Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm

Robert Devine

University of California at Berkeley
EECS Department, Computer Science Division
devine@cs.berkeley.edu

Abstract. DDH extends the idea of dynamic hashing algorithms to distributed systems. DDH spreads data across multiple servers in a network using a novel autonomous location discovery algorithm that learns the bucket locations instead of using a centralized directory.
We describe the design and implementation of the basic DDH algorithm using networked computers. Performance results show that the prototype of DDH hashing is roughly equivalent to conventional single-node hashing implementations when compared with CPU time or elapsed time. Finally, possible improvements are suggested to the basic DDH algorithm for increased reliability and robustness.

1 Introduction

Rapidly plunging hardware costs and increasing performance of CPUs and networks mean that future file and database systems are likely to be constructed as networked clusters of nodes. Algorithms should be devised to work in these environments. This paper describes the design and implementation of a distributed hashing algorithm.

Quick record retrieval is obviously crucial to overall file and database performance. The design of efficient retrieval algorithms has been a rich research area in computer science since the earliest days. Hashing algorithms can be classified as either static or dynamic. A static hash algorithm uses an constant sized hash table. A dynamic hash algorithm differs from a static hash algorithm because the table can grow and shrink from its initially allocated size to accommodate the continued insertion and deletion of records.

DDH (Distributed Dynamic Hashing) is an algorithm that gracefully expands and contracts without a central controller. A variable number of clients and servers participate in the

The performance of conventional, single-node hashing is determined by its CPU processing time and the number of I/Os needed. However for distributed hashing, the performance of messages must also be considered. Misdirected messages in DDH are possible but are shown to be minor if a local "hint" directory is used to provide a mapping of hash buckets to servers.

The paper is organized as follows: we present the design and implementation of DDH in sections 2 and 3, show the performance results in section 4, and conclude in section 5.

2 Goals for Distributed Hashing

The goals desired for distributed hash algorithms are listed below. High performance is of course a general goal. Other goals are certainly possible; Section 5 suggests several others. The above goals are used to evaluate and develop distributed algorithms. Section 3 tells how these goals are realized in DDH.

1. Location of distributed buckets and servers

   Two location requirements must be met. First a method is needed for locating candidate nodes that are eligible to participate in the distributed hash structure. The second requirement is to map a hash bucket to a server at run-time.

   If we restrict the selection of nodes to those that are within a few multiples of an access to a locally mounted disk, then all nodes that are within 50 milliseconds average network time away can be used. Nodes not on the same LAN can be used. From [3] it is shown that even many computers on the Internet wide area network have a mean response latency of under 50 milliseconds. While this fact argues for having a widely distributed hash tables, the concerns of administration boundaries and network overload (especially during bucket splits and merges) argue against extending a hash table distributed beyond the locally administered boundaries.

   A directory service can be used to identify which nodes can participate and converting their names to a network address. This information is relatively static because changes happen on a human time-scale as nodes are added or removed.

   Simple directory services are ill suited for dealing with the second requirement of tracking the rapidly changing bucket location information. Moreover if multiple directory services are used they must provide a consistent view of the bucket mappings. Because bucket splits and merges are happening at execution speed, the bucket directory problem must be handled dynamically.

2. Collision resolution

   There are several possible policies for handling filled buckets. The bucket could be temporarily expanded; records could overflow to an overflow bucket or a "buddy bucket" or the bucket can be immediately split.

   The effect of choosing one of the collision resolution methods is reflected in the overall performance and efficiency of the algorithm. When overflows are employed for a bucket that has a high rate of collisions, that bucket becomes unbalanced with respect to the average bucket. Alternately the policy of splitting evens out the load but may cause an imbalance in other ways such as an increased load on a single server.

   Expanding the bucket size is undesirable because it leads to more complex algorithms. Requiring permission before splitting means that agreement must be achieved using non-local information. Splitting immediately, while simpler, may cause complication in other areas, notably in the bucket location algorithm.

3. Load balancing across servers
   Ideally, no server is responsible for more than its proportional share of the total data size. This principle can be enforced by actively checking the local load level against the global load level and then performing adaptive load balancing to ameliorate hot spots. A naive approach would be to designate a load controller that partitions work among servers according to fairness criteria.
   The need for scalability obviates the naive approach. Any centralized solution would soon become a bottleneck as the hash table grows in size.
4. Parallel insertion and retrievals
   It is desirable to support multiple concurrent accesses to the same distributed hash table. Multiple readers can be easily supported because no changes are occurring. However, in a mixture of multiple readers and writers, the servers must institute a consistency control, such as locking to serialize access to its portion of the hash table.
   Each individual server should use the necessary consistency protocols to guarantee correctness when multiple clients are inserting records. In addition, if an insertion causes a split, the affected servers must mutually ensure consistency of their parts of the distributed table.

2.1 Previous Work

Multiple dynamic hashing methods have been proposed for single nodes with either a single processor [5, 2, 6] or with multiple processors [10].

LH* [7] is one proposal for a distributed hashing algorithm. It extends the notion of linear hashing [6] to allow multiple nodes in a network to participate in the same distributed data structure.

LH* is a directoryless algorithm like the single node version of linear hashing. It locates buckets through one of two algorithms based on the current split level and the bucket number. A client maintains what it thinks is the current split level and highest numbered bucket. Because the table can grow or shrink without the client knowing, the actual bucket may be located elsewhere.

As proposed in [7], LH* has some drawbacks:

First, each server has only one bucket. The limit is required for bucket location calculations. However, this can be easily overcome by using a logical server numbering scheme that maps to the actual server number (a round robin assignment would work fine). The larger problem is that a single bucket per server implies large buckets that result in a high cost to split a single bucket so each server should hold multiple, small buckets that are less expensive to split.

Second, splits must be ordered for the clients. Because the clients have no directory to allow unbalanced splitting, all splits are required to follow the bucket numbering order within each level. The bookkeeping of this ordering requires that a server send at least one more message than needed to just convey the split records. While the overhead of sending agreement messages with every split, the problem becomes most acute when dealing with failures. If the message controlling the ordering is lost or the server that owns the current split token crashes, all other servers are affected because they can not split until a new token is regenerated. This is analogous to the complexity of token ring networks compared to Ethernet.

Third, when multiple clients are inserting records there exists a timing window where it is possible that a client requires more than the expected maximum of two forwarding messages. If the client is slower than the rate of bucket splittings, the client's view of the LH* hash table will lag the actual configuration. While it is true that only two hash levels can exist at a single time, client actions are not time synchronous. Therefore it may see more than two versions of the hash table as it evolves. Each version may cause client addressing errors.

Fourth and most important, determining when a bucket can be split is not an autonomous decision that can be made by the affected server. The ordering of splits is imposed on the buckets to support the directoryless character. This restriction in inherent in LH* because of the need for all buckets to be determined from one of two bucket location functions. Several undesirable characteristics result from this. First, to strictly order the splits, the paper proposes a special split coordinator that participates in the bucket split operation. This is contrary to the goals of no load balance and high availability. Second is that buckets that are not allowed to immediately split must handle overflows locally. This leads to poorer performance and hot spots. Finally, because all buckets on a level must split before the next level can start to be split, this causes premature splitting of non-full buckets.

3 Distributed Dynamic Hashing

In this section, we introduce the design of DDH. The guiding philosophy for DDH is local autonomy. No changing global information guides the actions of individual servers; each server decides for itself when to split or merge buckets. Clearly some common, constant policy must be shared by all serves to avoid anarchy but the policy is constant can be easily coded into all servers. An interesting research question is what minimum level of shared policy is required yet still preserve a collaborative effort that is necessary for a distributed data structure.

A DDH server program runs at every server node that is participating in the distributed hash algorithm. It is responsible for controlling the storage of its portion of the entire table. Each server maintains the following information about each bucket: the bucket number, its split-level and the contents of the bucket. The server is responsible for handling all of the messages that a bucket may receive. This includes forwarding requests to another server, sending the appropriate response to a request and analyzing the replies to its own requests from other servers. Each DDH servers maintains a small local directory to store the bucket to server mapping. However, all DDH clients do not need to use such a directory but it is advantageous if they use one (performance results given in Section 4 show why).

A client calls the DDH library routines to do the insert or retrieve operation. A client program computes the hash key for the record, locates the *likely* bucket for that hash key, and then sends the request to the server that owns the bucket.

Each server maintains a directory containing location information about other server's buckets. The directory is not exact for remote buckets but gives likely location. Upon receiving any request concerning a data item from a client, the server determines if it is the correct recipient of the message by comparing the data item's key to the key range for its buckets. If the server discovers that it is not the right one, then it forwards the message to the correct recipient. Otherwise the operation is performed locally. A reply is sent from the server to the client indicating whether or not the operation was successful. The reply contains the current hash level of the bucket at which the operation was eventually performed. This allows the client to apply the DDH algorithm to update its local perception of the hash table.

3.1 Distributed Hash Table

The distributed hash table in DDH is a distributed main memory data structure composed of buckets spread over one or more servers. Each bucket holds up to some fixed number of records. It is not required that all servers use the same bucket size although There is one or more server processes per node, although it is expected that the usual arrangement is one server per node for the best performance. Client processes send requests to servers to insert or retrieve a record.

Bucket addressing is based upon binary radix trees, or *tries*. At split level N, the lowest order N bits are used to form the bucket number. As an example, suppose a record's key hashes to the value 0x81F0639C. Since 'C' hex is 1011 binary, if the current level is 1 then the bucket number is 1. If the level is 2 or 3, the bucket number is 3. And if the level is 4, the record must be in the bucket number 11 (= 'C' hex).

The syntax of bucket numbering is written as the pair (level, bucket number). When the algorithm begins there is a single bucket, numbered as bucket (0,0), that matches all hash values. When it fills, its contents are split into two buckets by using bit 0 of the hash value; these are then buckets (1, 0) and (1, 1). In general, a level L bucket split uses bit L of the hash value to move records into its child buckets. When bucket (L, N) splits, it forms the children buckets ($L+1$, N) and ($L+1$, $N+2^L$).

Bucket and Server Location: Bucket location is described by the tuple *(number of servers, server distribution function, hash function, key, current hash table configuration)*. The first two items are defined for the entire table and must be known by all servers and clients. A client chooses the *hash function* that generates a *key* for each record. The fifth field, *current hash table configuration*, is changed with every insert or delete operation so it is known definitively only by the client and servers that were involved in the operation; all other nodes must discover the configuration.

A server when starting, knows its logical server number and the total number of servers. A fresh client without any location knowledge starts with the server for bucket (0, 0) because that is the only bucket whose existence is guaranteed. Thereafter a client progressively learns the current configuration by making requests that are either correct or are wrong but come back with the correct location information. The client sends requests to the server it believes owns the bucket but will adjust its directory if told that the guess was incorrect.

Two approaches are possible for the client to use the bucket-to-server mappings it has learned. First, a client can maintain a directory of mappings that it has learned from previous messages. Then future messages use this directory to select servers. Second, a client can use heuristics to guide guesses. For example, based upon the current average split level information a client has gleaned from previous message replies, a client can produce a reasonable guess by using that level for the next message. Performance trade-offs are compared in Section 4.

Likewise two approaches are possible to implement the address correction protocol. The client can have its incorrect guesses returned to it so that it has to retry a different server or the server can silently forward the message to the correct server. If it is the case that the client expects a reply within a bounded time else it will time-out and resend the message, then it is better for the client to retry to avoid time-outs caused by too long server forwarding. DDH chooses the method of requiring the server to forward misdirected messages.

Collision Resolution: When a bucket is filled, whether to split the bucket is entirely a local decision. Unlike Linear Hashing [6], no order of bucket splitting is necessary. Unlike Extensible Hashing [2], the entire table is not split as one operation. Splitting is an autonomous operation that does not require global knowledge. Skewed data can therefore be efficiently handled with the minimum number of split buckets and no special overflow areas.

The implementation of DDH practices an uncontrolled splitting policy. The splitting server sends a message to the new site server that will manage the new bucket. If this server accepts it, a series of messages are sent to the server with the split records. Multiple records are sent in the same split message as a performance optimization.

Load Balancing: There are only data servers in DDH unlike LH* that used an index or split manager. Because its buckets can be small, DDH achieves a fine-grained load sharing across servers. When starting, all requests go to bucket 0 but as the number of buckets increases, all servers soon get roughly the same number of buckets.

Parallel Operations: Multiple clients can insert and retrieve records in parallel with each other. From the viewpoint of message reception, all servers operate atomically. Clients do not see the internal state of servers where inconsistent operations may result. Rather, the consistency points are defined by servers as they send and receive messages. All server operations are locally serialized through complete processing of a message before starting another. Multiple servers synchronize their behavior when dealing with page splits.

3.2 DDH Networking Implementation

The three basic services offered by the DDH session layer implementation are packet encoding/decoding, retransmission, and response matching. As only simple data types are used, the packet is rebuilt each time it is sent.

Because it is unlikely that purely homogeneous clusters of systems would be using DDH, it was implemented as a portable session level service. All communication to the DDH servers uses the Internet User Datagram Protocol (UDP) [8] datagram service and messages are encoded in network byte order.

The DDH network protocol is designed to the request / response model. Requests messages are sent by clients to servers, and also by servers to other servers during bucket splits and merges. If the request is sent to the wrong server, the server puts the network address of the requester inside the packet and forwards it to the correct server. The receiving server uses this address to send back a reply directly to the requesting client. As network failures may cause one or more packets to be lost, all operations are self-contained and idempotent. Messages are retransmitted if no reply is received before a time-out.

Each packet consists of a fixed format header followed by a specified number of (key, value) pairs. Both the key and value are preceded by a short word containing the length. All integer values are converted if necessary before transport to the network byte order. The following request messages are supported: `RETRIEVE`, `INSERT`, `DELETE`, and `SPLIT_BUCKET`. Only servers send `SPLIT_BUCKET` requests other servers before performing a split or merge operation and to prepare it for a bulk data transfer using `INSERT` messages. These requests result in one of these responses: `DUPKEY`, `ERROR`, `NOTFOUND`, and `SUCCESS`.

4 Performance Experiments

To show the implementation of distributed hashing, the application of a phone service "white pages" lookup for finding a phone numbers is used. Given one of a large number of names, the phone number is returned. Up to 50,000 pairs of names and phone numbers were used. Each name was less than 32 bytes long and served as the key field.

We compare the performance of the following hashing packages:

1. DDH – the distributed hashing algorithm introduced in this paper
2. NDBM – the single node hashing package from AT&T [1]
3. SELYIT – the single node linear hashing package from [9]
4. GDBM – the single node dynamic hashing package from GNU freeware

4.1 Environment

Multiple DECstation 5000/133 workstations, an approximately 20 SPECmark machine, with ample memory (at least 16 megabytes) were used. The systems were running ULTRIX 4.2a, a variant of BSD UNIX. Each workstation communicated with either a 10 megabits/second Ethernet or a 100 megabits/second FDDI LAN.

Program execution was timed using the operating system's `getrusage()` system call to record actual resources used. The timer was started before any calls were made to the hash packages and stopped immediately afterwards to remove from consideration all of the normal process initialization overhead from the performance measurement. Tests were run twice to allow all code pages to be fetched from the disk so that no start-up overhead affected this area. and were run on idle systems at off hours to minimize any possible interference from the network. However, testing ran at the "multi-user" level rather than "single user" so that some interference from system level daemons running in the background was possible. It was necessary to run at the normal level so that the network could be used.

4.2 Time to Insert and Retrieve N Records

Records were loaded using the different hash packages. During the first phase of the test, every record in the test was read from an on-disk file, hashed, and inserted into the table. During the second phase, the records is reread and every record is retrieved. Note that there is a high probability of finding the record in memory for each of the single node hashing packages.

All times given below are the total time to insert and retrieve the same N records. The elapsed time needed to run the tests is the complete time needed to run the program from the shell prompt. The values for 30K, 40K, and 50K are not shown for GDBM because those times became quite large and would have dominated the graph.

The graph in Figure 1 shows that DDH is comparable to other hashing packages that operate in a single process on the same node. This experiment presented the disk-based hashing package in the best light because all disk blocks were likely to be in memory. Hence the only I/Os performed were during the flushing of blocks to disk. In contrast, the DDH numbers are both the best and the worst because the messages have to be sent to a remote node in either case.

There were several surprises in the performance results. First, the widely different elapsed times between the SelYit and the GDMB package are because the SelYit package did not flush its dirtied pages while the GDBM explicitly called `sync()` system call to flush its directory pages from the ULTRIX file system buffer cache to disk. In addition, the GDBM package used an extensible hashing algorithm [2] that doubled its directory size periodically. This explains the "knee" in the curves above the 10,000 name pairs point for GDBM. Finally, the user time for DDH is dwarfed by the system time with the ratio being roughly 1:4. If DDH were run on a system with operating system with higher performing messaging, the elapsed level would be lower.

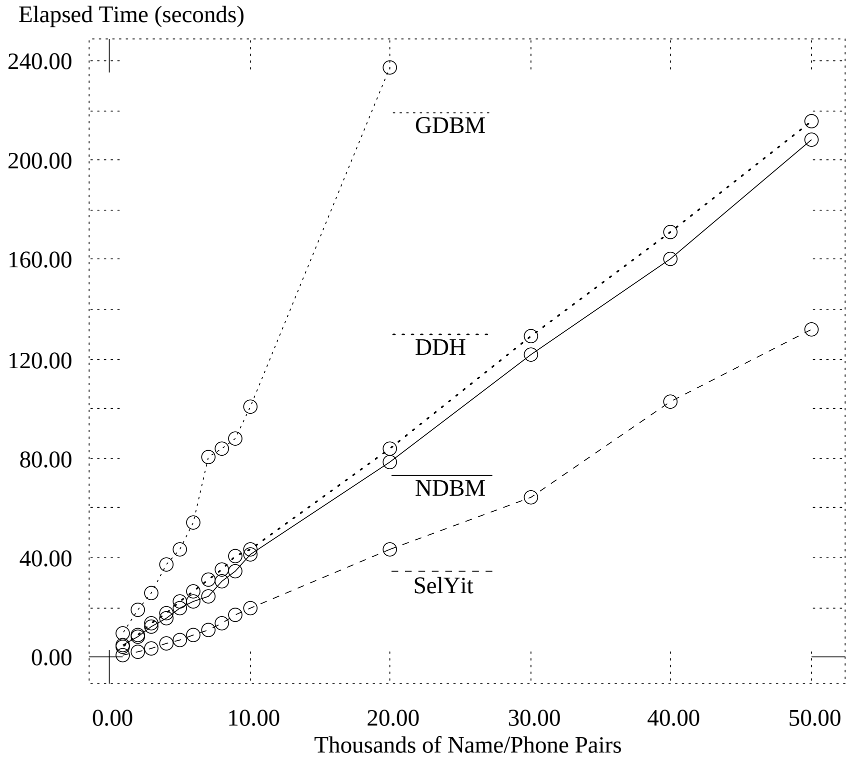

Fig. 1. Elapsed Time to Insert and Retrieve N Records

4.3 Bucket Location Strategies

The question of whether the DDH method of autonomous splitting causes the clients to make frequent addressing mistakes was examined. The experiments in this paper used a round-robin distribution to map buckets to servers but others are possible.

The number of server messages caused by addressing errors is also related to the number of split levels – i.e., because each bucket may have split to a new server, its current state is unknown unless a probe is made.

Three server locating strategies were compared. Table 1 shows the number of messages that result from addressing errors for a fresh client retrieving records from a 249 bucket hash table (9000 records stored in buckets holding a maximum of 64 entries) on 3 servers. The experiment used 3 servers because the value 3 a non-multiple of 2 and therefore causes every bucket split to send records to a different server.

The *heuristic* algorithms are based on a guess of the bucket's split level. The simple heuristic just uses the previous reply's split level to guess the next request's split level. The complex heuristic uses a *moving average* of all previous

Table 1. Bucket Addressing Errors

Strategy	Extra Messages for 249 buckets
Heuristic (simple)	1063
Heuristic (complex)	627
Incremental	62
Quick Start	9

replies so that the average would gradually converge on the actual size. Both heuristics assume that the hash table is reasonably well balanced. Like Linear Hashing [6], these are directoryless and neither store definite information about the whole hash table. Therefore they constantly makes the same mistakes and produce more errors than do the directory based strategies. Moreover, the mistakes made are more often because they result from individual record access unlike the directory strategies that use bucket granularity.

A client using the *Incremental Convergence* algorithm maintains a directory of definitive bucket locations the client receives from servers. Additionally, the location of a bucket can be inferred from the definitive knowledge of a sibling bucket. This strategy will incorrectly guess the bucket addresses about 25% (in the above table, 62/249 is approximately 25%) because of the four cases, it can correctly deduce two. When a bucket is authentically determined, it and its sibling are now known. That accounts for 50%. For remaining 50% buckets that it doesn't know about, in the worst case half will be located at the same node and half will be at a different node.

The *Quick Start* algorithm is similar to incremental convergence but a special probe message returns statistics about server 0's hash table when the client starts. The statistics are used to construct a complete directory so that the slow learning of the incremental strategy is avoided. While the resulting directory might not be completely accurate, it has the advantage of being mostly accurate and should some errors. This strategy made only a few bucket addressing errors because is able to completely guess the majority of the current directory by constructing a directory of the current depth. It is only wrong on the fringes where it had incorrectly assumed a higher split level.

For each of the above strategies, it is possible to propose examples that produce poor results. There are timing windows that would invalidate any previous knowledge and result in a miss on every bucket except bucket 0. However, real-world cases are unlikely to cause degenerative performance. Therefore the ability of a simple client directory produces very good results.

4.4 Number of Messages

Just as a main memory hashing is judged on the number of memory accesses and a disk based hash algorithm is judged on the number of I/Os it uses, a distributed hashing algorithm must show how many messages were used. This experiment found the average number of messages needed to insert or retrieve all records. DDH comes very close to the optimal of only two messages.

The number of messages is calculated as:

$$messages\_per\_request = request + reply + \varepsilon + \beta + \gamma \qquad (1)$$

The request and reply messages are the 2 normal user visible messages. The DDH message protocol requires an acknowledgment by the server for every client message received. Algorithms that assume reliable messages and only count the user visible messages fail to account for the low level acknowledgments used to enforce message reliability. Any low-level network messages are represented as ε and includes the messages invisible to user code such as low-level flow control, network name resolution messages, and message retransmissions. Since ε is unmeasuraable at the user level, it is not counted although it does affect performance. The effect of bucket splits is given by the β term which counts the number of bucket splits that occur between servers when a bucket fills. It is an inverse function of bucket size therefore it is advantageous to have the message size be slightly more than roughly half of the bucket size so that all of the migrated records can fit into one message.

Finally the γ term counts the addressing errors which are occurrences of forwarded messages between servers when a client asks the wrong server due to a change in the configuration of the distributed hash table. This is a non-linear function of number of servers and amount of data because a server can autonomously decide to split bucket while multiple clients are inserting or retrieving records means that clients will have addressing errors. There are also addressing errors resulting from the period that a client is learning the current hash table. Section 4.3 measured these. Table 1 shows actual counts of addressing errors. They are less than 1% of all messages when using the *Quick Start* algorithm.

The measured number of messages sent on average for every DDH request is 2.02 which is very close to the optimal message total of 2.

4.5 Network Performance

DDH used a synchronous request/response protocol for all of its communication using UDP messages. As a result, the network was the bottleneck. All performance monitoring showed that the CPU utilization rate was always in the low to mid 40% level for the client side. Of this amount, approximately 75% of the time was spent in system time as the kernel sent and received the UDP messages.

There is a common belief that because the raw speed of FDDI is 100 Mbps, it is 10 times faster in all dimensions than a 10 Mbps Ethernet. We found that FDDI produces only about 25% better response time. The proportion of user to system time was about 1:3 for FDDI as it was for Ethernet. This suggests that the network software layer is quite heavyweight.

To discover how the network performance affected the DDH performance, a comparison measurement was done using the `ttcp` network performance analysis tool. For the UDP protocol, by sending 80 byte packets to imitate the average message size used for DDH in the study, the CPU is nearly 100% busy. The workstations used in this study can send slightly under 800 UDP packets per second at its maximum. The ULTRIX kernel has a much higher code length for networking calls than it does for file system calls. By comparison, several thousand read and write calls per second are possible. Some modern microkernel OSs can perform a small-message RPC call in about a millisecond.

4.6 Multiserver Performance

One very strong advantage with distributed hashing is the ability to involve multiple servers to share the work load. A series of experiments were conducted to test the effect of adding more servers.

The first group of experiments quantify the speed-up of adding more servers. Speed-up means that increasing the available performance by N while keeping the workload constant should yield an N times speedup. Each record was inserted and then retrieved.

Table 2. Speed-up performance (elapsed time)

Records	1 Server / 1 Client	2 Servers / 1 Client	3 Servers / 1 Client
1000	6.2 seconds	5.5 seconds	7.0 seconds
2000	13.3	11.7	14.0
3000	19.1	16.8	20.1
4000	24.6	22.4	27.1
5000	30.5	28.5	33.6
6000	37.6	33.2	40.6
7000	44.0	38.7	46.3
8000	51.8	44.2	53.7
9000	56.8	50.9	63.0

Table 2 does not exhibit an increase of speed as one client uses one, two, or three servers. There is only marginal speed-up when using a second server. The most like explanation for the minimal speedup is that all messages are synchronous. Message throughput is still the bottleneck so that any added server capacity is unusable if the client waits.

The second group of experiments looked at the scale-up of increasing the workload commensurate with the increased available performance. Table 3 lists the resultant scale-up.

Table 3 shows that DDH has very good scalability when the load and capacity re both doubled. Based upon the DDH design, it seems very likely that scale-up exists when using higher multiples of server/client pairs. However those experiments where not performed so this remains unproven.

Table 3. Scale-up performance (elapsed time)

Records	1 Server / 1 Client	2 Servers / 2 Clients
1000	3.1 seconds	3.2 seconds
2000	5.9	6.1
3000	8.9	8.8
4000	13.9	12.8
5000	15.0	14.8
6000	19.1	19.0
7000	21.5	21.2
8000	25.8	24.4
9000	27.4	27.3

5 Conclusions and Future Work

With the growing number of networked systems that share information in a distributed manner, the argument for a distributed data structure is compelling. Freedom from single node limitations, easy scalability, and better overall performance are the goals. In comparison to a single-node hash table, a distributed table theoretically allows growth of the table to the composite size of all the workstation's memory before the hash table is forced to reside on disk.

DDH, a distributed dynamic hashing algorithm, was implemented on a group of workstations to quantify the benefit of using a distributed solution. The best environment for DDH is a group of servers under the same administrative control (either directly or indirectly) and have a "shared-nothing" architecture so that there is no interference between systems to diminish performance. Systems organized into *workstation farms* are likely the best match to the needs of DDH if they can provide network communication protocols with low latency.

Performance results show that it is comparable to other current implementations when measured with using CPU time or elapsed time. However, because DDH sends messages between systems, the network becomes the bottleneck. In our largest test that used 50,000 records, the test required approximately 200,000 network messages to insert and then retrieve all records. Because a single network message takes about 2 milliseconds elapsed time, even between fast workstations on the same Ethernet segment, the cumulative elapsed time is quite high. The cost of distributed access is better than a disk access, but far worse than a memory access.

The area of distributed data structures introduces several challenging research problems in the areas of concurrency control algorithm, server failure, and supporting variable number of servers. Since servers can fail, some method of using data redundancy or server redundancy should be employed [4]. However, the drawback is the increased complexity of dealing with replication or data consistency. The actual case may be more complex due to server failures that are not independent. Dynamic hashing freed hashing from the fixed sized hash tables. Distributed hashing expands hashing from a single node. The next logical step beyond is to allow for varying number of servers. The question is whether this is possible and still preserve retrieval using one message.

In summary, we find that DDH offers a useful approach for structuring distributed storage systems. DDH can prove to be useful when the data to be stored exceeds the size of a single system's memory. Because the network latency is less than the latency of a disk request, huge single table can be accessed from multiple servers with better performance. DDH is also more tolerant of skewed data than other dynamic hashing methods because it allows for bucket splitting on demand.

References

1. AT&T, DBM(3X), *Unix Programmer's Manual, System V.3*, pp. 506-8, 1985.
2. Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong, "Extensible Hashing – A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems*, Volume 4, No. 3, pp. 315-34, September 1979.
3. Richard Golding, "Accessing Replicated Data in a Large-Scale Distributed System", University of California at Santa Cruz technical report, June 1991.
4. H. I. Hsiao and David DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines", *Proceedings of the 6th International Conference on Data Engineering*, February 1990.
5. Per Larson, "Dynamic Hashing", *BIT*, 1978 Vol. 18(2), pp. 184-201.
6. Witold Litwin, "Linear Hashing: A New Tool for File and Table Addressing", *Proceedings of the 6th International Conference on VLDB*, October 1980.
7. Witold Litwin, Marie-Anne Niemat, and Donovan Schneider, "LH* – Linear Hashing for Distributed Files", *Proceedings of the 1993 ACM SIGMOD*. May 1993.
8. John Postel, "User Datagram Protocol", *USC/Information Sciences Institute*, Internet RFC 768, August 1980.
9. Margo Seltzer and Ozan Yigit, "A New Hashing Package for UNIX", *USENIX Conference Proceedings - Winter '91*, January 1991.
10. C. Severance, S. Pramanik, and P. Wolberg, "Distributed Linear Hashing and Parallel Projection in Main Memory Databases", *Proceedings of the 16th International Conference on VLDB*, Brisbane, Australia, 1990.

This article was processed using the LaTeX macro package with LLNCS style