Efficient Queries in Peer-to-Peer Systems

Prasanna Ganesan Hector Garcia-Molina
Stanford University
$\{$*prasannag, hector*$\}$*@cs.stanford.edu*

Abstract

A peer-to-peer (P2P) system consists of a large, dynamic set of machines, distributed over a wide-area network, collaboratively storing and managing data in a fully decentralized fashion. The scale, dynamism and network-wide distribution of P2P systems provide new twists to age-old data management problems, requiring the development of novel techniques for data storage and retrieval. In this article, we describe solutions developed by the Stanford P2P group for enabling efficient queries in P2P systems. We focus particularly on how to optimize data storage to support various kinds of queries, including range queries, multi-dimensional queries and similarity search.

1 Introduction

The peer-to-peer (P2P) model of distributed computation first rose to prominence thanks to the popularity of internet-scale file-sharing applications such as Napster and Kazaa. Evolving from those humble beginnings, P2P data management has now emerged as a major new area of computer science research with applications to large-scale networking infrastructure, wide-area monitoring, collaborative data sharing, data archival and, in some cases, even to traditional parallel and distributed databases. In this article, we discuss some of the research undertaken at Stanford in the quest for the holy grail of peer-to-peer data management — a system that offers powerful, database-like declarative queries while being implemented over a peer-to-peer substrate[1].

What is a P2P system? The difference between P2P systems and traditional parallel and distributed systems may be captured by the following three "laws" of P2P:

I. **All peers are born equal.** P2P systems are completely decentralized, with the constituent peers being functionally identical; there are no central servers in the system, and the system load is divided equally across all peers.

II. **The set of peers is dynamic.** Peers may join and leave the system at will, and the system size may itself vary dramatically over time. For example, the Kazaa network contains twice as many peers during the day as it does at night.

III. **Peers are distributed over a wide-area network.** The participant peers may potentially be spread out across the internet. Consequently, communication characteristics can be very heterogeneous; a peer may experience very low-latency communication with some other peers that are geographically close to it, but may also experience higher latencies when communicating with other peers that are further away.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] A holier grail to aspire for would be a system that additionally offers highly reliable storage and strong transactional semantics, two areas of active research that we do not consider in this paper.

Each of the above three laws present significant challenges to efficient query processing in a P2P system. The first law forces decentralized design and the enforcement of load-balance guarantees in the system. The second law requires highly adaptive designs for data storage and query execution to deal with peers frequently joining and leaving the system. The third law requires us to deal with widely distributed peers by localizing communication as much as possible, thereby reducing query latency and optimizing the usage of network bandwidth. The problem of tackling all the above challenges and ensuring efficient queries can be broken down into two partially independent sub-problems: (1) **Storage:** Determining how to partition and store data across peers in an efficient fashion; and, (2) **Retrieval:** Determining how to retrieve the relevant data from across the network when a query is posed.

In the following sections, we will discuss how both these problems may be solved to efficiently support various classes of queries, by combining appropriate storage solutions with overlay networks that enable efficient routing of queries and consequent retrieval of relevant data. Due to space limitations, our description will focus on the storage methods and provide a far more cursory overview of the associated retrieval algorithms and routing networks.

2 Key Lookups and Distributed Hash Tables

Distributed Hash Tables (DHTs) implement a simple dictionary abstraction, storing *(key,value)* pairs and allowing retrieval of all values that correspond to a given key (a selection query with an equality predicate) as well as insertion and removal of *(key, value)* pairs. While a large number of solutions have been proposed for implementing this abstraction [1, 2, 3, 4], all of them are founded on the same basic principles – principles that we illustrate with one example system, Chord [2].

Storage. Chord uses a technique known as consistent hashing [5] to partition data across nodes (peers). In consistent hashing, data keys are hashed into a large (circular) N-bit identifier space, and each node is assigned a unique *ID*, drawn at random from this same N-bit space. At any time, the set of node IDs induces a partitioning of the hash space, and each node stores data falling in the hash bucket ranging from its predecessor's ID to its own. Note that the assignment of hash buckets changes in the obvious fashion when nodes join and leave. The fact that IDs are chosen at random suffices to guarantee that each node manages a hash bucket of roughly the same size (within a logarithmic factor), which provides load balance as required by the first law.

Retrieval. When a lookup query is posed at some node X, it needs to be sent along to the node with the relevant data – the node Y whose ID is the closest successor of the query key's hash. Chord constructs an *overlay network* in which each of the n nodes maintains logical links to $O(\log n)$ neighbors, enabling X to send the query across to Y through a sequence of $O(\log n)$ links. The low network degree ensures that overlay maintenance is cheap as new nodes join or existing nodes leave. The low network diameter ensures that queries can be routed, and data retrieved, efficiently.

It is also necessary to optimize the overlay network *for the underlying physical network*, in accordance with the third law, to ensure that the routing path between any pair of nodes has low latency. While many different solutions exist for this problem, one approach that we have pursued [6] is to exploit the hierarchical structure of the physical network and ensure, among other things, that overlay routing between two nodes within some subtree of the hierarchy never involves any intermediate nodes that are outside that subtree. In addition, we are able to couple such overlay networks with a hierarchical caching scheme to further reduce query latency [6].

3 Enabling Range Queries with Online Load Balancing

Distributed hash tables are effective for enabling efficient queries for individual keys. However, they are problematic if we want to extend the querying power of our system to support range queries. For example, if we wanted to find all tuples (*(key,value)* pairs) with a key that begins with the letter "a", hashing offers no locality

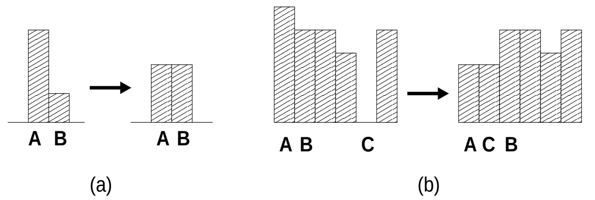

Figure 1: (a) NBRADJUST involving A and B and (b) REORDER involving A and C. The height of a bar represents the load of the corresponding node.

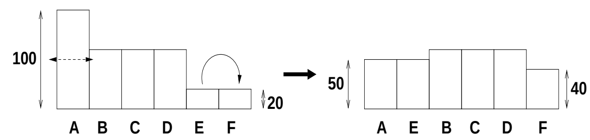

Figure 2: The cost of load balancing using REORDER is 70 while using successive NBRADJUST operations costs 250.

of reference and forces us to probe every single node for answers to the query, which is expensive. Our solution to the above problem is to do away with hashing in the storage solution, and instead distribute data across nodes by range partitioning. (The retrieval solution involving overlay networks and routing remains almost the same as that for DHTs as we discuss later.)

More precisely, data is divided into n range partitions on the basis of a key attribute, with partition boundaries at $R_0 \leq R_1 \leq \ldots \leq R_n$, with node N_i managing the range $[R_{i-1}, R_i)$ for all $0 < i \leq n$. We desire two properties from this range partitioning: (a) the partitions must be well-balanced with each node storing an equal amount of data; (b) it should be possible to dynamically change n, the number of partitions.

Let us first consider property (a) which requires load balance. If the data distribution were fixed ahead of time, we could carefully construct our partitions to satisfy this property. However, as the system evolves over time, data distributions change, and partitions need to be constantly reorganized to maintain load balance, introducing not only a data migration cost but also a management headache: when and how should data be repartitioned? We tackle both these problems with a fully automated *online load balancing* scheme that guarantees load balance at all times by continuously and automatically migrating data in the background using algorithms that have a provably small overhead. In fact, the scheme generalizes naturally to also support property (b) which requires us to also enable nodes to join and leave seamlessly without affecting load-balance properties.

Load Balancing Operations. What operations can be used to perform load balancing? An intuitive operation is as follows: when a node becomes responsible for too much data, it can move a portion of its data to its neighbor and thus attempt to balance out the load. We call such an operation NBRADJUST (illustrated in Figure 1(a) and defined below).

NBRADJUST *A pair of neighboring nodes N_i and N_{i+1} may alter the boundary R_i between their ranges by transferring data from one node to the other.*

We could devise load-balancing algorithms based on just this operation, but such algorithms are provably expensive, requiring massive amounts of data migration [7]. The key to efficient load balancing lies in a second operation, REORDER, illustrated in Figure 1(b) and defined below.

REORDER *A node N_i with an empty range $[R_i, R_i)$ changes its position and splits the range $[R_j, R_{j+1})$ managed by a node N_j: N_j now manages range $[R_j, X)$ while N_i takes over $[X, R_{j+1})$ for some value of X, $R_j \leq X \leq R_{j+1}$. The nodes are re-labeled appropriately.*

Example 1: Consider the scenario shown in Figure 2, where node A has 100 tuples, the next three nodes (B, C, D) have 60 tuples each, while the last two (E, F) have 20 tuples each. The least expensive scheme to improve load balance while preserving key ordering is to transfer all 20 tuples from E to F, and then use REORDER to split the load of A between A and E. The cost of such a scheme is 70 tuple movements; in contrast, a NBRADJUST-based balancing would require 250 tuple movements.

Load-Balancing Algorithm. Our load-balancing algorithm makes judicious use of the above two operations in order to provide a guarantee on load balance while also ensuring that the total amount of data moved across nodes in the process is very small. The key idea is to create an infinite geometric sequence of *load thresholds*, e.g., all powers of two, and let each node attempt load balancing every time its own load crosses a threshold. We do not discuss our algorithm's details here, and instead summarize its performance with the following theorem.

Theorem 1: Using a geometric threshold sequence with ratio δ, for any $\delta \geq (\sqrt{5}+1)/2$, we may obtain the following properties: (1) All node loads are within a factor δ^3 of each other; and, (2) The (amortized) number of tuples moved per tuple insertion or deletion is constant.

The above theorem states that it is possible to maintain load balance within any constant factor larger than the cube of the golden ratio ($\simeq 4.24$). Moreover, the total cost of maintaining load balance is linear in the number of updates, meaning that each insertion or deletion induces a constant number of tuples to migrate on average. We may interpret this cost as a constant-factor slowdown of updates in the system, since each tuple migration may be viewed as an extra deletion followed by an insertion. In fact, even experiments with highly adversarial workloads designed to stress the system forced only 2 tuple migrations per insert/delete, which could be viewed as a slowdown of updates by a factor of 5 [7]. In reality, the slowdown will be much smaller, both because the tuple migrations are batched in much larger chunks and because the overhead is an order of magnitude smaller with realistic, non-adversarial workloads.

From Load Balancing to Range Queries. So far, we have described a method for maintaining range partitions over a set of nodes while ensuring load balance. In fact, this can be useful in the context of plain-old parallel databases [7], rather than just P2P systems. The next step involves combining this range-partitioning scheme with a P2P routing network for efficient retrieval. We use overlay networks known as *skip graphs* [10, 11], with properties almost identical to that of standard DHT overlay networks such as Chord, in order to obtain efficient routing properties. In consequence, we are left with a true generalization of the Distributed Hash Table: a range-partitioned P2P system that matches the asymptotic efficiency of DHTs for all the standard operations – insertion and deletion of tuples, arrival and departure of nodes, and simple lookup queries – while being able to answer range queries far more efficiently, in an asymptotically optimal fashion.

4 Multi-dimensional Range Queries

We next discuss how a P2P system can support *multi-dimensional* range queries, i.e., conjunctive queries containing range predicates on two or more attributes of a relation. One simple solution is to perform range-partitioning on one of the attributes, use the solution from the previous section to evaluate the predicate on that attribute, and evaluate the remaining predicates as a filter. However, this solution is likely to prove very expensive in the context of motivating applications such as P2P photo-sharing and massively multi-player games, where data is inherently spatial and tagged with multiple "coordinates" [8]. Applying a selection predicate on just one coordinate would still retrieve far too much irrelevant data.

What is needed, therefore, is a data storage and retrieval mechanism specialized for multi-dimensional queries. In the world of databases, there are many solutions available for multi-dimensional queries, including space-filling curves, quadtrees and kd-trees. We adapt these techniques to develop efficient storage methods in the P2P context, and combine them with new overlay networks and retrieval algorithms to architect our overall solution for supporting multi-dimensional queries. Here, we illustrate the storage methods underlying our two solutions: SCRAP and MURK. Both solutions are designed to ensure data *locality* – minimizing the number of nodes that need to process the query – since the dominant cost of answering queries in a P2P system is often simply the cost of transmitting and initiating the query at each node that needs to process it. For multi-dimensional range queries, data locality can be obtained by ensuring that tuples nearby in the data space are stored at the same node for the most part. Our two solutions attempt to achieve locality in two different ways.

SCRAP: Space-filling Curves with Range Partitioning. Our first approach to supporting multi-dimensional queries, *SCRAP*, uses a two-step solution to partition the data: (a) Data is first mapped down into a single dimension using a space-filling curve; (b) this single-dimensional data is then range-partitioned across a dynamic set of participant nodes.

For the first step, we may map multi-dimensional data down to a single dimension using a space-filling curve such as *z-ordering* [12] or the *Hilbert* curve (e.g., [13]). For example, say we have two-dimensional data that consists of two 4-bit attributes, X and Y. A two-dimensional data point $\langle x, y \rangle$ can be reduced to a single 8-bit value z, by inter-leaving the bits in x and y. Thus, the two-dimensional point $\langle 0100, 0101 \rangle$ would be mapped to the single-dimensional value 00110001. (This mapping corresponds to the use of z-ordering as the space-filling curve.) Note that this mapping is *bijective*, i.e., every two-dimensional point maps to a unique single-dimensional value and vice versa. In the second step, once data has been reduced to one dimension using the space-filling curve, we simply use range partitioning to split the data across the available nodes, just as in our solution of Section 3.

Locality is achieved since the space-filling curve attempts to ensure that nearby data points in the multi-dimensional space are also adjacent in the single dimension. However, as the number of dimensions increases, locality becomes worse since space-filling curves are afflicted by the curse of dimensionality. Load balance across nodes is ensured by using online load balancing of range partitions, as discussed earlier.

MURK: Multi-dimensional Rectangulation with kd-trees. Our second approach, MURK, partitions the data *in situ* in the high-dimensional space, breaking up the data space into "rectangles" (hypercuboids in high dimensions), with each node managing one rectangle. One way to achieve this partitioning is to use kd-trees, in which each leaf corresponds to a rectangle being stored by a node.

We illustrate this partitioning in Figure 3. Imagine there is initially one node in the system that manages the entire 2-d space, corresponding to a single-node kd-tree. When a second node arrives, the space is split along the first dimension into two parts of equal load, with one node managing each part; this corresponds to splitting the root node of the kd-tree to create two children. As more nodes arrive, each node splits the partition managed by an existing node, i.e., an existing leaf in the kd-tree. The dimensions are used cyclically in splitting, to ensure that locality is preserved in all dimensions.

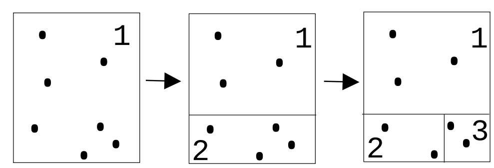

Figure 3: Evolution of partitions as nodes join the network. Each partition corresponds to a network node.

When a node leaves, its space needs to be taken over by existing nodes. The simple case is when the node's sibling in the tree is also a leaf, e.g., node 3 in the figure; in this case, the sibling node 2 takes over 3's space. The more complex case arises when the node's sibling is an internal node, e.g., node 1 in the figure. When node 1 leaves, a lowest-level leaf node in its sibling subtree, say node 2, hands over data to its sibling node 3, and takes over the position of node 1.

The MURK solution offers good data locality, since it partitions the data space into exactly as many rectangles as there are nodes. As the number of dimensions increase beyond two, MURK offers much better data locality than SCRAP and outperforms SCRAP in our experimental comparisons [8]. However, load balancing across partitions is trickier with MURK and, in case the data distribution itself is dynamic, we cannot provide any theoretical guarantees on the cost of load balancing. In contrast, we always obtain strong load balance guarantees with SCRAP. An interesting open question is whether it is possible to obtain both load balance and locality with some data-partitioning scheme.

5 High-dimensional Similarity Search

We next consider how to support similarity-search queries in P2P systems to find high-dimensional objects most similar to a given object. For example, objects could be text documents, with the query trying to find documents in the system that are most similar to a given document. In an end-user P2P system, objects could even be peers themselves; users may wish to find what other users are similar to themselves on the basis of the content they share. In either case, we would like an efficient method for finding similar objects in a P2P system. Similarity search is a non-trivial problem even in a centralized context and, in fact, our solution turns out to be as applicable in a centralized environment as it is in the P2P context.

Our solution [16] is based on the influential locality-sensitive hashing (LSH) scheme of Indyk and Motwani [15]. The basic idea behind (centralized) LSH is the following: objects are hashed using special locality-sensitive hash functions (designed for the particular similarity metric of interest), such that "similar" objects are much more likely to collide (hash to the same bucket) than dissimilar objects[2]. At query time, the query is also hashed to a bucket; objects which hash to the same bucket are retrieved as "candidate" answers, their actual similarity to the query is computed, and the most similar among them are returned as answers to the query. The results returned by LSH are guaranteed to have a similarity within a small factor ϵ (for any fixed $\epsilon > 0$) of the optimal.

While the LSH scheme provides many nice properties, and could be implemented in a distributed fashion with DHTs, it suffers from two problems: (a) various parameters of the hash function need to be carefully tuned for the particular data domain of interest, and may have to be changed periodically if data characteristics change; and (b) the scheme requires the data distribution to be "uniform" in order to work well while being space-efficient.

We develop a new solution called LSH Forest, which uses locality-sensitive hashing while avoiding both the above problems, eliminating the need for tuning and offering improved accuracy with lower space utilization. The core idea is to construct a tree structure with the objects forming the leaf nodes in the tree; the similarity between two objects is approximately captured by how close they are in the tree. In the P2P context, our implementation of LSH Forest once again exploits range partitioning to efficiently partition the leaves of the tree across the participant nodes. Query execution requires various primitives for navigating the tree, which can be implemented on top of standard P2P overlay networks [16].

Interestingly, in the special case where the objects being searched are the peers themselves, we can embed the entire data structure in the *overlay network* itself! The very fact of which peers connect to which others contains information about which peers are similar to which others, eliminating the need for any explicit data structures for similarity search. We refer the interested reader to [16] for details.

6 Related Work

Early P2P systems such as Gnutella were built as unstructured networks and supported only keyword searches, with no guarantees of completeness on the search results. Distributed Hash Tables [2, 1, 3, 4] established the notion of structured overlays and showed how to support simple lookup queries in a clean fashion. Recent research has considered how to support more powerful queries by using alternatives to hashing.

Concurrent with our work on range partitioning, Karger and Ruhl also proposed the use of range partitions for supporting range queries. Their solution was based on a slightly different online load-balancing scheme with weaker guarantees on load balance, albeit under a different analytical model. Reference [7] discusses the similarities and differences between our solution and that of [9] in greater detail. Ratnasamy et al. [14] propose using tries for enabling range queries. Such an approach is arguably simpler and can be implemented on top of standard DHTs; on the other hand, it introduces data-dependent query costs and does not support the powerful forms of load balancing possible with range partitions [7].

The PIER project [17] aims to build a P2P system with support for arbitrary SQL queries. Research in the project so far has relied on hashed data storage, and focused on building a SQL query engine with efficient operator implementations. It is conceivable that the storage solutions described herein could be incorporated

[2]Many similarity measures have corresponding LSH functions that provide this property. Examples of such similarity measures include Jaccard coefficient, the Hamming metric and the l_1 and l_2 norms [15].

into PIER to expand the space of design possibilities available for data storage. The Astrolabe [18] system also provides a SQL-like interface on top of a P2P system, but is specialized to handle only aggregate queries on a hierarchical structure.

7 Conclusions

We have described the key challenges involved in supporting efficient queries over P2P systems. We presented different storage solutions that help extend the querying power of P2P systems to support range queries, multi-dimensional queries and similarity search. In conjunction with suitably designed overlay networks, these solutions enable the efficient execution of powerful queries over a P2P system. Many questions remain, however. For example, we would like to integrate these techniques into a P2P database system that can support arbitrary SQL queries, optimize these queries, and efficiently execute them in a distributed fashion across the network. Another big open issue is to define clear semantics for the results of queries in a P2P system; when computing query results takes up a non-trivial amount of time, it is not always clear what the results of a query actually mean, since peers may be joining and leaving while the query is in a partial state of execution. Given the number of open questions remaining unanswered, it appears safe to suggest that the holy grail of P2P data management is still at least a few years away from being realized.

References

[1] S. Ratnasamy, P. Francis, M. Handley and R. M. Karp. A Scalable Content Addressable Network. In *Proc. SIGCOMM*, 2001.

[2] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. SIGCOMM*,2001.

[3] A.I.T. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In Proc. Middleware, 2001.

[4] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph and J.D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications 22(1)*, 2004.

[5] D.R. Karger, E. Lehman, F.T. Leighton, M.S. Levine, D. Lewin and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. STOC*, 1997.

[6] P. Ganesan, K. Gummadi and H. Garcia-Molina. Canon in G Major: Designing DHTs with Hierarchical Structure. In *Proc. 24th Intl. Conf. on Distributed Computing Systems (ICDCS)*, 2004.

[7] P. Ganesan, M. Bawa and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. In *Proc. VLDB*, 2004.

[8] P. Ganesan, B. Yang and H. Garcia-Molina. One Torus to Rule Them All: Multi-dimensional Queries in P2P Systems. In *Proc. WebDB*, 2004.

[9] D.R. Karger and M. Ruhl. Simple Efficient Load-Balancing Algorithms for Peer-to-Peer Systems. In *Proc. SPAA*, 2004.

[10] J. Aspnes and G. Shah. Skip Graphs. In *Proc. SODA*, 2003.

[11] N.J.A. Harvey, M. Jones, S. Saroiu, M. Theimer and A. Wolman. Skipnet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. USITS*, 2003.

[12] J. Orenstein and T. Merrett. A Class of Data Structures for Associative Searching. In *Proc. PODS*, 1984.

[13] H. Jagadish. Linear Clustering of Objects with Multiple Attributes. In *Proc. SIGMOD*, 1990.

[14] S.Ratnasamy, J.M.Hellerstein and S.Shenker. Range Queries over DHTs. *Intel Technical Report IRB-TR-03-009*, 2003.

[15] P. Indyk and R. Motwani. Approximate Nearest Neighbor - Towards Removing the Curse of Dimensionality. In *Proc. STOC*, 1998.

[16] M. Bawa, T. Condie and P. Ganesan. LSH Forest: Self-Tuning Indexes for Similarity Search. To appear in *Proc. WWW*, 2005.

[17] R. Huebsch, B. Chun, J.M. Hellerstein, B.T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, A.R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. In *Proc. CIDR*, 2005.

[18] R. Van Renesse, K.P. Birman and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management and data mining. *ACM Transactions on Computer Systems (TOCS), 21(2)*, 2003.