

## **IH\* : A New Hash-Based Multidimensional SDDS**

**Djelloul BOUKHELEF**

Post-graduate student (*Computer Systems*)  
boukhelef\_dj@yahoo.fr  
d\_boukhelef@ini.dz

**Djamel-Eddine ZEGOUR**

Maître de Conférence  
d\_zegour@ini.dz

**Institut National d'Informatique** (INI, *Oued-Smar*, Alger)

Classic one-dimensional data structures are very important. Nevertheless, the new data management systems (CAD, Web,...) need new access methods that support efficiently spatial operations on multi-attributes (*multidimensional*) data files.

Scalable Distributed Data Structures (SDDS) is a new class of data structures conceived specially for multicomputers: collection of PCs and workstations interconnected by a high speed network (Ethernet, ATM, Token Ring,...). Data of an SDDS are stored in the distributed memory of a multicomputer; which permits to implement large files and allows a very fast access time to the data comparatively to those stored on physical discs. These characteristics give the SDDS processing performances considerably higher than those of classic data structures.

Nowadays, several SDDS are developed such as: LH\* (*extension of LH to distributed environments* [KLR96,LNS93,LNS96]); RP\* (*scalable distributed version of the B-trees* [LNS94]); k-RP\* (*multi-attributes RP\**); k-DRT, ...

The SDDS presented in this paper named IH\*, is the first scalable distributed data structure based on multidimensional linear hashing (IH). IH\* constitutes an extension of the interpolation-based hashing of Burkhard [Bur83] to distributed environments and also, an introduction of the order and spatial aspects to Litwin's SDDS LH\* [LNS96]. Its main objective is to provide a scalable distributed multi-attributes file which preserves the order of records.

An IH\* file can start on a single machine (*server*) and extends dynamically through insertion over an infinite number of machines. This file is accessed simultaneously by several clients distributed on the multicomputer.

IH\* supports the simple and spatial operations (*insert, delete*) and manages efficiently exact and partial match queries.

### **1. IH\* FILE STRUCTURE**

An IH\* file is a set of records identified by a  $d$ -dimensional array key  $k = (k_1, k_2, \dots, k_d)$  and a non-key part. Records are stored in buckets of size  $b$  ( $b > 1$ ). Every bucket represents a region in the basic IH scheme and is stored in the multicomputer's main memory. It can also be stored on the disc. Buckets are numbered from 0 to M; where M is the total number of buckets present in the file. These numbers can be considered as the logical addresses of the buckets.

As for the LH\*, passing from IH to IH\* consists of storing each file bucket on a single server. Bucket management information (file ID, Dimension, bucket level ( $j$ ),...) are stored at the header of each bucket.

An SDDS client maintains a local image of the file composed of two integers ( $i$ ) and ( $n$ ) : where ( $i$ ) is the presumed value of the file level ( $j$ ), and ( $n$ ) is the position of ( $n$ ) ( $n$  points to the next server to be splitted). Initially ( $i=0$  ,  $n'=0$ ).

The exact file parameters ( $i,n$ ) are maintained in a particular site, called coordinator site (CoS). Other clients' images are cached copies (*can be out of date*) of the IH\* file image. In addition, the (CoS) manages the file splitting and shrinking. It maintains also the complete physical addresses table (PAT) of servers that can take part of the storage of the file buckets.

### **2. HASH FUNCTION**

When all the buckets have the same level, IH hash function ( $h$ ) [Bur83] is used to calculate the address of the bucket relative to a record  $k=(k_1,k_2,\dots,k_d)$ . Under splitting, buckets having different levels can coexist in the same file ( $j=i+1$  for splitted or recently created buckets,  $j=i$  for unsplitted buckets). In this case, the algorithm (*Access*) [Bur83] is used to

calculate for  $k$  the address ( $a$ ) of the corresponding bucket ( $a=0,1,...,2^i+n-1$ ).

To access an IH\* file, a client applies (`Access`) to his local image ( $i,n$ ). This technique can lead to addressing errors; which are not due to the algorithm but, rather, to the incompleteness of the client image. In the following sections, we present the basic principles of our SDDS and give solutions to the problems which can be met.

### 3. FILE EXPANSION

An IH\* file behaves like an IH file. It starts with the bucket 0 ( $R_0^0$ ) and (`Access`) uses in this case ( $n=0, i=0$ ). When bucket 0 overflows, it is splitted: bucket 1 ( $R_1^1$ ) is therefore created. Now (`Access`) uses ( $n=0, i=1$ ). At the next collision, bucket 0 ( $R_0^1$ ) is again splitted creating bucket 2 ( $R_2^2$ ). ( $n$ ) points now on bucket 1 and (`Access`) uses ( $n=1, i=1$ ).

Generally, a bucket ( $n, R_j^i, i$ ) is splitted into two buckets ( $n, R_{2j}^{i+1}, i+1$ ) and ( $n+2^i, R_{2j+1}^{i+1}, i+1$ ) having the same level and size. During this operation, at average, half of the records of bucket ( $n$ ) will be moved to the bucket ( $n+2^i$ );

### 4. SPLITTING

According to the linear hashing principle (LH and IH), the IH\* file extends through the linear splitting of the file buckets one by one and in the order. Splitting can be either controlled (*occurs when the load factor reaches a given threshold  $\tau$* ) or uncontrolled (*at every collision*). Basic IH scheme uses the algorithm (`Split`) to accomplish this task [Bur83].

**Distribution of algorithm (`Split`)** : The application of classic (`Split`) in a distributed environment leads to some problems: How to keep the coherence of the parameters maintained by the (CoS) and those maintained by the clients? To solve this problem, we have divided (`Split`) into two independent algorithms:

- *Bucket splitting* : Since the bucket to be splitted is stored at the server ( $n$ ), the initialization of the new bucket ( $n+2^i$ ) and the transfer of records towards it are ensured by the server ( $n$ ) without passing by the (CoS).
- *File parameters updating*: The (CoS) deals with the file parameters updating because they are exclusively accessed by it.

**Controlled and uncontrolled splitting** : Algorithm (`Split`) used in IH manages both controlled and uncontrolled splitting. However, this distributed manner arises an other problem: when does the (CoS) start a split operation?

*Uncontrolled split* : When the (CoS) receives a notification message from the server that underwent a collision, it sends a “*split order*” to server ( $n$ ). This last one proceeds to the splitting. As soon as it finishes, it sends a notification message to the (CoS). By receiving “*end of splitting*” message, the (CoS) calculates the new values of ( $n$ ) and ( $i$ ) and updates his (PAT).

*Controlled split* : In this mode, the problem is : how the (CoS) can know the exact load factor of the file to decide whether a split is needed or not?. The reason is that insert and delete operations do not pass by the (CoS). One solution (*deterministic*) consists of sending a notification message to the (CoS) for every insert or delete, which increases the number of messages across the network and render the (CoS) a hot-spot; which oppose the SDDS principles.

A second solution (*probabilistic*) developed by Litwin for LH\* SDDS [LNS96] permits to estimate the file load factor using the information received from the overflowing server. Let  $S$  the server that underwent a collision,  $b$  its capacity,  $x$  the number of records that it contains.  $S$  sends a “*collision*” message containing ( $b$ ) and ( $x$ ) to the (CoS). This last uses ( $b$ ) and ( $x$ ) to estimate the load factor of the file in order to decide if a splitting operation is necessary or not (load factor > threshold  $\tau$ ).

We note that our SDDS uses the same formula as LH\* to estimate the file load factor. The development of a new and optimal formula is addressed to future works.

## 5. ADDRESSING

The file buckets are manipulated simultaneously by several clients. Each client maintains a local image which is not necessarily identical to the global file image maintained by the (CoS) (*typically unknown by the other machines*). The client image is updated only when the it manipulates the file. As for LH\*, the IH\* addressing scheme relies on following three principles:

- a). Client address calculation** : The client sends the request (*insert, delete...*) relative to the record  $k=(k_1, k_2, \dots, k_d)$  to the server ( $a$ ). To calculate ( $a$ ), the client uses (`Access`) applied to  $(i', i)$  instead of  $(n, i)$ .
- b). Server address calculation** : Using (`Access`), a client can lead to an addressing error (*sends its request to an incorrect server*) due to the obsolescence of its local image. To resolve this problem, the server that receives the request has to verify if it is really the correct server. If so, it evaluates the request and returns the adequate answer to the client. Otherwise, it forwards the request to a second server. This one can, in its turn, forward the request to a third server. For LH\*, Litwin & al. proofed that the third server is the last one (*LH\* property*) : in the worst case, two “forward” messages are needed (*three buckets are visited at most*) [LNS93,LNS96]. This property remains valid for IH\*.
- c). Client image adjustment** : If an addressing error occurs, an “*image adjustment message*” (IAM) is sent to the client enabling it to adjust its local image  $(i', i)$ . The (IAM) is sent by one of the servers participating in the forwarding process. This message contains the server address ( $a$ ) and level ( $j$ ). In our SDDS, it is the first server ( $a$ ) which sends the (IAM). Our aim is to have  $(i', i)$  more close to the real file image as soon as possible. This choice guaranties that, at least, the same error will not occur since there are no new splits.

## 6. RANGE QUERIES

RP\* SDDS (*Range Partitioning*) compared to LH\* provides the better throughput for range queries. Records are not arranged in bucket using a hash function as for LH\*, but they are arranged according to their primary keys; which guaranties that a few number of servers have to process and answer a range query [Tsa00].

The IH\* SDDS takes advantage of these two SDDSs (RP\* and LH\*) since:

- a)** it preserves the order of records stored in the file. So, a limited number of servers have to answer the query;
- b)** it uses a hash function that requires only two integers to index the entire file. Contrary to RP\* where a whole B-tree is required at every client or server;
- c)** The length of a forward chain does not exceed 2: three servers are visited, in the worst case, to reach the correct server.

A parallel query  $Q$  is an operation (*update or select records according to some constraints*) sent by a client to several servers of an IH\* file ( $F$ ). The executions of  $Q$  by the file servers are mutually independent. A server replies only if it receives a copy of  $Q$ . If  $Q$  is a search operation, the answer contains the records satisfying  $Q$ . Query  $Q$  can ask for an answer from each server even though the answer is empty. This can be useful for the client to know if all the servers have answered its query.

One particularity of LH\* and IH\* is that a client knows only the file extensions viewed by its local image. If a server is created after the last update of this image, their descendants will be unknown by the client. This situation must not prevent

those servers to receive  $Q$ .

On the other hand, records resulting from parallel query arrive from several servers due to the distributed storage of  $F$ . The problem of termination of parallel queries arises now: The client can stop the reception process if all the answers are successfully received. If there are missing answers, the client sends back the query to the missing servers. To do this, the client has two strategies: the first method (*probabilistic*) uses a timeout  $T$ . At its expiration, the client can stop the reception. In the second one (*deterministic*), the client stops the reception when it verifies that the union of the received answers covers the scope of  $Q$ . For these strategies, restart mechanisms are envisaged to send back the query to the missing servers.

In the following sections we present the  $LH^*$  broadcast and unicast modes which can be applied easily to  $IH^*$ . we present also the  $IH^*$  unicast mode. This new mode takes advantage of the order in the file and uses our new range query portioning mechanism. As for the  $LH^*$  unicast mode,  $IH^*$  mode guaranties that:

1. only the buckets of  $F$  receive the query message;
2. servers do not need to communicate with the (CoS) ;
3. a concerned bucket receives a unique copy of  $Q$ ;
4. non concerned buckets (*according to the client image*) do not receive  $Q$ ;
5. non concerned buckets (*according to the CoS image*) do not process  $Q$ ;

Based on (3,4,5),  $IH^*$  method represents a good strategy comparatively to  $LH^*$  since it reduces the number of the servers participating in the evaluation of  $Q$ .

**a).  $LH^*$  broadcasting mode (*multicast, broadcast*) :** In this technique, the client sends a range query  $Q$  concerning  $F$  through multicast or broadcast messages to all the servers on the multicomputer. Only the servers that store buckets of  $F$  evaluate  $Q$  and send their answers to the client using TCP or UDP (*according to the answer size*) [LNS96].

*Probabilistic termination* : In this technique, only the servers that store records satisfying  $Q$  will answer. At the reception of an answer, the client resets a timeout  $T$ . When  $T$  expires, the client can stop or continue the reception process. A practical choice of  $T$  consists of tacking a delay time that minimizes the probability of answers losing. This choice depends not only on the network performances but also on the servers throughput.

**b).  $LH^*$  unicast mode** : This technique is proposed by Litwin for he  $LH^*$  [LNS96]. A query  $Q$  initiated by the client is sent to all the servers that appear in its local image through unicast messages. Each message includes the query level ( $j$ ). A server that receives  $Q$  evaluates it and send it towards its child servers (*do not appear in the client image*).

*Deterministic termination* : Server that receives  $Q$  sends to the client the selected records, its address ( $a$ ) and level ( $j$ ). On the other side, the client waits for answers arriving from several servers and follows up their evolution. It can decide to stop the reception if all the servers have answered  $Q$ . In this case, the client uses the parameters collected from these answers to update its local image. Otherwise, the client blocks the reception and resend  $Q$  to the missing servers.

## 7. $IH^*$ UNICAST MODE

$LH^*$  mode can be applied easily to  $IH^*$ . However, the  $LH^*$  strategy requires an answer from each server in order to satisfy the termination condition; which increases the traffic on the network. Furthermore, it does not take advantage of the order property since all the servers receive the query and have to send an answer (*even empty*) to the client. To avoid this

inconvenience, we have developed a new propagation method based on the IH range query subdivision mechanism. This solution aims to minimize the number of messages across the network by reducing the number of explored servers [BZ02].

- a). **Calculation of starting set A** : Client that wants to launch a parallel query  $Q$ , has to divide it into a set of separate sub-queries and determine for each one the corresponding server according to its local image  $(i, n)$ ; which represents its own vision on the file. It sends then to each server of this set the corresponding sub-query using point-to-point messages. A message holds the query level  $(j)$ , which is the server's level in the client image ( $j=i+1$  for  $2^j < a < 2^{j+1}$ ;  $j=i$  otherwise).
- b). **Query propagation by servers** : When a server receives a parallel query  $(Q, j)$ , it calculates the intersection between the region of  $Q$  and the region of the stored bucket. If it is not empty, the server sends the selected records, its address  $(a)$  and level  $(j)$  to the client. If it detects the incompleteness of the client image, it behaves like a client. It allots the region of  $Q$  into refined sub-queries and sends them to its child servers (*do not appear in the client image*). This refinement process continues until arriving to a server having no child or  $Q$  does not concern them.
- c). **Client image adjustment** : When it receives an answers, a client can refine the sending card and determine more exactly the missing servers as well as the corresponding sub-regions. This allows it to send back  $Q$  to the missing servers. Its can also use the collected parameters to adjust its local image.

## 8. CONCLUSION & FUTURE WORKS

We have presented in this paper a new multidimensional SDDS based on Burkhard's interpolation-based hashing. Our objective is to develop an client/server SDDS that supports efficiently exact and partial match queries on large spatial databases and takes advantage of multicomputers (*parallel processing and large distributed RAM*).

Our SDDS, called IH \*, supports efficiently range queries comparatively to LH\* since it minimizes the number of messages needed for its execution by reducing the number of explored servers. However, a serious work remains to perform in this field. It consists of developing a new and optimal strategy, which allows the client to receive servers' answers without blocking or losing messages.

Our future objective will be the fulfillment of an IH\* system on a real multicomputer environment, the detailed performance analysis as well as the conception of new variants of IH\* as done for other SDDSs (*without coordinator, concurrent splits, internal bucket structure, load balancing...*). Another important work is the study of height available, fault tolerant and secure variants of IH\*. Future works will be oriented to the parallel queries processing (*subdividing and collecting answers from servers, spatial and parallel joint...*) and the coupling of the IH\* SDDS with powerful DBMS (*object, object-relational*).

## 9. REFERENCES

- [Bur83] **W.A. Burkhard** - *Interpolation-based Index Maintenance*. Proc. of the 2<sup>nd</sup> Symp. On Principles of databases Systems. PODS 1983. pp.76-88.
- [BZ02] **D. Boukhelef & D.E. Zegour** - *IH\* : Hachage Linéaire Multidimensionnel Distribué et Scalable*. (Submitted to CARI'02). January 01<sup>st</sup>, 2002.
- [KLR96] **J.S. Karlsson, W. Litwin & T. Risch** - *LH\*lh: A Scalable High Performance Data Structure for Swiched Multicomputers*. Int. Conf. on Extending Database Technology, EDBT-96, Avignon, March 1996.
- [LNS93] **W. Litwin, M-A. Neimat & D. Schneider** - *LH\* : Linear Hashing for Distributed Files*. ACM-SIGMOD Int. Conf. On Management of Data, Washington D.C. p. 327-336, May, 1993.
- [LNS94] **W. Litwin, M-A. Neimat & D. Schneider** - *RP\* : A Family of Order-Preserving Scalable Distributed Data Structures*. 20<sup>th</sup> Intel. Conf. on Very Large Databases (VLDB-94), Santiago, Chile, 1994.
- [LNS96] **W. Litwin, M-A. Neimat & D. Schneider** - *LH\* : A Scalable Distributed Data Structure*. ACM-TODS. Dec. 1996
- [Tsa00] **M Tsangou** - *Mesure des performances de la scalabilité des requêtes parallèles sur les SDDS RP\**. Mémoire de DEA d'Informatique, U.Cheikh Anta Diop. April 15, 2000.