Comparison of structured peer-to-peer networks for low-dimensional range search

Spyros Blanas

July 2006

Contents

List of Figures

Chapter 1

Introduction

During the last decade there has been increasing scientific interest in a new distributed computing model. In this model, *peer-to-peer computing*, the traditional distinction between clients and server back-ends disappears and every participant plays both roles. Peer-to-peer networks are being studied extensively and are expected to achieve greater scalability, availability and ease of deployment than existing solutions.

Early peer-to-peer systems, such as the file-sharing networks Napster and Gnutella, had a number of limitations on the indexing scheme used to locate information. Napster stored such an index on a central server, thereby greatly limiting the scalability, availability and privacy of the service. Gnutella did not employ a centralized index, but due to its simplistic design it commonly failed to find content that was actually in the system.

Soon afterward, the research community proposed distributed infrastructures that solved this indexing problem efficiently. These approaches were *Distributed Hash Tables* and provided a mapping of keys onto values on extremely large, Internet-scale systems [17, 20]. One drawback of distributed hash tables is that, due to the hashing mechanism employed to ensure load balance, only exact match queries were evaluated efficiently.

Lately there has been significant research work in an attempt to bring relational database semantics into the peer-to-peer computing model. The work in the area is far from complete and the original goal has only been achieved partly, because Distributed Hash Tables are designed to efficiently support only equality queries. Other interesting problems include efficient range queries, joins and aggregation. Recent work has extended this functionality to perform efficient range search on a single attribute or in multiple dimensions [16, 4, 5]. This work focuses on range search, because it is currently the most important open problem and is connected to another popular subproblem, the nearest-neighbor problem.

Range queries on one dimension can be evaluated on a number of existing peer-to-peer networks, like P-Grid [1] and PHT [16]. Both networks actually

store data items in a virtual overlay network, in this case a binary tree, which is partitioned and distributed across the participating computers. Queries are forwarded from the computer who initiated the query to other computers who lie 'closer' to the answer, until the computer who has the answer to the query is reached.

On more dimensions, the range query problem starts to become more complicated. An approach to hide this extra complexity is using a space-filling curve to map all multi-dimensional data onto one dimension. In this technique the data items and the queries are mapped onto one dimension prior to insertion. Then, a network supporting one-dimensional range queries is used to retrieve answers to incoming queries. Finally, the inverse function is applied and the multi-dimensional data are extracted from the space-filling curve mapping. Another approach is to store data items and queries natively in a multi-dimensional space. Such an idea is used in CAN network [17], which stores data in a multi-dimensional torus.

Another distinction in the design of peer-to-peer networks is the partitioning policy. Some networks, when a new computer joins the system, the data space of a random node is split in two equal parts. This idea was proposed based upon the assumption that input data is uniformly distributed throughout the whole data space. Recent proposals split the data load in two equal parts when a new computer joins, irrespectively of the data space associated with the two new nodes.

The number of applications proposed for peer-to-peer networks is gigantic. However, only few such systems have been successful, mainly those providing file-sharing capabilities. The number of proposals for applications on peer-to-peer networks with range query support is significantly smaller, mainly because range-query support hasn't been very popular outside the academic community.

An application requiring range query support is GHT [18], which provides a functionality similar to a hash table for geometric data. The idea focuses in sensor networks and attempts to take advantage of the topology of the network for optimal routing and query evaluation.

Another possible application would be the ability to provide location-based services on ad-hoc networks. As the wireless connectivity has become very popular, Internet access is available almost anywhere, for example using GPRS. Users might use the Internet connectivity to query other mobile users and obtain information about nearby points of interest.

Finally, range query support is an essential feature when building massive and reliable databases from small and unreliable computers. If a group of people want to put information in a database and publish it online, current practices would require all these people to meet and buy a server and a high-bandwidth connection. Under the peer-to-peer model, each participant would use his desktop computer and his home connection to maintain and make the information publicly available. Thus, new applications be-

come possible, like Internet-scale distributed databases, run only by people interested in their content.

In this work we will evaluate and compare the behavior of different structured peer-to-peer networks when storing low-dimensionality data and being queried with orthogonal range queries. The nature of our attempt —comparison— requires precision and careful assessment of the different aspects of the compared objects. The object in question —peer-to-peer networks— increases the complexity significantly.

Peer-to-peer systems grow to a very large size, often with millions of nodes. Also participants join and leave continuously and the network topology is constantly changing. Moreover, networks of that scale are much more susceptible to network failures and therefore a good peer-to-peer system must take into account the possibility of a temporary network failure and ensure that the system will maintain its good characteristics in such occasions. These properties, most of which are specific to the peer-to-peer networks, make research work in this area very challenging.

In order to compare two networks, we must specify what a 'good' network features and a 'bad' network lacks. One desired property of such networks is good load balance, that is the efficient distribution of the processing load equally to all participating peers. Another desired property is the minimal disruption of peers who are unrelated to a query, as ideally only the relevant peers should be contacted for an answer to a query. Also the ability to give back answers quickly is very important. This can be broken down to the ability to provide a few relevant answers quickly (short response time) and the ability to quickly provide all relative information (short total answer time). Finally, a very important property is the ability to locate information while the network is evolving, that is while new computers join and connected computers leave or fail, but we will not deal with this last issue in this work.

Perhaps the most difficult obstacle is the lack of a proper and realistic model for evaluating new proposals. Work in this area is still in its infancy, with few experimental observations. Researchers therefore rely on statistics and well-known distributions to prove the efficiency of their ideas but there is no widely-adopted standard. Another problem is the absence of realistic behavior patterns of the users participating in peer-to-peer systems, an essential aspect of peer-to-peer networks. Again statistical properties are frequently employed to combat the last problem.

One important aspect of this work is the introduction of some metrics to make meaningful quantitative comparisons. Our metrics provide a systematic methodology for evaluating network efficiency and focus on the performance of the desired properties of the networks, as described in the previous paragraphs. This work, however, doesn't introduce a complete theoretical model for analysis of peer-to-peer networks; it relies on simulation.

All in all, in this work we will evaluate the performance of structured

peer-to-peer networks when performing range search queries. Although comparison is difficult, we carefully created metrics to evaluate the performance of the network with respect to the desired properties. After analyzing the results, we will attempt to demonstrate good design practices which achieve better performance and are not tied to a specific network.

Chapter 2

Related work

The first efficient distributed indices appeared in the early 1990s, being the essential building block for *parallel databases*. An example of such work is the LH* distributed data structure, which was designed to support efficient data expansion to new servers in order to ensure good load balance and required no central server, for example in the form of a master directory [13]. Such data structures, however, were inadequate for peer-to-peer systems. These systems differ in two major characteristics from parallel databases: There is no distinction between clients and servers, and the network is changing rapidly as users are free to join and leave the network at any time.

2.1 Distributed hash tables

Attempts to create a fully decentralized indexing scheme in a peer-to-peer network can be traced to the original design of the Gnutella network, in March 2000. Each user of this network maintains a list of other users and in order to locate information, each query is forwarded to every known user who recursively forwards the query to new users. This scheme, however, has been shown to require exponential bandwidth to the number of connected users [19]. Therefore, a search request will sooner or later be dropped and most queries will actually reach only a very small percentage of the network. Making matters worse, the totally unstructured nature of this network makes such an approach the only solution for locating data.

This observation spurred the academic community to invent distributed indexing schemes which work well on Internet-scale environments. The main functionality such structures support is the ability to store and retrieve data items identified by unique keys, with a simple interface that supports the following operations:

- put(key, data) which inserts the data in the structure, under key.
- get(key) which retrieves the data item associated with the specific key.

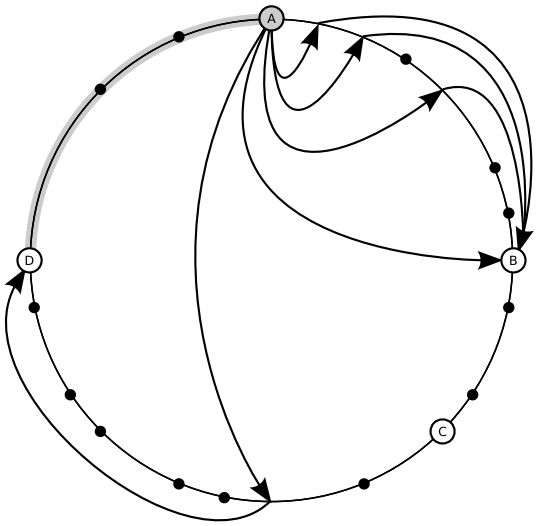


Figure 2.1: Typical Chord topology: the shaded region is the responsibility area of the shaded peer and the arrows indicate the entries in the finger table.

It can be observed that the interface follows the conventions used for retrieving and storing data in regular hash tables and that the distribution of data to the appropriate peers is completely transparent to the application. Such structures are named *Distributed Hash Tables*.

2.1.1 Chord

One distributed hash table is the Chord data structure, proposed by Stoica et al. [20]. Chord makes use of consistent hashing [12] to assign keys to the peers, a technique which is designed to let peers enter and leave the network with minimal disruption. This decentralized scheme tends to balance the system load, since each peer receives roughly the same number of keys. Chord manages to locate information with $O(\log n)$ messages.

The consistent hash function in Chord assigns each key and peer an m-bit identifier. All identifiers are ordered in an identifier cycle. Each key is assigned to the first peer whose identifier equals or follows the key in the identifier circle. Also, in order to achieve logarithmic lookup performance, each peer maintains a *finger table* which points to the peer responsible for every 2^{i-1} interval from this peer, with $1 \leq i \leq m$.

Figure 2.1 shows a Chord network with $m = 5$.

2.1.2 CAN

Another distributed hash table is the content-addressable network (CAN), proposed by Ratnasamy et al. [17]. The data space in the CAN network is

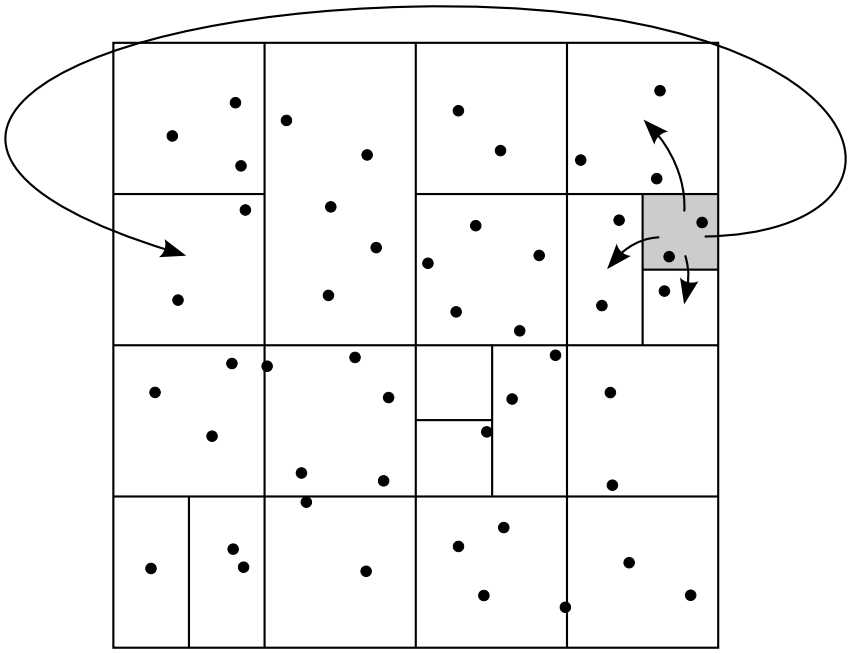


Figure 2.2: Typical CAN topology: the shaded region is the responsibility area of the peer and the arrows indicate the peers which the shaded peer knows.

a virtual k-dimensional Cartesian coordinate space on a k-torus. CAN is designed to be scalable, fault-tolerant and self-organizing and has a routing performance of $O(kN^{\frac{1}{k}})$.

Each peer is assigned a partition of the entire coordinate space. The hash function maps a key onto a point in the coordinate space. The key-value pair is stored in the peer that owns the area within this point lies. Also, for routing purposes, each peer also maintains a routing table that holds the coordinate zone and the network address of each neighboring peer. Lookup messages are forwarded by a simple greedy algorithm that attempts to minimize the distance to the destination.

Figure 2.2 shows a CAN network with 20 peers.

2.1.3 P-Grid

The P-Grid is a distributed hash table which is based upon a virtual binary search tree and was introduced by Aberer et al. [1]. In P-Grid, the data items hash to m-bit identifiers. Each peer is assigned all identifiers which begin with a given prefix, in such a way that each peer is responsible for a partition of the entire data space. For routing purposes, each peer maintains a link to a peer in the other side of the virtual binary tree, for every bit of its prefix. Lookup messages are forwarded to the peer which has the longest

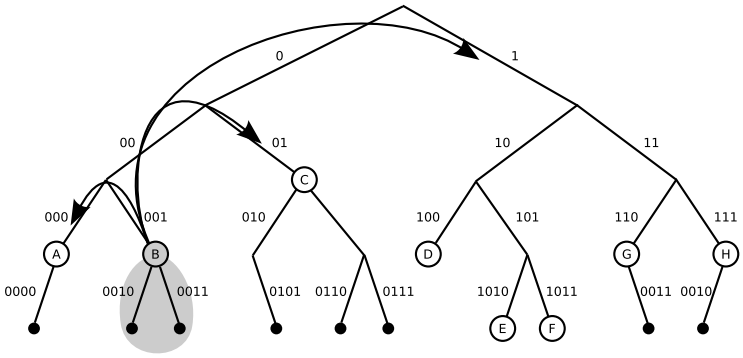


Figure 2.3: Typical P-Grid topology: the shaded region is the responsibility area of the shaded peer. The arrows indicate the subtrees in which the shaded peer knows at least one other peer.

common prefix with the destination. Figure 2.3 shows a P-Grid network with $m = 4$.

2.2 Range queries

One important type of queries in centralized databases is *range queries* (sometimes called *orthogonal* range search queries, to differentiate them from other more general types of queries) which return every data item within the range boundaries. Formally, if M is a domain on which the total order $<$ is defined and $S \subset M$ is an one-dimensional data items set, then when we *evaluate* the one-dimensional range query (q_1, q_2) with $q_1, q_2 \in M$ we retrieve $\{d \in S \mid q_1 < d \wedge d < q_2\}$. For k dimensions, the data items set is $S^k \subset M^k$, a k-dimensional range query is defined by (q_1, q_2) with $q_1, q_2 \in M^k$ and when the query is *evaluated* we retrieve $\{d \in S^k \mid \bigwedge_{i=1}^{k}\big(\pi_i(q_1) < d \wedge d < \pi_i(q_2)\big)\}$, with $\pi_i(q)$ being the projection of k-dimensional point q on dimension i.

2.2.1 Space-filling curves

Space-filling curves (also known as Peano curves) are curves whose ranges contain the entire 2-dimensional unit square, or the entire 3-dimensional unit cube, or for more dimensions the entire k-dimensional unit hypercube. This interesting property makes space-filling curves a flexible tool for converting any high-dimensional problem into an one-dimensional problem. Space-filling curves are used for low-dimensionality problems in the hope that spatial locality in the k-dimensional domain is preserved —to some extent— on the one dimension of the space-filling curve.

One of the most often used space-filling curves is the *Z-order* space filling curve. It is very popular in computer science because mapping points of

the k-dimensional space to the one-dimensional curve is easy: if the point is at the two-dimensional binary coordinates $(x_n \cdots x_2 x_1, y_n \cdots y_2 y_1)$, the one-dimensional mapping is done by interleaving the coordinate bits and this point is mapped on point $y_n x_n \cdots y_2 x_2 y_1 x_1$. Figure 2.4(a) shows the Z-order space filling curve applied over a two-dimensional unit square.

One of the biggest disadvantages of Z-order space filing curve is that the abrupt transitions it exhibits destroy spatial locality. One curve performing better in this aspect is the *Hilbert* space filling curve, shown in Figure 2.4(b). However, mapping k-dimensional points to one dimension using the Hilbert curve is more complex than performing bit interleaving as in Z-order curve.

Space-filling curves allow k-dimensional queries to be also mapped on the one-dimensional curve. However, the one-dimensional range that is produced represents a *superset* of the query space, which means that the evaluation of the one-dimensional query will return more data items than the evaluation of the k-dimensional query. Therefore, the data items domain cannot be collapsed to one dimension using this technique; space-filling curves provide merely an overlay: the k-dimensional query must be evaluated in order to obtain the proper answer set.

Despite this major drawback, however, space-filling curves present an easy way to adapt low-dimensionality problems to work with existing, widely used and thoroughly studied data structures for one-dimensional data, like B-trees. This is achieved by first locating the superset of nodes relevant to the query, using a space-filling curve, and then evaluating the k-dimensional query on these specific nodes only.

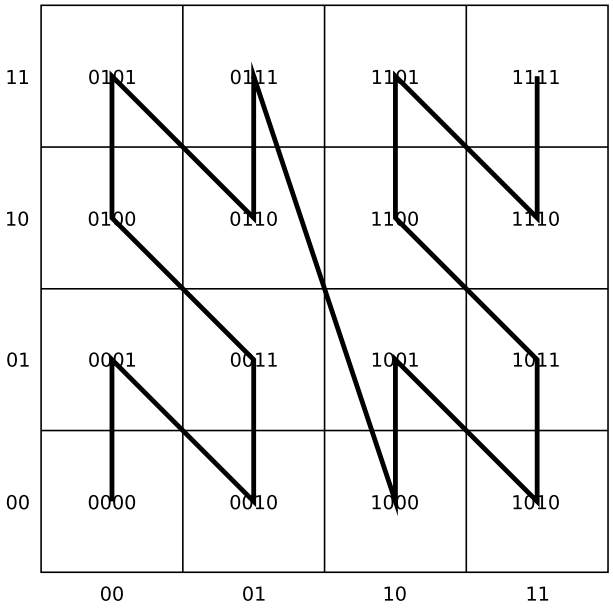
2.2.2 kd-tree

The first data structure to natively support multi-dimensional range queries is the kd-tree, proposed by Bentley [3]. A kd-tree is a tree data structure that organizes points belonging to a k-dimensional space. It uses splitting planes that are perpendicular to one of the coordinate system axes, partitions the data set and assigns each partition to a tree node[1].

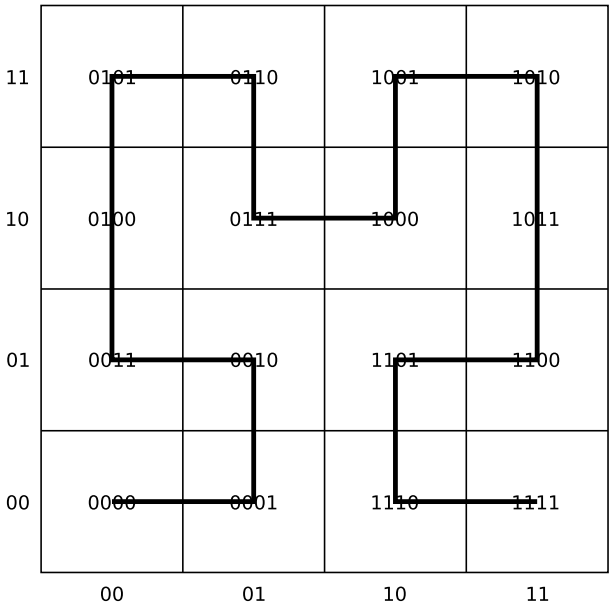
In order to construct a kd-tree, we construct a root node which is responsible for the whole data set. We then cyclically select the dimension on which the plane will be split and we split on the median of the points in this dimension in this node. Tree construction requires $O(n\log n)$ processing time and $O(n)$ memory space. Figure 2.5 shows an example of a kd-tree built upon two-dimensional data.

In order to evaluate a range search query, we start from the root and check if the query region intersects the cell the current node defines. If it

[1] Originally, kd-trees assign to each tree node one point only. The idea of assigning each node a partition of the data set is newer and is formally known as a kd-trie. Because the algorithms and structure are extremely similar, we use the term kd-tree to describe both approaches and focus on the kd-trie for the rest the text.



(a) Z-order space-filling curve



(b) Hilbert space-filling curve

Figure 2.4: Space-filling curves applied on two dimensions.

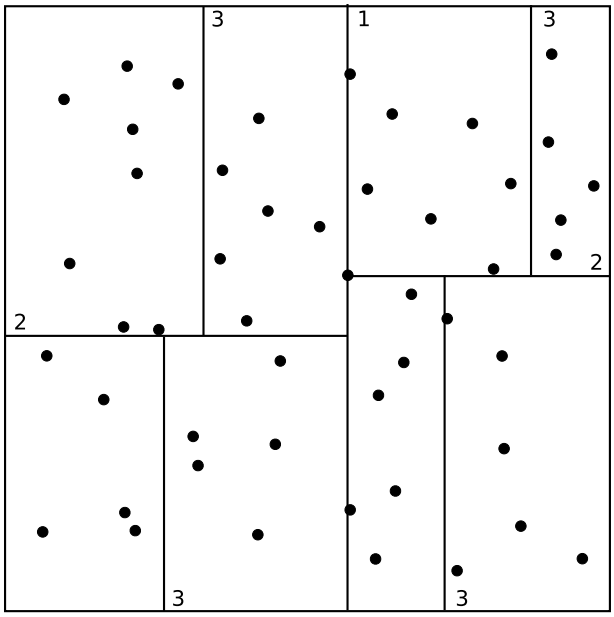


Figure 2.5: Typical kd-tree for two-dimensional data: the lines indicate the splitting planes and the numbers show the level of the subtree when performing the split.

does, then we repeat this check for both children. If it doesn't, no child node will also contain points of interest, so we don't check further down this tree. Applying this algorithm recursively quickly prunes irrelevant portions of the space and the time required to evaluate a range query is shown to be $O(n^{1-1/k} + s)$, where s is the number of data items in the answer set.

2.2.3 R-tree

R-tree is a tree data structure proposed by Guttman [7]. R-trees are similar to B-trees but are used for indexing multi-dimensional data items and support spatial access methods. The R-tree data structure partitions the data space in hierarchically nested boxes, which can even overlap. Each internal node of the R-tree stores a variable (but limited) number of child nodes and each child node's bounding box. Leaf nodes only store data elements and usually correspond to disk pages, like in B-trees.

The insertion and deletion algorithms use the hierarchy of bounding boxes to decide in which leaf node will the insert or delete operation take place. Insertions and deletions result in splitting or merging of existing nodes, so that the tree remains balanced. The searching algorithm also uses bounding boxes to decide which child node to visit to reach the required leaf node.

R-trees do not guarantee good worst-case performance and initial par-

tition of the data space is a key factor in good performance. Kamel and Faloutsos [11] propose the Hilbert R-tree and show that by ordering all data items using the Hilbert space-filling curve there can be significant improvements. Newer variations, like the priority R-tree proposed by Arge et al. [2], provide good worst-case performance.

2.3 Distributed structures with range query support

All distributed hash tables described implement several desired features, such as logarithmic number of messages to the number of users for retrieving information. However, they also share one major drawback: they rely on uniform hashing functions to achieve probabilistically good load balance. This is catastrophic for range and proximity queries, because processing such a query efficiently means benefiting from spatial locality. Locality however is destroyed when keys are uniformly hashed before being stored in the network, so distributed hash tables can not handle such types of queries natively.

This problem has been identified and alternative solutions have been proposed which retain the good characteristics the distributed hash tables demonstrate and, at the same time, can handle range queries and their additional complexity. Such proposals have been inspired by scientific work on distributed hash tables and in fact share routing algorithms and structures with the distributed hash tables that are based on.

2.3.1 PHT

Ramabhadran et al. [16] present a data structure which is based upon a distributed hash table and augment it to support one-dimensional range queries. They name this data structure *Prefix Hash Tree*, or PHT for short. The PHT data structure consists of two discrete layers: a binary trie over the data set and a network layer providing distributed hash table functionality.

The domain being indexed is m-bit binary strings, called keys. Every key is stored at the leaf node whose label in the trie is a prefix of this key. The corresponding peer responsible for this leaf node is obtained by querying the distributed hash table with the label of this node. Therefore, load balancing is as good as the underlying distributed hash table layer can achieve.

An interesting property of the PHT data structure is the definition of a threshold, B, which reflects the maximum number of data items each leaf node can store. If a leaf node stores more than B data items, it must split in two. If any sub-trie stores less than $B+1$ data items, two leaf nodes must merge. With the introduction of this threshold, the shape of the trie becomes dependent on the distribution of the keys. The trie is deep in regions which

are densely populated and shallow in regions which are sparsely populated.

A definite advantage of the PHT data structure is that it can be implemented on top of *any* distributed hash table and inherits its load-balance characteristics.

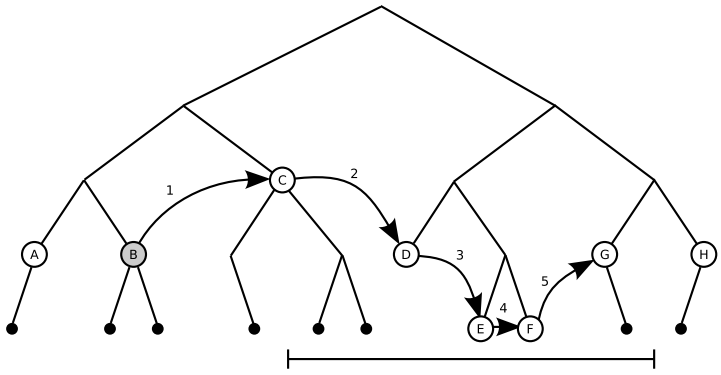
2.3.2 Range queries on P-Grid

Datta et al. [4] present two algorithms for supporting one-dimensional range queries in structured peer-to-peer networks on top of a trie abstraction in $O(\log n)$ complexity. They use the P-Grid network as an example of a binary trie and propose two algorithms for processing range queries.

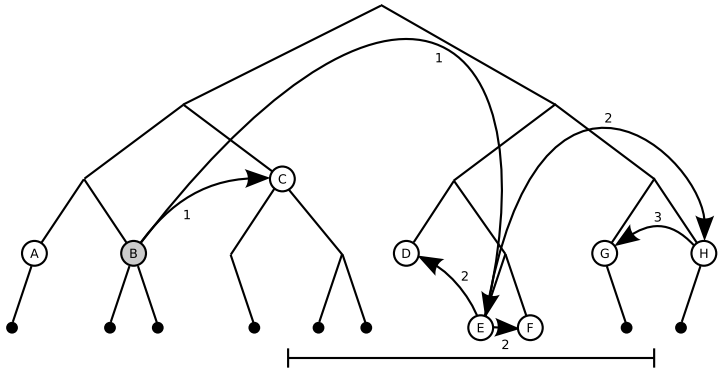
The first one, the *min-max traversal algorithm*, relies on links pointing to the peer that is responsible for the next partition of the ordered key space. This way, when a range query is processed, a P-Grid lookup for the lower bound of the query interval is performed. When the peer responsible for the minimum bound is located, it answers back the relevant data items in his partition and forwards the query to the peer responsible for the next partition in the key space. This exhaustive procedure is applied recursively until the peer responsible for the maximum bound has been located. Figure 2.6(a) shows an example of this algorithm. One drawback of this approach is that it requires new links between peers and cannot be used directly on top of an existing P-Grid network. Another drawback is that a single peer failure can terminate a query prematurely, without reaching the upper bound. Thus, one node failure can easily compromise the reliability of the system.

The second algorithm, the *shower algorithm*, forwards queries in parallel to peers who have part of the answer. When a peer receives a range query, it forwards it to an arbitrary peer responsible for any of the key space partitions within the range of the query. The query is then recursively forwarded to other peers in the query interval using this peer's routing table. Each peer can discover the subset of peers he is responsible for forwarding the message to by taking into account the relative position of the peer who forwarded the query to him in the virtual binary tree. Although it is possible to forward a query to a peer outside the query range, it is guaranteed that this peer will forward the query back to a partition within the range. Figure 2.6(b) provides an example of this algorithm. This technique can retrieve answers faster and is less prone to peer failures but is shown to require more messages than the min-max algorithm on average. Also, it utilizes only the existing P-Grid links to provide answers to range queries.

The shower algorithm has definite advantages over the min-max algorithm and we will focus on this algorithm for the remainder of the text.



(a) Min-max algorithm



(b) Shower algorithm

Figure 2.6: Range query support on P-Grid: The shaded peer initiates the query and the query boundaries are indicated by the line stop markers. The arrows represent the messages sent and the numbers indicate the order the message forwarding takes place.

2.3.3 MURK-CAN

Ganesan et al. [5] experimentally compare, in a peer-to-peer environment, two popular spatial-database solutions to the multidimensional range query problem. The solutions they experiment with are space-filling curves and kd-trees.

The first approach of supporting multi-dimensional queries first maps the data down into a single dimension using a space-filling curve. Afterwards, the single-dimensional data is partitioned among a dynamic set of participating peers. The underlying network is responsible for locating the peer responsible for the minimum bound in the query range and using neighboring links to traverse all peers until it locates the peer responsible for the maximum bound of the query. One drawback of this technique is that every query will reach some non-relevant peers as well, because the space-filling curve's one-dimensional range may actually map to peers irrelevant to the query in the native query space.

The second approach, named *multi-dimensional rectangulation with kd-trees* or MURK for short, partitions the data directly in the high-dimensional space which is the domain of the problem. This technique breaks up the data space into hypercubes, with each peer managing one hypercube. The process of creating this partitioning actually resembles kd-trees: Initially, one peer manages the entire space. When a new peer arrives, the space is split along the first dimension into two parts of equal load, with one peer managing each part. This corresponds to splitting the root node of the kd-tree to two children. As more peers arrive, each splits the partition managed by an existing peer; that corresponds to splitting an existing leaf in the kd-tree. The dimensions are split cyclically to ensure that locality is preserved in all dimensions. The partitioning scheme is very similar to that employed in CAN, with the crucial difference that MURK partitions equally the data *load* instead of the data *space*. Figure 2.7 shows a typical MURK topology on top of a synthetic geographical dataset that reflects points of interest in Greece. (Section 3.1 describes the details of the dataset construction process.)

In the MURK approach, each peer knows his neighboring peers in all dimensions, that is the peers that share a boundary with this peer. For more efficient routing each peer maintains also a random list of $2\log n$ peers, obtained, for example, from random walks. The queries in MURK are answered greedily: the query is sent to the neighboring peer that is minimizing the distance to the query range centroid, where the distance from a peer P to a hypercube Q is defined as the minimum Manhattan distance from any point in P's area to any point in Q. Once the query reaches a relevant peer, the query is flooded to all relevant neighbors recursively.

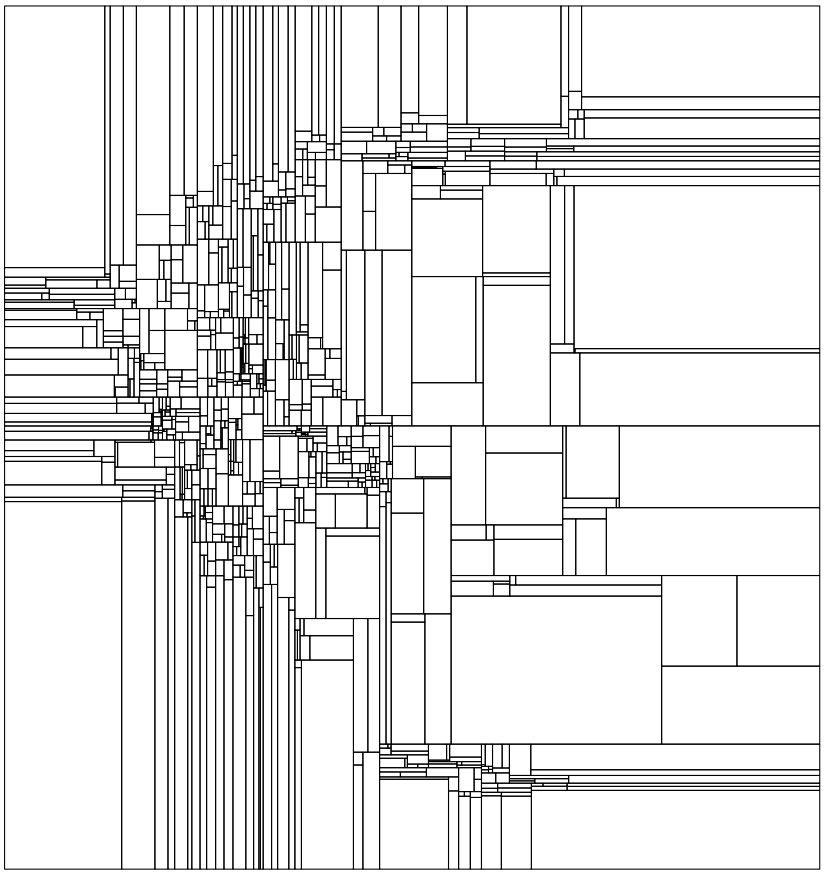


Figure 2.7: Typical MURK-CAN topology on a dataset with geographical locations of points of interest in Greece.

2.3.4 VBI-tree

Jagadish et al. [9] present a balanced tree structure overlay for peer-to-peer networks which supports a variety of centralized multi-dimensional tree structures and guarantees that range queries require at most $O(\log n)$ messages for evaluation.

The VBI-tree is a tree data structure that partitions the problem space and assigns it to the responsibility of some peers. Leaf nodes are responsible for data items and internal nodes are used only for routing purposes. Each node maintains pointers to its parent and children nodes, to its adjacent nodes and to neighbor nodes. Neighbor nodes are nodes at the same level of the tree. Links are maintained to neighbor nodes whose distance from this node is a power of 2. Also each node maintains links to all ancestor nodes and stores the height of the subtrees of all its children. Finally, each internal node is aware of the area each node it links to is responsible for. Every physical peer is responsible for one internal and one leaf node at all times[2].

Range queries are evaluated by first forwarding the query to all neighboring and children nodes that have an intersection with the query boundaries. Then, if the query in not answered completely, it is forwarded to the parent. By recursively applying the above algorithm, every query can be evaluated in $O(\log n)$ number of network hops.

The VBI-tree is a much newer work than the other techniques for performing range queries in peer-to-peer environments which have been presented so far. It addresses problems that were previously very difficult to solve, like specifying the boundaries of the dimensions of the data set, which is a very hard problem in real-time applications. The VBI-tree provides by design a solution to this problem, with the introduction of *discrete data* which actually represents data items that lie outside the current data set's boundaries and with network restructuring algorithms that allow the enlargement of the problem boundaries.

Another problem which is handled by the VBI-tree structure is the problem of a new node joining the network. Previous work relies on unverified statistical assumptions, like that each new peer joins the network by selecting any existing peer with equal probability and splitting its assigned space. This uniform probability is what most peer-to-peer networks require for good load balancing. VBI-tree structure, however, employs a simple but effective algorithm for selecting which peer will further partition his assigned data space and thus manages to enforce good load balance.

[2]In fact, there is an exception: there will always be exactly one peer who will be responsible only for one leaf node, but for no internal node. This follows from the fact that the VBI-tree is always a perfect tree.

2.3.5 Other work

Other research work in the area has been done by Ratnasamy, Karp et al. [18], as part of the GPSR, a research project on geographical routing for mobile wireless networks. In this publication, a geographic hash table is proposed, named GHT. Although GHT has many similarities to the problem we are dealing with, it mainly focuses at the network level, at routing protocols for communication between mobile clients and doesn't provide higher level algorithms.

2.4 Simulators of peer-to-peer networks

Typical peer-to-peer networks involve thousands of computers, make heavy use of the network communication capabilities of the underlying hardware and typically computers join and leave continuously. These facts make researching and obtaining experimental results for peer-to-peer networks a very difficult, error-prone and time-consuming process. The only feasible way to approach such problems is to use realistic simulation models to extract useful information about the (virtual) network, before proceeding to real implementations.

The academic community has created many specialized simulation tools for a variety of peer-to-peer networks. There is no single widely accepted framework for performing experiments on peer-to-peer networks; each individual solution has its own strengths and weaknesses and often focuses on different aspects of the problem.

2.4.1 p2psim

A simulator which is widely used for obtaining experimental data from virtual peer-to-peer networks is p2psim [6]. It is developed by Gil et al. at the Computer Science and Artificial Intelligence Laboratory at Massachusetts Institute of Technology. p2psim is an open-source, multi-threaded, discrete event simulator for peer-to-peer protocols. At the time of this writing, most DHT protocols are supported and have been already implemented: Chord, Accordion, Koorde, Kelips, Tapestry and Kademlia.

2.4.2 PeerSim

PeerSim [10] is another popular peer-to-peer simulator in use. It is developed by Jelasity et al. and is a free-source, Java^{TM}-based simulator. It is designed with scalability and dynamicity in mind and comes with two engines, a cycle-based one and an event-driven one. The simulator comes with no protocol implementations and provides only rudimentary support for many essential

aspects of peer-to-peer networks, but it is extremely flexible, extensible and easily configurable.

PeerSim is the simulator we used to get the experimental results from our prototype implementations. A more thorough description of the PeerSim simulator follows in Section 3.2.1.

2.4.3 Other simulators

Another simulator for peer-to-peer networks is 3LS [22], a generic 3-layer peer-to-peer simulator proposed by Ting and Deters. It's main goal is to handle extremely complex peer-to-peer networks.

Other peer-to-peer networks have specific simulators, like GPS [24] simulator for BitTorrent traffic and GnutellaSim [8] for modeling large-scale Gnutella networks.

2.5 Comparisons between peer-to-peer networks

Although many different peer-to-peer networks and algorithms have been proposed, there hasn't been much work in the area of comparing similar ideas to reveal good practices and possible bottlenecks.

Lua et al. [14] provide a survey and comparison of many peer-to-peer overlay schemes, both structured and unstructured ones. Specifically, they describe in detail CAN, Chord, Tapestry, Pastry, Kademlia and Viceroy structured networks and FreeNet, Gnutella, FastTrack, BitTorrent and eDonkey unstructured networks. The comparison between these networks, however, is rudimentary and qualitative. All networks are compared on some specific aspects (like upper-bound for information lookup cost, reliability and security) but no experiments are performed, only descriptions are given.

Tsoumakos and Roussopoulos [23] make a detailed experimental comparison of different search methods on unstructured peer-to-peer networks. The metrics they compare on are efficiency in discovery (search accuracy), bandwidth consumption and adaptation to topology changes. They evaluate different approaches in searching: flood-based schemes (like breadth first search), other blind methods (like random walks) and some informed methods.

Chapter 3

Results

A brief overview of the work will be provided in this chapter. We will first describe the process used to obtain the different input data sets for the experiments. We will then give an abstract but consistent description of our simulation framework for evaluating different structured peer-to-peer networks with range query support. Special attention will be given to specific issues which prove to be dominating factors for good results, like the initial topology of the network and the scheme used to assign partitions of the problem space to physical peers.

3.1 Dataset creation

In order to evaluate and compare the different approaches in low-dimensional range search, good datasets are required. Input data have two forms: they are points, which correspond to data keys, and rectangles, which correspond to query ranges.

For 2-dimensional data, we started by downloading some geographic datasets from "The R-tree portal" web site [21]. Specifically, we found two datasets describing the roads and rivers of Greece (using a piecewise linear approximation) and a dataset containing the geographic locations of cities and villages in Greece. These original datasets didn't contain range queries.

In order to extract a list of points from the piecewise linear datasets, which will be then used as the data input for the simulations, we used the following technique: First, we choose at random one line segment. Then we pick a point on this segment and we displace it by a (small) random amount. By applying this iteratively we can get a large number of points. The final input data set is constructed by merging[1] the results obtained from this procedure with the datasets originally containing geographic points.

[1] All the original datasets were using the same reference point for the axes, so no transformations were required.

Number of points	Number of queries	Maximum query size
10000	500	50
1000000	50000	50
100000	5000	10
		30
		50
		100
		250
		500
		1000

Table 3.1: Two-dimensional datasets

The query construction process is totally synthetic. We pick one point from the input data set, we displace it by a random amount and then we create a square query around this point, of random distance. One important parameter here which can affect the results is the number of data items which will be retrieved if this query is evaluated. In order to keep this parameter under control, we actually pre-evaluate such queries and keep only the queries whose size satisfies our constraints.

The two-dimensional datasets which were produced by this process are listed in Table 3.1.

For higher dimensions, we only used synthetic data obtained. We created datasets using a uniform distribution for 2, 3, 4, 5, 6 and 8 dimensions. For each dataset, we created 100000 data points and 5000 queries, with each query evaluating to at most 50 data points.

3.2 Network simulation

We will now present the simulation framework we created to obtain the experimental results. The framework is based upon the PeerSim simulator which offers tremendous flexibility but very limited support for running peer-to-peer experiments. For example, PeerSim doesn't natively support storing data items in the class representing a peer; the implementor must take care of that.

In this work, we will simulate the P-Grid, CAN, MURK, PHT and VBI-tree networks and some variations. The PHT network requires an additional DHT layer and we implemented Kademlia[15] for this purpose.

In this section we will first describe the PeerSim internal structure, its initialization and configuration. We will then describe the actual algorithms we used to initialize the topology for each different network. Finally, we will provide an abstract description of the simulation model we created to

experimentally evaluate these networks.

3.2.1 PeerSim internals

The PeerSim simulator structure is based upon components and makes it very easy to quickly create pluggable building blocks, which in fact correspond to Java™objects.

PeerSim supports two different simulation models: a cycle-based and an event-driven model. The second model is the most accurate and can simulate with accuracy continuous-time delays, like communications latencies. The first model is simpler and thus makes it possible to achieve extreme scalability and performance, at the cost of some loss of realism. Special care needs to be taken when the cycle-based model is selected, in order to fully specify what a simulation cycle reflects and how will it affect the output results.

Under the event-driven model, each operation is associated with a time delay. All events carry time-related information, in the form of a timestamp, and get sorted in a queue according to their timestamp. The event which has the timestamp which is closer to current simulation time is processed first: The specified operation is performed, then the timestamp is updated by adding the old timestamp and the delay associated with this operation and finally this or any new events are re-inserted in the event-queue.

Under the cycle-based model, peers communicate with each other directly and are given the control of the simulation periodically, in sequential order. During this time they can perform arbitrary actions, such as call methods of other objects or perform computations. In the application level, this is achieved by having all building blocks (objects) implement the same interface. The most interesting interfaces PeerSim provides are listed in Table 3.2.

In order to make modular programming easier and avoid lengthy recompilation processes, the simulation building blocks (like protocol type) and parameters (like network size, input data, etc.) can be specified at run-time from a configuration file.

The general idea of the simulation model is:

1. Choose a network size (number of nodes).

2. Choose one or more protocols to experiment with and initialize them.

3. Choose one or more *Control* objects to monitor the properties you are interested in and to modify some parameters during the simulation (like the size of the network, the internal state of the protocols, etc.).

4. Run your simulation invoking the *Simulator* class with a configuration file, that contains the above information.

Class	Description
Node	The peer-to-peer network is composed of nodes. A node is a container of protocols. The node interface provides access to the protocols it holds, and to a fixed ID of the node.
CDProtocol	It is a specific protocol, that is designed to run in the cycle-driven model. Such a protocol simply defines an operation to be performed at each cycle.
Control	Classes implementing this interface can be scheduled for execution at certain points during the simulation. These classes typically observe or modify the simulation.

Table 3.2: Interesting subset of interfaces provided by PeerSim.

The life-cycle of a cycle-based simulation is as follows: The first step is to read the configuration file, given as a command-line parameter. The configuration contains all the simulation parameters concerning all the objects involved in the experiment. Then, the simulator sets up the network initializing the nodes in the network and the protocols in them. Each node has the same kinds of protocols; that is, instances of a protocols form an array in the network, with one instance in each node.

At this point, initialization needs to be performed, that sets up the initial states of each protocol. The initialization phase is carried out by *Control* objects that are scheduled to run only at the beginning of each experiment. In the configuration file, the initialization components are recognizable by the `init` prefix.

After initialization, the cycle driven engine calls all components (protocols and controls) once in each cycle, until a given number of cycles, or until a component decides to end the simulation. It is possible to configure a protocol or control to run only in certain cycles, and it is also possible to control the order of the running of the components within each cycle.

The whole scenario is illustrated in Figure 3.1.

3.2.2 Topology construction

Before starting the simulation, the initial network topology needs to be constructed, that is for each virtual peer we must assign links to other peers. These links will be used to forward queries to peers who lie 'closer' to the data items which answer the question. The initial topology and the partitioning of the data items onto peers is an important part of the

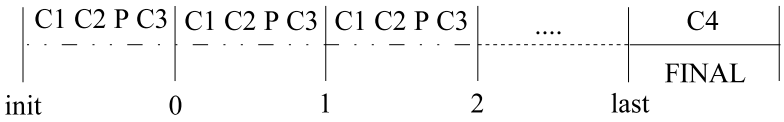


Figure 3.1: Scheduling controls and protocols. The "C" letters indicate a control component, while letter "P" indicates a protocol. The numbers in lower part of the picture indicate the PeerSim cycles. After the last cycle, it is possible to run any final controls to retrieve final snapshots.

comparison of different peer-to-peer networks.

All structured peer-to-peer networks provide a specific procedure which is executed when a new peer wants to join the network. This procedure typically involves knowing at least one participating peer, sending this peer a request to join the network, taking responsibility of a specific area of the problem space and discovering other peers to forward incoming queries to. This is a fully distributed procedure, requires no central server, and is usually associated with a significant communications cost.

This work focuses on the stable state of the system, not on the transient state during which new peers join or existing peers leave the network. Although in peer-to-peer networks peers join and leave continuously and thus a global stable state is impossible to achieve, the assumption of a stable system provides some significant advantages when analyzing each network's performance.

First of all, the performance achieved in the stable state reflects the optimal ability of the network to retrieve answers to queries. Thus this assumption enables us to identify the upper bound associated with each network's performance. More importantly, when we compare two peer-to-peer systems in their stable state, we actually only compare the effectiveness of their algorithms for locating data items. If we were comparing the systems in their transient state, not only would the results be inevitably affected by the effectiveness of the algorithms for node join, node removal and node failure but would also be affected in an unforeseeable and unmeasurable way. Since this work focuses on revealing good practices for peer-to-peer network design, such a comparison would make the system much more complicated and in turn would make the comparison exponentially harder.

In simulation, we benefit from the global view of the network and use it to extract information about other peers. For performance and simplicity, we transform the distributed join algorithms to simpler, centralized algorithms. Such transformation however, must be made with care, to ensure that the original properties of the algorithm are preserved. After the topology initialization, each peer only has a few neighboring links which he uses and is not allowed to use global knowledge to forward a query.

For the P-Grid network, we partition the problem space using Algorithm 1, and invoking it with:

createPrefix(0, Network.size(), 0)

This algorithm assigns each peer a binary prefix in the virtual overlay tree. The implementation of the function `calcSplitPoint`, in line 1 is important as it specifies the final shape of the tree. If `calcSplitPoint` always returns $\frac{start+end}{2}$ the resulting tree will always be a perfect tree. In our implementation, the value returned is random, sampled from a normal distribution with the mean $\mu = \frac{start+end}{2}$ and variance $\sigma^2 = (\frac{end-start}{8})^2$.

Algorithm 1 PGrid::createPrefix(int start, int end, int level)

```
1: split ← calcSplitPoint(start, end)
2: for i = start to split − 1 do
3:    Network.getPeer(i).ID.clearBit(level)
4: end for
5: for i = split to end do
6:    Network.getPeer(i).ID.setBit(level)
7: end for
8: if split − 1 > start then
9:    createPrefix(start, split − 1, level + 1)
10: end if
11: if end > split then
12:    createPrefix(split, end, level + 1)
13: end if
```

After each peer has been assigned a partition of the problem space, Algorithm 2 is run. This algorithm runs iteratively for each peer and locates other peers who will be linked to the first one, while maintaining the P-Grid invariants.

For the CAN network, the space is partitioned and assigned to peers using Algorithm 3. In this algorithm, we create a kd-tree structure which will be used later, in the peer discovery phase. Invocation of this algorithm requires knowing the dimensionality of the problem and a random point p on which the splitting will take place. We calculate the random point with two different methods, either a random point from a uniform distribution over the problem space or a random point from the input dataset. We will henceforth refer to the uniform distribution as *random selection policy* and to the other one as *data selection policy*. Although the data selection policy is unrealistic in practice, it makes a very interesting comparison.

An important decision in the space partitioning phase, is the calculation of the splitting plane, in line 3. In the traditional CAN approach the peer splits his area on two parts of equal space. The position of the plane is therefore the *mean* of the boundaries parallel to the splitting plane. In the

Algorithm 2 PGrid::assignNeighbors()

```
1: for all p ∈ Network do
2:    k ← 0
3:    key ← invertBit(p.ID.getBit(k))
4:    key.append(getRandomBits(119))
5:    g ← getPeerResponsible(key)
6:    while g ≠ p do
7:       p.addNeighbor(g)
8:       k ← k + 1
9:       if p.ID.getBit(k − 1) = 1 then
10:         key.setBit(k − 1)
11:      else
12:         key.clearBit(k − 1)
13:      end if
14:      if p.ID.getBit(k) = 1 then
15:         key.clearBit(k)
16:      else
17:         key.setBit(k)
18:      end if
19:      g ← getPeerResponsible(key)
20:   end while
21: end for
```

MURK approach, the peer splits his area on two parts of equal data load. The position of the plane is then the *median* of the boundaries parallel to the splitting plane.

Algorithm 3 CAN::partitionSpace(Point p, int dimensionality)

1: $n \leftarrow$ Network.getPeerResponsible(p)
2: $splitdim \leftarrow n$.getDepth() mod $dimensionality$
3: $splitline \leftarrow n$.calcSplitPoint($splitdim$)
4: n.left, n.right $\leftarrow n$.splitArea($splitline$)
5: n.left.parent $\leftarrow n$
6: n.right.parent $\leftarrow n$
7: move data items from n to n.left and n.right, as appropriate

In order to discover peers to link to, Algorithm 4 is run. This algorithm locates all peers who share a boundary with each peer. It relies on the kd-tree structure created in Algorithm 3 to achieve good performance. Additionally, each peer maintains $2 * \log n$ links to other random peers, where n is the size of the network. In practice, the addresses of the random peers would be obtained by random walks in the network.

For the PHT network, the space is partitioned by Algorithm 5 which creates the virtual binary tree over the whole network. There is no need to perform the additional step of locating peers to link to, as this is handled automatically by the DHT layer.

The VBI-tree network specifies its own algorithm for selecting peers to answer the join requests. We simply implemented the algorithms the authors described in their original publication [9].

3.2.3 Simulation model

In order to be able to run experiments with big networks (like networks with 100000 simultaneously connected peers) we implemented all protocols using the cycle-based model. The benefit of this approach is mainly a large performance gain. Even with massive input sets, all simulations finish in a few minutes.

The price we have to pay for using the cycle-based model is the lack of the ability to simulate varying continuous-time delays. This can be epitomized in the lack of transport model simulation, which means that we associate a fixed cost to each network operation. Another drawback is that the programming complexity rises substantially because we must take care of many details and quirks the cycle-based model has[2].

In the cycle-based simulator we implemented, we associate a fixed time delay of one simulation cycle each time a message must pass through the

[2] An example of this complexity is the controls.BufferSwapper class, described later.

Algorithm 4 CAN::assignNeighbors()

```
 1: for all p ∈ Network do
 2:   t ← p
 3:   q ← createEmptyQueue()
 4:   while t.parent exists do
 5:     if t.parent.left = t then
 6:       sibling ← t.parent.right
 7:     else
 8:       sibling ← t.parent.left
 9:     end if
10:     if sibling and p share a boundary then
11:       q.push(sibling)
12:     end if
13:   end while
14:   while q is not empty do
15:     tree ← q.pop()
16:     if tree is a leaf node then
17:       p.addNeighbor(tree)
18:     else
19:       if tree.left and p share a boundary then
20:         q.push(tree.left)
21:       end if
22:       if tree.right and p share a boundary then
23:         q.push(tree.right)
24:       end if
25:     end if
26:   end while
27: end for
```

Algorithm 5 PHT::partitionSpace(int blocksize)

```
1: tree ← createEmptyTree()
2: addAllData(tree.root)
3: while ∃n ∈ tree which has more than blocksize data items do
4:    n.left = createNode()
5:    n.left.ID = n.ID.append('0')
6:    n.right = createNode()
7:    n.right.ID = n.ID.append('1')
8:    for all item ∈ n.getData() do
9:       if item starts with n.left.ID then
10:         move item to n.left
11:      else
12:         move item to n.right
13:      end if
14:   end for
15: end while
16: for all nd ∈ tree do
17:   if nd is leaf node then
18:      DHT.put(nd.ID, nd)
19:   else
20:      DHT.put(nd.ID, INTERNAL_NODE)
21:   end if
22: end for
```

network to another peer. The time to process the queries and the time to prepare the answers to queries is considered trivial. These assumptions were based upon the fact that the dominant delay is the network communication, which typically takes a few seconds, while processing and retrieving data from memory typically takes a few milliseconds. On simulation initialization, random peers are assigned queries, randomly. On each cycle the query is duplicated or forwarded throughout the network until all answers have been retrieved. If all queries have retrieved all their answers, the simulation ends.

This simulator is implemented on the PeerSim framework using a custom set of classes implementing the `Control` and `CDProtocol` interfaces. In each simulation cycle we usually start by obtaining statistics, using the classes `PrecisionRecallStats`, `AnswerTimeStats`, `ExtendedAnswerTimeStats` and `NodeStats`. Then we run an auxiliary set of classes, like `SimulatorStopper` which checks if simulation should stop, `Shuffle` which is an internal PeerSim class for shuffling the order of execution of each peer's protocol. Another important auxiliary class is the `BufferSwapper` which ensures that if a peer has not yet processed his incoming queue in this cycle, new messages (from this cycle) will not interfere with messages in his queue (from previous cycles).

In an attempt to provide a more realistic network model with limited bandwidth between peers, we constructed the `CongestionCompensation` class which actually limits the input queue of each node to accept only a few messages per cycle. If a message is sent while the buffer is full, message delivery is postponed until the next cycle. This design is consistent with the common practice when building network applications: If a network socket cannot be connected to another computer, an error message is returned after a certain amount of time has passed (time-out limit) and the attempt is repeated at a later time.

Also, in order to verify the simulation, we created a special class, `AnswerValidator`, which at the end of the simulation compares the answers received from our model with the answers obtained by querying the whole input dataset. If this two sets are not the same, the simulation is invalid. This class was very helpful in quickly evaluating if new changes affected the simulation validity.

The classes implementing the network protocols and initializations can be found in Table 3.4 and the classes implementing the additional functionality, like statistics processing, are listed in Table 3.3. A sample configuration file is given below:

```
1 # simulation parameters
2 random.seed 1234567890
3 simulation.cycles 500
4 network.size 10000
5
6 # specifying hashing
7 hash.dimensions 2
8 hash.min1  100000
```

```
9  hash.max1  950000
10 hash.min2 3800000
11 hash.max2 4700000
12
13 # protocols
14 protocol.lnk peersim.core.IdleProtocol
15
16 protocol.skip peersim.core.IdleProtocol
17
18 protocol.can sim.protocols.CANProtocol
19 protocol.can.linkable lnk, skip
20
21 # initializers
22 init.tplinit sim.init.CANInitNeighbors
23 init.tplinit.protocol lnk
24
25 init.skipinit peersim.dynamics.WireKOut
26 init.skipinit.protocol skip
27 init.skipinit.k 2*13
28
29 init.qinit sim.init.FileInserter
30 init.qinit.file Q5K.dat
31 init.qinit.networkadapter sim.init.CANQueryAdapter
32 init.qinit.networkadapter.protocol can
33
34 init.dinit sim.init.FileInserter
35 init.dinit.file D100K.dat
36 init.dinit.networkadapter sim.init.CANDataAdapter
37 init.dinit.networkadapter.protocol can
38 init.dinit.networkadapter.selpolicy data
39
40 order.init dinit, tplinit, skipinit, qinit
41
42 # controls
43 control.shf peersim.cdsim.Shuffle
44
45 control.answ sim.controls.ExtendedAnswerTimeStats
46 control.answ.protocol can
47 control.answ.FINAL
48
49 control.swap sim.controls.BufferSwapper
50 control.swap.protocol can
51
52 control.stop sim.controls.SimulatorStopper
53 control.stop.protocol can
54
55 control.stat sim.controls.NodeStats
56 control.stat.protocol can
57 control.stat.FINAL
```

```
58
59 control.gc sim.controls.CollectGarbage
60
61 control.pnr sim.controls.PrecisionRecallStats
62 control.pnr.protocol can
63 control.pnr.FINAL
64
65 order.control pnr, shf, swap, answ, stop, stat, gc
```

Class	Description
controls.BasicStats	Abstract base class for obtaining statistical data.
controls.NodeStats	Keeps track of messages received and forwarded for each peer.
controls.ExtendedAnswerTimeStats	Keeps track of answer messages to queries and records their statistics.
controls.PrecisionRecallStats	Calculates precision and recall statistics.
controls.BufferSwapper	Isolates the next cycle's input buffer from the current input buffer, ensuring consistent simulation results.
controls.SimulatorStopper	Stops simulation if all queries have been answered.
controls.CollectGarbage	Forces garbage collection.
controls.AnswerValidator	Validates the query answers obtained by simulation with the answers obtained by querying the whole input dataset.
controls.CongestionCompensation	Provides a more realistic network model by allowing only a specific number of messages to be received by peers on each cycle.
misc.Query	Interface for all query objects.
misc.CANQuery	CAN-specific query object.
misc.PGridQuery	P-Grid-specific query object.
misc.KademliaQuery	Kademlia-specific query object.
misc.VBIQuery	VBI-specific query object.
misc.CANNode	Custom CAN node storage.
misc.VBINode	Custom VBI node storage.
misc.BitString	Handles arbitrary length bit strings.
misc.Hasher	Performs hashing and one-dimensional space-filling curve mapping.

Table 3.3: Description of the classes our framework consists of, regarding auxillary operations like statistics processing. Classes in italics represent abstract classes.

Class	Description
init.BaseInserter	Provides an interface for all data loaders.
init.FileInserter	Loads data from a specific file.
init.RandomInserter	Generates random data with specific properties.
init.NetworkAdapter	Interface for all network-specific adapters.
init.PGridDataAdapter	Data adapter for data insertion in P-Grid.
init.PHTDataAdapter	Data adapter for data insertion in PHT.
init.CANQueryAdapter	Query adapter for query insertion in CAN.
init.CANDataAdapter	Data adapter for data insertion in CAN.
init.VBIQueryAdapter	Query adapter for query insertion in VBI.
init.VBIDataAdapter	Data adapter for data insertion in VBI.
init.PGridInit	Initializes the P-Grid network topology.
init.KademliaInit	Initializes Kademlia topology.
init.CANInitNeighbors	Initializes neighboring links in CAN.
protocols.DHT	Interface for all DHT implementations.
protocols.KademliaProtocol	Kademlia DHT implementation.
protocols.ControlInterface	Interface for extracting statistical data from protocols with range-query support.
protocols.CANProtocol	CAN protocol implementation.
protocols.PGridProtocol	P-Grid protocol implementation.
protocols.PHTProtocol	PHT protocol implementation.
protocols.VBIProtocol	VBI-tree protocol implementation.

Table 3.4: Description of the classes our framework consists of, regarding network implementations. Classes in italics represent abstract classes.

Chapter 4

Analysis

In this chapter we will first present a description of the metrics we use to evaluate the comparison between different structured peer-to-peer systems. Then we will present the graphs with the results of the comparison.

4.1 Description of the metrics

For each query, we can observe a partition on the set of peers. Some peers have data items which are relevant to the query and some that don't. Also for each simulation cycle, some peers have been accessed by this query (either in order to obtain answers or for routing purposes) and some peers haven't. Each peer belongs to any combination of these sets. That means that for each query in a specific simulation cycle each peer belongs either to the relevant & accessed set, to the relevant & non-accessed set, to the non-relevant & accessed set or to the non-relevant & non-accessed set.

By observing the ratios between these quantities, we can measure some interesting phenomena. The ratio between the number peers in the relevant & accessed set to the number of peers relevant to the query will reveal how quickly we retrieve answers to this specific query. This ratio is expected to be 1 when the simulation ends because in a static network all algorithms retrieve every answer. Another interesting ratio is between the number of peers in the relevant & accessed set to the number of peers accessed by this query. This ratio will show us the how efficient the query processing is. A ratio of 1 is optimal, meaning that only peers relevant to the query were disturbed. Therefore, for each query we define:

$$precision(t) = \frac{\text{number of peers accessed and relevant until cycle } t}{\text{number of accessed peers until cycle } t}$$

$$recall(t) = \frac{\text{number of peers accessed and relevant until cycle } t}{\text{number of relevant peers}}$$

We record these two metrics as the simulation evolves in time (in simulation cycles).

Another interesting metric is the number of the peers who were accessed by the end of the simulation and were relevant to the query. This is the actual number of peers storing the answers to the query. Analogously, we also measure the number of peers who were accessed by the end of the simulation but were not relevant to the query. This is the number of peers who were disturbed while having no relation to the query.

We keep track of the last two metrics in relation to the answer size of the query, for all networks.

In addition to these metrics, we also count the number of messages each peer receives, forwards or sends as query answers. If the distribution of these values is greatly dispersed this means that some peers do significantly more work than the others and the load is not balanced equally among all participants.

4.2 Plots

In this section we will present the graphical plots which were generated from our simulations. We will see how the networks perform against each other and we will describe and focus on interesting phenomena. 'data' and 'random' appearing on legends show the current selection policy.

Results obtained by running the PHT simulations are missing. We omitted the presentation of the results because PHT performs poorly in terms of speed. PHT performs successive lookups on the DHT layer, in order to receive answers. All DHTs require a few cycles to answer back with data and the overall PHT performance is dominated by this fact. PHT requires around 250 simulation cycles, whereas most networks finish in 30 cycles. Therefore, including PHT results in the following plots would ruin their legibility and has been left out.

4.2.1 Recall measurement

The first thing we were interested in was seeing how fast would a random query be answered. From a user perspective, answer speed is one of the most important factors and can be split into two subproblems: how fast will the total number of answers be retrieved and how fast will the first few answers be retrieved. The generated plot is in Figure 4.1.

Another interesting question is what part of the total effort is spent on retrieving data items from peers who have an answer and what is spent on routing, with respect to the query answer size. This is depicted in Figures 4.3 and 4.2.

As observed, the number of peers is linear to the query size and for simplicity from now on, we will refer to these plots using their least squares

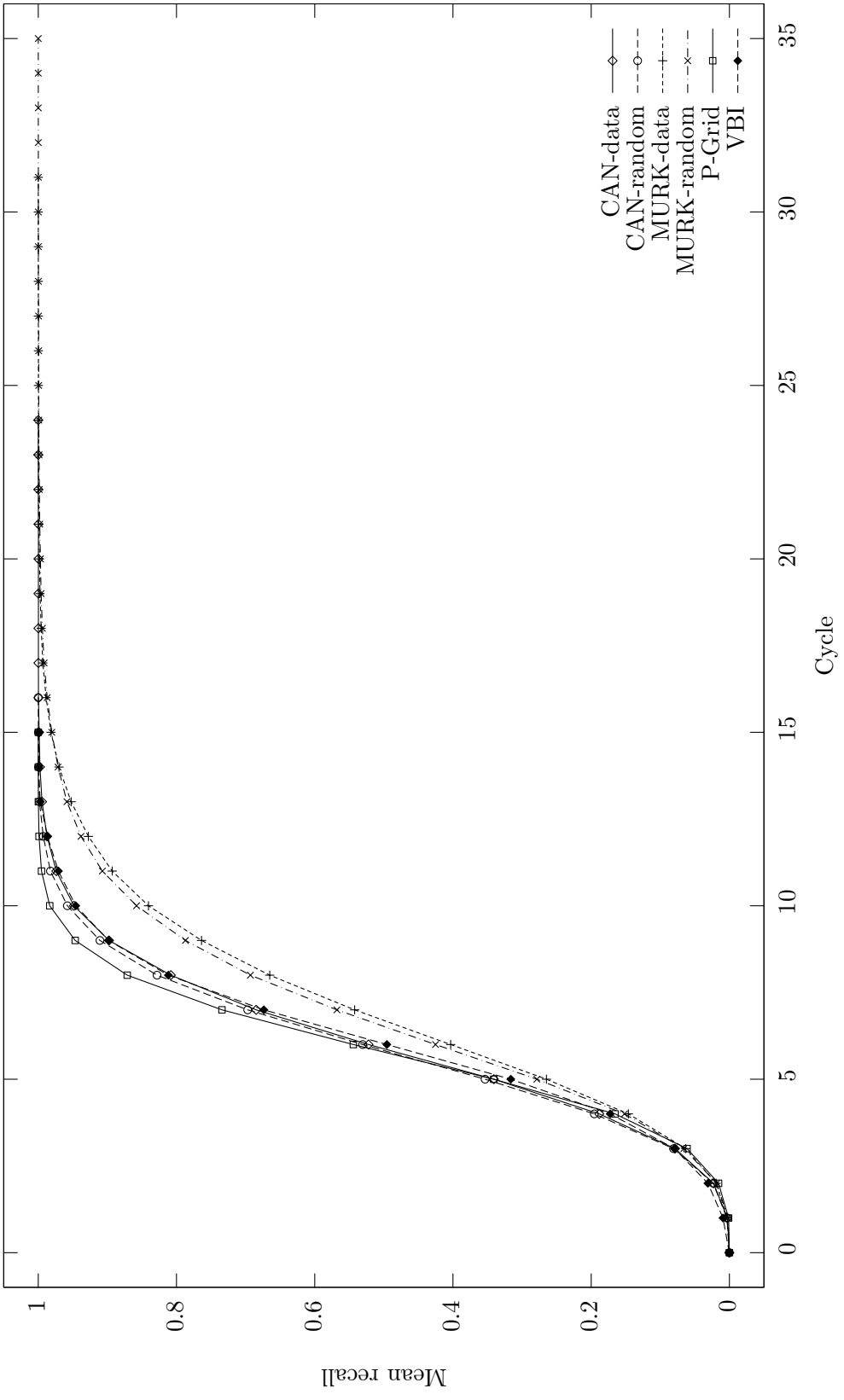


Figure 4.1: Recall graph for the real 2-d dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

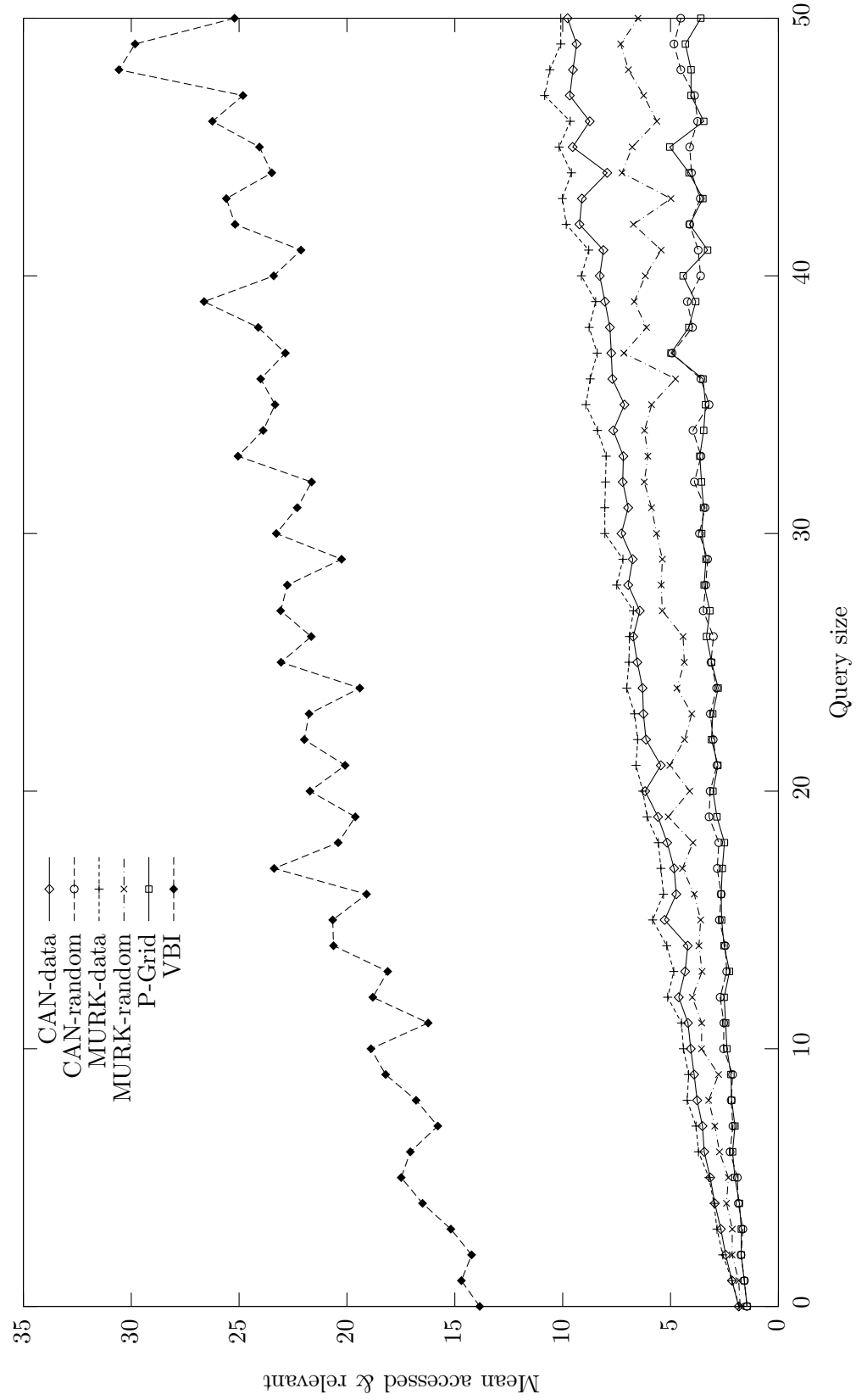


Figure 4.2: Plot of the relevant & accessed peers versus the query size (in data items) for the real 2-d dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

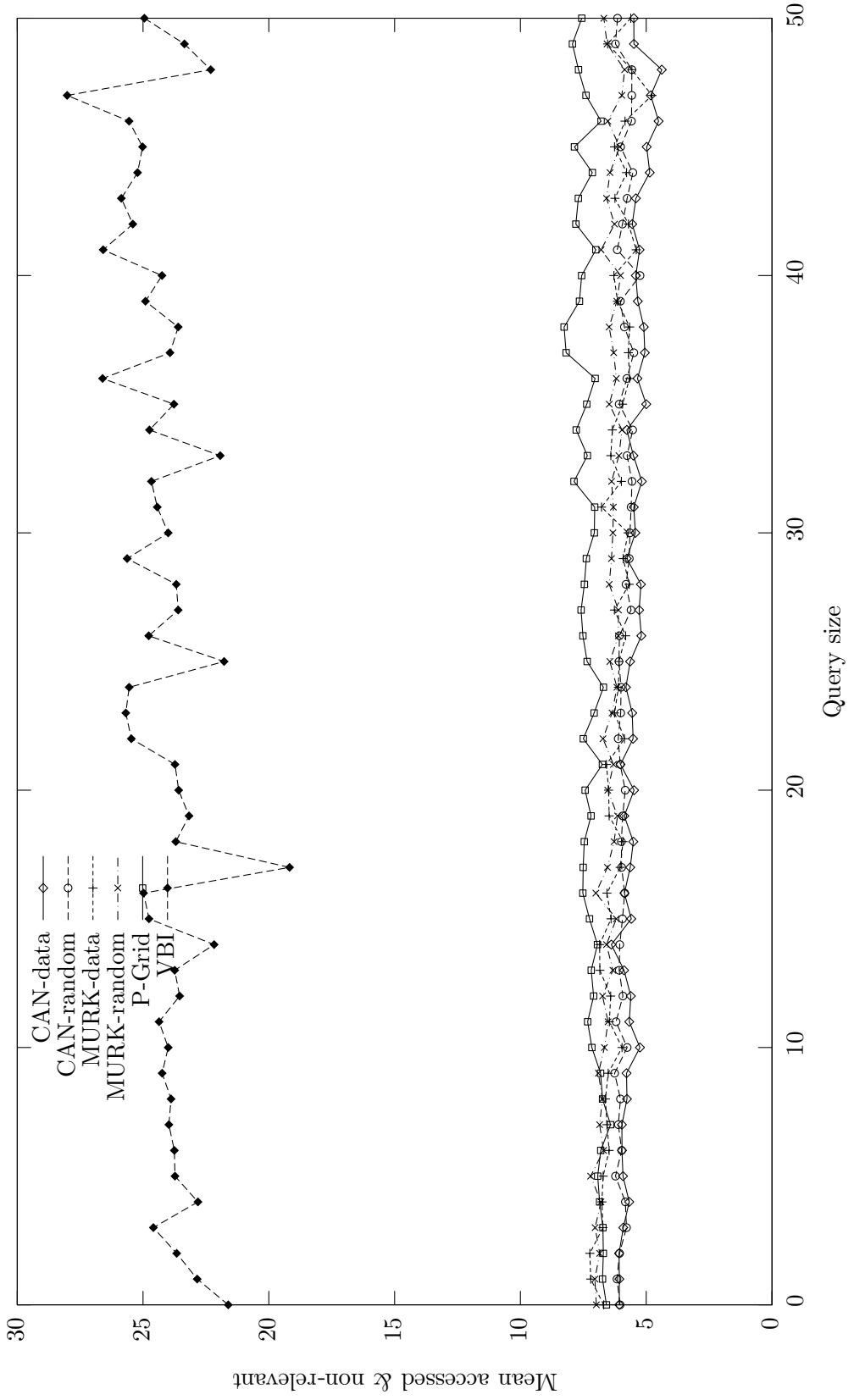


Figure 4.3: Plot of the non-relevant & accessed peers versus the query size (in data items) for the real 2-d dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

approximation. That is, we will find four parameters, a_{RA}, b_{RA}, a_{NRA}, b_{NRA} that minimize the error of these two equations, where qs is the query size:

$$RA = a_{RA} * qs + b_{RA}$$

$$NRA = a_{NRA} * qs + b_{NRA}$$

4.2.2 Effect of dimensionality

In this section we present the performance of these networks perform in higher dimensions. The Figures 4.4, 4.5 and 4.6 were produced by running our synthetic dataset for 2, 3 and 5 dimensions respectively.

As for the number of peers who were required to get the results depicted above, we plot their least squares estimation parameters in Figures 4.7, 4.8, 4.9, 4.10.

4.2.3 Effect of network size

In order to evaluate the scalability of each network, we performed simulations, using the same input datasets but with a varying number of participating computers. Figures 4.11 and 4.12 show how recall performance is affected when the network has 1000 and 50000 peers, respectively.

4.2.4 Load balancing

A very important aspect of peer-to-peer systems is their ability to equally distribute load to all participants. In order to evaluate this, we obtained simulation results for the precision metric, shown in Figure 4.13. The final precision value of each network represents the percentage of disturbed peers who actually had answers for our query.

However, the precision metric only gives a vague idea of the load balance quality and not a quantitative result. In order to be able to obtain such results, we order the number of incoming messages in descending order and for each peer in the horizontal axis we plot the sum of the messages received from the peers encountered so far, normalized by the total number of messages. The optimal load balancing would separate load equally and would produce a line from the axis starting point to the upper right corner. The resulting plot is shown in Figure 4.14. This plot allows us to see that CAN-data and MURK-data are performing best, VBI follows closely (albeit some small percentage of peers is very heavily loaded) and P-Grid and CAN-random networks behave poorly, with 10% of the peers being responsible for 50% of the load.

Interesting as the results might be, they still don't reflect the actual number of messages sent, since one network can be more expensive than

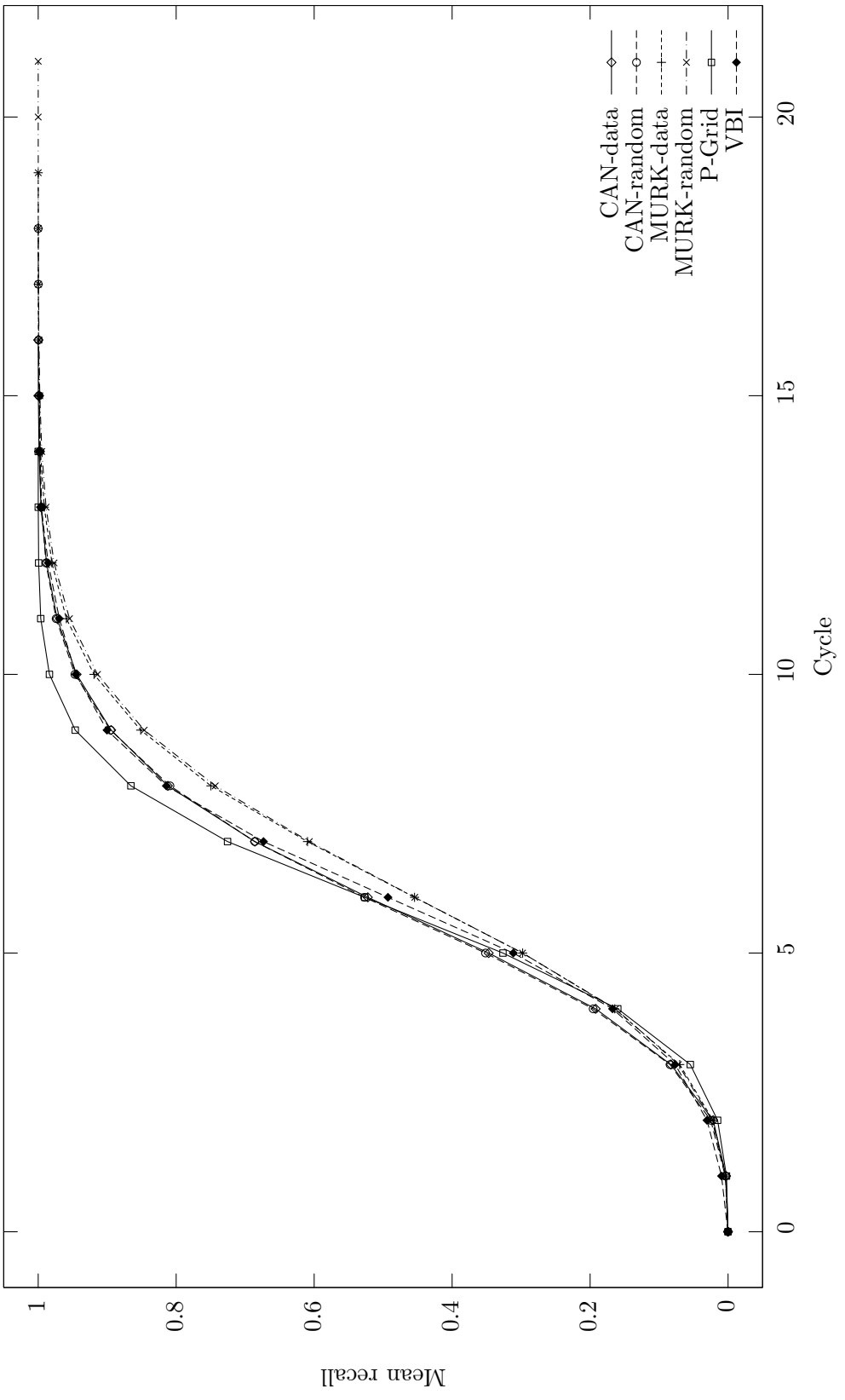


Figure 4.4: Recall graph for the 2-dimensional synthetic dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

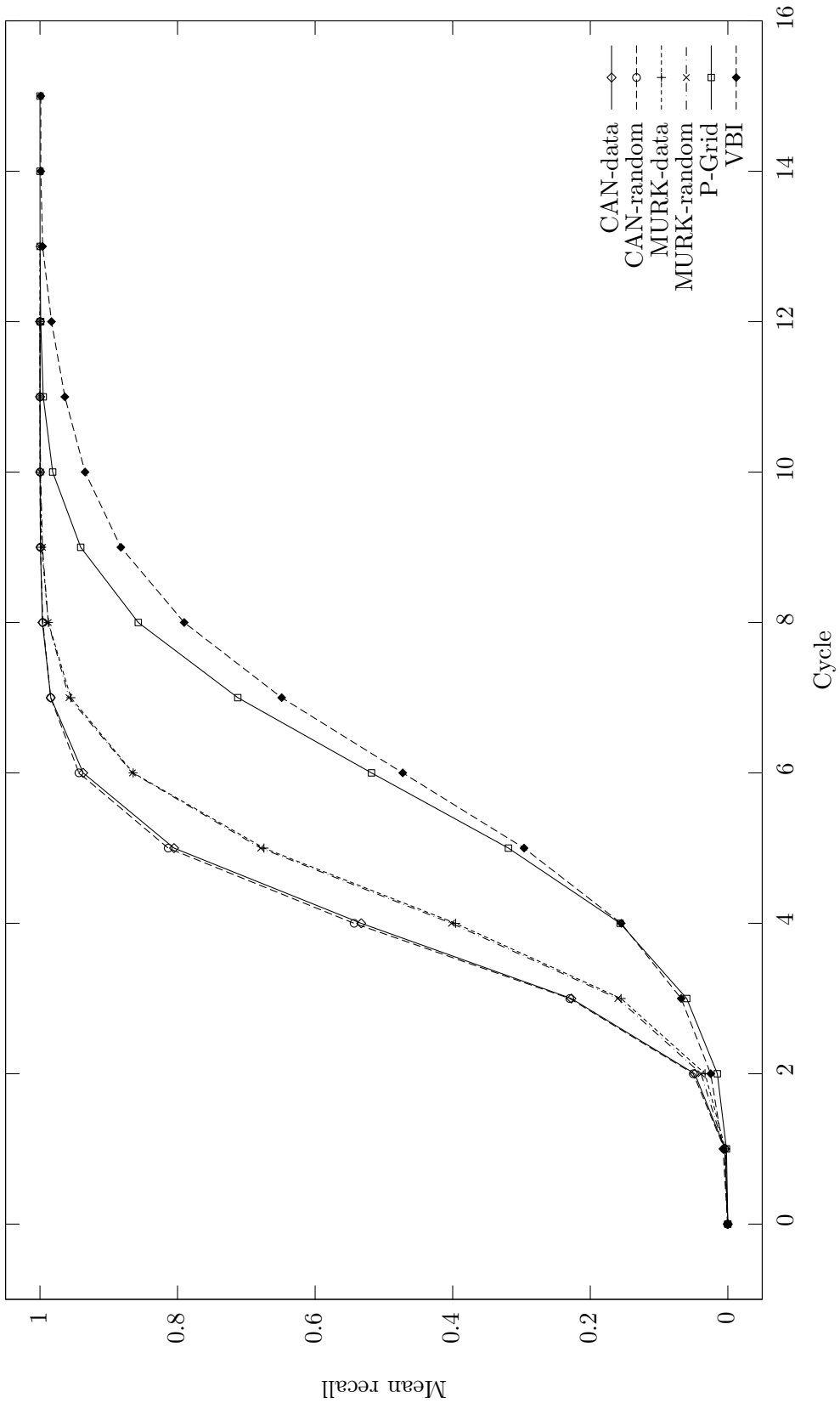


Figure 4.5: Recall graph for the 3-dimensional synthetic dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

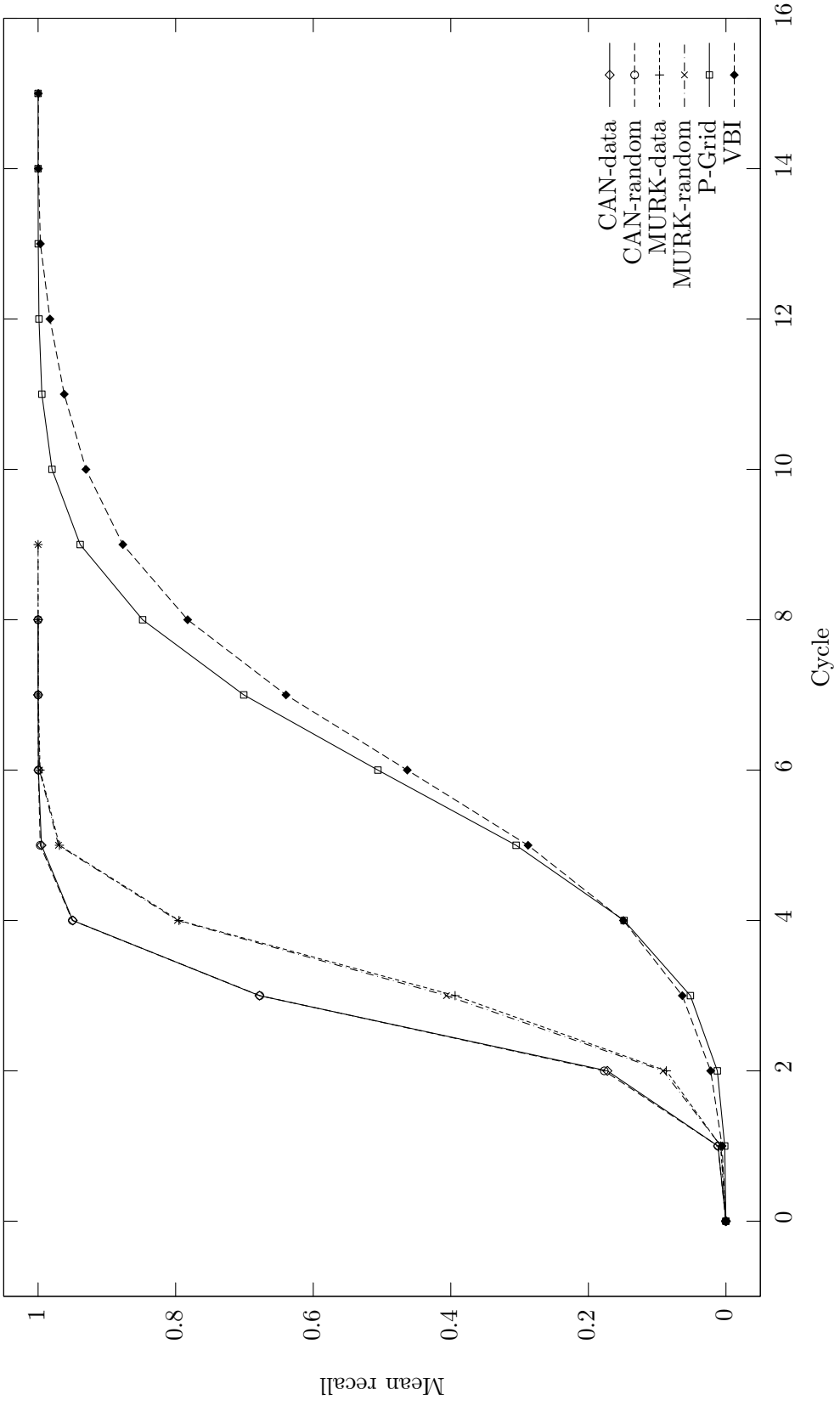


Figure 4.6: Recall graph for the 5-dimensional synthetic dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

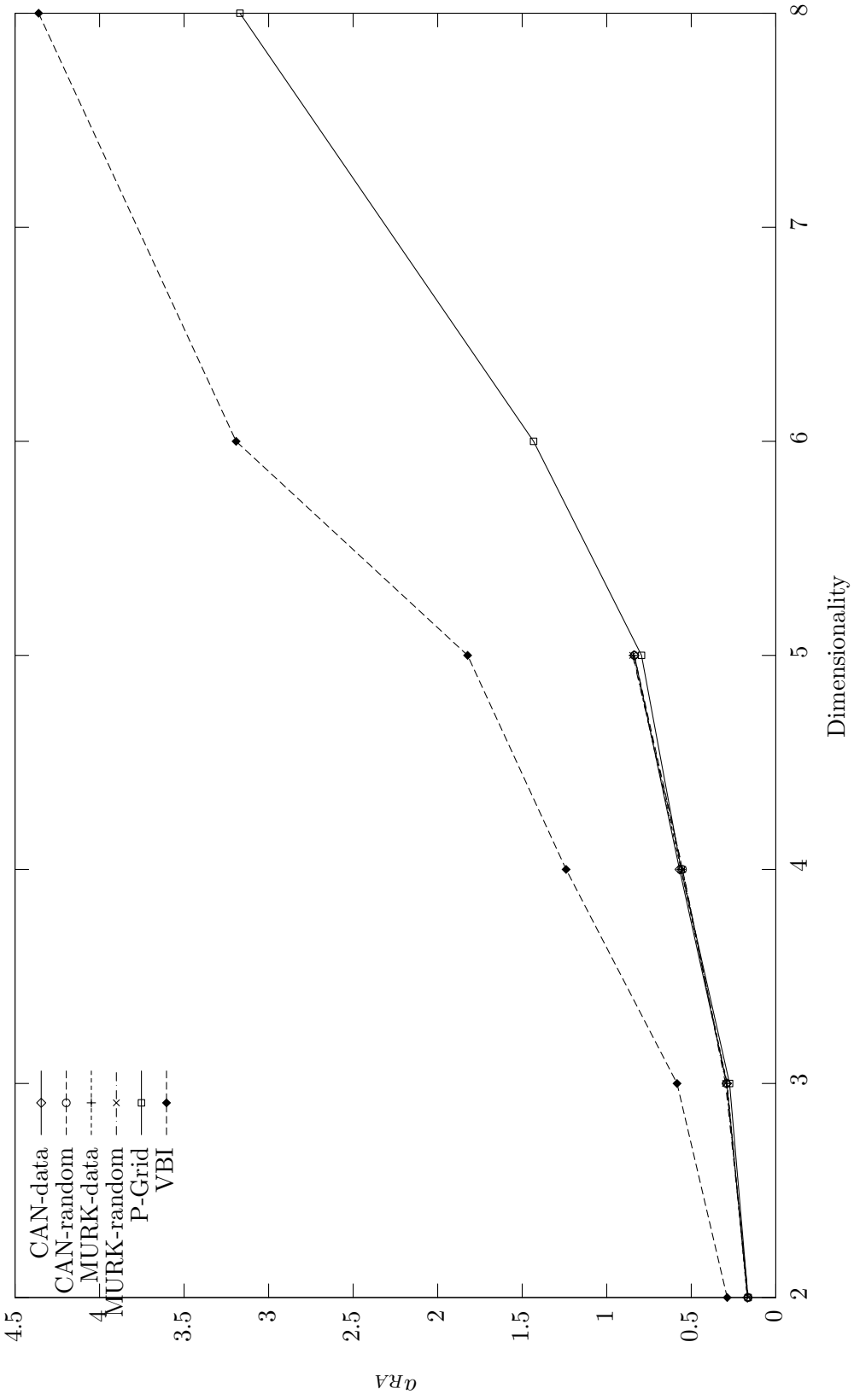


Figure 4.7: Plot of the a_{RA} parameter versus the dimensionality of the problem for synthetic datasets with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

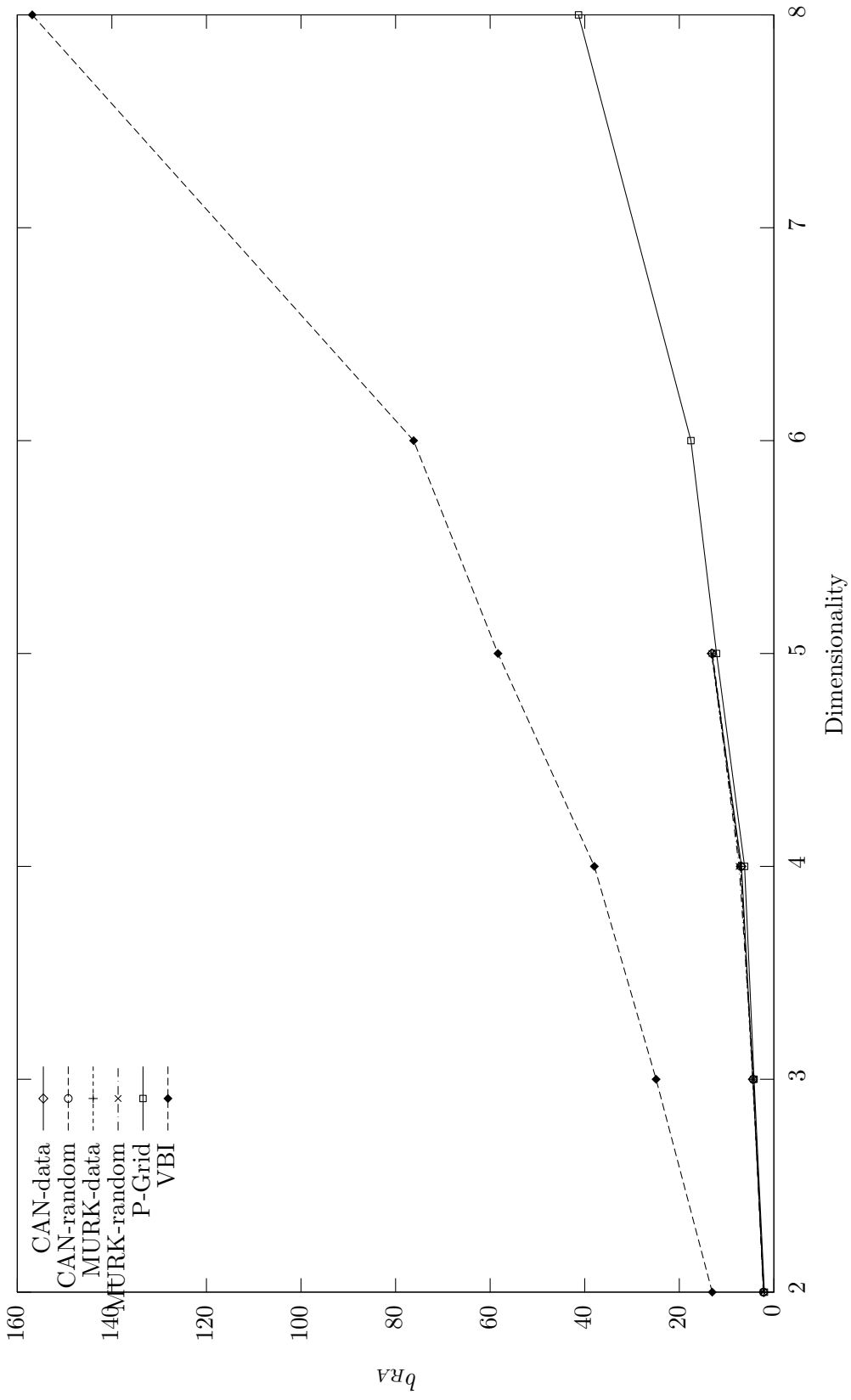


Figure 4.8: Plot of the b_{RA} parameter versus the dimensionality of the problem for synthetic datasets with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

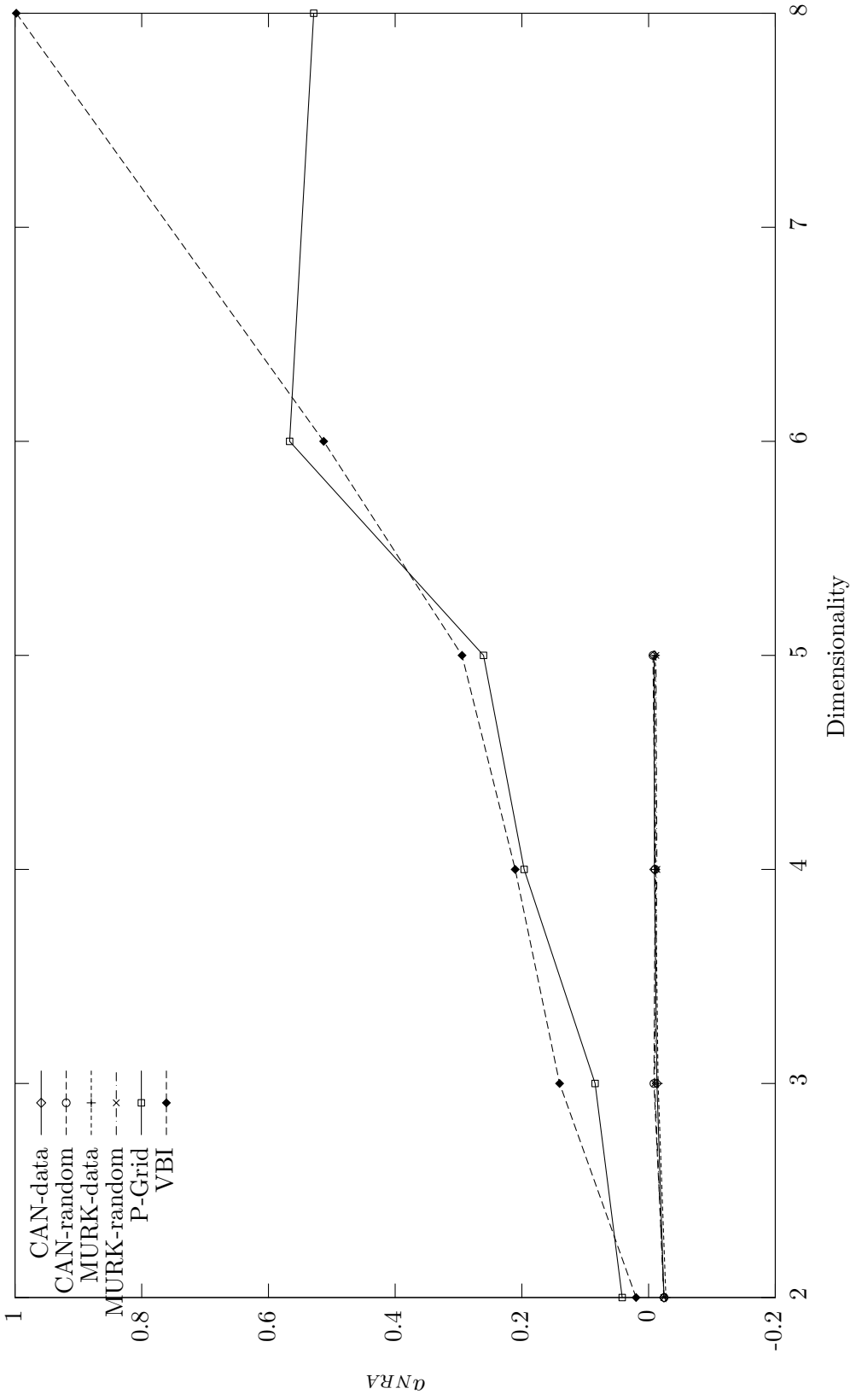


Figure 4.9: Plot of the a_{NRA} parameter versus the dimensionality of the problem for synthetic datasets with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

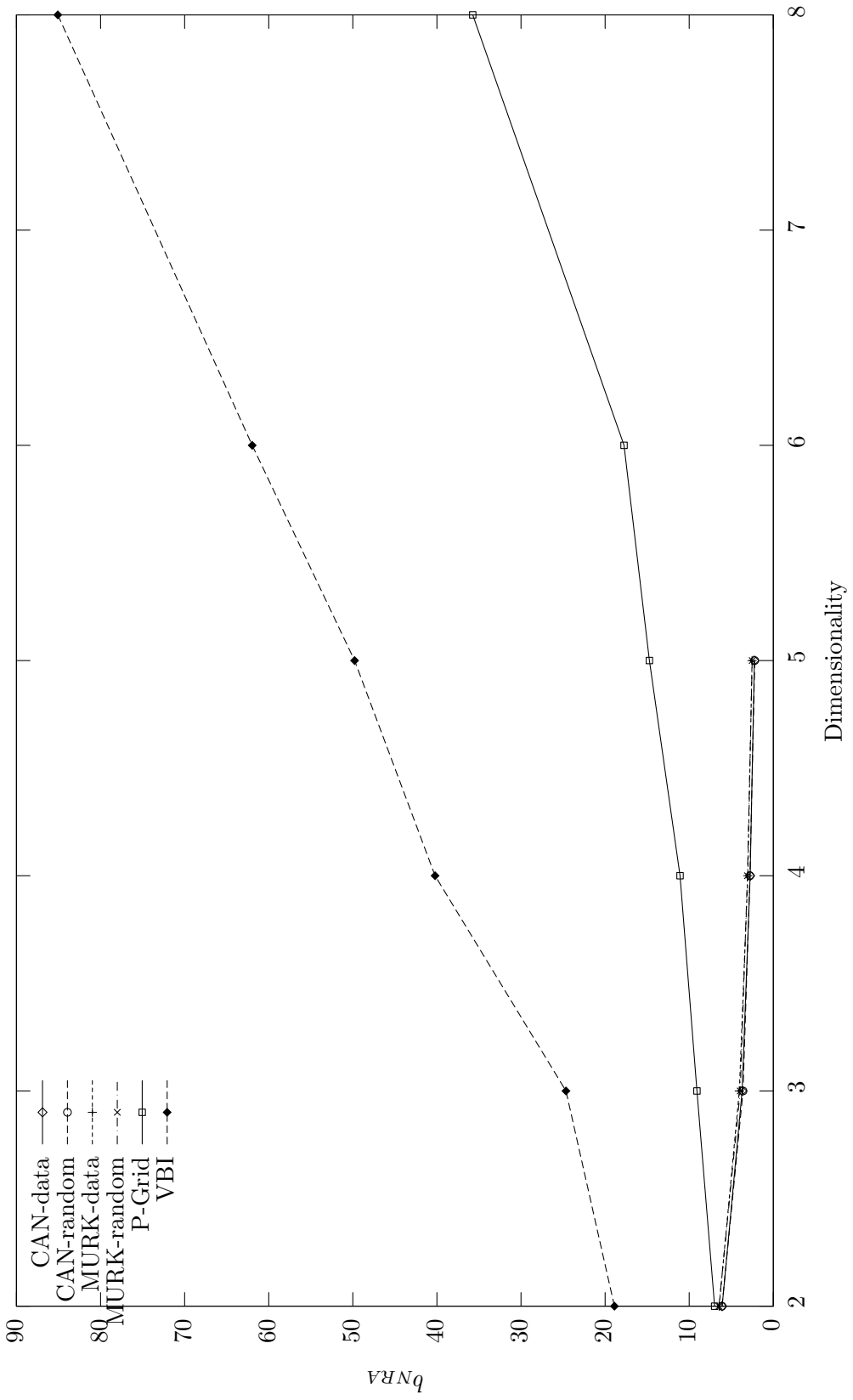


Figure 4.10: Plot of the b_{NRA} parameter versus the dimensionality of the problem for synthetic datasets with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

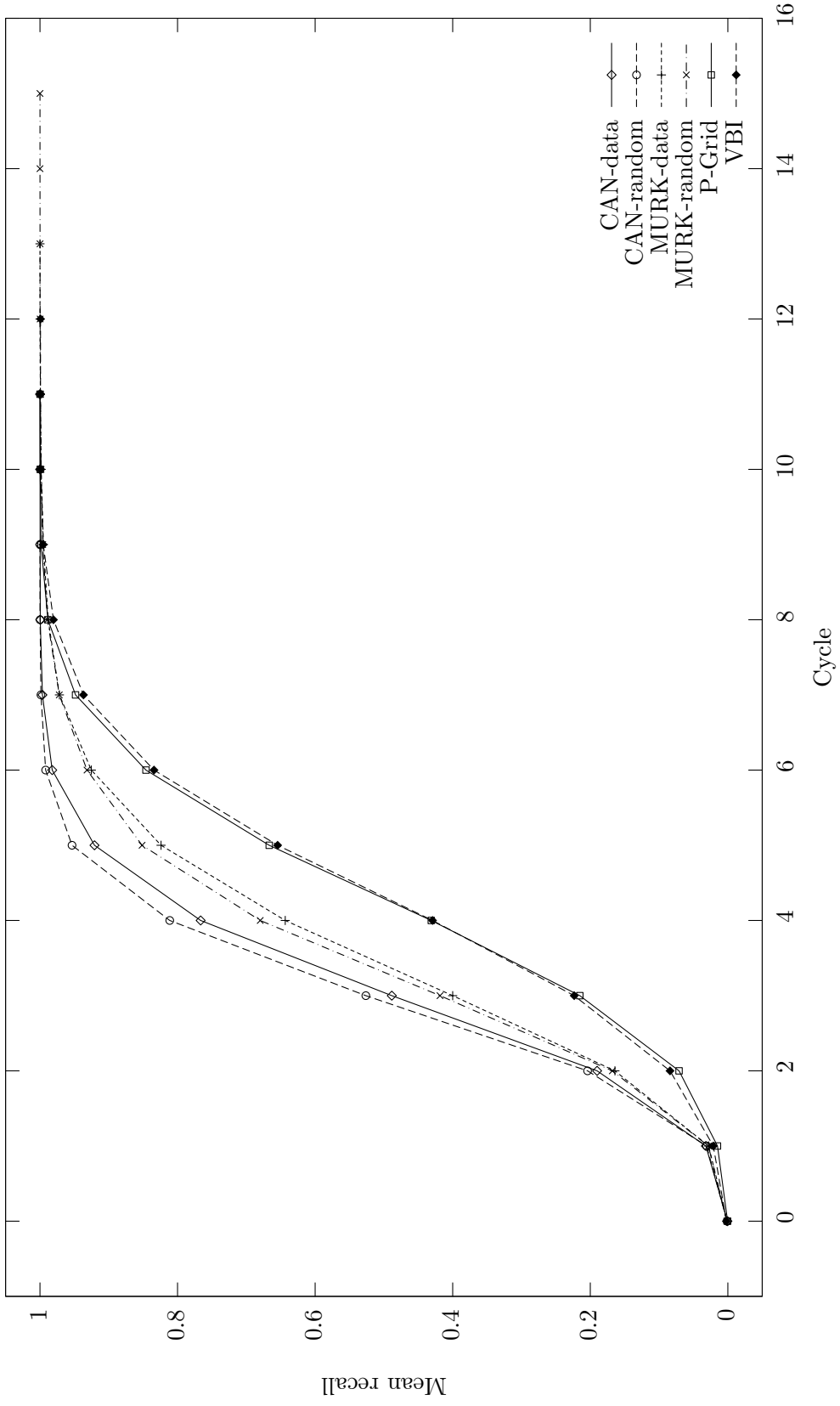


Figure 4.11: Recall graph for the real 2-d dataset with 100000 data items, distributed among 1000 peers and performing 5000 range queries.

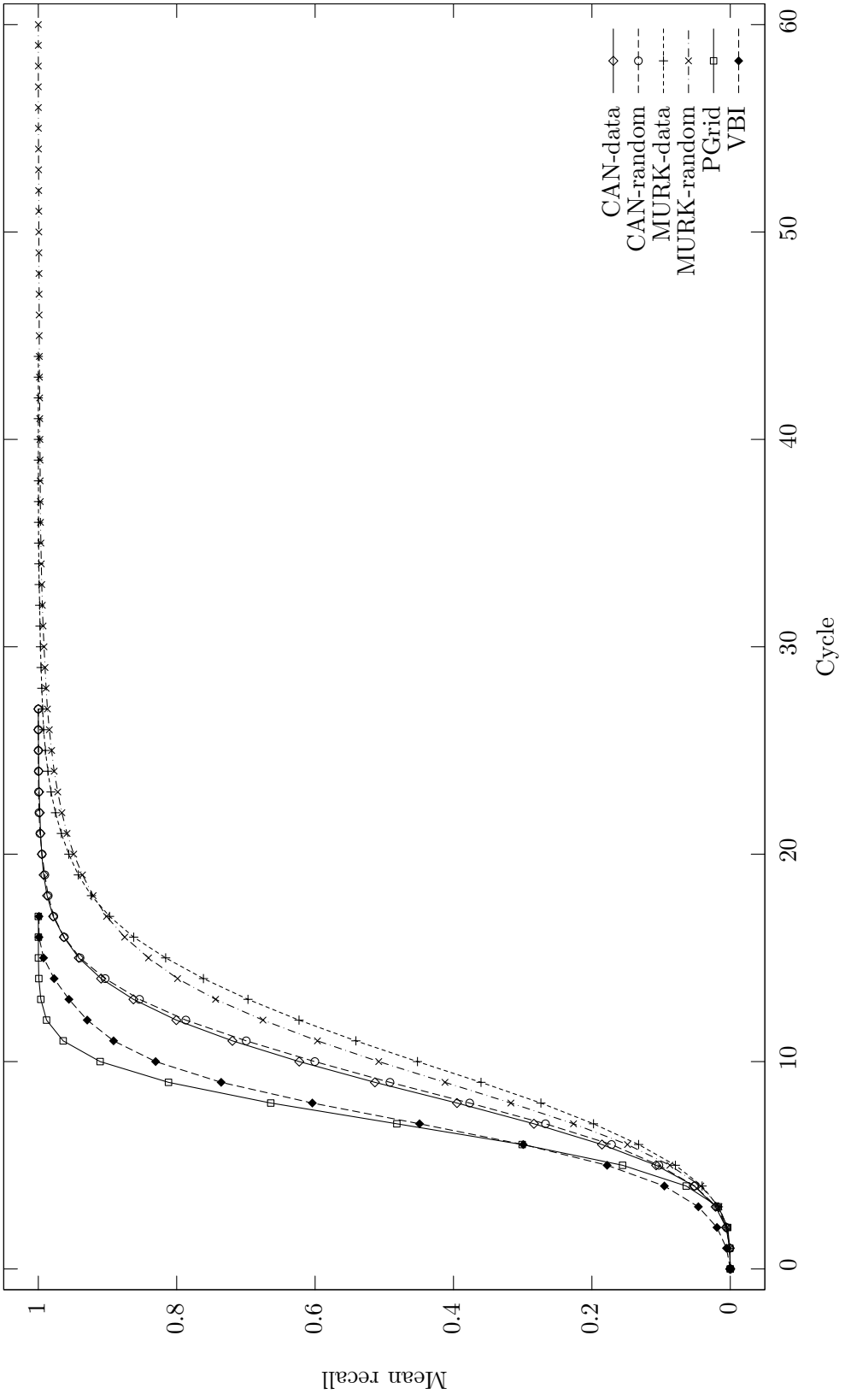


Figure 4.12: Recall graph for the real 2-d dataset with 100000 data items, distributed among 50000 peers and performing 5000 range queries.

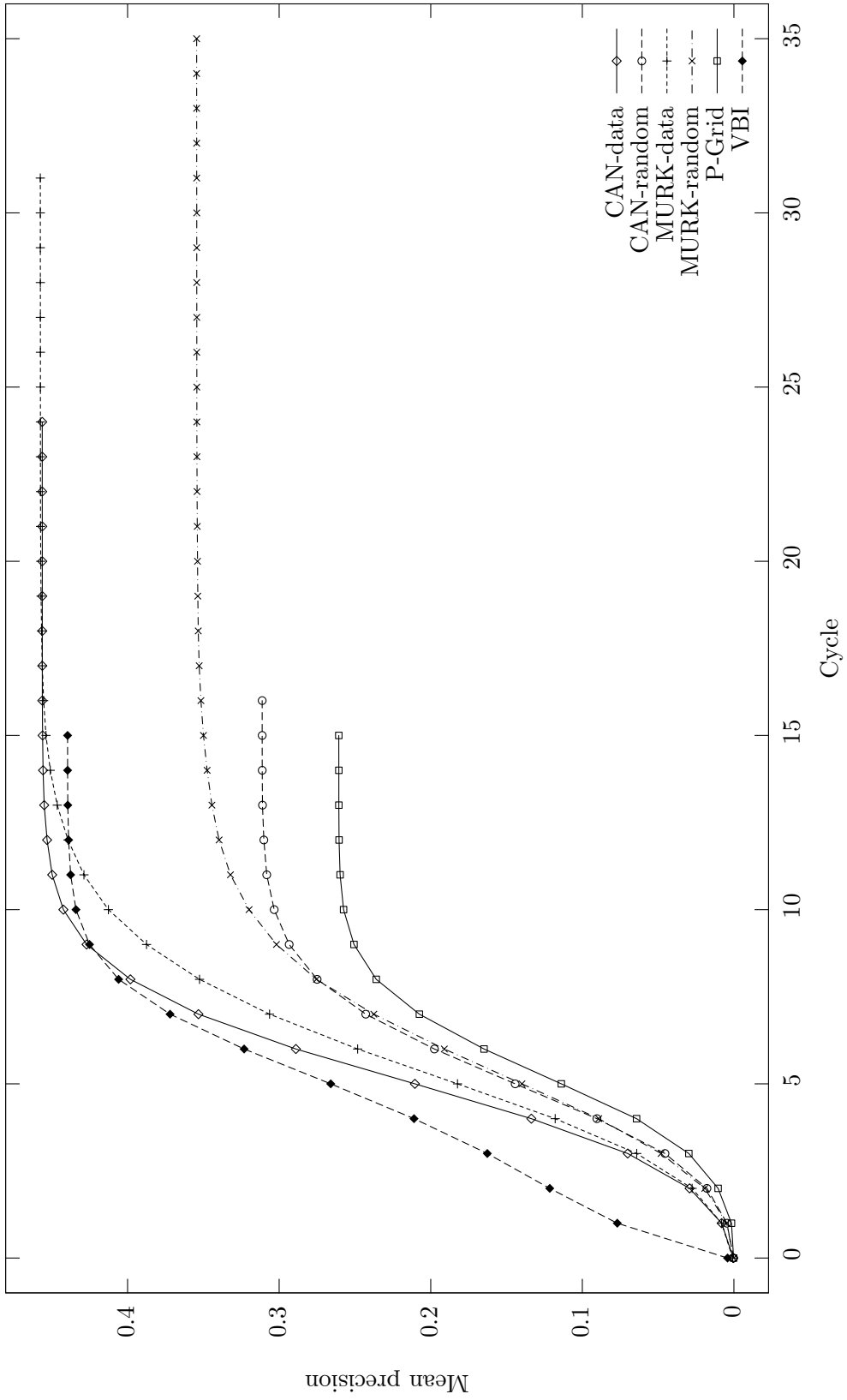


Figure 4.13: Precision graph for the real 2-d dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

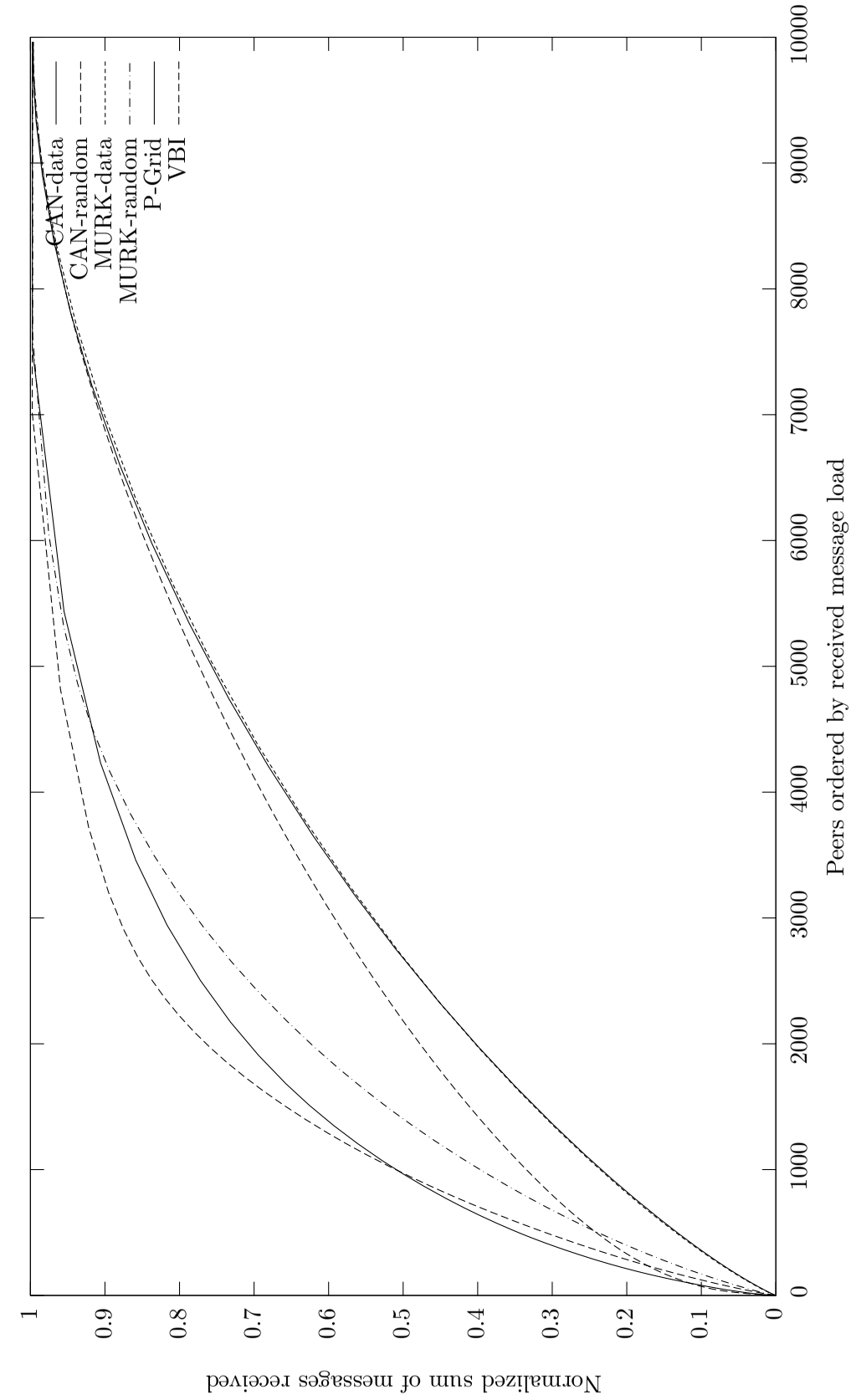


Figure 4.14: Load distribution graph for the real 2-d dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

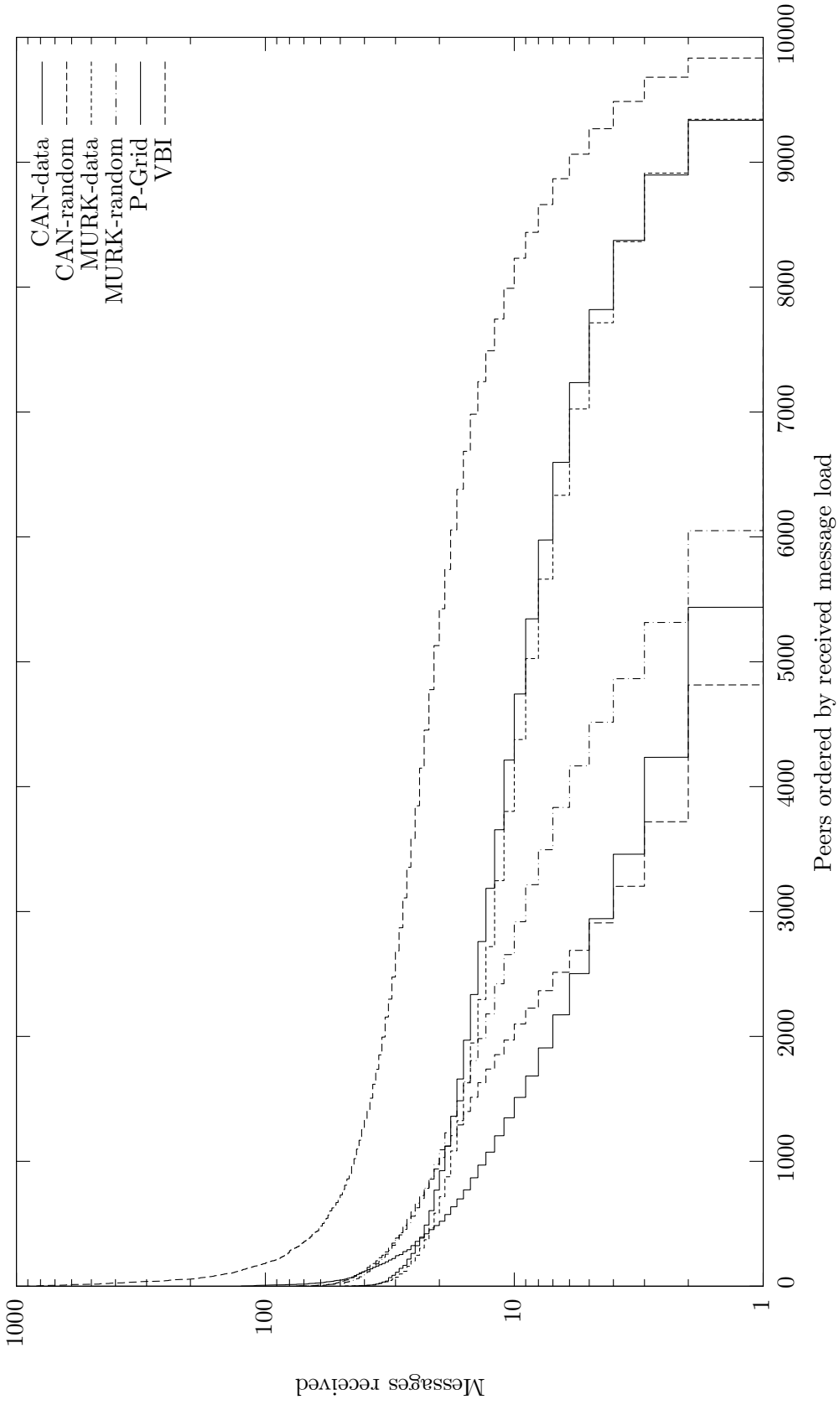


Figure 4.15: Load distribution graph for the real 2-d dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries.

the other. For this purpose, we also plotted the number of messages each peer receives, in descending order, with no summations and normalizations. The result is the logarithmic plot it Figure 4.15, in which we can see that the VBI network is very expensive in communications cost, with one peer having received more than 600 messages.

4.2.5 Effect of realistic network model

Finally we want to compare how would these networks perform under a realistic network model, in which there is limited bandwidth and thus only a partial set of the messages sent in each cycle will be delivered to the recipients in the next one. Although we already know that some networks, like VBI-tree, are more expensive than others, we don't know how this would affect their performance. We therefore run the simulations and allowed only 10 and 5 messages to be delivered per cycle. The results are plotted in Figures 4.16 and 4.17.

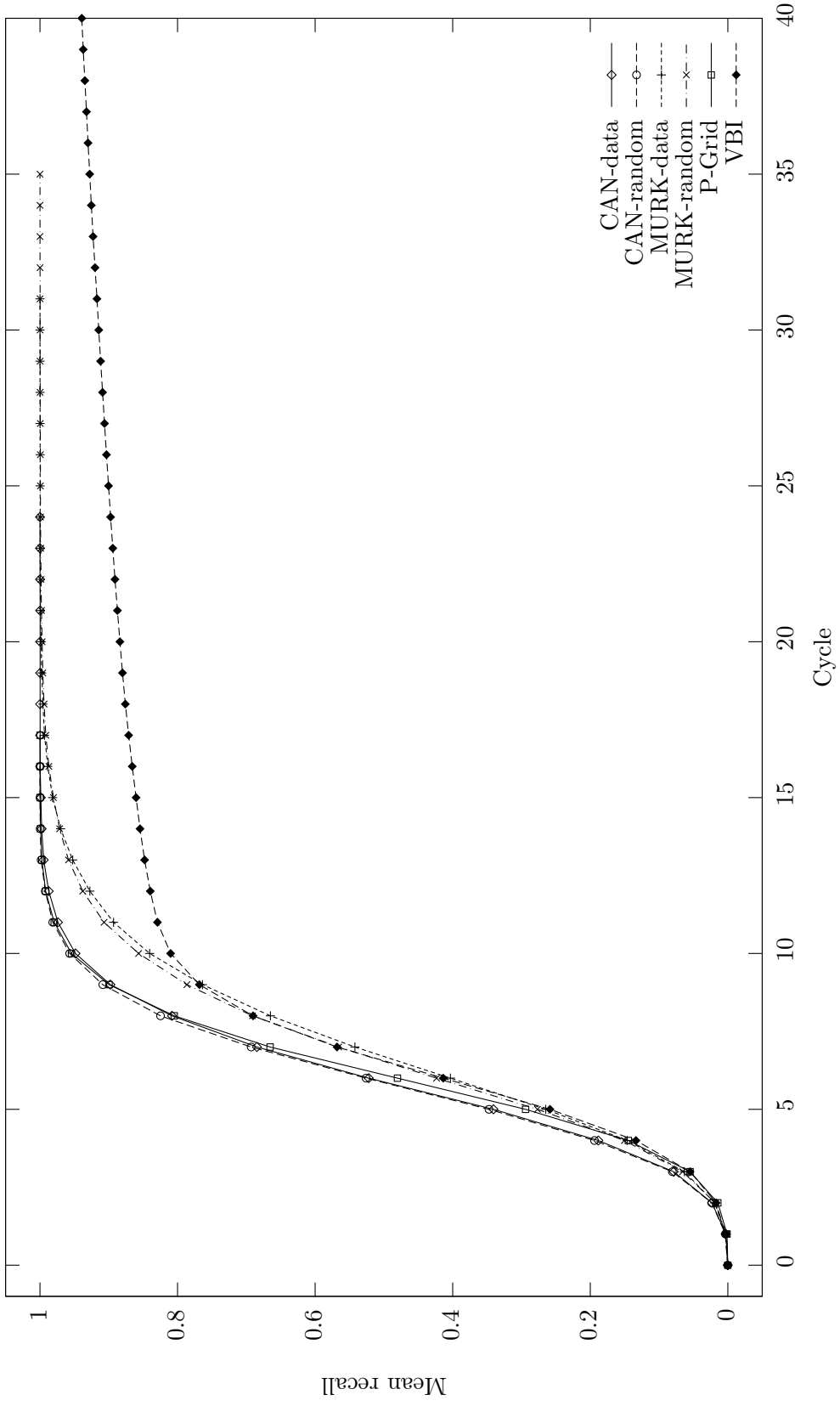


Figure 4.16: Recall graph when allowing only 10 messages per cycle to be received for the real 2-d dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries. VBI-tree continues linearly and reaches 1 around cycle 90.

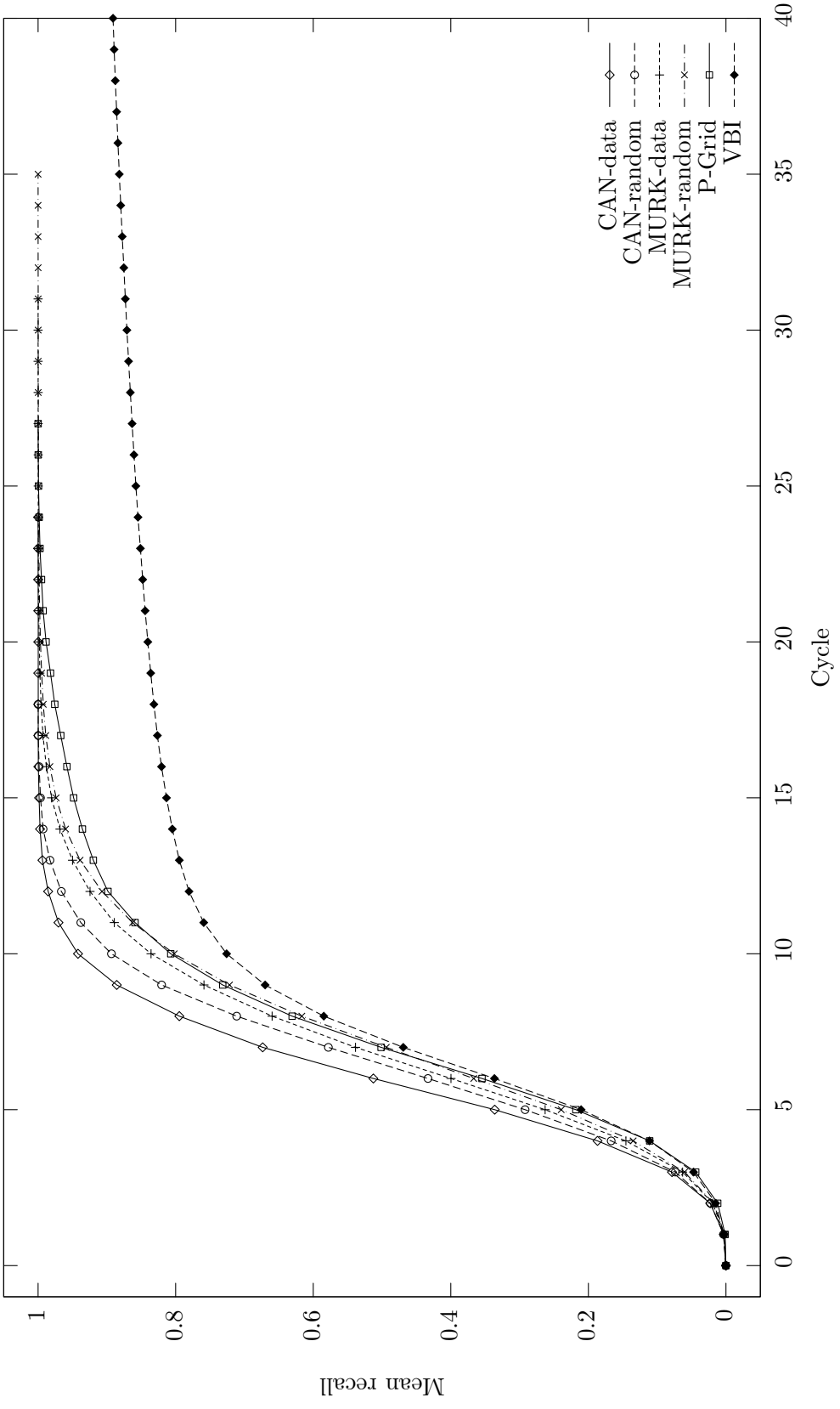


Figure 4.17: Recall graph when allowing only 5 messages per cycle to be received for the real 2-d dataset with 100000 data items, distributed among 10000 peers and performing 5000 range queries. VBI-tree continues linearly and reaches 1 around cycle 170.

Chapter 5

Conclusion

As observed from the previous chapter, no currently proposed solution is best in all aspects. Before making a choice, one should consider all the alternatives and specify the parameters of interest (e.g. fast retrieval, good load balancing, dimensionality or combinations). Some general-purpose observations extracted from our simulation results are the following:

- CAN-based schemes which store data items in the native problem space clearly outperform other approaches in higher dimensions and their effectiveness is increased as the dimensionality increases.
- The way existing peers are selected for allowing new peers to join (e.g. via splitting) is a very important issue. This is shown experimentally with the difference between data and random selection policies in the previous chapter. VBI-tree is currently the only solution which takes care of this issue by design.
- As for the load balancing, CAN-based solutions with data selection policy outperform other approaches. VBI-tree follows closely but is more expensive in communications cost.
- All solutions are scaling well, with tree-based approaches like P-Grid and VBI-tree performing better in bigger networks, like in the case of 50000 peers.
- Query answer size, when modified from a few data items to a thousand, didn't change the results significantly.

All in all, there is no clear winner. VBI-tree is newer and takes some new issues into consideration, is scalable, fast and gives good load balance, but has high communications costs. CAN is fast and is the definite answer for higher dimensionality problems but in the general (random) case behaves poorly in load balancing. MURK technique achieves better load balancing

but reduces the retrieval speed considerably and this worsens as the network size increases. P-Grid appears to be scaling very well, but doesn't balance the load equally.

In the future, we would like to experiment more with higher dimensionality spaces and give more realistic datasets for such problems. Also we would like to do a more in-depth evaluation of the load balancing properties, counting data items and accesses per peer.

A future desire is to adapt the framework described here to work with dynamic networks, but this also requires work in terms of specifying realistic behavior of peers who join, leave and fail. Also it would require a careful assessment of different metrics to perform the comparison.

Finally, an open challenge is adapting the framework to work with dynamic datasets and evaluate the performance of such methods.

Bibliography

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A self-organizing structured P2P system. *SIGMOD Record*, 32(3), Sept. 2003.

[2] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. In G. Weikum, A. C. König, and S. Deßloch, editors, *Proceedings of the 2004 ACM international conference on management of data (SIGMOD)*, pages 347–358. ACM Press, June 13–18, 2004.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 57–66. IEEE Computer Society, 2005.

[5] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multidimensional queries in P2P systems. In S. Amer-Yahia and L. Gravano, editors, *Proceedings of the Seventh International Workshop on the Web and Databases (WebDB)*, pages 19–24, June 17–18, 2004.

[6] T. M. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling. p2psim. Online webpage at http://pdos.csail.mit.edu/p2psim, July 2006.

[7] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM international conference on management of data (SIGMOD)*, pages 47–57. ACM Press, June 18–21, 1984.

[8] Q. He, M. Ammar, G. Riley, H. Raj, and R. Fujimoto. Mapping peer behavior to packet-level details: A framework for packet-level simulation of peer-to-peer systems. In *Proceedings of the 11th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, page 71. IEEE Computer Society, 2003.

[9] H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, page 34. IEEE Computer Society, Apr. 3–8, 2006.

[10] M. Jelasity, G. P. Jesi, A. Montresor, and S. Voulgaris. Peersim. Online webpage at http://peersim.sourceforge.net, July 2006.

[11] I. Kamel and C. Faloutsos. Hilert r-tree: An imporved r-tree using fractals. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 500–509. Morgan Kaufmann, Sept. 12–15, 1994.

[12] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC)*, pages 654–663. ACM Press, 1997.

[13] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - A scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[14] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2), Apr. 2005.

[15] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In P. Druschel, M. F. Kaashoek, and A. I. T. Rowstron, editors, *IPTPS*, pages 53–65, Mar. 7–8, 2002.

[16] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix hash tree, an indexing data structure over distributed hash tables. Technical report, Intel Research Berkeley, Feb. 2004.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 161–172. ACM Press, Aug. 27–31, 2001.

[18] S. Ratnasamy, B. Karp, Y. Li, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage. In *Proceedings of the first ACM international workshop on wireless sensor networks and applications (WSNA)*, pages 78–87. ACM Press, Sept. 28, 2002.

[19] J. Ritter. Why gnutella can't scale. No, really. Online article at http://www.darkridge.com/ jpr5/doc/gnutella.html, Feb. 2001.

[20] 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 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 149–160. ACM Press, Aug. 27–31, 2001.

[21] Y. Theodoridis. The R-tree portal, 2003.

[22] N. S. Ting and R. Deters. 3LS – A peer-to-peer network simulator. In *Proceedings of the third international conference on peer-to-peer computing (P2P)*, page 212. IEEE Computer Society, 2003.

[23] D. Tsoumakos and N. Roussopoulos. A comparison of peer-to-peer search methods. In V. Christophides and J. Freire, editors, *Proceedings of the sixth International Workshop on Web and Databases (WebDB)*, pages 61–66, June 12–13, 2003.

[24] W. Yang and N. Abu-Ghazaleh. GPS: A general peer-to-peer simulator and its use for modeling BitTorrent. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 425–434. IEEE Computer Society, 2005.