Performance Measurements of RP* : A Scalable Distributed Data Structure For Range Partitioning

Aly Wane Diène & Witold Litwin
CERIA
Université Paris 9 Dauphine
{Ali-Wane.Diene ; Witold.Litwin}@dauphine.fr

Abstract

The RP* scheme generates the scalable range partitioning. The intervals at the data servers adjust dynamically so that new servers accommodate the file growth transparently for the application. We have implemented variants of RP* on a Windows 2000 multicomputer. We have measured the performance of the system. The experiments prove high efficiency of our implementation. RP* should be of importance to future main-memory parallel DBMSs.

Key words: *Multicomputer, scalability, scalable distributed data structures, parallel processing.*

1 Introduction

A Scalable Distributed Data Structure (SDDS) dynamically partitions data over a multicomputer, [LNS93]. The data are distributed over server nodes. They are stored for processing entirely in the distributed RAM, for much faster access than to the traditional disk-based structures. The applications access data through the client nodes. The partitioning adjusts to the file growth, transparently for the application. The overflowing servers split and send the records towards new servers, dynamically appended to the file. These capabilities should interest many applications. The SDDSs should enhance in particular the scalability of parallel DBMSs. These use only static partitioning schemes at present, and disk disk-based data structures.

Several SDDS schemes are now known, [SDDS]. Some are for scalable hash partitioning, e.g., the LH*$_{LH}$ schema, [BDNL00]. Some LH* schemes are the *high-availability* SDDSs that support the unavailability of some server nodes, e.g., the LH*$_{RS}$ schema [LS99]. Other schemes, especially the RP* SDDS, provide the range partitioning [LNS94]. Finally, there are schemes for the *k*-d partitioning etc.

For every SDDS schema, the theoretical analysis has predicted its access and storage performance.

However, we believe that nothing replaces the experimental validation. A prototype SDDS management system termed SDDS-2000 is therefore being developed at CERIA, in cooperation with U. Uppsala[1] and U. Dakar[2] (ceria.dauphine.fr). It runs on Windows multicomputer. It supports at present the LH*$_{LH}$ and LH*$_{RS}$ schemes, and RP* schemes.

[1] Research group of Prof. Tore Risch, formerly with U. Linkoping (Sweden)

[2] Research group headed by Dr. S. Ndiaye and Dr. T. Seck.

Below we report on experiments with variants of the RP* scheme. These experiments, *first ever reported for this schema*, show fast access times, usually in the range of a fraction of a millisecond. These confirm the theoretical predictions in [LNS94]. The times are also orders of magnitude faster than for more traditional disk files. The measures show also good scalability of all the variants under the study.

Section 2 recalls the principles of RP* schemes. It also presents the RAM data structures used at SDDS-2000 servers. Section 3 describes the SDDS-2000 system architecture. Section 4 reports on the experiments. Section 5 analyzes the scalability of RP* files as it appears through the experiments. Section 6 contains the conclusion.

2 RP* Schemes

2.1 File Structure

An RP* file consists of records identified by primary keys, [LNS94]. The keys and records of an RP* file are completely ordered. A record consists of the key and of a non-key field(s). The records on every server are stored in memory space called *bucket.* Buckets are usually in the main memory, for fast access. The set of the keys in the bucket corresponds to an ordered partition of the key space. An interval called *bucket range* and noted $(\lambda, \Lambda]$ is associated with each bucket. The value λ is the *minimal key* and Λ is called the *maximum key* of the bucket. A record with key c is stored in the bucket with the range such that $\lambda < c \leq \Lambda$. The union of the ranges covers the key space, assumed $(-\infty,+\infty)$. The file initially consists of single bucket 0, with $\lambda = -\infty$ and $\Lambda = +\infty$.

Each bucket contains a maximum of b ($b >> 1$) records. The value b is the bucket *capacity.* When the number of records to store in a bucket exceeds b, the bucket overflows. The records with keys within the upper half of the range move to a new bucket at a new server appended to the file. Both bucket ranges are adjusted accordingly. No message is sent to clients about the split.

The client requests (queries) to the file are as usual the inserts, updates, deletions and searches based on key values. A *key* search is the search for a record with given key. A *range query* searches for all the records with keys in the range given by the query. The range query is basically multicast and buckets deliver the records in parallel. Alternatively, one traverse the relevant buckets in ascending or descending order. Finally, the *general query* searches the non-key fields. It usually scans in parallel all the buckets.

The client has the choice of two termination strategies for the range query deal with as the parallel query. With the *deterministic* termination, every server that receives the query sends its range and the records in the query range, if any. The client terminates the processing *successfully* when the union of the received ranges subsums that of the query. The client also has a timeout, to terminate *unsuccessfully* if not all answers are received within this time. With the *probabilistic* termination, only the servers with records in the query range reply. The replies must be collected within the timeout *T*. The client always terminates *successfully* after *T* expires. The choice of *T* involves the probability of losing an answer. This depends on the performance of the network, on the processing speed of the servers…

One distinguish three RP* variants in [LNS94], labelled: $RP*_N$, $RP*_C$ and $RP*_S$. An $RP*_N$ client sends the requests to the servers using multicast messages. An $RP*_C$ file is an $RP*_N$ file with a specific index on each client. The index contains a (perhaps partial) image of the partitioning. An element of the index may contain a bucket range and its address. There are also elements, symbolically noted '*' whose correspondence to buckets is (yet) unknown. Initially, for any new client, there is only one element in its index which is '*' for the whole file.

The client uses unicast messages for key based requests within a range of a bucket with known address. It multicasts the request with the key fitting the range of the '*' element. A unicast request with key *c* may reach an *incorrect* bucket such that $c \notin (\lambda, \Lambda]$. Such a bucket has split since the index element with its range on the client was created or last updated. The server that receives such an *out-of-range* request and key, multicasts the request together with its own range to all the servers. The *correct* bucket with the range $c \in (\lambda, \Lambda]$ processes the request. It then sends back to the client the *Image Adjustment Message* (IAM). Each IAM contains the range of the bucket that sends it and perhaps of the other bucket(s) that the record has visited. The client adjusts its image accordingly.

Finally, $RP*_S$ adds to $RP*_C$ an index distributed over servers, indexing all the buckets. One uses then the unicast messages for all types of requests, and the redirections among the servers, except perhaps for the range queries.

Our SDDS-2000 prototype currently supports these three RP* variants. It also supports a variant of $RP*_C$ called $RP*_{Cu}$. The client of $RP*_{Cu}$ uses only unicast messages for the key based requests. The key that does not match an index element with known bucket address is unicast to the bucket with the highest preceding range. Performance experiments reported below for RP*c concern $RP*_{Cu}$. For convenience, we denote the latter simply $RP*_C$.

2.2 Bucket Structure

The RP* buckets are supposed stored in distributed RAM. One may expect the processing time orders of magnitude faster than the traditional disk-based structures [LNS94]. The multicomputer supporting the RP* may involve very many nodes. Hence, one can also reasonably expect enough distributed RAM for even very large files. The RP* schemes originally defined leave open the internal structure for an RP* bucket. We now present our design at SDDS-2000.

We use two *Memory Mapped File* (MMF) of Windows 2000. The bucket is basically a specific kind of RAM B^+-tree, [D98], [DNB00]. We distributed the bucket storage space into three zones, Figure 1:

1. The *Header* - It includes the bucket range, the address of the index root, the bucket size, the number of records in the bucket and, , the index size.
2. The *Index* - It is a variant of B+-tree, Figure 2.
3. The *Data*. This zone contains the index leaves with the data.

One MMF stores the header and the B+-tree index. The other stores the data.

Header	B+-tree index	Data (Linked list of index leaves)

Figure 1: RP* bucket storage space.

The index is a hierarchical structure of nodes, as usual for a B^+-tree, Figure 2. In addition, we have linked the successive index nodes at the same level. This makes the index split during the bucket split more efficient. Each node contains at most *n* entries, in the (*key*, *pointer*) form. Each pointer refers to a next level node. Except for the root, each node has at least *n/2* entries. The root can contain even a single entry.

We have limited our tree to three levels, as at the figure. It suffices for very large files and our tests. The bottom *leaf headers* of our tree point to the leaves. The pointers between leaf headers make the sequential traversal faster, as in the linked B-tree. The records in the leaves are logically ordered linked lists of records. An insert adds the record at the end of those already in the bucket. A deletion is only logical. The space is reclaimed during the split or garbage collection (not implemented yet).

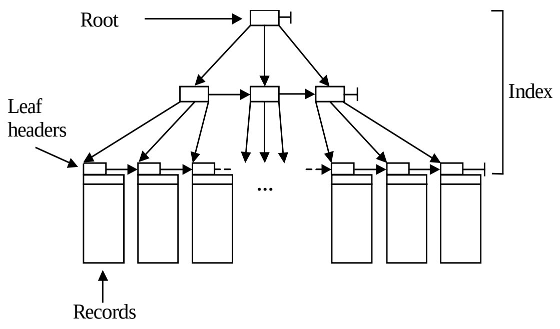

Figure 2: RP* bucket structure

3 System Architecture

SDDS-2000 system architecture is common to all schemes it supports. Below, we describe only its features relative to the scope of this work, i.e., the experimental performance analysis of RP* schemes. We present successively the capabilities of the messaging architecture, the functional architecture of the servers, and that of the clients.

3.1 Messaging

Our machines are on the Ethernet network. We use unicast, point-to-point, messages to addresses any single site. We also use the multicast, to address with single message the entire *multicast group* of sites, identified by a shared address. The transport level is managed by the Windows Sockets Interface [S96]. The sockets support the TCP/IP and UDP protocols. TCP/IP provides the reliable service connection based service. UDP works with datagrams, without connection and guarantee of the delivery in order and without losses. We use UDP protocol for short messages (< 64K) and the TCP/IP otherwise. In addition, we have a dedicated flow control protocol for requests using UDP messages in SDDS-2000, if losses are unacceptable, [D&al00].

The protocol uses a sliding window, as in the algorithm proposed by Van Jacobson for the congestion control in TCP/IP [J88]. The server sends an acknowledgement with the record key to the client for every individual request. The client has a buffer where it stores the requests sent but not yet acknowledged. The client stop sending when the buffer reaches its capacity, of 10 requests at present. The acknowledgement removes the request from the buffer. This enables a new send out. For each request sent, a timeout is automatically started. The client resends the request without the acknowledgement within the timeout.

3.2 Servers

The servers use the *multithread* processing, Figure 3. Several threads take care of the processing, asynchronously and in parallel. They communicate through queues, buffers and events.

The *ListenThread* receives the client request, and puts it into a FIFO *RequestsQueue*. It announces then the *ArrivedRequest* event and waits for next requests. The even wakes up a *WorkThread* from a pool of such threads, noted W.Thread *i* (*i* = 1,.., *N*) in Figure 3. The number of active WorkThreads depends on the server load. Each *ArrivedRequest* event, wakes up a *work thread*, if any is still available. The thread reads the next request in *Requests queue*. It identifies the operation to perform. If the request requires the flow control, the thread puts the acknowledgement in the FIFO *Ack*Q*ueue* (Ack queue in Figure 3). The *SendAck* thread asynchronously reads AckQueue and sends the acknowledgements to the clients. WorkThreads continues with the processing of the requested operation. At the end, it returns the answer to the client, if required. If there is no new *ArrivedRequest* event pending, it finally goes to sleep.

For the clients, the server is identified by its IP address and UDP port. A site can support multiple servers on different ports. A server is created empty, and listening for a bucket creation request. These requests may create several buckets from different SDDS files. The buckets are allocated using the Bucket Allocation Table (BAT). BAT contains the bucket address, size and the ID of its file.

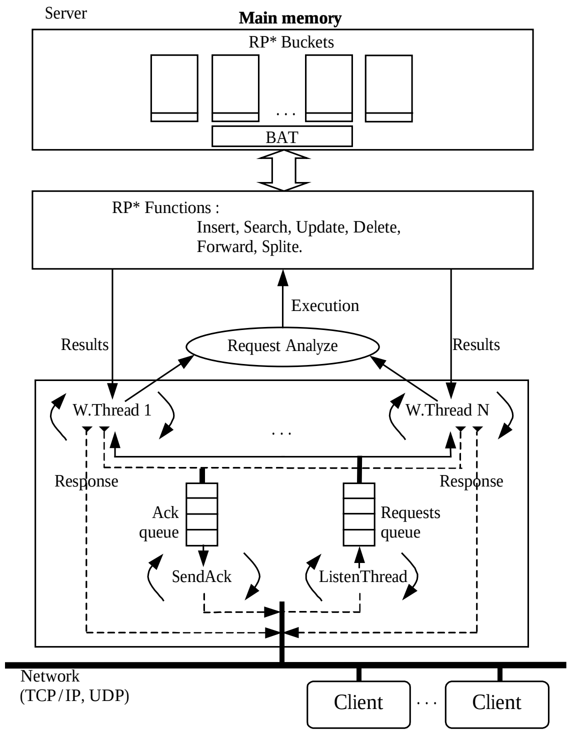

Figure 3: Server architecture.

The bucket creation request comes from a client for a new file or from a server with a bucket to split. The latter case subsumes the former so we present only that one. Let S_O the splitting server be, and let the S_i be any server among M currently started ; $1 \le i \le M$. The split at S_O follows the steps. First, S_O sends the *SplitRequest* multicast message with the requested bucket size. Each S_i that has enough space replies. S_O selects the first site S_k to reply. It opens a TCP port and sends the *ResponseAccept* unicast message to S_k requesting it to connect to S_O. S_O locates also the middle key c_m in its bucket and splits the bucket into two groups. One group, let it be G, contains records with keys $c > c_m$ and their B+-tree sub-index. The other group contains all the other records and their sub-index. S_O, sends G to S_k. It then updates its index and data zones so to remove the allocation gaps and sets the range to $[\lambda_O, c_m[$. Finally, S_k creates the bucket with the received records and range $[c_m, \Lambda_O[$.

The client requests in *RequestsQueue* received during the split are dealt with as usual once the split is over.

3.3 Clients

The RP* client architecture is structured into two modules, termed *send* and the *receive* module. They run in parallel, and are further structured as in Figure 4. The *send* module processes the application request and expedites it to the servers. Its *GetRequest* thread waits for the application request. It puts the incoming request ID (Id_Req) and that of the application (Id_App) in the RequestJournal table. Both Ids are provided by the application. Next, it processes the request and puts it in a FIFO queue. The *SendRequest* thread reads the queue. It builds the messages to the servers accordingly and sends them. For RP*$_C$, it consults the client image, to determine the IP address and the type of the message to use.

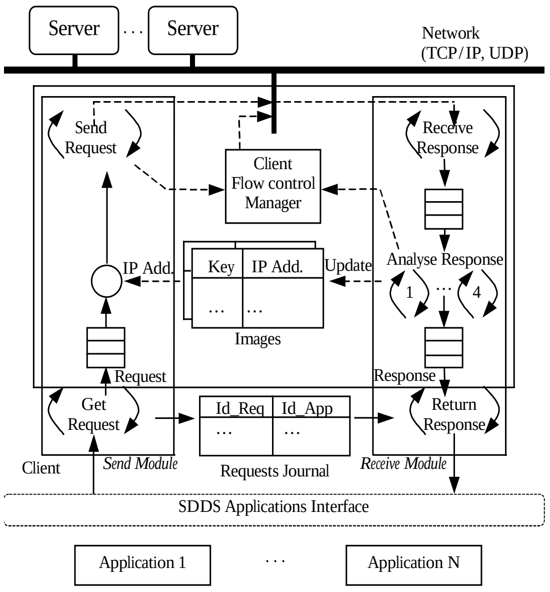

Figure 4: Client architecture

The *receive* module manages the replies from the servers. It works into two different ways. For key based requests, the *ReceiveResponse* thread awaits the replies from the servers. It inserts them into a FIFO queue, read by the *AnalyseResponse* threads. Up to four such threads may run simultaneously at present. Each *AnalyseResponse* reformats the received data for the application and possibly update the RP*$_C$ client image from the IAMs. Then, it puts data and the request id into another queue for *ReturnResponse* thread. *ReturnResponse* searches Id_App in the request journal for each Id_Req in the reply retrieved from the queue and returns the data to the application.

For a range query, the *ReceiveResponse* thread opens a TPC port before the expedition of the request to the servers by the *send* module. The relevant servers use this port to establish simultaneous TCP connections on other ports with the client. To receive the data, through each accepted server connection request, the *ReceiveResponse* thread creates a dedicated *ReceptionThread*. It keep the TCP connexion until the end of the data transfer from the server.

4 Performance Analysis

4.1 Experimental Environment

Our platform consisted of six Pentium III 700 MHz machines with four Professional Windows 2000 and two Server Windows 2000. Each site had 128 MB of RAM. The machines were linked by a 100 Mb/s Ethernet. One or two of the machines served as clients.

The message size was 180 bytes (80 bytes for the message header and 100 bytes for the record). The message for the key search had 80 bytes. The size of the sliding window for the flow control is set to 10. The keys for the experiments are random integers within some interval and without duplicates. The keys for search and insert operations are between 1 and 100.000. Those for the file creation are between 1 and 150.000. The capacity of the internal index node in the RP* bucket is set to 80 keys. That of a leaf is 100 records.

All times measured are in milliseconds (ms). The measures are collected at the clients for inserts and searches, and at the servers for the split time. The client measures the time requesting a specific acknowledgement for every *i*-the key or record sent; *i* = 1, 10000, 20000… in our case. It records the time when it receives the acknowledgements. The splitting server measures the split time as the difference between the time it starts the split and that when it gets the acknowledgement from the server of the new bucket that its creation is finished.

4.2 File creation

We measured the creation of the RP*$_C$ and of the RP*$_N$ files. The performance factors of interest were the total time, and the time per record inserted. The bucket capacity was 50.000 records. In 1st experiment, a single client inserted the 150.000 records. The file scaled up from the empty bucket to five buckets (on five servers). The communication was through UDP messaging, one per record (no bulk loading). Table 1 shows the final numerical results. Figure 5, 6 and 7 present the graphics. Each point (*x*, *y*) of a curve of the time *y* per insert, shows the total time to create the file of *x* records divided by *x*. This time measures the throughput, and should not be mistaken with the average individual insert time. The latter is measured by the average time per insert at a single server only. Losses without the flow control were negligible (< 0,01%).

RP*$_C$			
With flow control		Without flow control	
Ttl time	Time/Ins.	Ttl time	Time/Ins.
67838	0.452	45032	0.300
RP*$_N$			
With flow control		Without flow control	
Ttl time	Time/Ins.	Ttl time	Time/Ins.
69209	0.461	47798	0.319

Table 1: File creation by a single client

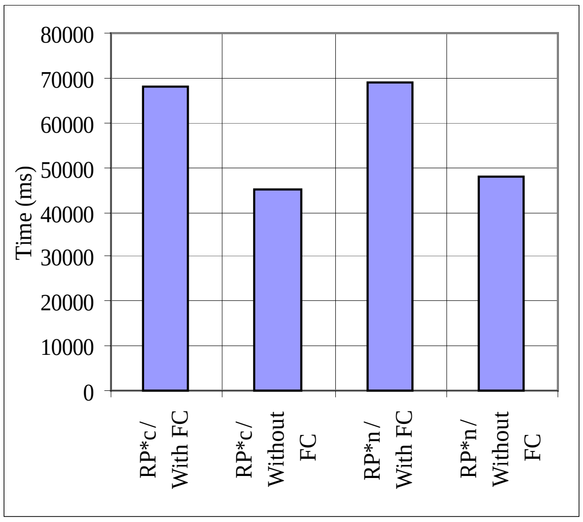

Figure 5: RP*$_C$ and RP*$_N$ file creation by one client.

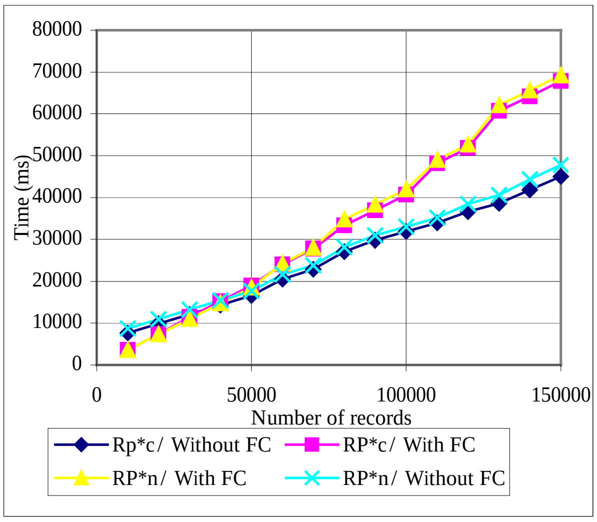

Figure 6: File creation by a single client.

All the curves become flat. The time per insert is thus valid for further scaling of the files. As one could expect, RP*$_C$ appears also slightly faster than RP*$_N$ in all the cases. The difference is of 2% with the flow control and of 6% without. These results match the intuition. They made us in addition believing that the performance of RP*$_C$ without flow control could be bound by the throughput of a single client. The processing of an insert should be indeed longer at the server than at the client. But, when the file scales over enough servers, their collective throughput should become faster than that of any single client. This could happen also for RP*$_N$, or not. The multicast sends indeed every record to every server. A record to be stored at the server requires than longer processing than that to drop. But, while these records are produced at the constant rate by the client, they scatter on more and more servers. On the other hand however, each server gets relatively more records to drop. The result of the match is unclear.

To test the conjecture we have created the file by 120.000 inserts from 2 simultaneous clients. The file scaled to four buckets, which is the limit of our configuration for two clients. Table 2 and Figure 8 and 9 present the results.

The conjecture appears true for RP*$_C$. The time per inserts decreases indeed by 7% for RP*$_C$, It does not in contrast for RP*$_N$. RP*$_C$ becomes for 2 clients notably faster than RP*$_N$, by 15%. The curves of total time are straight and those of time per insert become flat. The file scales thus again linearly and the curves may be used to predict the creation time for further scale-up.

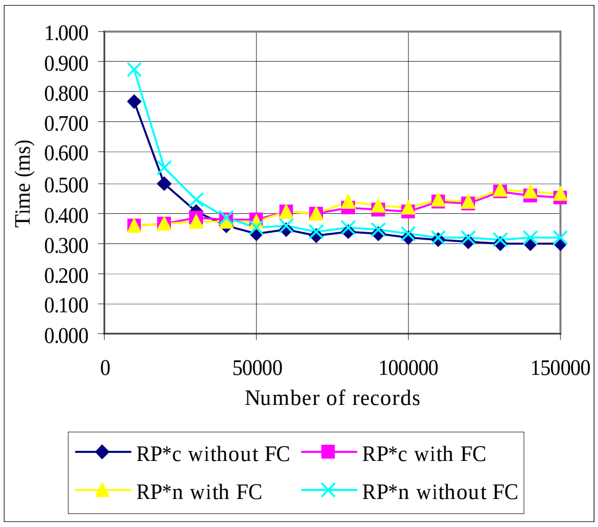

Figure 7: Insert time by a single client.

RP*$_C$		RP*$_N$	
Without flow control		Without flow control	
Ttl time	Time/Ins.	Ttl time	Time/Ins.
33553	0.279	38432	0.320

Table 2: File creation by two clients.

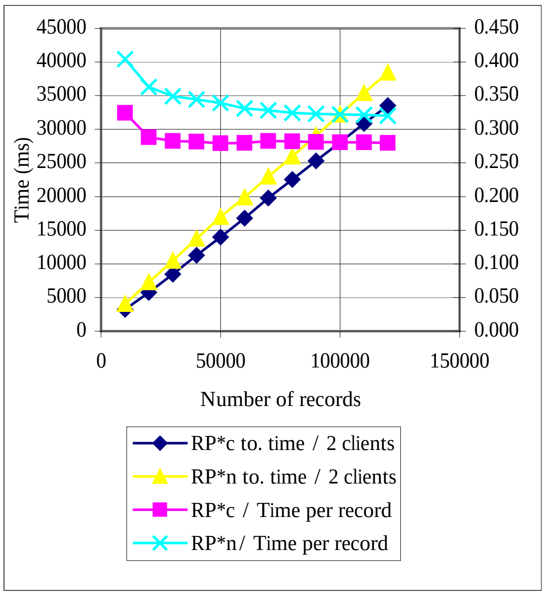

Figure 8: File creation by two clients without flow control.

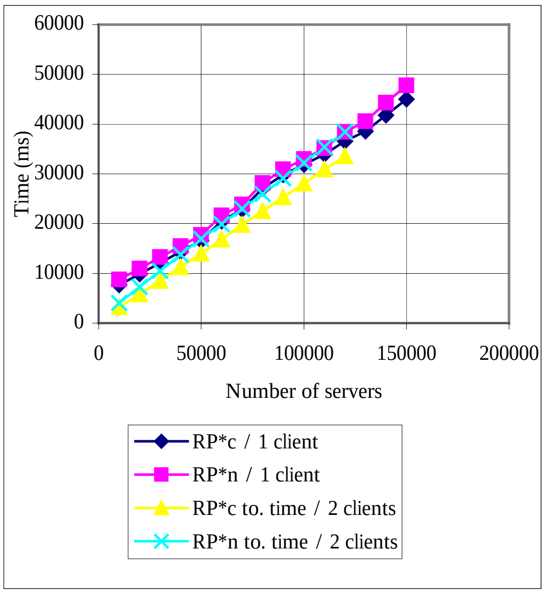

Figure 9: File creation times without the flow control by one or two clients

4.3 Split time

These experiments determine the split time and its scalability as the function of bucket capacity *b*. Table 3 and Figure 10 show the results.

All times decrease for larger buckets. The time per record decreases from 0.137 ms four *b* = 10000, to 0.037 ms for *b* = 100000, i.e., by about 73%. Splits are thus much more effective for larger buckets. The curves have a tendency to about flat, but we do not reach yet the server speed. Hence for even bigger *b*, one can expect still slightly better efficiency.

b	Time	Time/Record
10000	1372	0.137
20000	1763	0.088
30000	1952	0.065
40000	2294	0.057
50000	2594	0.052
60000	2824	0.047
70000	3165	0.045
80000	3465	0.043
90000	3595	0.040
100000	3666	0.037

Table 3: Split times for different bucket capacities

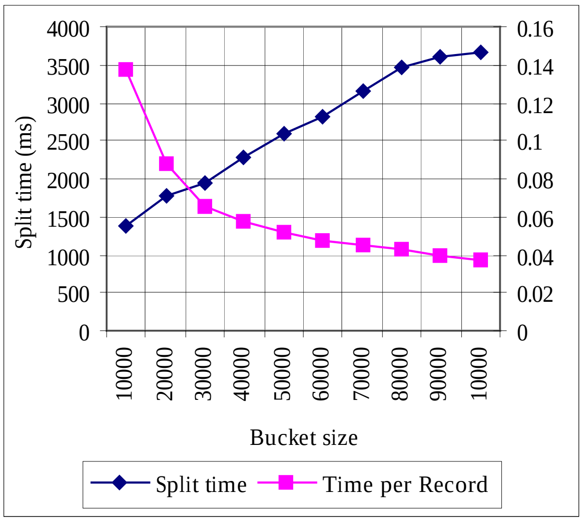

Figure 10: Split times as function of bucket size.

4.4 File manipulation

We studied the time of inserts, key searches and of range queries issued by a single client. We measured the time for a series of requests and the average time per request for both RP*$_N$ and RP*$_C$. We expected again the processing time of a request at the server longer than at the client. The client should reach thus full load and the system best performance only when it faces several servers. To determine how many, and all the times, we have varied the number of servers from 1 to 5, using one bucket per server.

4.4.1 Inserts

We measured the insert performance for a series of up to *x* = 100000 inserts with and without flow control. These were carried out to the initially empty *k*-bucket file. The key values were uniformly distributed. The inserts did not lead to any splits, unlike for the file creation. However, on the one hand we consider a new client, hence there are IAMs for RP*$_C$ client. On the other hand, for inserts without the flow control, we consider the client with the correct image. The idea is to evaluate the incidence of IAMs as discussed below. Table 4 presents the numerical results. Figure 11 and Figure 12 show the curves with the times in function of *x*.

All the times measured decrease when *k* grows. Thus, the processing time at the server is indeed again higher than on a client. The values for $k = 1$ basically measure the former. The performance is basically the same for RP*$_C$ and RP*$_N$. The multicast used for RP*$_N$ in this case is indeed a unicast. A slight overhead always appears nevertheless for the multicast. Mostly, for the inserts with the flow control, when it reaches about 1%.

The times per insert show that our RP* server is effective. Whether the flow control is used or not, the server processes an insert in about 1/3 ms. We recall that this is about 30 times faster than an average insert to a disk file. Increasing *k*, gradually decreases the average insert time. The limits are bound by the client's speed. The best times that appear are about 0.2 ms with the flow control and under 0.1 ms without. In the case of RP*$_C$ without the flow control, the client (and the whole system) reaches its full speed (throughput) for $k = 4$. For all other cases, it is not yet the case even for $k = 5$. The curves become however about flat, so this should happen for $k = 6,7$ servers, reaching the time slightly under 0.22 ms per insert.

The ratio between the processing time for $k = 1$ and for $k = 5$ is over 1.5 for inserts with the flow control for both RP*$_C$ and RP*$_N$. Without the flow control, this ratio is about 3. *The client may thus process at least three times more inserts than a server.* This figures confirms more precisely the conjecture from Section 4.2. Furthermore, the ratio between the times with and without the flow control, is about 1.3 for $k = 1$ and increases to about 2.4 for $k = 5$. Thus the flow control becomes increasingly expensive when the file scales. It appears thus as the predominant slow-down factor for larger files.

It also appears that RP*$_N$ throughput is slower for less servers, closing on RP*$_C$ when there are more servers, although remaining slightly slower. This is logical as every request is multicast to all the servers. The overhead is the greatest for $k = 2$ when there is no flow control. It reaches then 27%. It decreases gradually to 3% for $k = 5$. This behavior seems contrary to the intuition. By its nature, the multicast messaging scales indeed less efficiently than unicast messaging.

The observed behavior could be due to the IAMs. Before the client sending inserts without the flow control gets an IAM triggered by an incorrectly addressed insert, it has time to send several other such inserts. All these inserts are multicast among the servers. More there buckets in the file, more IAM have to be generated and more records have to be forwarded in this way. Each forward also generates a useless IAM that slows down the client. In our experiment, there are also progressively less inserts per bucket from 50.000 per bucket for 2 servers, to 20.000. The effect of the additional forwarding has more incidence. RP*$_C$ could becomes less effective, with performance closing on RP*$_N$.

To prove this explanation we have generated the correct index at the client so to avoid any IAM. As Table 4 shows, the times were only slightly better. Hence, our explanation does not hold. At present, we do not have any other.
RP*$_N$ is a simpler structure than RP*$_C$. The fact that its performance is almost that of RP*$_C$ for a larger file is a nice features of that schema. The processing of the out-of-range records at each server appears relatively negligible. As it was expected while the scheme was proposed, [LNS94]. If there are more clients, RP*$_C$ should nevertheless again reveal notably faster. For two clients, one can expect the speed-up similar to that for the file creation. The experimental confirmation remains to be done.

k	RP*$_C$						RP*$_N$			
	With flow control		Without flow control				With flow control		Without flow control	
			Empty image		Correct image					
	Ttl time	Time/Ins.	Ttl time	Time/Ins.	Ttl time	Time/Ins.	Ttl time	Time/Ins.	Ttl time	Time/Ins.
1	35511	0.355	27480	0.275	27480	0.275	35872	0.359	27540	0.275
2	27767	0.258	14440	0.144	13652	0.137	28350	0.284	18357	0.184
3	23514	0.235	11176	0.112	10632	0.106	25426	0.254	15312	0.153
4	22332	0.223	9213	0.092	9048	0.090	23745	0.237	9824	0.098
5	22101	0.221	9224	0.092	8902	0.089	22911	0.229	9532	0.095

Table 4: Insert times.

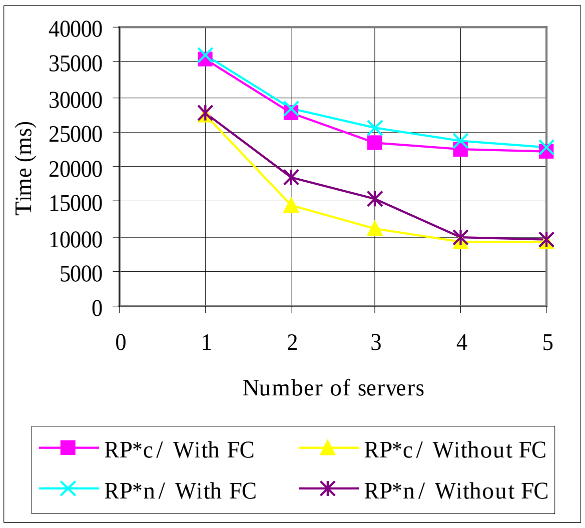

Figure 11**:** Total insert time for initially empty image**.**

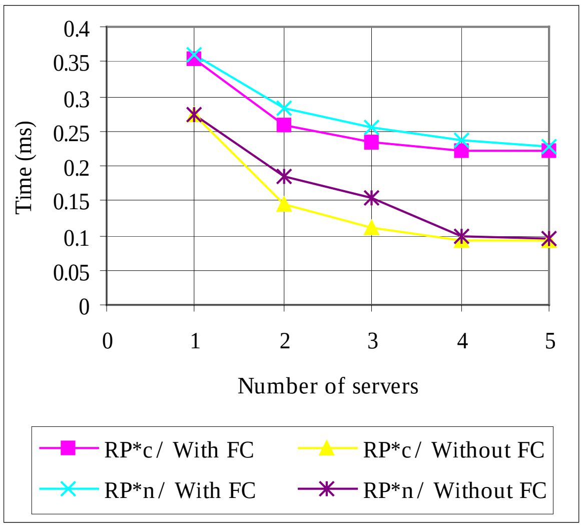

Figure 12: Insert time per record for initially empty image.

4.4.2 Key search

We measured this operation using files at *k* servers. In the experiments, there were up to *k* = 5 servers, supporting in each case the file of 100.000 records. In each experiment, a single client sent 100.000 successful random search requests. The flow control means here that the client sends at most 10 requests without reply. Table 5 tabulates the results. Figure 13and Figure 14 present them as curves depending on *k*.

The results confirm the efficiency of the system and of the analyzed RP* schemes. The individual key search time is always about 0.3 ms, i.e. 30 times better than for a disk file. The throughput reaches 0.2 ms per search with the flow control and 0.14 ms otherwise. The curves become are about flat, but we do not reach the client speed yet. Hence for more buckets one can expect still slightly better throughput. Probably around 0.13 ms per search.

. *k*	RP*$_C$				RP*$_N$			
	With flow control		Without flow control		With flow control		Without flow control	
	Ttl time	Avg time	Ttl time	Avg time	Ttl time	Avg time	Ttl time	Avg time
1	34019	0.340	32086	0.321	34620	0.346	32466	0.325
2	25767	0.258	17686	0.177	27550	0.276	20850	0.209
3	21431	0.214	16002	0.160	23594	0.236	17105	0.171
4	20389	0.204	15312	0.153	20720	0.207	15432	0.154
5	19987	0.200	14256	0.143	20542	0.205	14521	0.145

Table 5 : Search times.

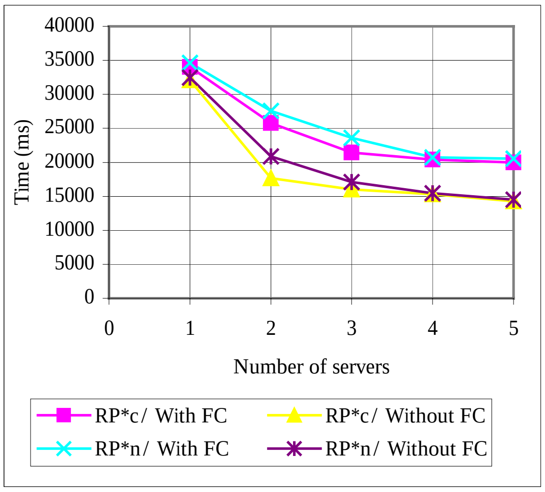

Figure 13: Total search time.

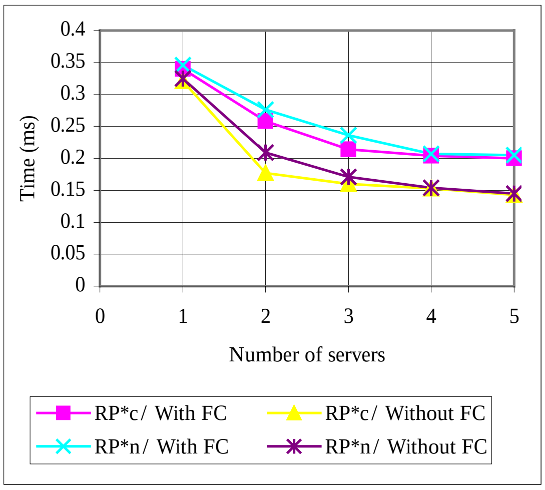

Figure 14: Search time per record.

4.4.3 Range query

We have measured the time of the range query with the deterministic termination scanning in parallel the entire file. Its execution was the same for RP*$_N$ and RP*$_C$. We had 100.000 records to bring to the client. The records were distributed between the *k* servers. Table 6 shows the total time and the throughput. The value *n* is the number of records per server. Figure 15 and Figure 16 show the results as the function of *k*.

n	*k*	Total time	Time/Record
100000	1	3495	0.035
50000	2	2083	0.021
33333	3	1692	0.017
25000	4	1373	0.014
20000	5	1048	0.010

Table 6: Range query times.

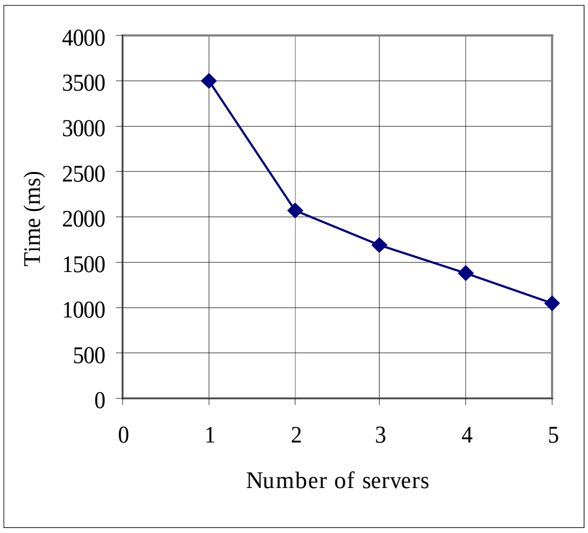

Figure 15: Range query total time.

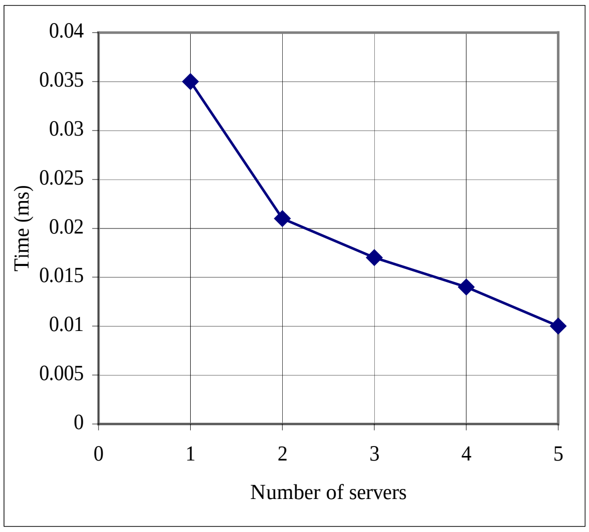

Figure 16: Range query time per record

The outcomes show that the range queries are effective whether for one or more servers. As one could expect, it appears nevertheless useful to distribute the records among multiple servers. The throughput improves from 0.035 ms for the file on one server, to 0.010 ms for the same file in five buckets. This is a strong improvement, by about 71%. As the curves are not flat yet, more servers should further improve this performance.

5 Scalability analysis

The experimental curves we have obtained for various times become about linear for total times or flat for the throughputs. One may thus infer the performance of files much larger than those we experimented with. At first one may study in this way the largest file one could possibly create at our current configuration. The full capacity of 3-level index leads to *b* = 640.000 records per bucket. For 100-Byte record assumed in the study this corresponds to 64MB buckets. These can be supported by our servers as they have 128MB of RAM. As the average load factor of an RP* file is 70 % under random inserts, our file may scale to 448.000 records per bucket on the average. Its total capacity may then reach 2.240.000 records and 320 MB. A single RP*$_N$ client would create this file in 2.240.000 x 0.095 + 51.2 = 264 s. The 1st term in the expression reflects the total insert time, and the 2nd one totalizes the split times. A single RP*$_C$ client would be faster by almost 7 s. For the sake of completeness let us mention that if one should create that file on a disk, it would take 22.400 s, i.e. over 6h.

The machines used support in fact 256 MB of RAM. That capacity could be used at present rather to store large records, up to 300 B. To store more records, a 4-level index would be needed.

On the other hand, one may infer the cost of the example file if it scaled to a much larger size, e.g., 10.000.000 records. The average bucket load would be of 70 % x 50000 = 35000 records. The file would reach 10.000.000/35000 = 286 buckets. There are many more machines available at U. Paris 9, hence this size is feasible. The (random) inserts would last 950 s for RP*$_N$ or 920 s for RP*$_C$. The splits would require 285 x (25000 x 0.04) = 285 s. The RP*$_N$ and RP*$_C$ files would be created respectively in 1235 s and 1205 s, i.e., each one in about 20 minutes. The creation of the disk file would require about 28 hours.

The operational use of that large RP* files would probably require nevertheless some changes to the client architecture with respect to the range queries. At present the messaging is quite basic: a connection and a thread is created per each responding bucket. The current TCP/IP stack architecture does not handle efficiently a large number of simultaneous connections. The related study in [TS00] analyses the corresponding pitfalls, and propose more scalable messaging alternatives for an SDDS client.

6 Related works

The papers on SDDS schemes present usually the theoretical access performance analysis measured in number of messages. The advantage of that measure is to be network and CPU speed independent. The evaluation of actual elapsed times are rare. Below, Table 7, compares our results to the theoretical predictions in the original RP* paper [LNS94]. It also compare them to the experimental results for LH*$_{LH}$ on SDDS-2000, [BDNL00].

The theoretical performance of the RP* scheme in [LNS94] were predicted in only for RP*$_N$. The network model was from [G88]. Site speed was accordingly assumed 100 MIPS, and OS time to process a message was 25 µs (2500 instructions). No flow control was considered. A key search query was 100 B long, and a message with a record, was assumed 1 KB long.

Some theoretical measures in Table 7, e.g., t_c, are not in [LNS94]. They were calculated here a new, on the basis of the same assumptions, to complete the comparison. Their computation was quite obvious, so we omit it.

	LH* Imp.	RP*$_N$ Thr.	RP*$_N$ Imp.		RP*$_C$ Impl.	
			With FC	No FC	With FC	No FC
t_c	51000	40250	69209	47798	67838	45032
t_s	0.350	0.186	0.205	0.145	0.200	0.143
$t_{i,c}$	0.340	0,268	0.461	0.319	0.452	0.279
t_i	0.330	0.161	0.229	0.095	0.221	0.086
t_m	0.16	0.161	0.037	0.037	0.037	0.037
t_r		0.005	0.010	0.010	0.010	0.010

Table 7: Comparative Analysis

t_c: time to create the file
t_s: time per key search (throughput)
t_i: time per random insert (throughput)
$t_{i,c}$: time per random insert (throughput) during the file creation
t_m: time per record for splitting
t_r: time per record for a range query

Our experimental results basically confirm the theory. The values without flow control are quite similar except for t_m. The reason is that the theoretical value was calculated for an individual record transfer. This one is close to the 0.137 ms obtained for the smallest *b* in Table 3. In Table 7 however, we have chosen to show the best performance, for the largest *b* measured, and corresponding to our most efficient TCP/IP stream transfer.

The results for LH*$_{LH}$ in Table 7 were reported at the same configuration. The bucket capacity was however only $b = 5.000$ records, instead of our $b = 50.000$. The file was also of 20.000 records only and records were of 50 bytes, hence two times shorter. The time t_c Table 7 is therefore inferred for LH*$_{LH}$ for our 150.000 record files. It is comparable, but longer than all those for RP*. This is due to much smaller *b* used, hence relatively higher incidence of the splits.

The search time t_s for LH*$_{LH}$ is established for a series where next search starts when the reply to the previous one is received. Hence, it is naturally higher than that indicated for RP* and confirms incidentally the efficiency of the flow control algorithm used. Times $t_{i,c}$ are similar, the differences steaming from the respective *b* values. The relatively high value of t_i for LH*$_{LH}$ has no explanation at present. It is perhaps simply in error. Time t_m of LH*$_{LH}$ should rather be compared to that of 0.137 ms for $b = 10.000$ in table. This performance appears then about similar for both schemes.

Finally, t_r was not measured for LH*$_{LH}$. There is no concept of range query for that scheme. The scan of the entire LH*$_{LH}$ file could be measured nevertheless. Similar performance for both schemes should appear then. Both schemes use indeed SDDS-2000 and performance of a scan is basically independent of the SDDS scheme used. Notice however that different internal bucket structures, as well as termination algorithms should somehow influence the result.

7 Conclusion

SDDS-2000 is a prototype system for Windows multicomputer supporting various SDDSs. In particular it supports several variants of the RP* schema for the scalable range partitioning. We have presented the design choices with respect to the internal bucket structure and the multithread system architecture. We have shown also the experimental performance measures of the file manipulations.

The behavior of the scheme and the processing times appear in line with the expectations. RP*$_C$ is generally more effective, especially when the performance is limited by the servers or there are several clients. The system is nevertheless efficient enough to provide times per insert or key search of a fraction of a millisecond for both measured variants. The client finds or inserts a given record in about 0.3 ms. This value is mainly due to the server processing speed. Hence it should decrease as PC servers become faster. It is already at present about 30 times faster than a disk access. The throughput time per insert in a series of random inserts by a client may reach 0.09 ms. The throughput time per random search by a client may reach 0.14 ms. These limits are due mainly to client CPU speed. Hence, they should not improve when the number of servers scales. But should decrease further for a faster client, e.g. on a GHz PC. The range query time reaches 0.01 ms per retrieved record. The curves show that if the file scales, it should further improve. Finally the split time is in the order of a second for smaller buckets to four seconds for the largest measured. The times per record show that larger buckets are relatively more efficient.

Further work concerns more in depth performance analysis. There are also design variations we plan to experiment with. We will also experiment with actual applications.

Acknowledgements

This work was partly supported by a research grant from Microsoft Research. We thank Jim Gray, David Lomet, and Pierre-Yves Saintoyant for helpful discussions. Earlier work on SDDS-2000 was partly supported by grants from HP Laboratories, Palo Alto, CA, and from IBM Almaden Res. Cntr., San Jose, CA.

References

[BDNL00] Bennour, F., Diène, A. W., Ndiaye, Y. Litwin, W., Scalable and Distributed Linear Hashing LH*$_{LH}$ under Windows NT. SCI-2000 Orlando, Florida, USA. July 23-26, 2000.

[D98] Diène, A. W. Internal organization of RP* SDDS family. DEA Report (in French). U. of Dakar, 1997.

[D&al00] Diène A. W. & al. An Architecture for a Manager of RP* SDDS under Windows NT. CERIA Res. Rep. U. Paris 9, 2000.

[DNB00] Diene, A. W., Ndiaye, Y., Bennour, F. Scalable ordered and distributed file under SDDS-2000. CERIA Res. Rep., U. Paris 9, 1999.

[G88] Gray, J. The Cost of Messages. 7th ACM Symp. on Principles of Distributed Systems, 1988.

[J88] Jacobson, V. Congestion Avoidance and Control, Computer Communication Review, vol. 18, no 4, pp. 314-329 (Aug.), 1988.

[L80] Litwin, W. Linear Hashing : a new tool for file and tables addressing. Reprinted from VLDB-80 in READINGS IN DATABASES. 2-nd ed. Morgan Kaufmann Publishers, Inc., 1994. Stonebraker , M.(Ed.).

LMRS99] Litwin, W., Menon, J., Risch, T., Schawrz, T. J. E., Scalable Availability LH* Shcemes with Record Grouping. DIMACS Workshop on Distributed Data Structures, Princeton U., (May 1999), Carleton Scientific, (publ.). 1999.

[LNS93] Neimat, M-A., Schneider, D. LH* : Linear Hashing for Distributed Files. ACM-SIGMOD Intl. Conf. On Management of Data, 1993.

[LNS94] Litwin, W., Neimat, M-A., Schneider, D. RP*: A Family of Order-Preserving Scalable Distributed Data Structures. 20th Intl. Conf on Very Large Data Bases (VLDB), 1994.

[LN95b] W. Litwin, M-A Neimat. LH*s : a high-availability and high-security Scalable Distributed Data Structure. IEEE Workshop on Research Issues in Data Engineering. IEEE Press, 1997 (to app.)

[LN95a] Litwin, W., Neimat. k-RP* : a Family of High Performance Multi-attribute Scalable Distributed Data Structure. to app. in IEEE Intl. Conf. on Par. & Distr. Systems, PDIS-96, (Dec. 1996).

[LS99] Litwin, W., W. and Schwarz, J. E.: LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. CERIA Res. Rep. 99-2, U. Paris 9, 1999.

[RAC96] Rodrigues, S.H., Anderson, T.E., Culler, D.E. Hight-performance local communication with fast socket, NOW, UC Berley, 1997.

[S96] Alok k. Sinha. Network Programming in Windows NT. Addison-Wesley Publishing Company.

[SDDS] Bibliographie
http://ceria.dauphine.fr/SDDS-bibliograhie.html.

[TS00] Tsangou, M., Mesures de performances de la scalabilité des requêtes parallèles sur les SDDS RP*. Mémoire de DEA, Université de Dakar, Avril 2000.