 1\}$, and $list$ is a list of objects that intersect the block of $B(u) = (x_1, y_1, x_2, y_2)$ defined by $u = \big((x_2 + x_1)/2, (y_2 + y_1)/2\big)$ and are maintained by the owner peer of that control point. Each u has a parent control point, $P(u)$, and a set of four children, $\{C(u,1), C(u,2), C(u,3), C(u,4)\}$. Given a control point u, it is always possible for any peer to determine its block $B(u)$. This requires an *origin block*, $B(o) = (0, 0, 1, 1)$ (the unit square) with control point $o = (0.5, 0.5)$ and a method of sub/super-division that covers all space. Call o a (or the) 0-level control point. Let $L(u)$ be the level of control point u. For example, in Fig. 6, $L(o) = 0$ and object Y has been mapped to a block at level 2.

5 Preliminary Experiments

We have built a simulation-based evaluation environment for our work. Our goal is to see how well, in terms of average query time, our index will perform in comparison to a regular central index. We used the Network Simulator (ns-2; www.isi.edu/nsnam/ns/), which is a popular simulator for networks, in tandem with the Georgia-Tech Internetwork Topology Generator (GT-ITM; www.cc.gatech.edu/projects/gtitm/), to create a simulation environment. We ran an experiment consisting of 6 data points. Each data point is obtained after a series of 5 consecutive runs with the same input parameters. We averaged these 5 runs to obtain a data point value. For each data point obtained for our index, there is also a comparison point obtained from additional runs on a simple client-server system. For the client-server case, the queries come to a central index from various nodes in the network and the server starts by sending out the hits (i.e., the objects that satisfy the query) back to the requesting clients. Each run yields a value in seconds for the average query time obtained from all of the queries processed in that run. A given set of queries coming to a system in a random fashion as a flash crowd (i.e., a sudden stream of queries) formed the query list to be processed by the system. Hence, the average time for a query shows how fast our index or a simple client-server based one can respond to queries in such an environment and scenario.

Before each run, we created a transit-stub type network at random using GT-ITM. Basically, this was our perception of the Internet. Our networks had 2 transit domains. Each transit domain can be considered as representing a different metropolitan area network. We had 8 transit nodes in each transit domain. Also, each transit node hosts a group of 6 stub domains where each stub domain can be considered as representing a different campus/agency network. A stub node is used to represent a local area network. We had 12 stub nodes in a stub domain. We did not allow for extra transit to stub edges or stub to stub edges across domains. We had a connection probability of 0.6 between the transit nodes in a domain. The connection probability for stub nodes in each stub domain was 0.2. The bandwidth between stub nodes was 2.5Mbps and the bandwidth between each transit node was 80Mbps. The stub to transit bandwidth was 2.5Mbps. The links between transit domains had a capacity of 40Mbps to properly represent the most congested links in such networks.

Peers are randomly distributed over stub nodes. Queries are also randomly distributed over stub nodes. Each peer had one data object associated with it. We had 1000 peers and hence 1000 data objects. For

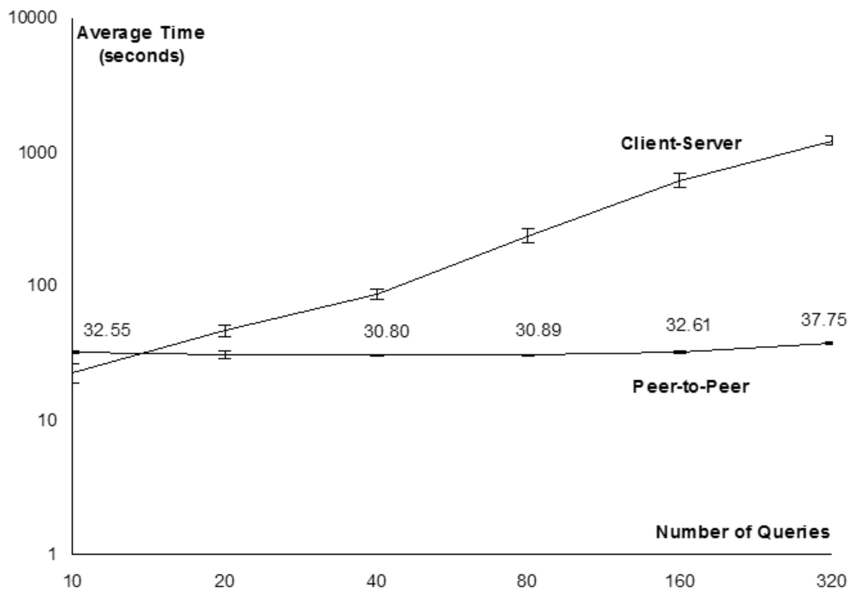


Figure 7: The comparison of our index with a simple client-server based index. The vertical lines at each data point show the standard deviation. Note that the x-axis and the y-axis are logarithmic.

the client-server case, the server was also located on a stub node. The object data size was 100Kbytes (i.e., the object definition and other data attached to it). The data was generated using US Census Bureau data on US postal codes over the Washington, D.C.–Baltimore metropolitan area where we have generated small rectangles representing houses using the population distribution over postal codes (each forming a data object). Queries were generated as rectangles from postal codes themselves. So, users are assumed to make postal code selections for their queries over a housing data set. Finally, messages between peers are simulated as 50Bytes. We used a fundamental level of 3 and the MX-CIF quadtree also had a maximum depth of 10.

We altered the number of queries that entered the system and this gave us our first results (Fig. 7). As we can see from the figure, our index can handle many more queries than a client-server based index. While the client-server system degenerates linearly, the P2P network shows no increase in average query answer time as there is a much larger overall bandwidth that is still not used in the network. The P2P system has a larger cost for a small number of queries as it is much easier to go to a single server that is not congested to obtain data. The time needed to download a large number of hits determines the performance when we have a large number of queries.

6 Concluding Remarks

For P2P networks, searching complete content in the form of files using keywords is well-known. There are many shortcomings to this approach. More effective mechanisms that allow users to access complex distributed data over the Internet are needed. In this paper, we described a hashing-based index that can be used to access complex data over a distributed dynamic network. In particular, we are interested in spatial data; however, we believe that our index can be extended to arbitrary spaces. We adopted the Chord approach for our solution to querying such complex content. We showed that our index can scale well while a client-server based index will be swamped with a few queries and downloads. We are in the process of refining our algorithms and we plan to further test our work in the near future. We believe many Digital Government applications can use the results of this research to access massively distributed data sets. For example, a large spatial data set may be generated and maintained by several local governmental entities and centers. By creating such an adaptive distributed index, we enable these entities to use a distributed data set without an extensive effort for really reallocating the pieces to a central server and then maintaining the data in that server.

References

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

[2] Napster. http://www.napster.com/.

[3] SETI@Home. http://setiathome.ssl.berkeley.edu/.

[4] F. Banaei-Kashani and C. Shahabi. Searchable querical data networks. In *Proceedings of the International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (held in conjunction with VLDB)*, Berlin, Germany, September 2003.

[5] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, Berkeley, CA, July 2000.

[6] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the IEEE HotOS VIII Workshop*, pages 65–70, Schloss Elmau, Germany, May 2001.

[7] M. Harren, J. M. Hellerstein, R. Heubsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the IPTPS'02*, pages 242–249, Cambridge, MA, March 2002.

[8] A. Harwood and E. Tanin. Hashing spatial content over peer-to-peer networks. In *Proceedings of the Australian Telecommunications, Networks, and Applications Conference-ATNAC*, Melbourne, VIC, December 2003.

[9] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 321–332, Berlin, Germany, September 2003.

[10] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent store. In *Proceedings of the ACM ASPLOS'00*, pages 190–201, Cambridge, MA, November 2000.

[11] W. Litwin and T. Risch. LH*g: A high-availability scalable distributed data structure by record grouping. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):923–927, July 2002.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM'01*, pages 161–172, San Diego, CA, August 2001.

[13] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the ACM Middleware'01*, pages 329–350, Heidelberg, Germany, November 2001.

[14] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM'01*, pages 149–160, San Diego, CA, August 2001.

[16] M. Truong and A. Harwood. Distributed shell over peer-to-peer networks. In *Proceeding of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 269–275, Las Vegas, NV, 2003.

[17] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB-CSD-01-1141, Department of Computer Science, University of California, Berkeley, April 2001.