Publication avant LAFMC

Titre	Range Queries to Scalable Distributed Data Structure RP*
Auteurs	Mianakindila Tsangou, **Samba Ndiaye**, Mouhamed Seck, Witold Litwin
Référence	Proceedings of the Fifth Workshop on Distributed Data and Structures, Thessaloniki, June 2003 (WDAS 2003)
Editeur	WDAS
Pages	1 - 14
Année	2003
DOI	
URL	http://www.docstoc.com/docs/47769536/Range-Queries-to-Scalable-Distributed-Data-Structure-RP
Index	
ISBN	
Encadreur	Non
Extrait d'une thèse	Non

Range Queries to Scalable Distributed Data Structure RP*

Article · January 2003 *with* 4 Reads

Mianakindila Tsangou

Samba Ndiaye
4.82 · Cheikh Anta Diop University, Dakar

Mouhamed Seck

+ 1

Witold Litwin

Show more authors

Abstract

A Scalable Distributed Data Structure (SDDS) is a data structure of a new type specifically designed for multicomputers, P2P and grid computing systems. SDDS RP* provides a range partitioned file scaling up dynamically over distributed nodes. Our concern is the efficient execution of RP* range queries. The query may address an a priori unknown number of data servers.

Range Queries to Scalable Distributed Data Structure RP*

MIANAKINDILA TSANGOU
Université Cheikh Anta Diop de Dakar, Sénégal

SAMBA NDIAYE
Université Cheikh Anta Diop de Dakar, Sénégal

MOUHAMED T. SECK
Université Paris 9, Dauphine, France

WITOLD LITWIN
Université Paris 9, Dauphine, France

Abstract

A Scalable Distributed Data Structure (SDDS) is a data structure of a new type specifically designed for multicomputers, P2P and grid computing systems. SDDS RP* provides a range partitioned file scaling up dynamically over distributed nodes. Our concern is the efficient execution of RP* range queries. The query may address an a priori unknown number of data servers. The client may get replies in parallel from very few or very many servers, perhaps thousands. One needs a scalable data reception strategy. We experiment with three such strategies using the typical UDP and TCP/IP protocols. In two strategies, the server sends the relevant data immediately. In the third strategy, the server sends the data only when the client polls it. We show experimentally that this strategy scales best and becomes the fastest for replies from even a few servers. Our results generalize to distributed/parallel queries other than range queries.

Keywords

Scalability, range query, performance, SDDS, multicomputer, P2P, Grid computing

1 Introduction

A multicomputer, P2P or grid computing system consists of a large number of PCs and workstations interconnected through a high-speed network. Such system need new data structures, scaling efficiently over very many nodes [1, 2]. The Scalable Distributed Data Structures (SDDSs) aim at this goal [6]. An SDDS

file consists of records with keys stored on SDDS *servers*. The records at every server are stored in a memory space called *bucket*. The file scales dynamically with inserts, from initially one SDDS server, to potentially any number of servers. An existing bucket that becomes full splits sending about half of its records to a new bucket dynamically appended to the file. An application addresses data through the SDDS service *client*. The client may not know the servers of the file. A node of a multicomputer (P2P system or grid system) can be a client or a server, or a client for some and a server for other applications.

Several SDDS families are now known. The LH* schemes [6, 8], use a distributed version of the linear hashing [5]. Some are the high-availability SDDS schemes [9, 10]. The family termed RP* (Range Partitioning) preserves the key order like a B-tree [7]. There are several implementations of SDDS-RP* file [3, 4, 11, 12, 13].

A RP* file is partitioned so that each bucket contains a maximum of b records with the keys within some interval $]\lambda, \Lambda]$ called bucket *range*. The parameter λ is the *minimal* key and Λ is the *maximal* key of the bucket. A record r with key c is in bucket identified by the range $]\lambda, \Lambda]$ only if $c \in\ ]\lambda, \Lambda]$. A split partitions a full bucket as in a B-tree. The ranges resulting from any number of splits partition the record key space and the file into a collection of intervals $]\lambda_i, \Lambda_i]$, where i designates the i-th file bucket among the Ncurrently existing. For any c, there is only one bucket in the RP* file which may contain it.

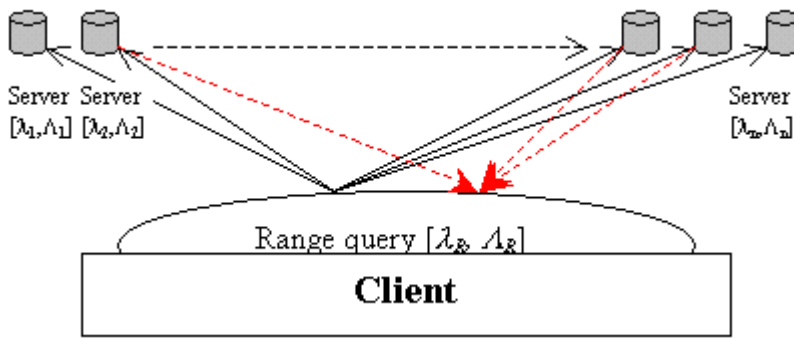

Figure 1: Range query in an SDDS

As in B-trees, an RP* file is intended to support efficiently the *range* queries. Such a query, let it be R, requests all records with the keys within some *query interval* $[\lambda_R, \Lambda_R]$. The application submits R, as any of its queries, to the SDDS *client* at its node. That one sends R out and receives the relevant data coming back from the buckets concerned by the query. This is the case of every bucket whose

interval $]\lambda_i, \Lambda_i]$ is such that $]\lambda_i, \Lambda_i] \cap [\lambda_R, \Lambda_R] \neq \emptyset$. The number N of those servers may be small or very high. N may potentially be over thousands, perhaps soon millions.

There is therefore a non-trivial problem on the client side, to most efficiently organize the reception of the corresponding messages. We analyse three strategies aimed at this goal. Through the implementation and the simulation of existence of over a hundred of server sites, we determine the performance, especially the response time, of each of them. We demonstrate that one of these strategies scales better than the others. We recommend therefore this one for the practical use. A variant of this strategy is now present in the prototype SDDS-2000 available for download at CERIA site http:\\ceria.dauphine.fr.

Section 2 overviews more in depth the SDDS range query data reception problem. It also presents our strategies. Section 3 discusses the performance analysis through experiments with our strategies. Section 4 addresses the related work. We conclude in Section 5 and point out further research directions.

2 SDDS Range Query Data Reception

In the typical environment, the communication of the records selected by a range query uses the UDP or the TCP messaging. UDP is unreliable by its datagram only nature. It is also limited to 64 KB per message. TCP is free of both limitations, but has a more costly connection management dialog. UDP appears a suitable support for service messages between the SDDS client and the servers. TCP appears the only practical candidate for the range query records transmission.

Accordingly, the basic strategy for a range query is that every server concerned requests a TCP connection with the client. If the number of connections to manage is too high, the client may get swamped in various ways. First, some demands may get refused, forcing the servers to repeat their requests, perhaps many times. Next, managing many simultaneous connections may lead to a loss of some incoming data, impossible to accommodate fast enough in the buffers available for the TCP/IP protocol. This triggers under TCP/IP costly retransmissions. Three following strategies appear suitable for implementation.

[Strategy 1] The client accepts as many TCP/IP requests for connection as it can. In practice, it may concern at most M requests, for some positive integer M. This one is the system parameter usually called backlog. The refused server drops out, abandoning the reply. The client may need to find which ones did it and request the missing data specifically from each drop out. This recovery is not the part of Strategy 1 itself.

[Strategy 2] Every server starts as for Strategy 1. The refused servers repeat the connection requests. The repetitions follow the CSMA/CD policy.

[Strategy 3] Each server that should reply requests the connection demand from the client through the UDP service message. Each message contains the interval and the IP address of the server. After receiving all the intervals and IP addresses, the client establishes TCP connections with the servers to receive the data. It may initiate as many simultaneous connections as it can accommodate given its storage and processing resources. However, we study below only the case of a single connection at the time.

The advantage of Strategy 1 is its simplicity. The drawback is that some servers drop out. The recovery of missing data by the client is a costly process if there are many drop-outs.

Strategy 2 offsets this limitation, at the price of additional complexity at the server side. It may in contrast naturally swamp the client. Strategy 3 avoids both drawbacks, but the price may be potentially a longer response time.

It does not seem that there is any easy way to determine the actual performance of these strategies through a purely formal analysis. Especially, to obtain their respective response times, in function of the number of replying servers. We therefore turned towards the experimental analysis that we describe now.

3 Experimental Performance Analysis

3.1 Environment and Measurements

We have developed a multithreaded client able to process in parallel multiple data streams from the servers. We have used Java to design the software able to run on different platforms.

Our hardware platform consisted of eight Pentium II 400 MHz PCs under Windows NT Server 4.0. These nodes were linked by 10 MB/s Ethernet network. Each PC had 64 MB of RAM and 450 MB of virtual memory.

Our main performance measure of a strategy was the response time of a range query. We measured the difference between the time of reception of the last data item and the time of query send out by multicast. We call this measure the *total* time. We also measured the corresponding *per (received) record* time.

For each strategy, our application built range queries whose intervals $[\lambda_R, \Lambda_R,]$ scaled up progressively. At the end, the query addressed all the buckets in the file. For each experiment, we measured the response time for given $[\lambda_R, \Lambda_R]$. We repeated each experiment five times, to get the average value.

We started 140 buckets on 7 PCs, with, uniformly, 20 buckets per PC. Every bucket had a distinct port number. Buckets used thus distinct TCP/IP connections, as required to simulate the actual 140 server nodes. The ranges of buckets were]0,50],]50,100], ...,]6950,7000]. As the data to retrieve, we have inserted 50 records into each bucket. We located our SDDS client and the application issuing the range queries and collecting the measurements at the eight PC.

3.2 Results

3.2.1 Strategy 1

Figure 2 and the 2nd column of Table 1 below show the total response time, measured up to forty responding servers. The points are our measurements. The line is the least square linear interpolation. The figure shows that the response time scales linearly with the number of servers.

Number of servers	Strategy 1		Strategy 2		Strategy 3	
	RT	PRT	RT	PRT	RT	PRT
1	20	0.40	24	0.48	250.6	5.01
5	40	0.16	44.2	0.18	274.2	1.10
10	74	0.15	476.4	0.95	344.2	0.69
15	558.8	0.75	504.2	0.67	440.4	0.59
20	578.8	0.58	997	1.00	492.2	0.49
25	1141.6	0.91	1203.4	0.96	560.4	0.45
30	1616.2	1.08	1536.4	1.02	725	0.48
35	1666.6	0.95	1732.8	0.99	755	0.43
40	1612.2	0.81	1754.6	0.88	791	0.40
45		0.40	2033	0.90	923.6	0.41
50		0.40	2169.2	0.87	1073.2	0.43
55		0.40	2069.2	0.75	1059	0.39
60			2591.2	0.86	1139.8	0.38
65			2600	0.80	1205.8	0.37
70			2708	0.77	1404	0.40
75			3477	0.93	1576	0.42
80			3568.8	0.89	1612.2	0.40
85			3320.4	0.78	1560.6	0.37
90			3214.4	0.71	1608.4	0.36
95			3397.2	0.72	1748.4	0.37
100			3653.4	0.73	1758.6	0.35
110			3773	0.69	2068.6	0.38
120			4185.8	0.70	2465.4	0.41
130			4191	0.64	2466.5	0.38
140			4877	0.70	2313.3	0.33

Table 1: Response times for all three strategies. RT: total Response Time, PRT: Per record Response Rime.

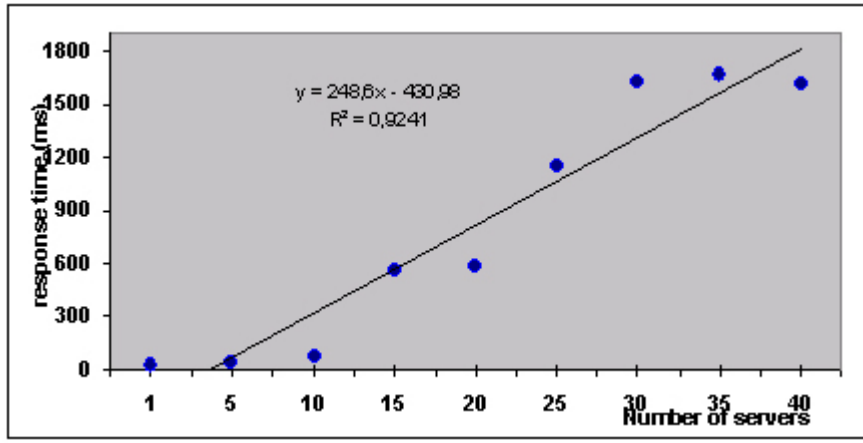

Figure 2: Response time for Strategy 1.

The correlation coefficient R^2 in Figure 2 is about 0.92. The figure also shows the formulae of the line. The number x of servers is less than 40. Over 40 servers, 80% of measurements fail. In fact, for every series of 5 measurements, 4 of them fail because the client refused many servers. Thus, the experiments show a limit on the scalability of Strategy 1. It is equal to 40 servers under our experimental criterion of 80% of drop out. The limit is due to the bottleneck at the client, itself induced by the backlog value.

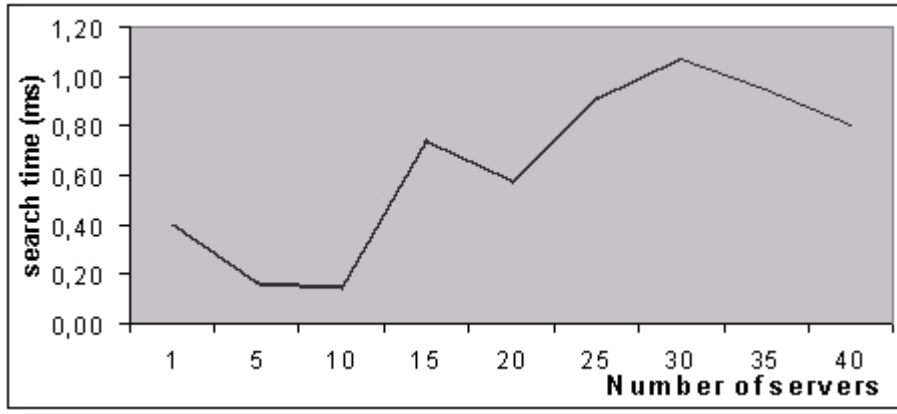

Figure 3: Per record response time of Strategy 1

Figure 3 and the 3rd column of Table 1 show the per record response of Strategy 1. This time increases with the number of servers, almost linearly between 10 and 30 servers. In fact, for 5 servers, this time is about 0.16 ms per record, and reaches about 0.8 ms for already 15 servers. Next it somehow oscillates around this value with the peak of 1 ms for 30 servers. This is a quite scalable behavior. However, as for the total response time, this behavior is limited to 40 servers only in our experimentation. Because of this limitation, one cannot qualify Strategy 1 as scalable for our purpose.

3.2.2 Strategy 2

Figure 4 and the 4th column of Table 1 show the total response time for Strategy 2. As before, the points are our measurements and the line is the least square linear interpolation.

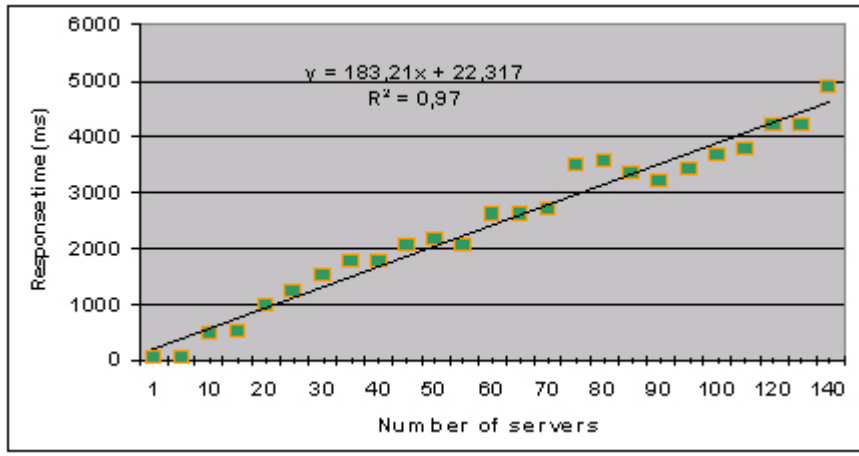

Figure 4: Total response time of Strategy 2

This strategy scales up linearly to 140 servers. Over 140 servers, we have, the same phenomenon like in Strategy 1, i.e., 80% of measurements fail. Thus, 140 is a limit for the second strategy.

Figure 5 and the 5th column of Table 1 show that the per record response time of Strategy 2. It becomes rapidly almost constant around 0.8 ms. It even presents a slightly decreasing tendency, after reaching 1 ms for already 20 servers. This behavior also proves the scalability of Strategy 2.

We also noticed, Figure 6, that only one connection attempt is necessary to servers when their number do not exceed 40. Strategy 2 boils then down obviously to Strategy 1. Beyond 40 servers addressed by the query, till about 70 servers, several servers need 2 attempts. Afterwards, some servers need up to 3 connection attempts.

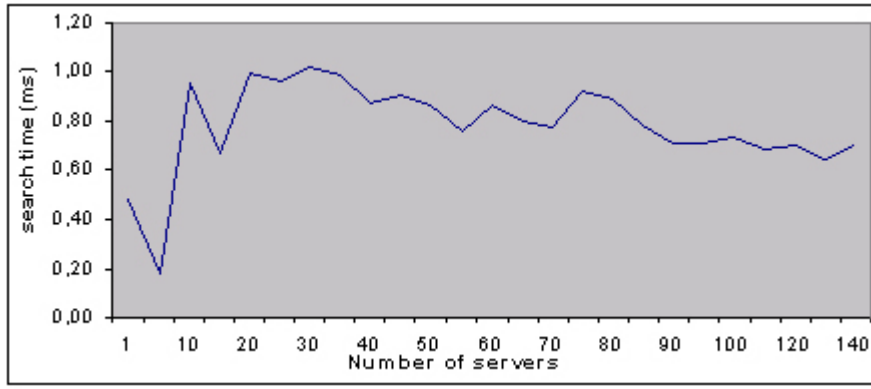

Figure 5: Per record response time of Strategy 2

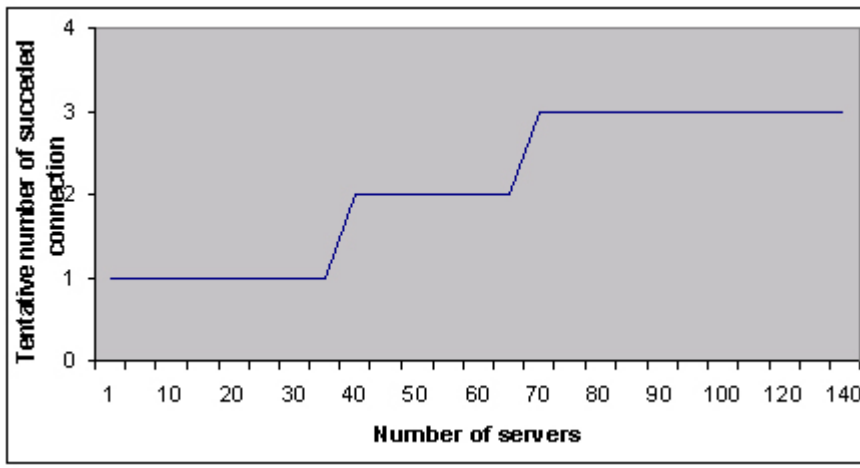

Figure 6: The maximum number of connection demands

3.2.3 Strategy 3

Figure 7 and the last column of Table 1 show the measured and interpolated total response time of Strategy 3. This time also scales linearly up to our 140 server limit. One could expect such behavior since the client keeps constant the number of simultaneous connections at each time. Hence, the communication load should remain basically also constant. The response time should appear about linear in consequence as well. In particular, it should be so also arbitrarily far beyond our 140-server limit.

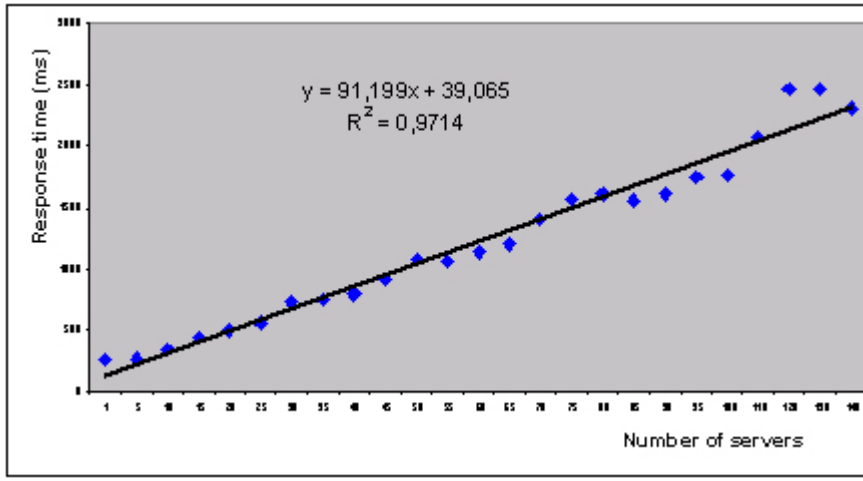

Figure 7: Strategy 3: Total response time

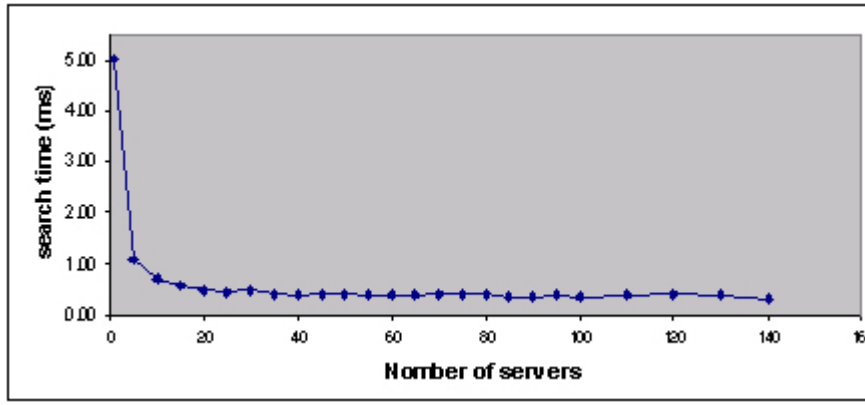

Figure 8: Per record response time of Strategy 3

The per record time, in Figure 8, starts by decreasing steeply. The initial 5 ms for one server, drops to 1 ms for 5 servers, reaches 0.49 for 20 servers. It then decreases only very slowly to the final 0.33 ms for 140 servers (figure 8). The curve appears in fact about flat between 20 and 140 servers, with small oscillations.

3.2.4 Comparative Analysis

Figure 9 and Table 1 show together the actual total response times of all the strategies. For a number of servers increasing from 1 to 35, Strategies 1 and 2 appear practically equal, as they should. The differences result only from the experimental nature of the collected measures. Figure 10 and Table 1 show together the per record time of all the strategies. After 15 servers, Strategy 1 gives best results.

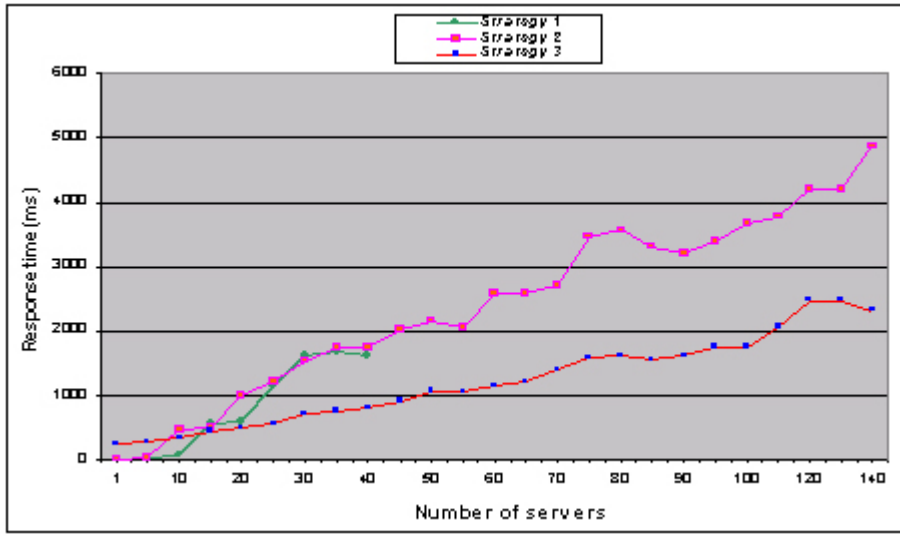

Figure 9: Comparison of total response times

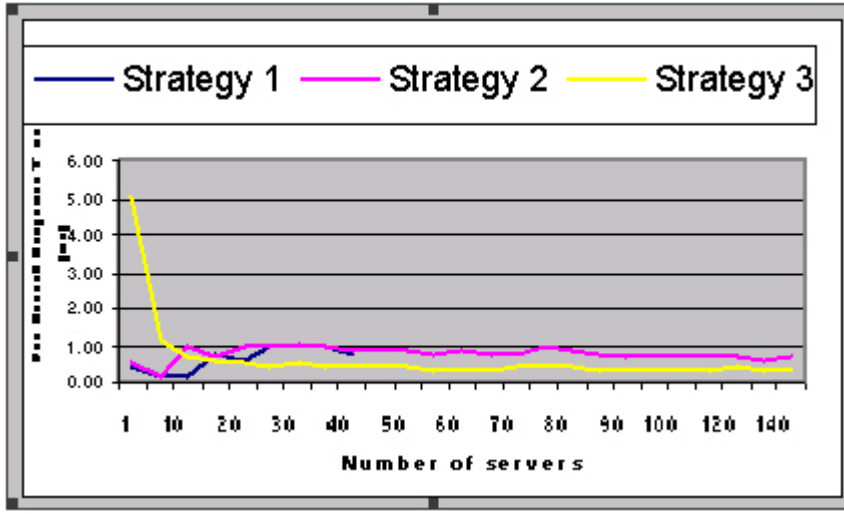

Figure 10: Comparison of per record response times

Strategy 3 is by far the worst performing for a small number of buckets. Actually, up to five buckets in our campaign. It is reasonably good between 5 and 15-bucket range. It turns to be the best performing beyond already 15 servers. For 140 servers, it is almost twice as fast as Strategy 2. The reason seems to be that Strategy 3 apparently avoids an increasingly important number of repeated connection demands, and of repeated datagrams. This outcome might surprise. One may recall however that the token ring protocol behaves similarly to Strategy 3, and is well known to offer better performance for a high network traffic than the collision based Ethernet protocols, managing Strategy 2. The superiority of Strategy 3 shows also that the client manages its load itself more efficiently than its TCP/IP layer flow control at present. This is also a highly instructive outcome.

4 Related Work

Somehow surprisingly, we could not find any similar analysis reported in the open literature. Perhaps, since scalability of data structures and their manipulations over a large number of multicomputer, or P2P, or grid nodes, is a relatively new concern. The closest application domain we could study were the commercial parallel (i.e. distributed, federated DBMSs). These were especially IBM DB2 Universal v7.x, Oracle 9, and SQL Server 2000 DBMSs.

These systems can, on the one hand, manage parallel queries on multiprocessor computers. The data communication they use local buses. The scalability does not seem there any concern.

These systems offer also the partitioning over clusters of the computers on a network. They further offer the federated or distributed partitioned views [14, 15, 16]. The number of server nodes one can address in this way seems a few dozens at largest for all these systems. We could only find the current maximal number for SQL Server 2000. This one is 32 nodes. Our experiments and concerns go far beyond these present technology limits.

We could find instructive information on parallel/distributed query optimization. But, no publicly available documentation seems to exist on the actual strategy for the retrieved data transfers. For the SQL SERVER 2000, the only information we could find is that such data exchanges use pipes.

However, it is clear that for any parallel (i.e. distributed, federated DBMS), the query optimizer should play a crucial role for the strategy to use for the parallel query data reception. One conclusion we would draw from our experiments is the use of Strategy 2 for smaller number of server sites, (what the commercial systems perhaps finally effectively do), and the unconditional switch to Strategy 3 for larger queries. The practical breakeven point can be about 10-15 buckets (server nodes) as in our case. Usually however, it should depend on the actual configuration of the DBMS.

5 Conclusion and Future Work

Strategy 3 appears the best choice for range queries, and in fact the parallel queries in general, to an SDDS. It offers the linear scale up, and the best response time. This one achieves the half of the response time of Strategy 2 for our largest file. Consequently, Strategy 3 provides also the best per record response time. That one reaches 1/3 ms, being thus much faster than a disk file response time. Using a faster PC and a popular 100 Mb/s Ethernet or an increasingly popular Gigabyte Ethernet should improve the response times of all the strategies. It should not change however our basic conclusion about the superiority of Strategy 3. Nevertheless, the breakeven point under which Strategy 2 performs better could move forward, accommodating more server nodes.

For a faster CPU, performance of Strategy 3 could also improve by opening at the client a few simultaneous connections instead of only one in our experiments. This generalization of Strategy 3 was integrated for the operational use into SDDS-2000 [3].

The strategies we have experimented delivered the result of each bucket directly to the client. An alternative approach could be that partial results are first grouped at some servers. This reduces the number of connections to manage at the client. Such strategies appear especially interesting for a range query with an aggregate function. The process can be organized into a hierarchy where each level reduces the number of servers aggregating the result. Ultimately one server could get then the final result, delivering the reply to the client in a single fast message. Such strategies constitute an interesting direction for our further work.

In general, the scalability of data structures and of their manipulations over a large number of storage nodes is a relatively recent concern. Nevertheless, such an analysis appears naturally of basic importance for the designers of future multicomputer, P2P, and grid computing systems. Our results reported here should be in this context instructive as well.

References

[1] K. Aberer and M. Hauswirth. Peer-to-Peer Information Systems: Concepts and Models, State-of-the-Art, and Future Systems. In *Proceedings 18th IEEE International Conference on Data Engineering (ICDE)*, (San Jose, CA, 2002).

[2] K. Aberer, M. Hauswirth, M. Punceva, and R. Schmidt. Improving Data Access in P2P Systems. *IEEE Internet Computing 6*, 1, (2002).

[3] A. Diene. Contribution à la Gestion de Structures de Données Distribuées et Scalables. Thèse de Doctorat d'informatique de l'Université Paris Dauphine, 2001. http:\\ceria.dauphine.fr

[4] A. Diene, Y. Ndiaye, T. Seck and F. Bennour. Conception et Réalisation d'un Gestionnaire de Structures de Données Distribuées et Scalables sous Windows NT Rapport Technique, CERIA, Université Paris Dauphine, 1999. http:\\ceria.dauphine.fr

[5] W. Litwin. Linear Hashing: a New Tool for File and Table Addressing. Reprinted from VLDB-80 in READINGS IN DATABASES, 2nd ed., M. Stonebraker (ed.), Morgan-Kaufmann, 1996.

[6] W. Litwin, M.-A. Neimat and D. Shneider. LH*: Linear Hashing for distributed Files. In *Proceedings International ACM Conference on Management of Data (SIGMOD)*, 1993.

[7] W. Litwin, M.-A. Neimat and D. Schneider. RP* : a Family of Order-Preserving Scalable Distributed Data Structure. In *Proceedings International Conference on Very Large Data Bases (VLDB)*, 1994.

[8] W. Litwin, M.-A. Neimat and D. Schneider. LH*: a Scalable Distributed Data Structure. *ACM Transactions on Database Systems*, 1996.

[9] W. Litwin and T. Schwarz. LH*$_{RS}$: a High Availability Scalable Distributed Data Structure using Reed Solomon Codes. In *Proceedings International ACM Conference on Management of Data (SIGMOD)*, 2000.

[10] W. Litwin and T. Risch. LH*g: a High-availability Scalable Distributed Data Structure by Record Grouping. *IEEE Transactions on Knowledge and Data Engineering 14*, 4, (2002).

[11] Y. Ndiaye. Protocole de communication SDDS RP*. Mémoire de DEA d'Informatique. Univ. Cheikh Anta Diop de Dakar, 1998.

[12] M.T. Seck, S. Ndiaye, A.W. Diene, W. Litwin and G. Levy. Implémentation des Structures de Données Distribuées et Scalables RP*. In *Proceedings CARI Conference*, (Dakar, Sénégal, 1998).

[13] M. Tsangou. Mesures des Performances des Requêtes Parallèles sur les SDDS RP*. Mémoire de DEA d'informatique. Univ. Cheikh Anta Diop de Dakar, 2000.

[14] http://msdn.microsoft.com/

[15] http://otn.oracle.com/

[16] L. Haas et al. Garlic: a New Flavor of Federated Query Processing for DB2. In *Proceedings International Conference on Very Large Data Bases (VLDB)*, 2002.

[17] S Ndiaye, T. Seck and M. Tsangou: Scalabilité des Performances des requêtes parallèles dans les SDDS-RP*. In *Proceedings International Workshop on Distributed Data and Structures (WDAS)*, (Paris, France, 2002).

Mianakindila Tsangou is with the Université Cheikh Anta Diop de Dakar, Sénégal. E-mail: tsangou@ucad.sn

Samba Ndiaye is with the Université Cheikh Anta Diop de Dakar, Sénégal. E-mail: ndiayesa@ucad.sn

Mouhamed T. Seck is with Université Paris 9, Dauphine, France. E-mail: seckm@ucad.sn

Witold Litwin is with Université Paris 9, Dauphine, France. E-mail: Witold.Litwin@dauphine.fr