hQT*: A Scalable Distributed Data Structure for High-Performance Spatial Accesses

Jonas S Karlsson
CWI, Box 94079, NL-1090 GB Amsterdam
The Netherlands
jonas@cwi.nl

Abstract

Spatial data storage stresses the capability of conventional DBMSs. We present a scalable distributed data structure, hQT, which offers support for efficient spatial point and range queries using order preserving hashing. It is designed to deal with skewed data and extends results obtained with scalable distributed hash files, LH*, and other hashing schemas. Performance analysis shows that an hQT* file is a viable schema for distributed data access, and in contrast to traditional quad-trees it avoids long traversals of hierarchical structures.*

Furthermore, the novel data structure is a complete *design addressing both scalable data storage and local server storage management as well as management clients addressing. We investigate several different client updating schemes, enabling better access load distribution for many "slow" clients.*

Keywords Scalable Distributed Data Structure, Spatial Point Index, Ordered Files, Multicomputers

1 Introduction

Research is increasingly focusing on using multicomputers [1] [20] [24]. Multicomputers are built from mass-produced PCs and Workstations, often having special high bandwidth networks. Such infrastructures have emerged and many organizations have typically thousands of machines with an impressive large total amount of RAM, CPU and disk-storage resources.

Multicomputers provided a challenge and promises to cope with the ever-increasing amount of information using new distributed data structures. Scalable Distributed Data Structures (SDDSs) form a class of data structures, first proposed in [13], that allows for scalable distributed files that are efficient in searching and insertion. The approach gives virtually no upper limit on the number of nodes (computers) that participate in the effort. Multiple autonomous clients access data on the server-nodes using their *image* to calculate where the data is stored. Their images might be outdated, but clients are updated when addressing errors occur.

SDDSs are especially designed for avoiding hot-spots, typically a central directory. Clients are autonomous and their directory information is incomplete. When a server receives a mal-addressed request from a client, the request is forwarded to the correct server, and an *Image Adjust Message* (IAM) is sent to correct the client, improving its addressing information.

So far several SDDSs have been proposed. Many of them are hash-based, such as LH*[13] followed by DDH[2], and [25] and LH*LH [6]. A number of B-tree style distributed data structures were also designed, e.g., RP* [14], DRT[10][11], and k-RP* [12]. k-RP* allows for multi-attribute distributed indexing.

Increased storage demands of larger amounts of spatial data gave birth to many different data

The 5th International Conference on Foundations of Data Organization (FODO'98), Kobe, Japan, November 1998.

structures. They can be divided into the following classes: Grid-style files, (quad)tree-structured files, directory-based, hash-based. Combinations of hash-based structures and search-trees are called *hybrid-structures*. Grid-files [16] experience problems with non-uniformly distributed data. So do many multidimensional order-preserving hashing structures, for example MOLPHE[8], PLOP-Hashing [9] and [5]. The basic principle of these structures it that they map keys of several dimensions into one dimension first, using for example *z-ordering* or similar algorithms[8][18], and applying *an order preserving hashing* algorithm [23][19] afterwards.

Balanced quad-tree-like structures (overview in [21]) were developed to solve this problem. However, much of the real-world data is clustered in a few spatial areas. The dense regions' data will, when inserted, be pushed down deep in the tree structure by inserting empty nodes. This leads to excessive navigation to reach data. The BANG-file [3] and BD-tree[17] were designed to create more "compact" trees.

We present hQT* which is a novel spatial (2-dimensional) *scalable distributed* data structure, that can be viewed as a hybrid hierarchical structure, with mostly hash-access performance. It imposes a successive more dense grid on square parts (regions) of the data domain. In hQT*, we access the buckets *bottom-up* instead of — the normally used method — top-down, avoiding long path traversals for data access. Furthermore, empty buckets need not be stored, they are created when data is first inserted. hQT* adapts to skewed and clustered data, typical for spatial data. Uniform distributions should achieve similar performance as DDH [2], but allowing for 2-dimensional data also.

hQT* is based on lessons learned from many data structures, it combines features from hB-tree [15] in that the splits are not restricted to a horizontal partitioning, and LSD-trees [4] in that subtree extractions are made. Finally, it has some similarities with the SDDS LH* [13] using hashing to enable as efficient performance for clustered data.

This is achieved through our bucket numbering schema. Each bucket, at any level in the imposed hierarchical grid, is given a unique number. These numbers are used for identifying the correct bucket. Distribution is managed similarly. The data structure is distributed by moving subtrees to new nodes. The bucket numbers identifying the subtrees are then marked as *ForwardBuckets*, and these make up the *image* of a server, clients have a subset of these entries in their image.

The rest of the paper is organized as follows. In Section 2 we give an overview of hQT*, followed by hQT* distribution, in Section 3. Section 4 describes the splitting algorithm. Measurements are presented in Section 5 and Section 6 concludes the paper.

2 hQT* Overview

General principles for SDDSs as defined in [13] apply to the hQT* data structure as well. The hQT* file is stored on *server nodes* (computers), and the applications access the data from *client nodes*. A server is assumed to be continuously available for the clients, but the clients are autonomous. Clients are not continuously available for access, may be off-line for longer periods.

The clients use an *image* for addressing data on the servers. This image can be *outdated*, causing the client to make *addressing errors*. Servers receiving such a request *forward* the query towards the correct server using its own image.

A file consists of records that reside in *buckets*. For the algorithm the relevant part is the *key*, which identifies a record and is used to locate the record. Buckets reside in main-memory (RAM) at servers. Each bucket has a fixed capacity of records. Similarly, servers have limited capacity, too. Each server keeps data for a number of different spatial areas (regions). For each area several local buckets are kept in main memory, virtually arranged in a (quad) tree hierarchy, using a numbering schema. The subtrees of hQT* stored at one server are there seen as root-trees. Each such tree can be uniquely identified by the number of the root-node's corresponding bucket. Each server keeps a mapping for the subtrees (buckets) it moved to another server giving the network address.

2.1 Records

The records in hQT* store the spatial coordinate x, y and the associated attributes $a_1, ...a_n$. The layout of the record is shown in Figure 1. For efficiency the calculated pseudokey p is stored as well, avoid-

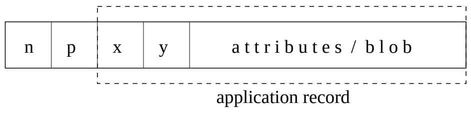


Figure 1: The Record Structure.

ing unnecessary recalculations. during reorganization. Additionally, a pointer n is used to group records together into a link list to create buckets. The application data is kept in the tail of the hQT* record for efficient application access. Typically the capacity of such a bucket is set ≤ 10 elements.

2.2 Pseudokey Construction

We map the 2-dimensional keys into a pseudokey, a bitstring of fixed length. For the current implementation of hQT* we construct the pseudokey as follows. The 2-dimensional coordinate is mapped to one bitstring by interleaving the bits of the bit-representation of the coordinate data (x, y). For example let X and Y be bit vectors of length 4 forming the pseudokey P of length 8. If $X = (x_3, x_2, x_1, x_0), Y = (y_3, y_2, y_1, y_0)$ the pseudokey is then $P = (x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3)$. As can be noted P is the reversed interleaved bitstring. This is to simplify low-level bit-programming, most algorithms like LH first consider the lowest ordered bits, and so do we.

2.3 Bucket Numbering

Buckets are numbered uniquely using the pseudokey and the level. Briefly, the idea is to impose a grid on each level of a quad-tree, using regular decomposition, such that all grids align with each other when superimposed. For hQT* any "space-filling curve" can be used, z-ordering of the buckets is shown in Figure 2. Our pseudokey use a reversed bitstring yielding a slightly more visually complicated numbering, but the principle is the same. Each layer is offsetted with the total number of preceding layers buckets. All buckets at all levels have a unique number. A bucket's number is calculated by first extracting the appropriate number of bits from the pseudokey, and then adding the offset stored in a table. As can be seen from the first 3 grid layers, in Figure 2, for level=0, the offset is 0; level=1, offset=1; level=2, offset=5.

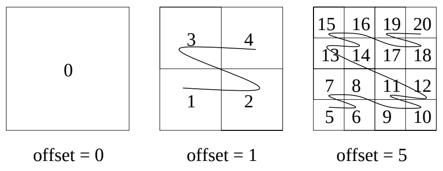


Figure 2: An offsetted space-filling curve — first 3 layers.

Navigation in the virtual tree is supported by functions that calculates *parent*, and *children* bucket number. In-expensive table lookups are used to make the mappings efficient.

2.4 Addressing

In many SDDSs (LH*, RP*) there are usually two different types of addressing algorithms, one for clients, and one for servers. The client calculates the address where it *believes* the information resides, and the server that receives a request *checks* whether the data is local or has moved due to a split. These calculations are, of course, similar. In hQT* we use the same algorithm and data structure for both client and server address calculation, the difference being that the client, conceptually does not store any data[1].

We start with point queries to address local buckets. This then easily generalizes to distributed bucket (server-nodes) accesses.

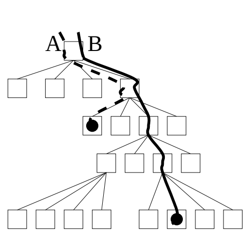
2.4.1 Local Point Queries

Given a point in 2-dimensional space, one way to find its associated data is to follow a path from the top-node, shown schematically in Figure 3a for two cases; A and B. A lies close to the root, and B is found further down the tree. However, this has been identified as an expensive operation, because clustered data will be pushed down down in the quad-tree. In a distributed setting, this is even worse due to communication resulting in hot-spots.

In hQT* we start from the bottom (highest numbered existing level[2]) of the tree style structure, as shown in Figure 3b. Here we see that we get a direct

[1]In hQT*, caching of data could be realized through letting the client store data, too.

[2]Another level could be used as a starting point, but then distributed addressing would be a bit more complicated.



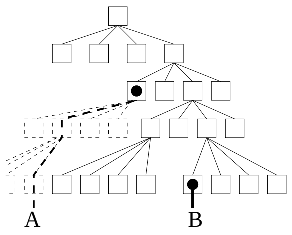


Figure 3: Navigation in a) Quad-Tree b) hQT*

hit for B from our calculations, but for A we have to walk up using the parent function — towards the root — through nonexisting buckets (dotted in the figure), until an existing bucket is found. Probing is done locally, using hash-table lookups. This allows for hash access to highly dense areas on the lowest levels, giving direct hits with minimal overhead, while less populated areas require local probes.

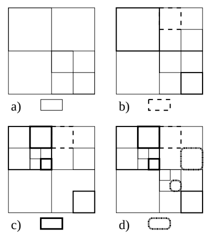
2.4.2 Local Region Queries

Rectangular region queries in hQT* use the implicit tree-structure among the buckets. Initially, the algorithm search for a local bucket that fully contains the queried region, in worst case giving the top-node, i.e. the bucket numbered 0. The subtree below is investigated, using the implicit tree-structure in hQT*. Each subtree, is first tested if it is contained, or overlaps with the region in question. All the appropriate local buckets are visited by traversing the implicit tree.

2.5 File Growth

Initially, when the hQT* file is created, only one server is used. Later, then this server's capacity is exceeded, i.e. the server is *overloaded* and *split*. Figure 4, left, shows a sequence of splits starting with the first split in Figure 4a. Successive splits, each moving data into a new server is shown through Figure 4b to Figure 4c. The first time 2 subtrees are moved, then second time only one, third (c) 2, and last (d) again 2 subtrees. The squares do not overlap, other by recursive decomposition.

Generally, a split is chosen in such a way that we minimize the number of squares (subtrees) to move, they have as large coverage as possible, and it should move half of the records (load). For now we just assume the existence of such a splitting algorithm, in the next chapter we will describe our splitting algorithm in detail.



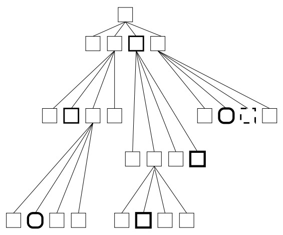


Figure 4: Left: hQT* file key space partitioning by 4 successive splits. Right: The equivalent quad-tree.

3 Distribution in hQT*

In hQT* both the clients *and* the servers use an *image*. The image stores the mapping from a bucket number to the actual bucket, or a *ForwardBucket* that stores forwarding information. Using an array for the mapping would fail for skewed data. Instead we store the buckets in a has-based structure. To test the existence, or to retrieve a bucket an inexpensive hash table lookup suffices.

3.1 Distribution (ForwardBuckets)

To handle distribution in hQT*, we introduce the *ForwardBucket*. A ForwardBucket is a replacement for a subtree that has been moved from a server, it replaces the removed subtree at its root. The spatial coverage of the ForwardBucket is identical to that of the moved tree. The ForwardBucket is associated with the address of the server where the data was moved.

3.2 Distributed Point queries

In a distributed setting, the client is first locally searched using the local hQT* addressing schema, resulting in one ForwardBucket instead of a leaf bucket. The operation is then *forwarded* to the node associated with the ForwardBucket. That (server) node is then searched, resulting in either a local bucket if the data is stored at that server or another ForwardBucket, in which case the request is again forwarded. Eventually, the correct server-node is found that stores the data in a local bucket.

Noticeable is that clients and servers use identical addressing operations, the only difference is that only servers stores buckets with data. In practice, hQT* client addressing can be seen as a single forwarding operation.

3.3 Distributed Region Queries

Distributed region search is first performed locally in the client, as earlier described. Whenever a ForwardBucket is found by the client, it is noted in a list. This list serves to forward our query to the associated servers, these servers are also given the list. These servers then repeat the same search as the initial server, processing their locally found records. Discovered ForwardBuckets are again noted. When all appropriate local records have been processed, the server sends a *finished-message* to the original server with the list of ForwardBuckets/servers where to it then will forward the query, excluding servers already identified at the original server. These servers in turn, perform the same operations till there are no more servers to forward to. The initial server keeps track of all servers it awaits responses from. Whenever, a finished-message is received, the corresponding ForwardBucket is removed from the list, and discovered ForwardBuckets on the remote server are added. The query is finished when this list is empty. This algorithm can be varied by requiring the forwarding-servers to collect the results themselves and then send the results back. In RP*[14], broadcast and multicast messages are considered. When broadcasts and multicast are available they can be used with similar performance.

Clients perform the query in exactly the same way as the servers, with the only difference that clients do not store records.

3.3.1 Forwarding: Image Adjust Messages

A client operation forwarded by a server incurs an overhead of extra messages. This is inevitable. However, it is crucial that an SDDS inhibits the client to repeat the same addressing mistake. Clients are therefore updated using *Image Adjustment Messages* (IAMs)[13]. An IAM contains information that improves the image of the mal-addressing client. In LH*, this is an *extremely efficient* procedure, at most 3 forward messages are needed[13]. This is achieved through its strict linearization of buckets, which indirectly inform the client of the existence of other servers.

In a tree data structure this is less so, but instead they cope with non-uniform distributed data better. In Section 5.1 we will see that different client (and server) image updating schemas can improve the performance substantially at the cost of more update messages.

3.4 IAM Policies

As explained earlier, a forwarded request yields an Image Adjust Message (IAM), this makes sure that the client does not repeat the *same* addressing error. Naively, only the client needs to be updated. However, as noted in RP*, DDH, and others, this puts a large load on the first server, that repeatedly has to update the same client for different mistakes. The ultimate solution is to let the servers be updated by IAMs too when their forwarded request is again forwarded by another server. Below we present different strategies used to investigate different aspects of image updating policies. The different strategies investigated are:

- ONLYCLIENT: only clients are updated
- FORWARDERS: all clients & forwarders are updated
- BROTHERS: as FORWARDERS, but the IAM also include all existing sibling[3] ForwardBuckets.
- UPDATE/FORWARDERS: "update" walks up the tree at each server that forwards, registering all ForwardBuckets unknown to the client. This is intended to give better load balance, by making servers share the relevant information to clients earlier.
- UPDATE/BROTHERS: as the previous one but with BROTHERS instead.

Naively, every server could send updates to *all* servers that participate in the current operation. In practice, the final server sends the IAM to the client directly, possible *piggybacked* on the reply message. Then it would back-trace the path the client's request was forwarded, updating these servers. The benefit of updating these servers is their decreased load by fewer mal-addressed client-requests.

[3]A sibling of a bucket is another bucket with the same parent.

4 Server Splitting

Unlike LH* [13], LH*LH [7] [2], and other hash-based data structures, our server splitting performs well also for splitting skewed data. LH* decides the split by a linearization of which buckets to create, which easily creates problems with unskewed data. Among the non-hashed based SDDSs; RP* [14], for instance, uses a simple interval division at the median value, but the efficiency of splitting and how to manage the locally stored records as such is not addressed.

4.1 hQT* Splitting

In hQT* a server is split by a simplistic, but yet effective algorithm that we call *Dissection Splitting*. In most cases we achieve a near 50% split. The best split is characterized by three properties. First, it should be as close to 50% as possible. Second, it should have as large a *coverage* as possible. Third, the number of identified squares should be as low as possible. Even if these properties seem to contradict, in most cases the expected number of selected subtrees is below 2, and seldom over 3, which means that every split mostly generates two new Forward-Buckets, and two subtrees are moved respectively. These subtrees may be rooted on different levels in the tree, but are not overlapping.

4.2 Dissection Splitting Algorithm

A server splits when it is *overloaded*, by dissecting the forest of *root-trees* stored at that server. The first server contains only one root-tree, covering the whole spatial domain of the data structure. A split of a server is the process of selecting a subset of (sub)trees to move. The subset selected will contain roughly half of the records (load). The *weight* of a tree is defined as the ratio between the number of records contained in the tree compared to the number of records at the server, usually expressed in percent(%). Dissecting a server involves "opening up" the quad-tree by decomposing it into its subtrees.

The algorithm, shown in Figure 5, works as follows. Insert all roots of the server to be split into the current working set; prune all below a certain *threshold*. Try all combinations; store the best combination. If there is a subset that is inside our allowed split range, $[50 - maxdiff, 50 + maxdiff]\%$, then the algorithm terminates. If the solution is not *GoodEnough*, we replace the heaviest tree in the working set with its children, and start over in the loop, till the algorithm terminates.

```
proc Dissection_Splitting(server) ≡
  $L := \{root\text{-}trees\ stored\ in\ server\}$;
  $S := nil$;
  while ($\neg GoodEnough(S)$) do
      if ($S \neq nil$)
         $B :=$ remove heaviest from $L$;
         $L := children(B) \cup L$; fi
      $L := \{r : weight(r) > Threshold\}$;
      for $\forall P \subseteq L$ do
         if ($BetterThan(P, S)$)
            $S := P$; fi od od.
proc GoodEnough(s) ≡
  $abs(Weight(S) - 50\%) \leq maxdiff$; .
proc BetterThan(a, b) ≡
  $GoodEnough(a) \wedge ||a|| < ||b||$; .
```

Figure 5: Split Dissection Algorithm

There are mainly two parameters that control the algorithm. First, the *Threshold* which tells us what sizes of trees to remove from the working set. Second, the *maxdiff* – the maximum deviation from 50% that we find *GoodEnough*. A solution is considered *BetterThan* than another solution if the number of subtrees (S) identified to move is fewer.

Experiments show that at most 4 trees are chosen, the mean value of number of trees is 1.8. Further on, we observe that setting *maxdiff* = *Threshold* performs well.

5 Measurements

Our measurements show the performance of hQT* in different settings. Scalability is shown using 456 servers in one experiment, and 3700 servers in another. We investigate the efficiency of different IAM policies, since this is a major concern for tree structured SDDSs. An efficient policy is vital for good load balancing among the servers. Another concern is how hQT* is affected by the data input order. For example loading data in the "wrong" order can cause many structures to degenerated[11]. We display results showing that hQT* automatically redistributes the load in case of ordered inserts.

5.1 Efficiency of IAM Policies

In line with other SDDS evaluations [13][2] we present performance figures measuring the overhead of forwarding. We show the performance of different IAM policies, in the end choosing the most optimal and load-balancing policy for hQT*.

The experiments randomly generate 262,144 data points (coordinates) in the spatial domain. They are *evenly distributed* over the key domain ($[0, 65535]$). The mean number of messages over a series of clients is measured. One client is used at a time in these tests. The activeness of a client determines the overhead, which is tested by *restarting* the client, with a probability P after every insert, we show the value $1/P$, roughly indicating the lifetime of a client.

During the experiments the file grows from one server to several by splitting when the servers are overloaded, reaching 3700 servers in the first experiment, and 456 in the second.

The acceptance interval for a satisfactory server split is set to a weight of $[45, 55]\%$, and the lightest subtree considered is set to 5% accordingly.

Update messages for clients and servers are not counted directly, instead the forwarding overhead is shown. There are several reasons for this. There are several differently efficient ways to implement each strategy, delayed update of servers from servers, etc. For example, in the OnlyClient option, there is either a single message per forwarding, in which case the displayed values can be recalculated as $new = (old - 1) * 2 + 1$, or the final server sends back an update message to the client, yielding considerable fewer messages. Other strategies, trace the updates back to the client from the final server through all the servers that did forward the request, in the end incurring the double overhead (new).

5.1.1 3700 Servers

First we investigate, in a scenario where each server can hold at most 100 elements. This gives approximately 3700 servers[4]. The values presented in the Table (below) are the average number of messages totally used for inserting data.

1/P	OnlyClient	Forwarders	Brothers	Update/Forw.	Update/Bro.
100	3.98	1.98	1.90	1.94	1.81
1 000	2.53	1.73	1.53	1.63	1.44
10 000	1.46	1.27	1.16	1.23	1.15
100 000	1.12	1.08	1.05	1.08	1.05

When a client inserts 100 elements in a file distributed over 3700 servers, it is likely to access around 100 servers. In the table the best strategy Update/Bro. gives only $1.81 - 1 = 0.81$ extra messages at an average, indicating the efficiency of SDDSs addressing and updating schemas.

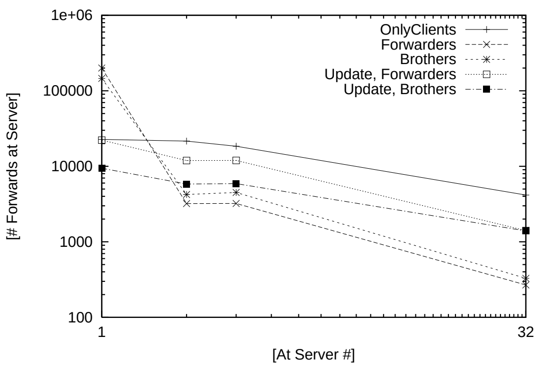
At a first glance, one would expect the OnlyClient strategy to give the worst overhead, furthermore that Brothers does not seem to give much improvement over Forwarders. However, this is an illusion. In Figure 6, we show the number of forwardings that each of the servers numbered 1, 2, 3, and 31 had to perform. In this run the probability (P) of a new client is 1/100, and the same amount of data is entered.

For OnlyClients, server 1 needs to perform nearly 23,000 inserts. But worse are Forwarders and Brothers that need nearly 200,000 and 146,000 forwards for the same data. Now, three candidates remain, depicted in Figure 6, OnlyClient, Update/Forwarders and Update/Brothers. The first two candidates both incur quite a few forwards for the first server, but Update/Forwarders wins for the remaining servers, in total reducing the number of forwards. However, Update/Brothers is the clear overall winner, since the work of forwarding is further reduced to less than half for the first server, sharing the load with the rest of the servers. This is achieved through its faster client and server image updating schema.

5.1.2 456 Servers

In the second scenario, each server can at most hold 1000 elements. This results in 456 servers for similarly generated data.

[4]Obviously, 3700 servers is a very high and unlikely number, full data availability will not be feasible, but it shows the good characteristics of the data structure going in an extreme.



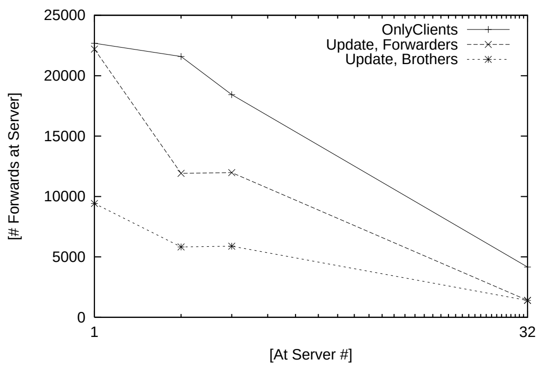


Figure 6: Different strategies: Left) Load on servers 1,2,3, and 31. Right) The 3 best strategies

1/P	OnlyClient	Forwarders	Brothers	Update/Forw.	Update/Bro.
100	2.30	1.60	1.39	1.51	1.32
1 000	1.32	1.17	1.07	1.14	1.06
10 000	1.05	1.03	1.01	1.01	1.01
100 000	1.01	1.01	1.00	1.01	1.00

Apparently fewer servers incur less messaging, comparing with the 3700 servers in the previous scenario. Again Update/Brother is clearly the winner.

5.2 Server Load Distribution

One major problem for many data structures, is that they assume that data is inserted unordered (random). Ordered input leaves these structures unbalanced with substantially reduced performance. To assess the performance of hQT* in very skewed distribution we use regional point data of the SEQUOIA-benchmark [22] (file ca) . The file contains 62,584 California place names. The data is skewed with several dense regions (cities) with many points, leaving other areas almost empty.

In the first experiment, we insert the data in the order it occurs in the file, sorted on the names of the places. In the second experiment we insert the data sorted first on the X and then on the Y coordinate values, resulting in a totally different input order.

By first studying how the server 0 splits for different distributions, we can assess that the load balancing is efficient through observing the number of regions server 0 forwards queries to. Second, a common goal for data structures is to linearize the splitting so that the splits are evenly distributed over the time that data is inserted, effectively spreading the cost of redistribution to occur evenly distributed over time.

91 servers were created for the unordered inserts, and 101 servers using ordered inserts. Servers were split when their local capacity of 1000 records is reached.

For the unordered set server 0 was split 6 times giving 11 regions (ForwardBuckets), and for the ordered set 23 times giving 39 regions[5]. However, in the ordered case, most of these areas were moved when the structure evolved, in the end giving it the responsibility of only 13 regions, effectively restructuring and *balancing* the distribution tree(forwarding structure). This is explained by viewing the number of ForwardBuckets that are *contained* in the subtrees which is to be moved. During a split, when a subtree is moved, contained ForwardBuckets from previous splits are also moved. In effect, the responsibility of these regions is given away to a new server with larger spatial coverage.

Viewed at a distance, one notices that hQT* splits top-down for unordered inserts, in a hierarchical manner, but for ordered inserts the splits partly occurs bottom-up, due to their clustered occurrence in the input stream. However, since splits

[5]The number of non-overlapping regions a server holds is an indication of how much forwarding it has to perform.

near the root occur later, they include already distributed domains, yielding a balanced tree in the end anyway. In the plots in Figure 7, we study the

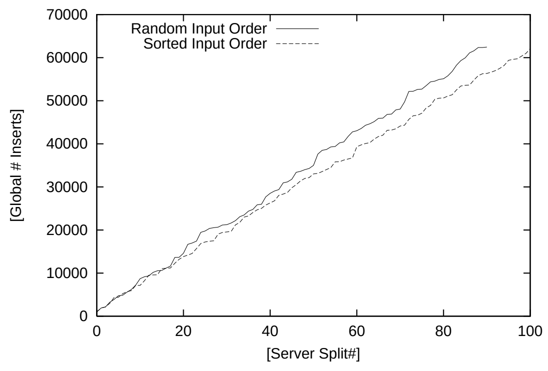


Figure 7: Split distribution over "time".

number of splits in respect to the number of inserts. The two curves show the ordered and unordered inserts, respectively. The server splits are shown on the X-axis, and the global number of inserts on the Y-axis. Even if the "Random Input Order" grows slightly faster, being more eager to split, both of them show quite a nice spread of the server splits over time.

5.3 Discussion

We have investigated several strategies for IAM updating of the clients (and servers), measuring the forwarding cost. One strategy, UPDATE/BROTHERS shows excellent performance compared to the others. The extensive measurements show the messaging overhead to be low, using a client to insert 10,000 records on 456 servers, incur a message overhead of 1% and 6% for 1,000 records. Further on, it allows for faster client image adjustments, distributing the access load evenly on the servers.

We show that the file can be scaled up to 3700 servers with reasonable added messaging cost. For example, a client inserting 10,000 records, incurs an overhead of 15% while being updated knowledge of 3700 additional servers. Comparing to a naive solution where only the client is updated, which gives an overhead of 46%. For more active clients (100,000) the overhead is down to 5%.

hQT* automatically adjusts the tree structure when ordered data (clustered in spatial areas) are inserted. This is achieved by our Dissection Splitting algorithm that first considers distributing larger spatial areas. So, for a skewed server splitting order the responsibilities of spatial areas are reassigned to different nodes.

Studying how server splits occur over the time, one can observe a near linear function, even for ordered data inserts, achieving results comparable to linearizing hashing structures.

6 Conclusions

We have shown that hQT* is a well-behaving Scalable Distributed Data Structure. It allows for spatial point data insertions and point and region retrieval. hQT* is a 2-dimensional order preserving hashing structure. It is a complete solution, in that it also stores and manages the local storage. Inserts can be performed in near 1 message, and point data access in near 2 messages. Using a tenfold more servers, the same client still achieves similar performance (the added cost is 5% for a client inserting 100, 000 records).

We have investigated different IAM updating strategies, choosing a strategy that updates the images on both the servers and clients. The chosen strategy allows for a very low messaging overhead for active clients, 6% overhead for a reasonable active client, and moderate overhead for less active clients.

We have shown that an hQT* file can scale to thousands of servers, still giving acceptable performance. Increasing the number of servers from a few hundred to ten times as many does not impose an increase in the overhead by ten, but substantially less, depending on how active the clients are.

In our experiments, the server splits occur evenly distributed in time, even for ordered inserts of data. Furthermore, it is shown that the tree-structure automatically restructures on spatially clustered inserts in time.

In the future we plan to extend hQT* to allow for n-dimensional data allowing for Data mining storage and queries. Extended objects are also a challenge for SDDSs. Further on, database query processing over SDDSs, a currently progressing project is underway to incorporate the hQT* data structure.

References

[1] D. Culler. NOW: Towards Everyday Supercomputing on a Network of Workstations. Technical report, EECS Technical Reports UC Berkeley, 1994.

[2] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Procedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, 1993.

[3] Michael Freeston. The BANG File: A New Kind of Grid File. In *ACM SIGMOD: Special Interest Group on Management of Data 1987*, pages 260–269, San Fransisco, California, December 1987.

[4] Andreas Henrich, Hans-Werner Six, and Peter Widmayer. The LSD tree: Spatial Access to Multidimensional Point and Nonpoint Objects. In *Fifteenth International Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989.

[5] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. Globally Order Preserving Multidimensional Linear Hashing. In *ICDE: Fourth International Conference on Data Engineering*, pages 572–579, Los Angeles, California, 1988.

[6] Jonas S Karlsson. *A Scalable Data Structure for a Parallel Data Server.* Licentiate thesis no. 609, Department of Computer Science and Information Science, Linköping University, 1997.

[7] Jonas S Karlsson, Witold Litwin, and Tore Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. In *Advances in Database Technology — EDBT'96*, pages 573–591, Avignon, France, March 1996. Springer.

[8] H.-P. Krigel and B. Seeger. Multidimensional Order Preserving Linear Hashing with Partial Expansions. In *International Conference on Database Theory*, pages 203–220, Rome, 1986.

[9] H.-P. Krigel and B. Seeger. PLOP-Hashing: A Grid File without Directory. In *ICDE: Fourth International Conference on Data Engineering*, pages 369–376, Los Angeles, California, 1988.

[10] Birgitte Kröll and Peter Widmayer. Distributing a Search Tree Among a Growing Number of Processors. In *ACM SIGMOD Conference on the Management of Data.*

[11] Birgitte Kröll and Peter Widmayer. Balanced Distributed Search Trees Do Not Exists. In *WADS Conference*, Mineapolis, 1995.

[12] W. Litwin and M-A Neimat. k-RP*: A Family of High Performance Multi-attribute Scalable Distributed Data Structure. In *IEEE International Conference on Parallel and Distributed Systems, PDIS-96*, December 1996.

[13] W. Litwin, M-A Neimat, and D. Schneider. LH*: Linear hashing for distributed files. In *ACM-SIGMOD International Conference On Management of Data*, May 1993.

[14] W. Litwin, M-A Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Procedings of VLDB*, 1994.

[15] David B. Lomet and Betty Salzberg. hB-tree: A Robust Multi-Attribute Search Structure. In *Fifth International Conference on Data Engineering*, Los Angeles, California, February 1989.

[16] Jörg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, March 1984.

[17] Yutaka Ohsawa and Masao Sakauchi. The BD-tree — a new-dimensional data structure with highly efficient dynamic characteristics. Information Processing 83, 1983. North-Holland, Amsterdam, 539-544.

[18] Ekow J. Otoo. A Mapping Function for the Directory of a Multidimensional Extendible Hashing. In *Tenth International Conference on Very Large Data Bases*, pages 493–506, Singapore, 1984.

[19] Ekow J. Otoo. Linearizing the Directory Growth in Order Preserving Extendible Hashing. In *ICDE: Fourth International Conference on Data Engineering*, Los Angeles, California, 1988.

[20] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Number ISBN 0-13-715681-2. Prentice Hall, 1991.

[21] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* January 1994 edition, 1989.

[22] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Storage Benchmark. In *19th ACM SIGMOD Conference on the Management of Data*, Washington DC, USA, May 1993.

[23] Markku Tamminen. Order Preserving Extendible Hashing and Bucket Tries. *BIT*, 21(4):419–435, 1981.

[24] Andrew S. Tanenbaum. *Distributed Operating Systems.* 1995.

[25] R. Wingralek, Y. Breitbart, and G. Weikum. Distributed file organisation with scalable cost/performance. In *ACM-SIGMOD International Conference On Management of Data*, May 1994.