Using Distributed M-Trees for Answering K-Nearest Neighbor Queries

Brent Bryan, Andrew W. Moore, Andrew Snyder, Jeff Schneider

April 2007
CMU-ML-07-106

Using Distributed M-Trees for Answering K-Nearest Neighbor Queries

Brent Bryan[1] **Andrew W. Moore**[2]
Andrew Snyder[2] **Jeff Schneider**[3]

April 2007
CMU-ML-07-106

Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

The proliferation of large dynamic data sets allows for unprecedented learning opportunities. However, collections of data are only a valuable resource if methods exist for quickly finding and extracting relevant information. Hence, it is desirable to store this data in an indexing structure, such as a tree. Traditional tree-based data structures typically store as much of the tree as possible in RAM, as writing nodes to the disk results in a significant performance hit. In order to eliminate the disk-based bottleneck, several techniques have been proposed to store large indexing trees over a series of machines. As the tree is no longer stored on a single machine, a trade off between tree balance and insertion cost (due to machine communication) arises. In this work, we present a general framework for maintaining a tree structure over parallel resources in a dynamic environment. We show that the technique results in a dynamic tree structure with *k*-nn query rates similar to those of the optimal tree for uniform data sets and significantly better when the data is either skewed or dynamic. In particular, the algorithm is ideally suited for querying server logs.

[1] Machine Learning Department, Carnegie Mellon University, Pittsburgh PA, USA
[2] Google Pittsburgh, Pittsburgh PA, USA
[3] Robotics Institute, Carnegie Mellon University, Pittsburgh PA, USA

Keywords: M-trees, parallelization, k-Nearest Neighbor

1 Introduction

With the creation and publication of many large data sets, it is imperative to have a mechanism to quickly locate interesting objects within these sets. This work is motivated by the task of querying server logs to find the k records which are most similar to a known fraudulent record. Server logs present two unique challenges to standard k-nearest neighbor (k-nn) techniques. First, server logs tend to be large. Our database contains over ten million records which collectively cannot fit into the RAM of a single machine. Secondly, the logs tend to be quite dynamic with hundreds of changes per second.

As each record in our log, $\mathcal{D}$, consists of dozens of both real-valued and symbolic (e.g. string) attributes, we cannot directly embed each record into a Hilbert space. Instead, we define the similarity between two records in terms of a distance function, d, which defines a metric space: for any three elements $x, y, z \in \mathcal{D}$:

1. $d(x, y) \geq 0$, and $d(x, y) = 0 \implies x = y$,
2. $d(x, y) = d(y, x)$, and
3. $d(x, y) + d(y, z) \geq d(x, z)$.

Generalizations of vector spaces, metric spaces allow us create rapidly locate data by employing pruning techniques to eliminate large groups of records which are provable not within the k-nearest neighbor result of a query. Much work has been done in the past 20 years to efficiently process k-nn and range queries in metric spaces in both single and multi-processor environments. We briefly summarize these efforts, and refer interested readers to [32].

We begin by discussing single processor methods. As with spatial data, metric data can be more efficiently indexed and searched when it is stored in a tree structure. However, since metric points do not have an absolute position in space, it is not clear how to group the data into branches and leaves. [30] suggested two methods for building a single static tree top down. The first idea, known as vantage point trees (VP-Trees) is to divide the data using spherical cuts [31]. For a specific point p, a distance, r, from p is computed which evenly divides the data set into two sets (those with distances to p less than r and those with distances to p greater than r). This partition defines the two branches of a binary tree. VP-trees can be expanded to encompass the use of multiple vantage points (MVP-trees) [5], as well as multiple cuts per vantage point.

The second method suggested by [30] is that of generalized hyperplanes (GH). Here, two seed points are selected from the data set and the remaining points are partitioned by associating each point in the data set with the nearer of the two seed points. This idea has been extended to use multiple seeds [6], as well as multiple servers by way of dynamic hashing techniques [2]. Additionally, [29] suggests that a better partitioning of the points can be accomplished (at roughly the same cost) by building a minimum spanning tree of the points under consideration, and then breaking the tree along the largest edge that roughly divides the points.

One major draw back of both VP-trees and GH-trees is that they are static structures. M-trees [8] overcome this restriction using a mechanism similar to R-trees [12] to reorganize portions of the tree during node overflows or deletions. Moreover, [7] developed methods for bulk-loading M-trees, similar to those for R-trees [25, 16, 19, 11].

Much work has been done to parallelize database structures in recognition that modern databases tend to contain billions of objects and computational power is fairly cheap. Methods for parallelizing both R-trees and M-trees including splitting the tree structure over multiple disks [15, 33], using multiple CPUs to compute distance functions for a single tree [33], pre-assigning each node to a processor for computation (just as each node is assigned to a disk) [1], as well as actually splitting the tree structure over multiple machines [17, 26]. Recent parallelizations involve peer-to-peer networks to eliminate issues associated with the primary index residing on a single machine, such as I/O bottlenecks and single points of failure [28, 24]. However, all current implementations suffer from extensive I/O requirements either from disk, or over the network.

In this paper, we present a simple, yet powerful algorithm for parallelizing search structures that minimizes communication costs, eliminates single points of failure and ensures that the indexing tree remains balanced with high probability. We show that this parallelized tree structure exhibits linear speedups for both build and query times as more machines are added. Moreover, query return delay is optimally minimized, while query throughput is within a constant factor of that achievable by the optimal balanced tree. For dynamic and/or skewed data sets, the proposed algorithm significantly out performs current state-of-the-art querying techniques.

2 Algorithm

Naively, one can perform a k-nn query on a data set, $\mathcal{D}$ of size m, in $O(m)$ by iterating over all m items in $\mathcal{D}$ and adding them to a heap of size k[1]. However, using information about the proximity of objects, tree based structures can partition the m data points into hierarchical non-overlapping sub-regions. If the number of points is divided roughly equally among the c non-overlapping children of a tree node, the k-nn query can be performed in $O(\log_c(m))$ expected time by pruning distance children [3]. Thus, it is desirable to construct data structures that partition the space into disjoint sub-regions that each contain an equal number of points.

However, such a partitioning scheme is not always possible. While points in a vector space can be easily partitioned, extended objects — those represented by shapes — cannot in general be partitioned into non-overlapping sub-regions. Moreover, for points in a generic metric space such a partition may not necessarily even exist (e.g. strings using string-edit distance). For these reasons, data structures have been developed which relax the non-overlapping restriction imposed above (e.g. [12, 30]). In practice, these structures perform better than a simple linear search. Additionally, artificially increasing the overlap among children can actually be beneficial for some search tasks [23, 21].

We expand on the idea of allowing overlapping children. Specifically, consider the case where the storage tree allows all c children of the root to overlap completely. In this case, an inserted object has no preference as to which of the c root children it will be inserted. If the child into which the object is inserted is chosen uniformly at random, then the sub-tree rooted by each of the c children will contain approximately the same number of objects with high probability. Moreover,

[1]Define the priority of an object in the heap to be the distance of the object to the query object, and structure the heap so the item with the largest priority is at the root. During iteration of the data set, $\mathcal{D}$, add the current object to the heap or replace the root if the heap is not full or if the current object is closer than the root, respectively.

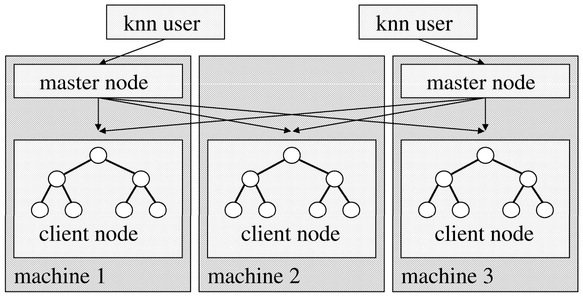

Figure 1: Pictorial design of the algorithm. Note, multiple users can connect to the k-nearest neighbor query clients using different servers.

each of the c sub-trees will be independent of the other $c - 1$ sub-trees. Since the root of the initial tree is equal to the roots of each of the c children (since all children overlap maximally), we can discard the root. Thus, we are left with a set of nearly identical, independent balanced trees. This situation is trivially parallelizable.

To be clear, allowing the children of the root node to maximally overlap is not free. Each of the c new trees cover the entire metric space, and as such k-nn queries cannot be answered without searching all c trees. Thus, we are replacing a $O(\log(m))$ solution with one that is $O(c \log(m/c))$. However, this strategy allows us to store larger trees without resorting to disk accesses. Moreover, the search time can be evenly spread over c processors. This significantly reduces the time for a single query, without drastically affecting the overall query throughput.

Concretely, our proposal is to take the data set $\mathcal{D}$ and to split it uniformly over c client machines. Each client will be responsible for storing and retrieving only those items contained within it. Even distribution of objects among the clients will be facilitated by using a master node to hash each object to one of the clients; insertions and deletions require determining which client should contain the object via the hashing function, and then forwarding the request to that client. Assuming a balanced hash function, objects will be uniformly distributed, and with high probability each client will have roughly the same number of objects, no matter what sequence of insertion and deletions is encountered. Performing a k-nn search requires the master node to forward the k-nn request to all c clients and then aggregating the results. Upon receiving the k-nearest neighbors for each client, the master node merges the c sets of results into a single sorted list and returns the k smallest objects.

The master node is essentially stateless, storing only a deterministic hash function and pointers to each of the c clients. Thus we can easily replicate the master node, ensuring that there is no bottleneck or single point of failure. Moreover, the master node is a lightweight component in terms of both memory and computational resources, and hence can placed on one (or more) of the client nodes with negligible performance degradation. Figure 1 shows an overview of our framework.

2.1 Partial Queries

When the number of clients becomes even moderately large, the requesting the k-nearest neighbors from each client will be excessively conservative. Since the data points from $\mathcal{D}$ are distributed uniformly over our c clients, we expect each client to contain k/c of the true k-nearest neighbors. Thus, our algorithm can be sped up by forwarding partial queries (h-nn queries, for some $h < k$) to each of the c clients. If we choose h too small, during the merge phase of the algorithm we may exhaust the h-nn result of one of the clients. When this happens, we can no longer prove that the result of the merge process is exact.

Imagine picking elements for the final k-nn result one at a time (from smallest to largest) from each of the c h-nn result sets. In order to prove that a specific element from one of the h-nn lists is a member of the true k-nn result, we need to be sure that there is not a closer neighbor on any of the other $c-1$ clients (that has not already been added to the k-nn result). This implies that we can guarantee that the k-nn result is complete if every time we select the next element to add to the k-nn result (the smallest remaining element in all c h-nn result sets), there is at least one element in every h-nn result set that is not in the k-nn result.

We say that a query is "complete" if we can prove that we have found the k nearest neighbors. If an h-nn query fails to result in a complete k-nn solution, we reforward a larger query to each of the clients. In practice if a partial query fails, we forward a k-nn query to each client.

For a particular probability of success, the optimal value of h can be determined using the solution to the classical occupancy problem [10]. Specifically, the probability that any given client, c_i, contains h of the k nearest neighbors when the data is uniformly spread over the c clients is

$$\binom{k}{h}\left(\frac{1}{c}\right)^{h}\left(1-\frac{1}{c}\right)^{k-h}.$$

Thus the probability that none of the c clients contain h or more of the k nearest neighbors when the data is uniformly distributed among the c clients is

$$\left[1-\sum_{\ell=h}^{k}\binom{k}{\ell}\frac{1}{c^{\ell}}\left(1-\frac{1}{c}\right)^{k-\ell}\right]^{c}. \qquad (1)$$

For a given number of clients c, and desired number of neighbors k, we can use Equation 1 to determine the smallest value of h which will produce a complete k-nn result with high probability. When $k = 100$ and $c \sim 1-10$, a value of $h = 2k/c$ results in a complete solution roughly 99% of the time. However, when $k \simeq c = 100$, then $h \geq 5k/c$ is required for the algorithm to return a complete solution for more than 70% of queries.

3 Experiments

In the previous section, we made no assumptions about the type of trees stored on each client. Indeed, any data structure can be easily implemented in this framework. In this paper, we use M-trees [8] on the clients, as they are general enough to handle both metric and spatial data. Moreover, M-trees easily handle insertion and deletions, which are required for our applications.

Unfortunately, we are unable to discuss the results of our algorithm on our server log problem. Instead, we present results from the following synthetic and real-world data sets:

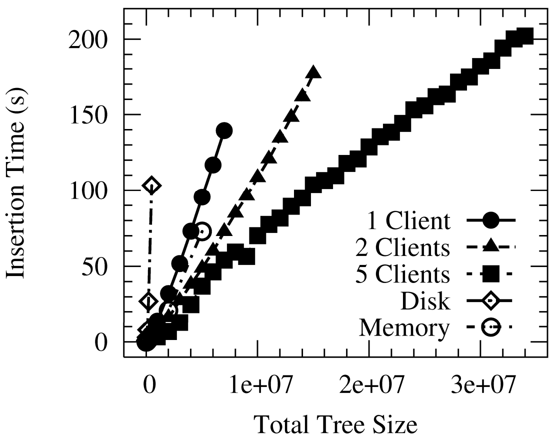

Figure 2: Insertion time of our algorithm on the SDSS data set for various number of clients. Open symbols correspond to standard M-trees stored in RAM or on disk.

SDSS: $\sim$ 2.6 million astronomical objects observed by the Sloan Digital Sky Survey DR5 [14]. We look at the colors of the objects in 5 filters (u,g,r,i,z), as similar objects (e.g stars, etc), show distinct color profiles.

Sierpinski: 100,000 points randomly generated from a 3D Sierpinski pyramid, where the x, y, and z axes range between zero and one.

Pacific NW: 2.1 million 2D road segments in the states of Washington, Oregon, Idaho, Montana and Wyoming taken from the U.S. Census Tiger database[2]. Data was normalized between 0 and 1 in a manner that approximately maintains the correct physical scale between the x and y axes (latitude & longitude).

Our data sets span a range of skewness. The Pacific NW data set is distributed roughly uniformly over the feature space, while the Sierpinski pyramid embedded in 3D has a fractal dimension of 2 leading to moderate skew. The SDSS data set is highly skewed, forming tight clumps (e.g. stars and galaxies) with extended tails. In fact, many real world data sets are highly skewed. [22] show that the distances of feature vectors extracted from images on the web follow a power law distribution, while [18] demonstrates a similar results for paper citation, US patent application, Internet routing map and co-authorship graph data sets.

For each data set, we performed both build and query operations. Data was randomly sampled from the data sets, using replacement when necessary. Queries were generated at random from among points in the data sets. The key measure of performance was wall clock run time, including all disk and network access costs. Results for build and search times using random samples from the SDSS data set are shown in Figures 2 and 3 for differing numbers of clients (filled symbols); search results include the time necessary for the master client to merge the k-nearest neighbor results for each of the c clients. Open symbols in Figures 2 and 3 correspond to standard M-trees where the tree nodes were either stored in RAM or on disk.

[2]See: `http://www.census.gov/geo/www/tiger/index.html`

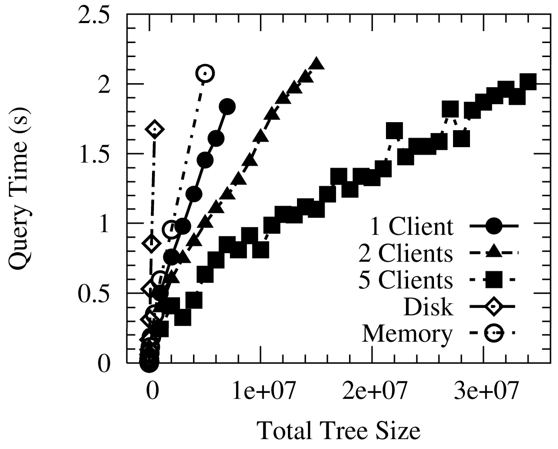

Figure 3: Query time for 100-nn for our algorithm on the SDSS data set for various number of clients. Open symbols correspond to standard M-trees stored in RAM or on disk.

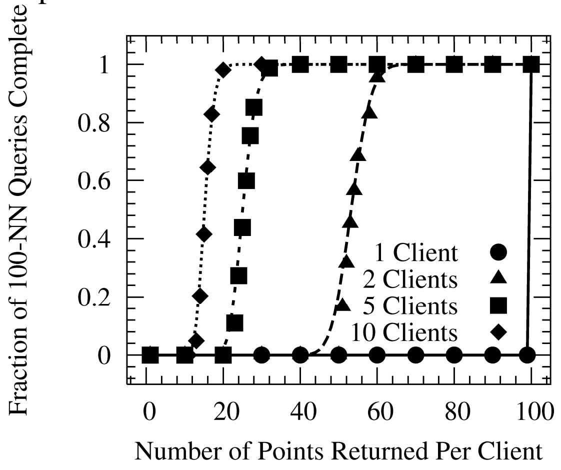

Figure 4: Fraction of queries complete for our algorithm on the SDSS data set, as a function of number of neighbors returned by each client. Lines indicate theoretical predictions.

As expected, the disk based implementation is about twenty times slower for all operations than the RAM based trees. The single processor RAM based M-tree was faster than our framework using a single client for both building the tree and querying it, due to the additional overhead of transferring data over the network (between the master and client nodes) in our algorithm. However, using two or more clients in our algorithm outperforms the single processor M-tree by wide margins. Results for the other data sets were similar.

As mentioned in §2.1, it is often unnecessary to request all k-nearest neighbors from each client. In Figure 4, we depict the experimental and theoretical probability (using Equation 1) that a 100-nn query will be complete as a function of the number of neighbors returned by each client for the SDSS data set. The figures shows that there is a small window over which the queries go from being completed with zero probability to being completed with probability near 1. Using five clients, the algorithm has a 11% completion rate when requesting 22 neighbors per client, but a

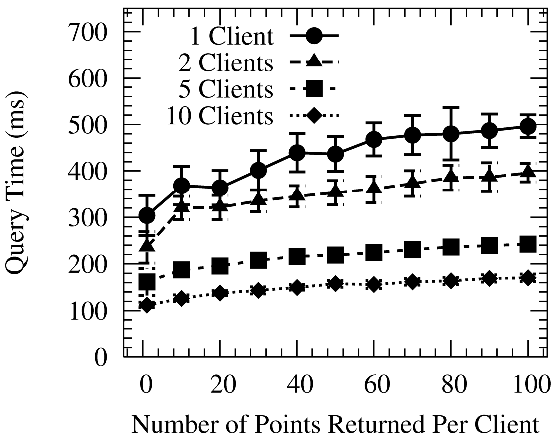

Figure 5: Query time for 100-nn for our algorithm on the SDSS data set for various number of clients as a function of number of points returned by each client.

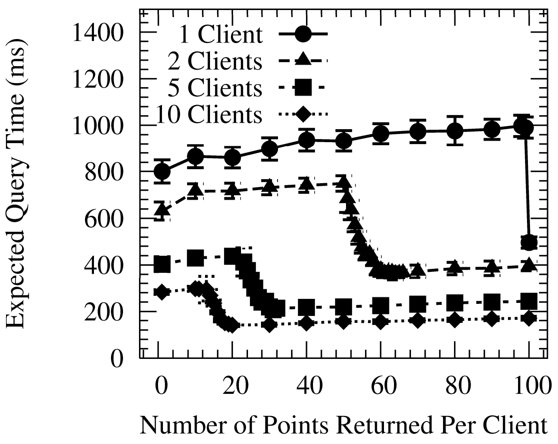

Figure 6: Expected query time for 100-nn for our algorithm on the SDSS data set as a function of number of neighbors returned by each client. Sudden drops in the plot are the result of partial queries returning complete solutions, eliminating the need to re-forward k-nn queries to the clients.

99% completion rate when requesting 32 neighbors per client.

In Figure 5, we display the query time as a function of the number of neighbors requested per machine, again with the SDSS data. For all numbers of clients, the average time required to complete a single search query is logarithmic in h, the number of neighbors requested. Combining the results in Figures 4 and 5, yields the expected search time as a function of the number of neighbors requested per client, shown in Figure 6. If a query is not completed after the first h-nn request to the clients, a second k-nn request is made. Therefore, there is a large disincentive to underestimate h. However, picking the smallest value of h for which the probability of completion is nearly 1.0 can yield significant savings; this point will become more obvious in the following figures.

Finally, while Figures 2 and 3 illustrate the fact that additional clients improve the build and

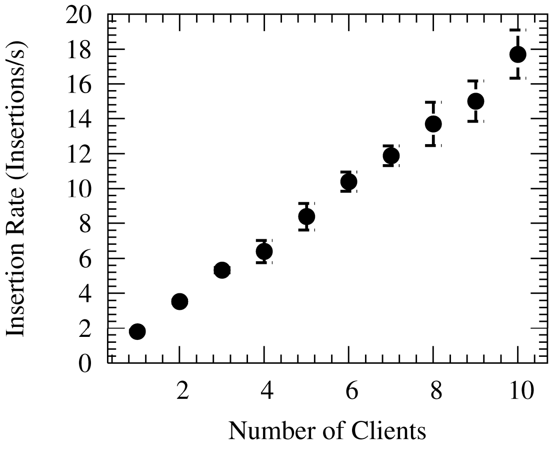

Figure 7: Insertion rate for our algorithm using 1 million points randomly sampled from the SDSS data set as a function of the number of clients.

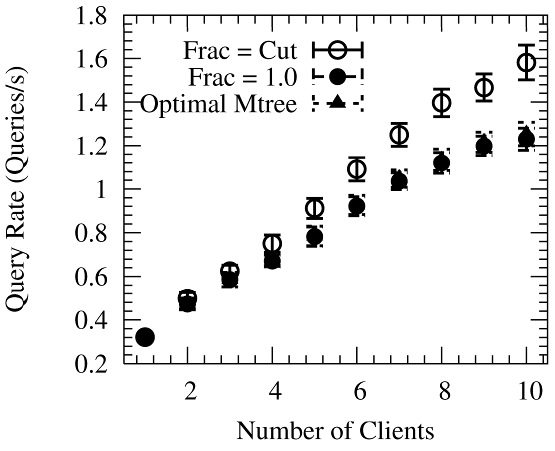

Figure 8: 100-nn query rate for our algorithm using 1 million points randomly sampled from the SDSS data set as a function of the number of clients. Points for the optimal M-tree and the full k-nn queries to each client fall on top of each other.

k-nn query times, determining the relative performance is difficult to ascertain from the plots. In Figures 7, 8 and 9 we plot the insertion and query rates — the number of inserts into the tree and the number of queries performed per second — as a function of the number of clients used for a fixed data size of 1 million randomly drawn points.

Figure 7 shows that including additional clients in our framework results in a near linear speedup for the build processes for the SDSS data set; the other two data sets showed a similar near linear speed up. As a linear speedup is the best that can be expected, this result shows that our framework is nearly optimally utilizing additional client's processing power during insertion operations.

However, this is not the case for nearest neighbor queries. Figures 8 and 9 show the query rates for our algorithm using both full queries (requesting k neighbors per client), and optimal partial

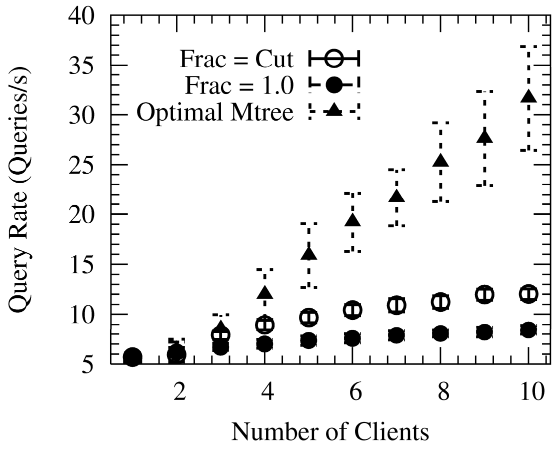

Figure 9: 100-nn query rate for our algorithm using 1 million points randomly sampled with replacement from the Sierpinski data set as a function of the number of clients.

queries (requesting h neighbors per client, such that the expected query time is minimal) for the SDSS and Sierpinski data sets, respectively. In both these figures, we compare our search time to that of an optimal distributed M-tree; other algorithms, such as the peer-to-peer networks of [28, 24], will be slower due to network I/O costs. The optimal tree was built by first looking at all of the data, and then optimally partitioning it to both minimize overlaps between the bounding shapes of the subtrees on each client as well as to ensure that the clients remained balanced. Since the optimal tree uses the standard prune steps of M-trees through out its structure, it may be able to prune off branches corresponding to some of the clients during the search process. Thus, we defined the query time of the optimal tree to be equal to the expected search time of the subtree on each client multiplied by the number of clients that the query would need to be sent to *assuming we knew the distance to the k^{th}nearest neighbor a priori*. Moreover, the estimated search time for the optimal tree does not include costs incurred by the master node or network communication costs, and hence is an underestimate of the true query time (overestimating the query rate) of any distributed M-tree built using the data.

Figure 8 shows that our algorithm achieves a near linear speed up in query rate by forwarding only partial queries to the clients. In fact the query rate obtained by our algorithm is better than that of the optimal tree, which has a query rate similar to our algorithm when using full queries. This phenomenon can be traced to the fact that the SDSS data set is highly skewed. In the case of 10 clients, the optimal tree must forward search queries to 9.8 clients on average. Our system clearly has an advantage on such skewed data sets, as even though it forwards queries to all clients, it need only forward partial queries to each client.

When the data sets are less skewed, our algorithm does not perform as well as the optimal distributed M-tree on k-nn queries. While Figure 9 shows that our algorithm still results in a near linear speedup when using partial queries on the Sierpinski data, the speed-up is a factor of three less than that obtained by the optimal tree; results of the Pacific NW data mirror those given in Figure 9.

However, if the queries themselves are highly skewed, then the optimal tree will also suffer, regardless of the data distribution. If all of the queries revolve around similar targets, then it is likely that the queries will be routed to the same set of clients. Thus, only a small fraction of the clients will be used to do a majority of the processing. This reduces the effective query rate of the optimal tree to that of our algorithm forwarding full queries to each client (filled circles in Figures 8 and 9).

4 Discussion

In the previous sections we have detailed our proposed parallelization algorithm, and showed how it linearly reduces the query time with increased number of clients, while maintaining a query throughput within a constant factor of an optimal distributed M-tree structure. We now discuss how our approach differs from other proposed methods. In particular, we focus on a M-tree master-client approach similar to that of [26] and a distributed peer to peer (P2P) network M-tree similar to [28]. Again, we consider a data set $\mathcal{D}$ of m objects being added to a M-tree distributed over c clients. Suppose the branching factor of the M-tree is b. We will assume that the m objects are roughly evenly distributed over the c clients to ensure optimal search performance. Thus each client will contain approximately $n = m/c$ objects.

4.1 Algorithm Advantages

4.1.1 Simply Parallelizable

As mentioned in §2, each of the clients contain an independent set of identically distributed objects from $\mathcal{D}$. Thus, we can easily split the workload among the c clients, ensuring that no client remains idle while tasks are waiting to be completed, for any number of clients c.

Conversely, parallelizing M-trees using a master-client approach is non-trivial. Ensuring an equal number of objects per client is difficult. Even when objects are nearly equally distributed, such an approach can perform poorly when presented skewed data sets. [22] show that a distributed master-client approach utilizing a variant of M-trees results in very few clients doing the majority of the processing. Additionally, the master-client algorithm is ideally designed for a number of clients that is a power of the tree's branching factor, b; this allows us to cut the tree at some level, and then assign each of the sub-trees below that level to a different client. It is not clear what the optimal distribution scheme is when the c is not a factor of b.

4.1.2 Mutable, Balanced Trees on Clients

M-trees are built from the leaves upward [8]; when a leaf overflows, it splits and promotes a representative sample and information describing the group variation upwards for each new leaf. If this causes the parent to overflow, the parent likewise splits and promotes information upward, until either the new representative sample fits within its parents, or until it is at the root (in which case the root is split and a new root is created). This algorithm is perfectly designed for a single processor system as the algorithm clearly dictates what actions the processor should be performing. However, generalizing the algorithm to multiple processors is not trivial. [33] describes a variant

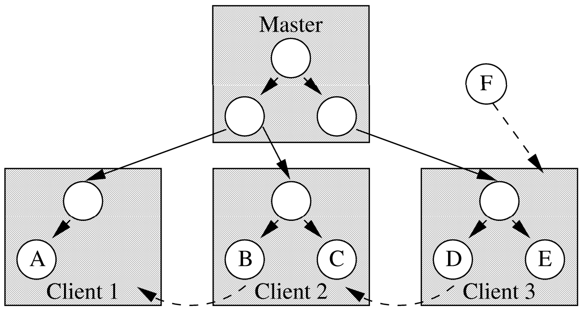

Figure 10: Updates required during the insertion phase for a master-client M-tree. The insertion of the object (or sub-tree) F requires that sub-trees B and D be bumped to clients 1 and 2 respectively, to ensure that each of the clients remain balanced and that dissimilar objects are placed on different clients.

of M-trees which uses a single processor to follow the search/update algorithm through the tree and the remaining processors to execute distance computations. However, such a technique can often lead to many of the CPU's being underutilized, especially if the distance function is trivial to compute.

On the other hand, if we spread the tree over multiple clients, placing each client in charge of a sub-portion of the entire tree, updating the tree becomes troublesome. If the data is not inserted uniformly, the sub-trees associated with each client can quickly become disparate in sizes, requiring a re-balancing step between the clients. This step requires clients with a higher than average number of objects to shuffle some of these objects to other clients. However, for the algorithm to remain efficient, it must ensure that the objects being transferred to a client that has a lower than average number of objects are still similar to objects the client currently contains. In order to ensure that the objects on each client remain similar, it may be necessary to swap objects stored on clients with an average number of objects with those from another client. For example, consider the equally sized sub-trees $A, B, C, \ldots, Z$, where the subtrees' similarity is given by their natural order (i.e. A is most similar to B, then C, etc., and least similar to Z). Suppose we add F to the tree in Figure 10. In order to balance the sub-trees on each client while ensuring that similar objects are on the same client, we would need to insert item F in client 3. In order to maintain balance between clients, item D would need to be moved from client 3 to client 2, and item B would move from client 2 to client 1. Thus, even though only one insertion was performed, the structure each of the clients had to be updated. In practice, it is often easier to rebuild the entire tree rather than try to figure out how to shuffle the objects among the clients. This, however, requires reprocessing all of the data in a central location, a I/O nightmare.

Instead, our algorithm, places an equal probability that a new object will go to any one of the clients. Thus, each client will have roughly the same number of objects with high probability. Clients will remain balanced even after a series of insertions and deletions, and there is no need to ever shuffle objects between them (with high probability). This reduces the insertion and deletion cost of the parallelized structure from being roughly $O(m \log_b(m))$ to $O(\log_b(n))$, slightly less than the P2P network.

4.1.3 Minimized I/O Costs (Disk & Network)

All three parallelization algorithms we are considering do not require disk accesses, as disk accesses are roughly two order of magnitude more expensive than simple RAM accesses[3]. Inter-client communication (e.g. using TCP/IP packets) requires 5-10 times more time than RAM accesses, and is a secondary candidate for I/O cost reduction. As the trees built by the three methods will be roughly similar in size, an estimate of the algorithms' latency is the number of inter-client requests.

Let us consider insertions (deletions will be similar). Using our framework, insertions require exactly one inter-client communication, as once we hash the object we know exactly which client it should be located on. In contrast, the P2P network, requires $O(\log_b(m))$ communications (as each node of the tree is (potentially) on a separate client) while the master-client approach can take as few as 1 communication (assuming the object fits in the sub-tree and no re-balancing is necessary) but may take substantially more. In the worst case, all objects must be shuffled to a single client to then redistributed.

Now consider query related costs. Our method will require exactly c communications[4]. The P2P approach will again require $O(\log_b(m))$ communications, while the master-client approach may use only one. In the worst case the master-client algorithm requires c communications in order to check points on all clients. In practice, we find that the master-client approach checks a constant fraction of the total number of clients, where the constant is a function of skewness of the data set. Thus, our algorithm has the minimal number of communications for insertions and deletions and is within a constant factor for search queries.

4.1.4 Fault Tolerance

While the algorithm as described in §2 does not contain precautions against faults, building in such checks is simple. First, since each master is stateless, they can be easily replicated. Only the IP addresses and ports of the clients must be copied from one master to another. Additionally, since each client is self-contained, we can easily maintain multiple copies of each client.

Specifically, if we have qc processors to use as clients, then we can treat processors 1 to q as the first client, processors $q + 1$ to $2q$ as the second client, etc. Note that the only change required is that the master nodes must now contain pointers to all qc processors; assuming no hardware failures, these pointers will not change throughout the lifetime of the algorithm. All requests (either insertion/deletion or query) will be forwarded to all q processors representing a single client. Thus all q processors will be mirror images of each other (similar to RAID1). For query requests, we need only wait for the first processor for each client to return before compiling the k nearest neighbors. If dedicated processors are used for clients, we expect all q processors representing a single client to return in approximately the same time, providing limited benefit for forwarding search requests to all q processors. However, if we place clients on user desktop machines, the benefit is much more substantial. The redundancy of clients allows us to bypass

[3]However, all three methods can be modified to use on disk storage, if the total size of the tree exceeds available the RAM.

[4]Here we define a single communication as both the request and the resulting response.

bottlenecks caused by intense tasks running on a specific machine. Run times using two copies of each client on different desktops at Google Pittsburgh were shorter and variances were much smaller than when we used the same machines but doubled the number of clients.

If the number of available processors is not divisible by c, then we can obtain redundancy by dividing the data set into c pieces and distributing the first q pieces to the first client, pieces 2 through $q + 1$ to the second client, etc. in a RAID5 fashion. Similar to the mirrored approach, we only need to wait for every qth machine to return, as only $1/q$ of the machines are needed to represent the entire data set.

Both the P2P and master-client algorithms can be replicated using mirroring. Mirroring the data structure results in q independent copies of the same structure[5]. Using such a scheme, the a single failure of one machine in each of the q copies is enough to break the system. Thus there are c^q out of $\binom{qc}{q}$ possible ways the failure of q machines can cause the structure to break, when qc machines deployed running q copies of c clients. Under our replication system, only c of the possible $\binom{qc}{q}$ machine failures break the algorithm. In general, if the probability of failure of each machine, f, is equal and independent of all other hardware failures, the probability that our system breaks is $1 - (1 - f^q)^c$, while the probability that the P2P and master-client system will break is $(1 - (1 - f)^c)^q$.

As a result, our approach is at 10 times less likely to fail with 10 clients and a replication factor of 2 than approaches using mirroring. The failure ratio of between the mirrored approach (P2P and master-client algorithms) and our method grows linearly with the number of machines and exponentially with the replication factor. This is especially important when dealing with a cluster of machines. In a cluster of 1000 machines with an expected lifetime of 5 years between failures, the expected time until a single machine fails is just 4 days. Clearly, with such a high hardware mortality rate, algorithms need to be robust to such failures.

4.1.5 Batch Implementation

Given the independent nature of our algorithm, it is not only trivially parallelizable, but amenable for batch applications. Unlike either the master-client or P2P approaches, the proposed algorithm can be easily implemented in the Map-Reduce framework [9], as each of the tree is independent of the others. The Map-Reduce framework provides check-pointing and ensures that the process completes, even in situations where clients hang or are removed from the cluster, augmenting the fault tolerance of the system.

4.1.6 Approximate Solutions

In many applications returning the exact k nearest neighbors is unnecessary [20, 21]. Our algorithm can be modified in two ways to return an approximate solution. First, we could use an approximate algorithm to process requests on each of the c clients, such as a spill-tree [21]. Such an approach does not require any changes to our framework.

[5]Mirroring the clients of each structure independently would be much harder, as every node would need to know of about q copies of all sub-nodes; updating such a structure would be laborious.

Secondly, we could forward partial queries to each client (described in §2.1) and approximate the final k-nn solution using one of the following approaches:

- Return the first solution from one of the c clients. In settings where clients are not run on dedicated machines, this may result in substantially faster run times. Since each client contains data that is identically distributed, the neighbors returned by such an approach will likely contain a subset of the k-nearest neighbors. The resulting query throughput rivals that of the optimal master-client approach.
- Using the partial results returned by all clients return the k-nearest objects to the query point. If we do not exhaust any of the c h-nn result sets during the merge phase of the algorithm, then this solution is complete. Otherwise, the result will contain at most $k - h$ objects which are not within the true k-nn result.
- Using the partial results returned by all clients return those neighbors which are provably in the true k-nn result. During the merge phase, if we exhaust one of the h-nn result sets, we return the merge set at that point. Our solution will be exact, but will contain fewer than k neighbors.

4.1.7 Ease of Implementation

Finally, our algorithm is easy to implement as each client contains a standard k-nn search data structures. The only additional code that is required is the interface between the master and the clients, and the algorithm for hashing objects. Communication can be easily accomplished using TCP/IP sockets, while hashing is well studied. For instance, [4] presents a simple implementation that we find results in near optimal division of objects among the c clients. Using our framework, it is easy to swap out storage algorithms, and hence use the ideal algorithm for the data at hand. We are not locked into any single storage algorithm.

4.2 Disadvantages

While faster than the full k-nn query, performing the partial query on each client (as described in §2.1), results in an query rate that is sub-optimal (e.g. see Figure 9). Since each client is independent and identically distributed, the M-tree structures contained on each client are nearly identical. Thus, the search algorithm is forced to perform nearly exactly the same search on each client. That is, clients 2 through c cannot directly benefit from prunes computed by client 1. Moreover, the data set on each client is only $1/c$ times smaller than the original data set, and so the expected number of comparisons will be reduced by only $\log(c)$. In this sense, the master-client approach is much better suited for searching, as it can prune clients out of the search algorithm, freeing these clients to work on subsequent queries.

However, note that if either the data set is skewed (as with the SDSS data set), or if the queries are skewed (as in the case of web searches [13, 27]), our algorithm outperforms the optimal distributed M-tree algorithm. This is because the optimal distributed M-tree algorithm distributes the objects based solely on relative distance, routing popular queries to small subset of the clients and leaving the other clients idle [22]. Our algorithm ensures that all clients remain active as long as there are queries to be processed.

Moreover, the cost to build the M-tree is substantial. For the Pacific NW data set, building a M-tree takes roughly 10 minutes but finding the 100-nearest neighbors requires only 60 milliseconds. Thus 10,000 queries can be completely for every tree build. Using 10 clients, the optimal distributed M-tree algorithm is roughly twice as fast as our algorithm. Therefore, if fewer than 20,000 queries are performed for every insert or delete, then our algorithm will be faster than rebuilding the best master-client tree upon insertions and deletions. For our server log application, the modification rate tends to be higher than the query rate making our framework substantially more efficient than the best master-client approach.

5 Conclusion

In this paper we present a novel approach to the problem of storing and searching large sets of dynamic data. We show that a simple approach of distributing the data independently over a number of clients results in a linear speedup for the building process. By forwarding h-nn queries (where $h < k$) to each client we can obtain a near linear speedup when computing the k-nn result for uniformly distributed data; faster k-nn query times result in only a small constant factor decrease in query throughput as compared to an optimal tree on these data sets. Moreover, when the the distribution of data or queries are skewed, which is common in real-world data sets, our algorithm performs better than the optimal tree. Additionally, our parallelization scheme results in may advantages over the traditional master-client framework. The algorithm performs well in dynamic environments, such as server logs, as it easily handles data modifications, and is currently being used in a real-world fraud detection system.

References

[1] A. Alpkocak, T. Danisman, and T. Ulker. A parallel similarity search in high dimensional metric space using m-tree. In *IWCC '01: Proceedings of the NATO Advanced Research Workshop on Advanced Environments, Tools, and Applications for Cluster Computing-Revised Papers*, London, UK, 2002. Springer-Verlag.

[2] M. Batko, C. Gennaro, and P. Zezula. Scalable and distributed similarity search in metric spaces. In *Proceedings of the 5th Workshop on Distributed Data and Structures (WDAS 2003)*, May 2003.

[3] J.L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4), 1980.

[4] J. Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, 2001. See Item 8, Chapter 3.

[5] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1997. ACM Press.

[6] S. Brin. Near neighbor search in large metric spaces. In *The VLDB Journal*, 1995.

[7] P. Ciaccia and M. Patella. Bulk loading the m-tree, 1998.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *The VLDB Journal*, 1997.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation*, San Francisco, CA, 12 2004.

[10] W. Feller. *Introduction to Probability Theory and its Applications*. John Wiley and Sons, New York, 3rd edition, 1968.

[11] Y.J Garcia, M.A. Lopez, and S.T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *GIS '98: Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, New York, NY, USA, 1998. ACM Press.

[12] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1984. ACM Press.

[13] B.J. Jansen and U.W. Pooch. A review of web searching studies and a framework for future research. *Journal of the American Society of Information Science*, 52(3), 2001.

[14] J.K. Adelman-McCarthy et al. The fifth data release of the sloan digital sky survey. Submitted to Astrophyical Journal Supplements, 2007.

[15] I. Kamel and C. Faloutsos. Parallel r-trees. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1992. ACM Press.

[16] I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM '93: Proceedings of the second international conference on Information and knowledge management*, New York, NY, USA, 1993. ACM Press.

[17] N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multi-computer architecture. In *Extending Database Technology*, 1996.

[18] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM Press, 2005.

[19] S.T. Leutenegger, J.M. Edgington, and M.A. Lopez. Str: A simple and efficient algorithm for r-tree packing. In W.A. Gray and P. Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.* IEEE Computer Society, 1997.

[20] T. Liu, A.W. Moore, and A. Gray. Efficient exact k-nn and nonparametric classification in high dimensions. In *Advances in Neural Information Processing Systems 16*, 12 2003.

[21] T. Liu, A.W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Advances in Neural Information Processing Systems 17*, 12 2004.

[22] T. Liu, C. Rosenberg, and H.A. Rowley. Clustering billions of images with large scale nearest neighbor search. *IEEE Workshop on Applications of Computer Vision*, 0, 2007.

[23] D. Neill and A.W. Moore. A fast multi-resolution method for detection of significant spatial disease clusters. In *Advances in Neural Information Processing Systems 16*, 12 2003.

[24] D. Novak and P. Zezula. M-chord: a scalable distributed similarity search structure. In *Proceedings of the 1st international conference on Scalable information systems*. ACM Press, 2006.

[25] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *SIGMOD '85: Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1985. ACM Press.

[26] B. Schnitzer and S.T. Leutenegger. Master-client r-trees: A new parallel r-tree architecture. In *Statistical and Scientific Database Management*, 1999.

[27] A. Spink, D. Wolfram, B.J. Jansen, and T. Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science and Technology*, 52(3), 2001.

[28] E. Tanin, D. Nayar, and H. Samet. An efficient nearest neighbor algorithm for p2p settings. In *dg.o2005: Proceedings of the 2005 national conference on Digital government research*. Digital Government Research Center, 2005.

[29] C. Traina Jr., A. Traina, B. Seeger, and C. Faloutsos. Slim-Trees: High performance metric trees minimizing overlap between nodes. *Lecture Notes in Computer Science*, 1777, 2000.

[30] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4), 1991.

[31] P.N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[32] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search - The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.

[33] P. Zezula, P. Savino, F. Rabitti, G. Amato, and P. Ciaccia. Processing m-trees with parallel resources. In *RIDE*, 1998.

MACHINE LEARNING
DEPARTMENT

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Carnegie Mellon