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

Aly Wane Diène & Witold Litwin
CERIA
University Paris 9 Dauphine
http://ceria.dauphine.fr

Plan

- SDDSs
- RP* Schemes
- Bucket Structure
- SDDS-2000 Client/Server Architecture
- Performance Analysis
- Conclusion and Future Work

Scalable Distributed Data Structures

- Introduced in 1993
- Specifically for multicomputers
- Files of records identified by keys
- Accessible from client sites
- Data on server sites
  - Usually in distributed RAM
- Overloaded servers split
- Clients are not synchronously informed of splits
- Clients may make addressing errors
- Servers forward incorrectly addressed requests
- Image Adjustment Messages sent back to improve the client addressing scheme (client image)
- Several SDDS known
  - Hash partitioning (LH*), Range Partitioning (RP*)…

Scalable Distributed Data Structures

- Early Prototype Implementations
  - Distributed Dynamic Hashing
    - UCB 1994 (Bob Devine), Unix
  - LH*
    - HPL 1994 (D. Schneider, J. Levy) on SUNs
    - U. Linkoping 1996 (J. Karlson) on Parsytec multicomputer)
    - …
- SDDS-2000
  - SDDS Manager for Wintel Multicomputer
  - Designed for any SDDS Schema
  - Supports at present LH*, LH*RS, RP*
    - LH*RS is a high-availability schema using Reed Salomon erasure correcting codes
  - Interfaces AMOS main memory DBMS for database query processing

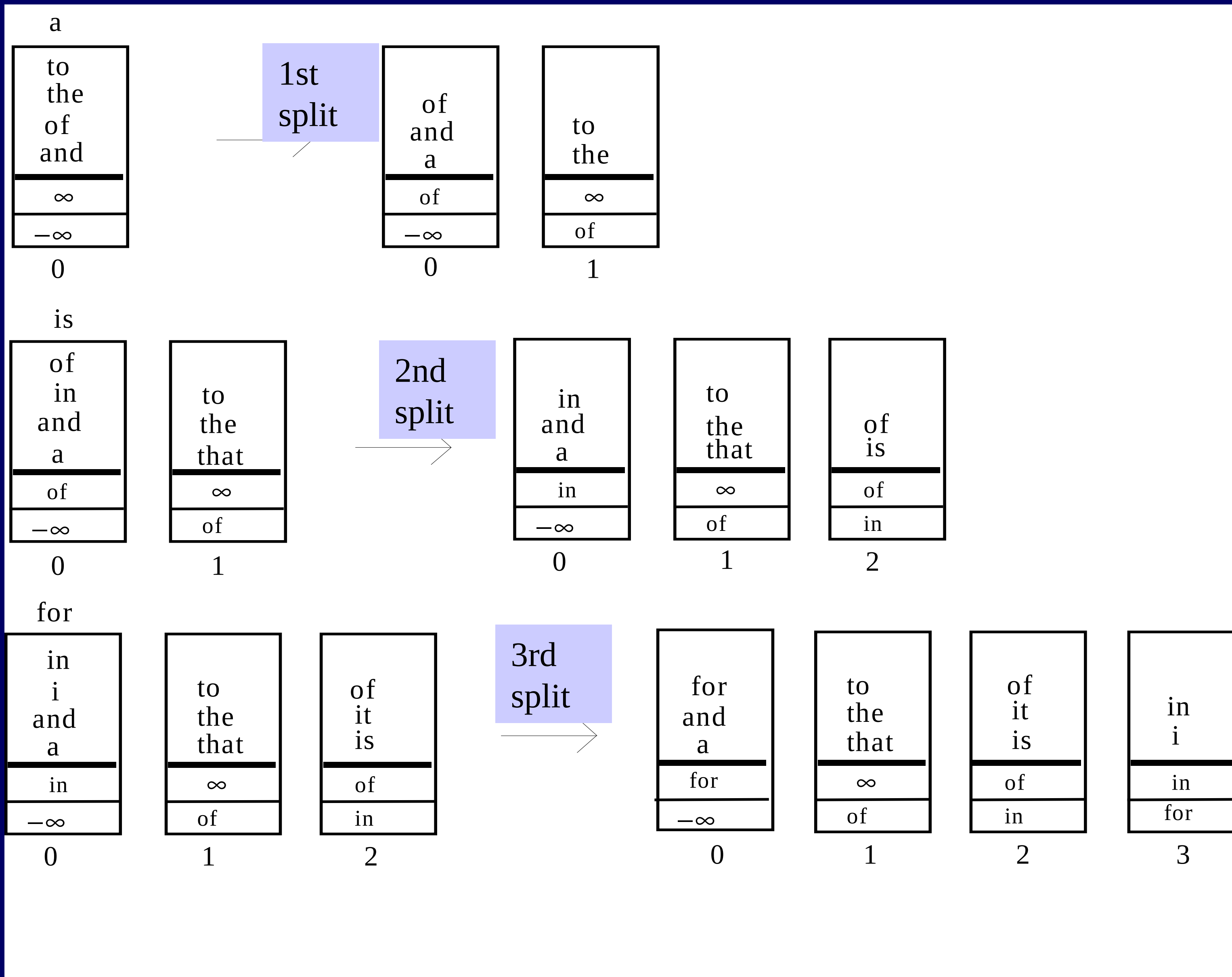
RP* Schemes

- Manage ordered files
  - Range partitioning
  - Bucket splitting using median key
    - Like in a B-tree
- Key search queries
- Range queries
  - Parallel all-records delivery
  - In order pipelined delivery
- Non-key parallel queries (scans)
  - Evaluated locally by servers' AMOS
  - With deterministic or probabilistic termination

RP* Schemes on SDDS-2000

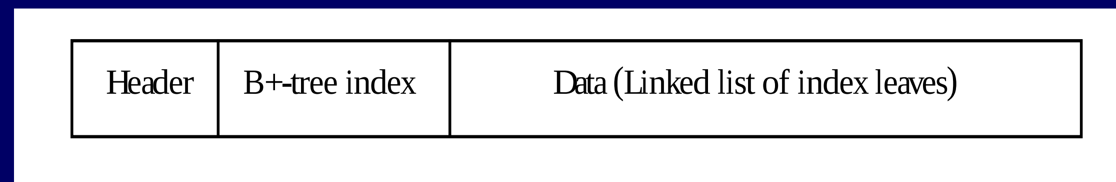
- RP^*_N
  - no index
  - query multicast
- RP^*_C
  - RP^*_N + client index
  - query multicast only if the bucket address not in the index
  - forwarding by multicast
  - RP^*_{Cu}
    - variant of RP^*_C, without multicast by the client
- RP^*_S
  - RP^*_C + servers' index
  - Optional multicast
    - For range queries only

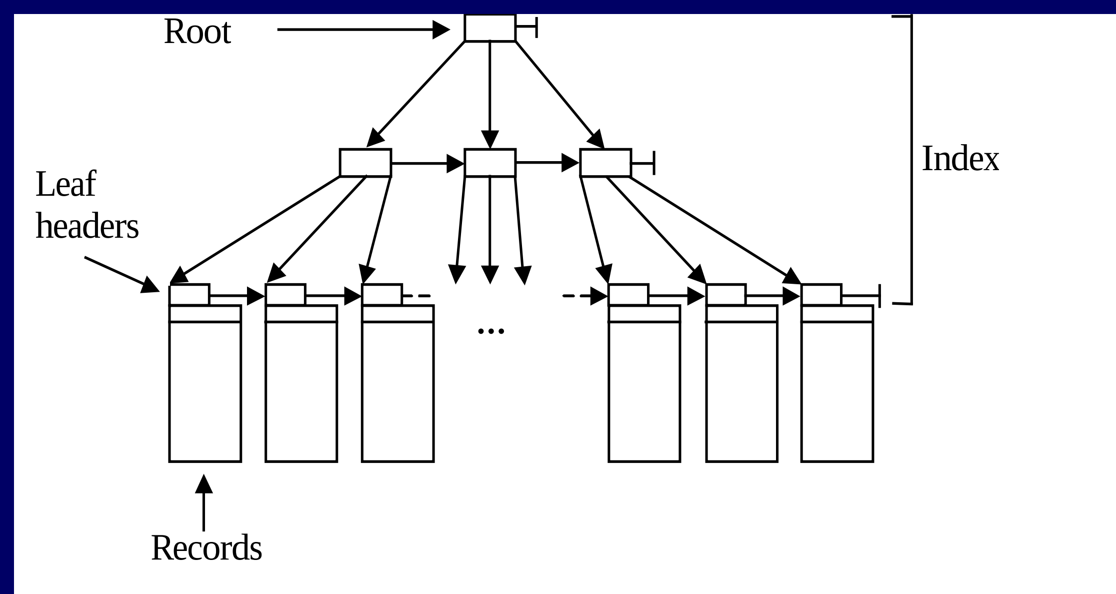
RP* File Expansion



RP* Bucket Structure

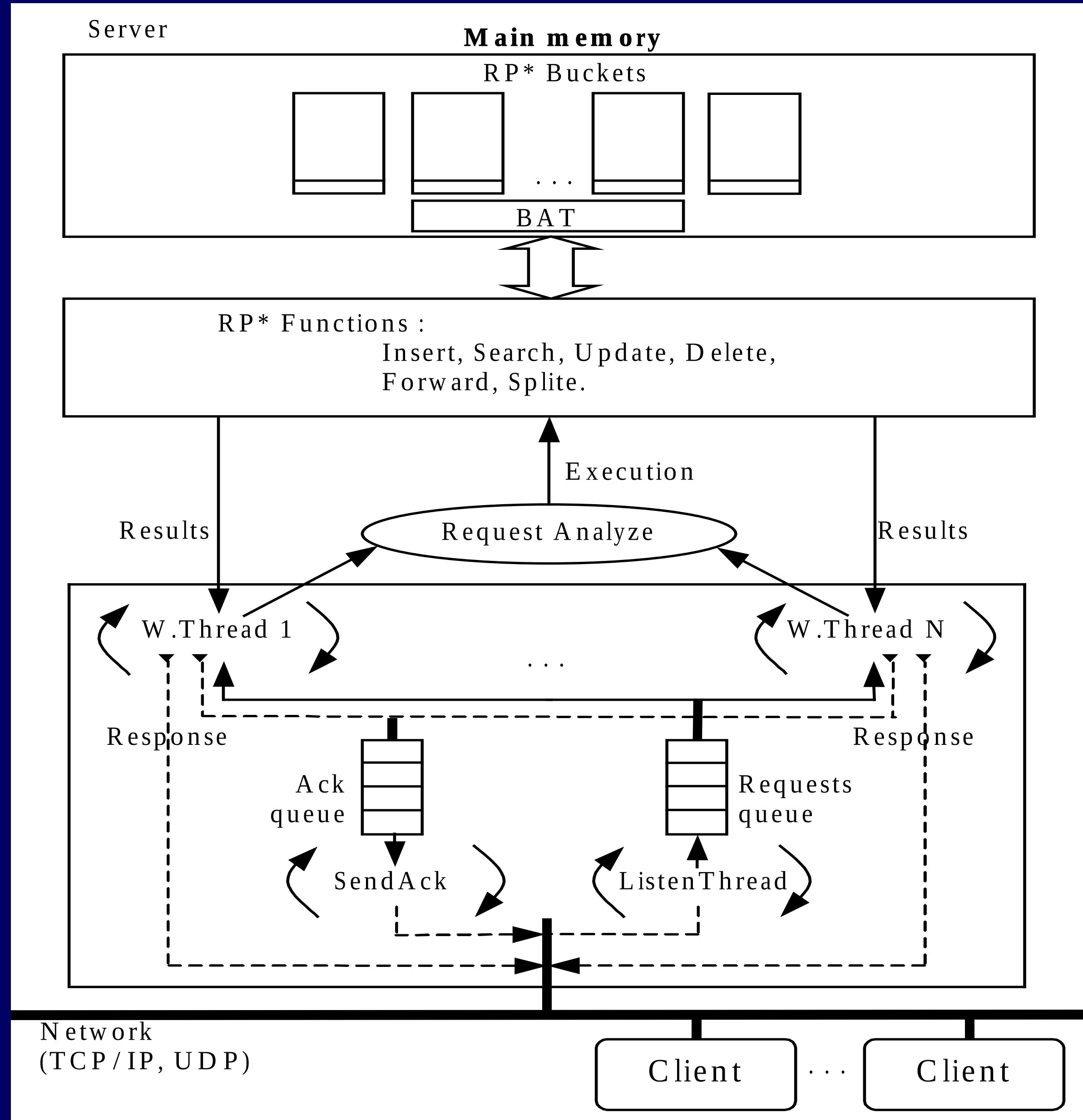
- ***Header***
  - Bucket range
  - Address of the index root
  - Bucket size…
- ***Index***
  - Kind of of B+-tree
  - Additional links
    - for efficient index splitting during RP* bucket splits
- ***Data***
  - Linked leaves with the data





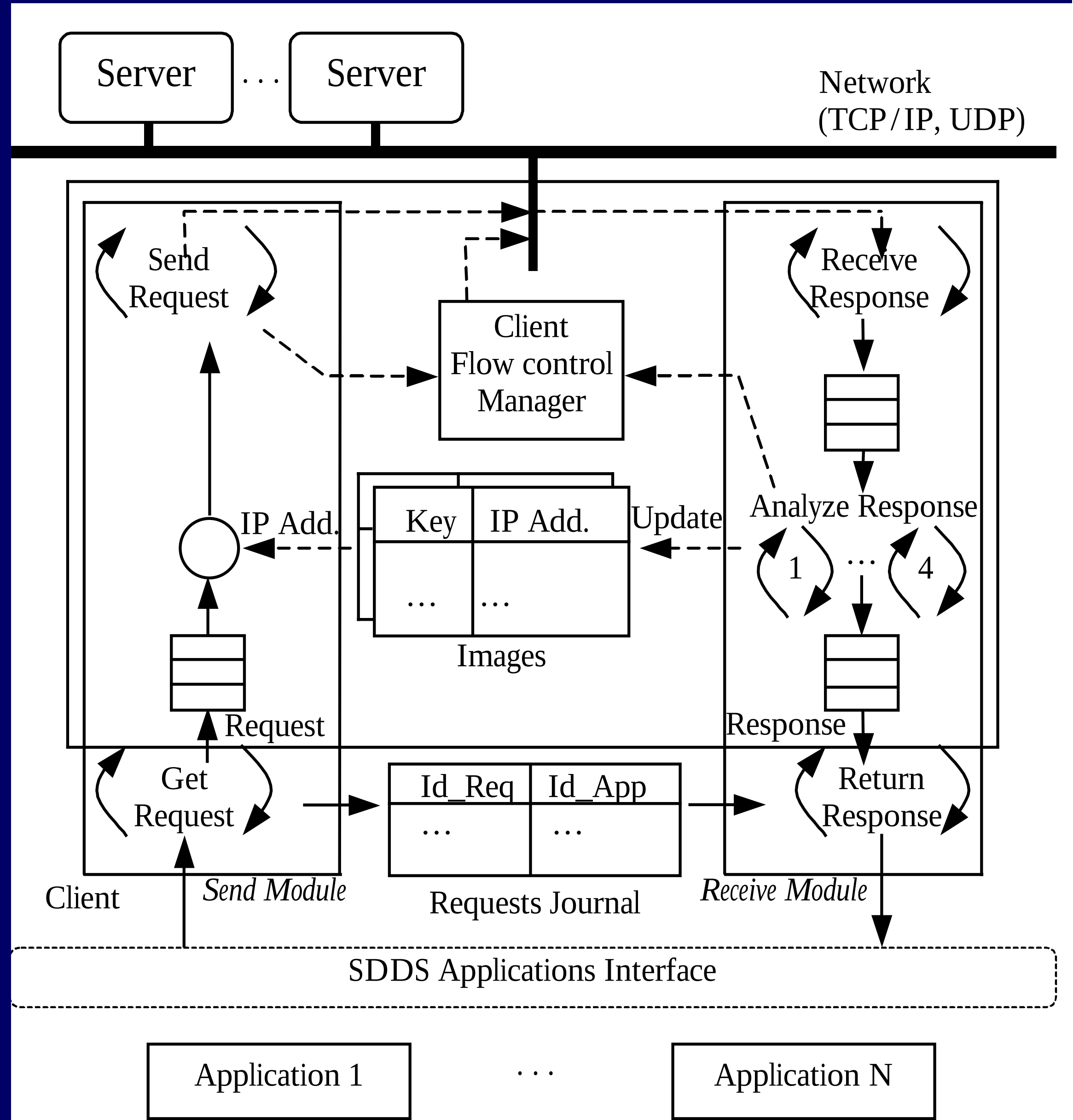
SDDS-2000: Server Architecture

- **Several buckets of different SDDS files**
- **Multithread architecture**
- **Synchronization queues**
- **Listen Thread for incoming requests**
- **SendAck Thread for flow control**
- **Work Threads for**
  - **request processing**
  - **response sendout**
  - **request forwarding**
- **UDP for shorter messages (< 64K)**
- **TCP/IP for longer data exchanges**



SDDS-2000: Client Architecture

- 2 Modules
  - Send Module
  - Receive Module
- Multithread Architecture
  - SendRequest
  - ReceiveRequest
  - AnalyzeResponse1..4
  - GetRequest
  - ReturnResponse
- Synchronization Queues
- Client Images
- Flow control



Performance Analysis

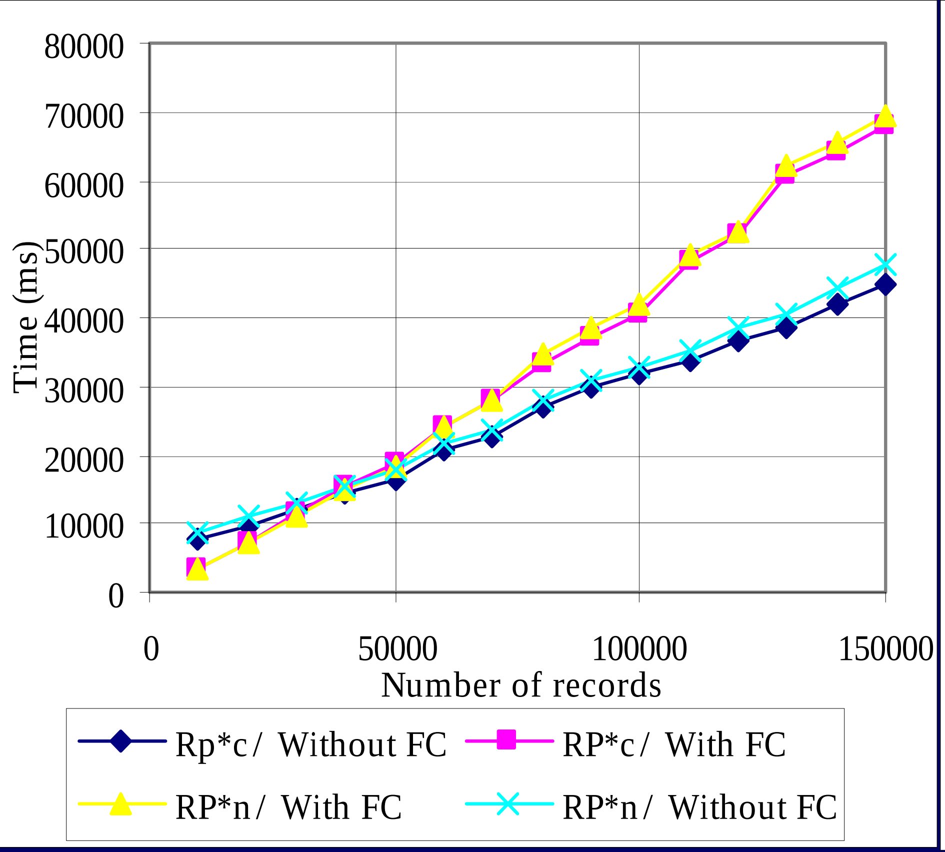
Experimental Environment

- **Six Pentium III 700 MHz**
  - **Windows 2000**
  - – **128 MB of RAM**
  - – **100 Mb/s Ethernet**
- **Messages**
  - – **180 bytes : 80 for the header, 100 for the record**
  - – **Keys are random integers within some interval**
  - – **Flow Control sliding window of 10 messages**
- **Index**
  - –**Capacity of an internal node : 80 index elements**
  - –**Capacity of a leaf : 100 records**

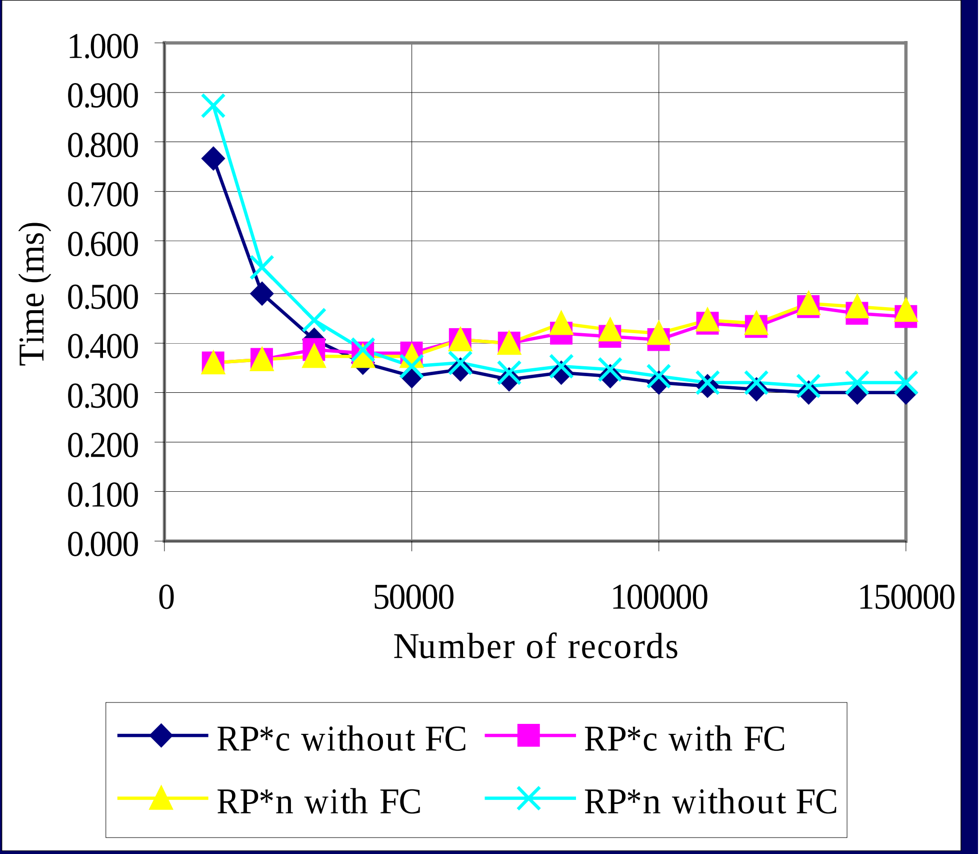
Performance Analysis

File Creation

- Bucket capacity : 50.000 records
- 150.000 random inserts by a single client
- With flow control (FC) or without



File creation time



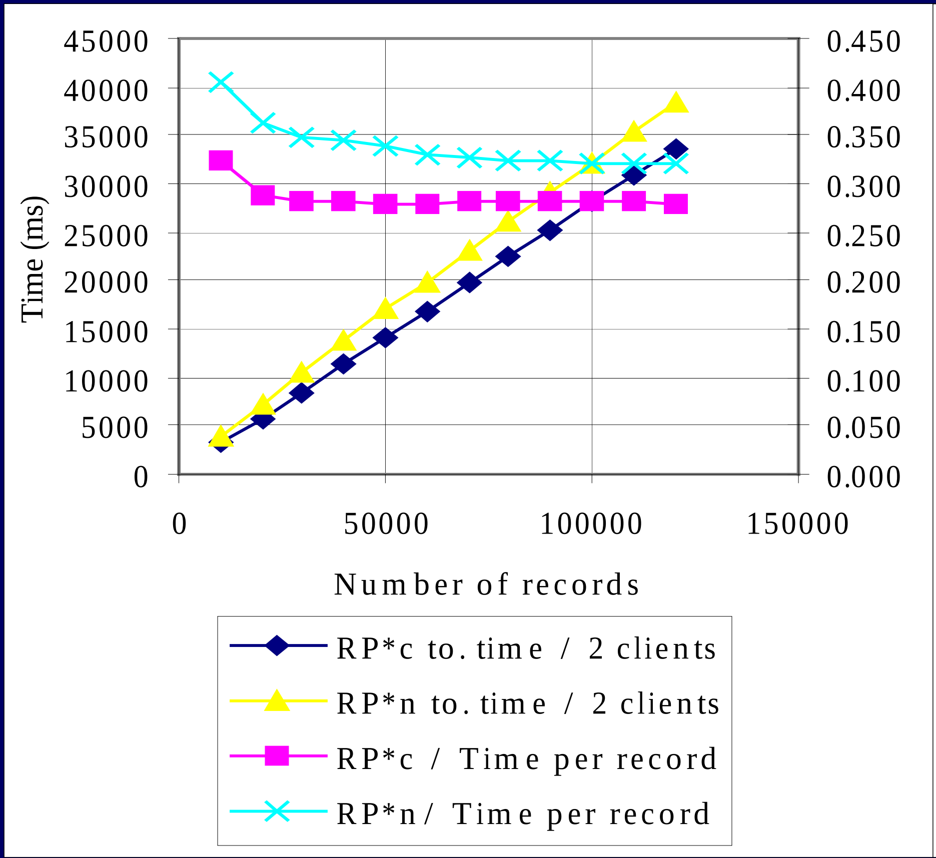
Average insert time

Discussion

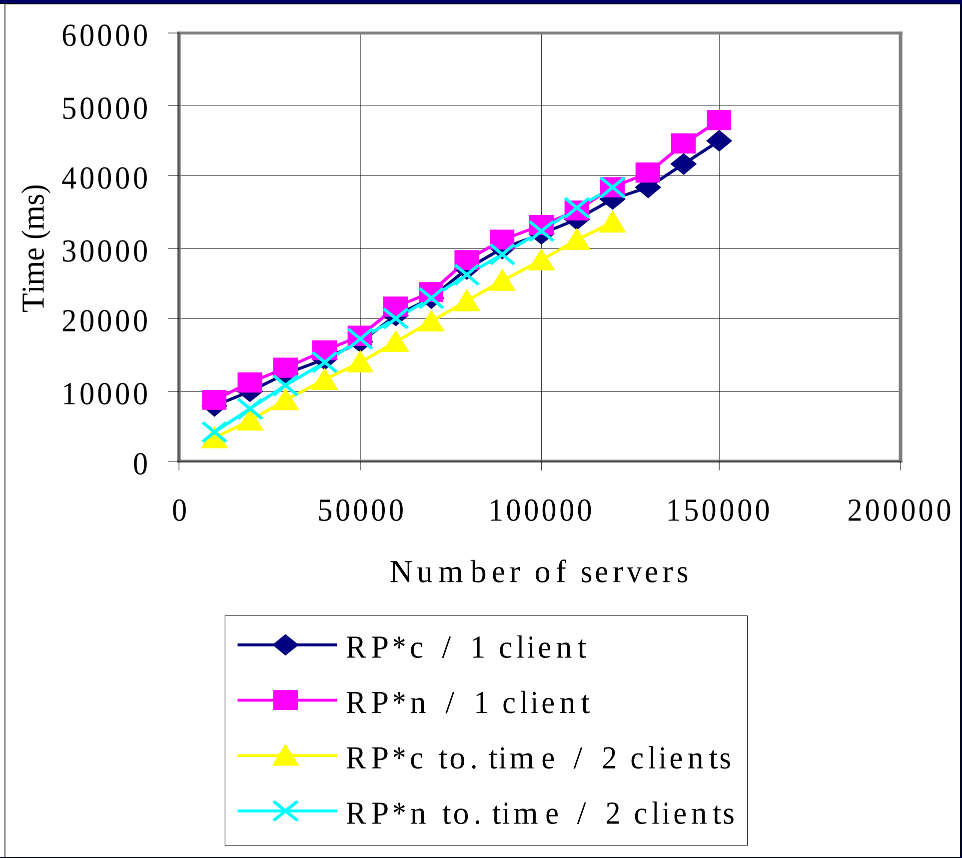
- Creation time is almost linearly scalable
- Flow control is quite expensive
  - Losses without were negligible
- Both schemes perform almost equally well
  - RP*C slightly better
    - As one could expect
- Insert time is 30 faster than for a disk file
- Insert time appears bound by the client speed

Performance Analysis
File Creation

- **File created by 120.000 random inserts by 2 simultaneous clients**
- **Without flow control**



File creation by two clients : total time and per insert



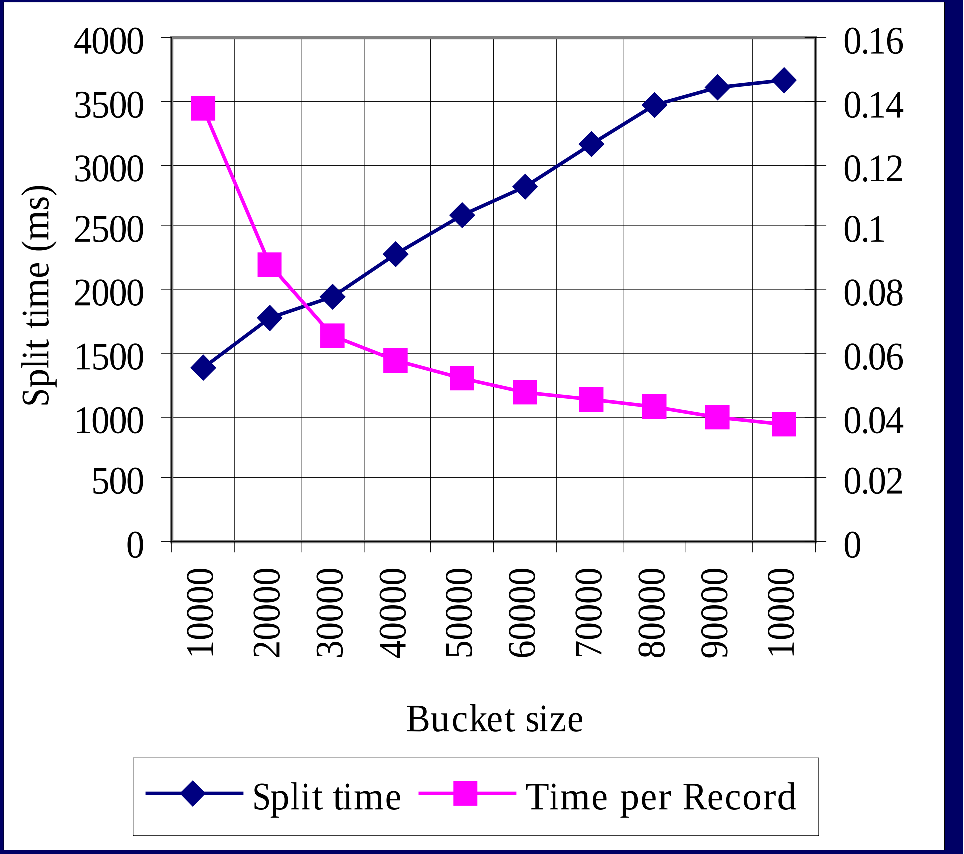
Comparative file creation time by one or two clients

Discussion

- Performance improve
- Insert times appear bound now by a server speed
- More clients would not improve performance of a single server

Performance Analysis
Split Time

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



Split times for different bucket capacity

Discusion

- About linear scalability in function of bucket size
- Larger buckets are more efficient
- Splitting is very efficient
  - Reaching as little as 40 μs per record

Performance Analysis

Insert without splits

- Up to 100000 inserts into *k* buckets ; *k* = 1…5
- Either with empty client image adjusted by IAMs or with correct image

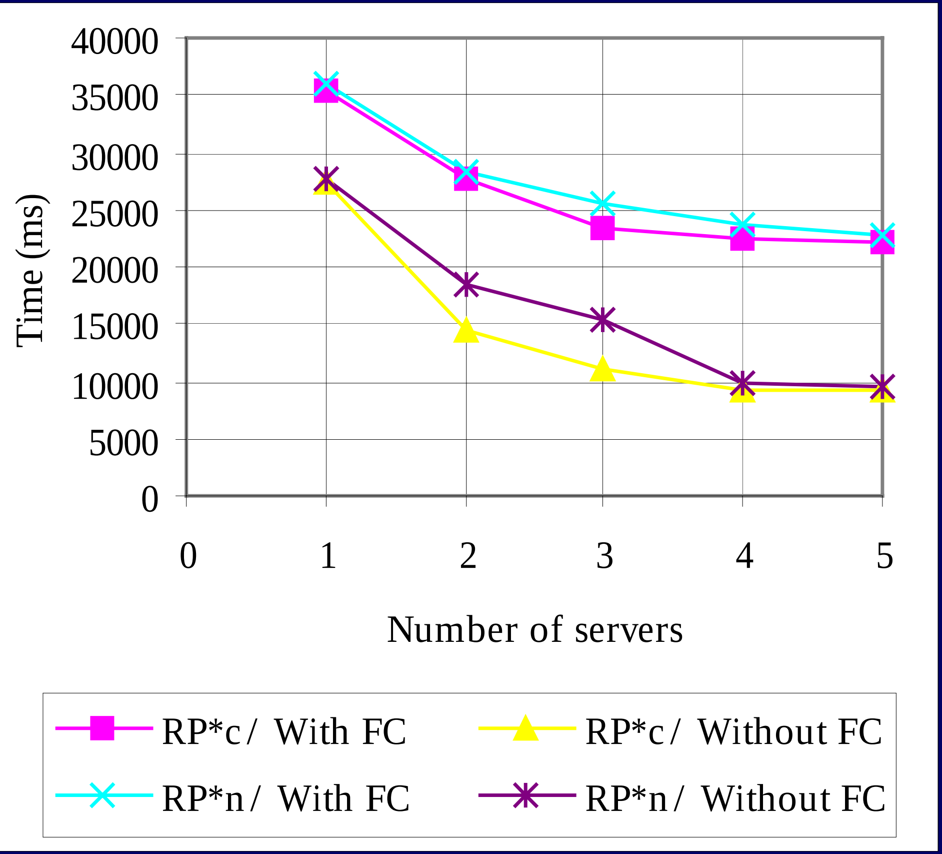
k	RP^*_C						RP^*_N			
	With flow control		Without flow control				With flow control		Without flow control	
			Empty image		Correct image					
	Ttl time	Time/Ins.	Ttl time	Time/Ins.	Ttl time	Time/Ins.	Ttl time	Time/Ins.	Ttl time	Time/Ins.
1	35511	0.355	27480	0.275	27480	0.275	35872	0.359	27540	0.275
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

Insert performance

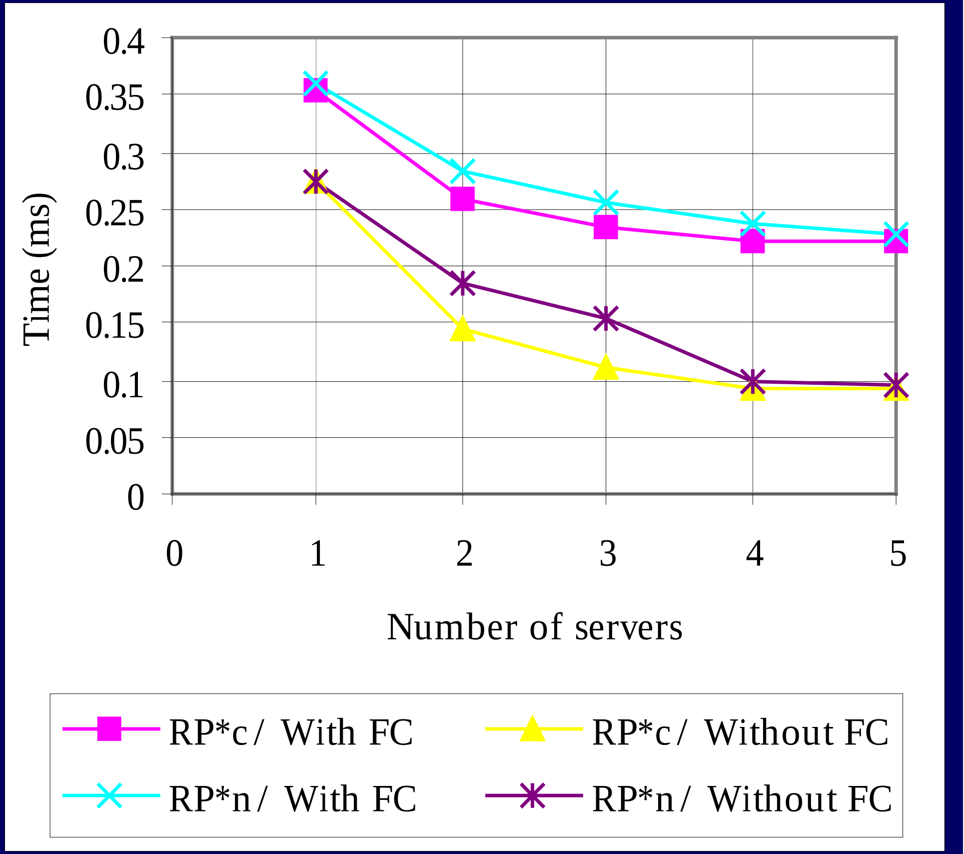
Performance Analysis

Insert without splits

- **100 000 inserts into up to *k* buckets ; *k* = 1...5**
- **Client image initially empty**



Total insert time



Per record time

Discussion

- Cost of IAMs is negligible
- Insert throughput <u>110 times faster than for a disk file</u>
  - 90 µs per insert
- RP*$_N$ appears surprisingly efficient for more buckets closing on RP*c
  - No explanation at present

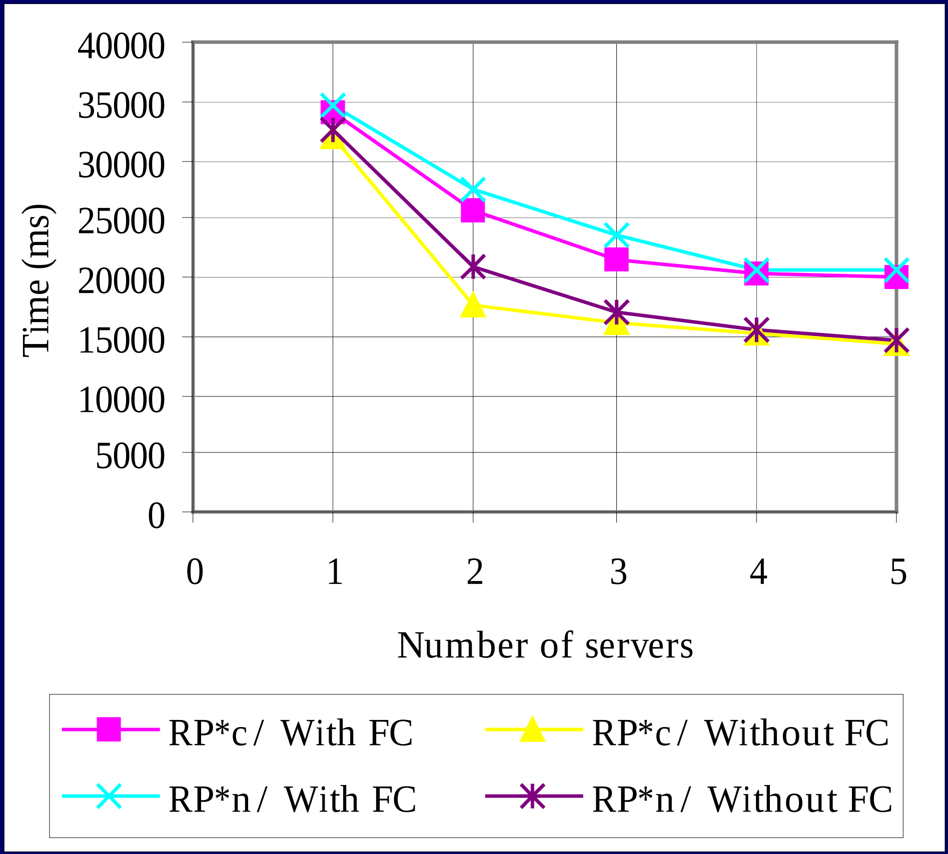
Performance Analysis
Key Search

- A single client sends 100.000 successful random search requests
- The flow control means here that the client sends at most 10 requests without reply

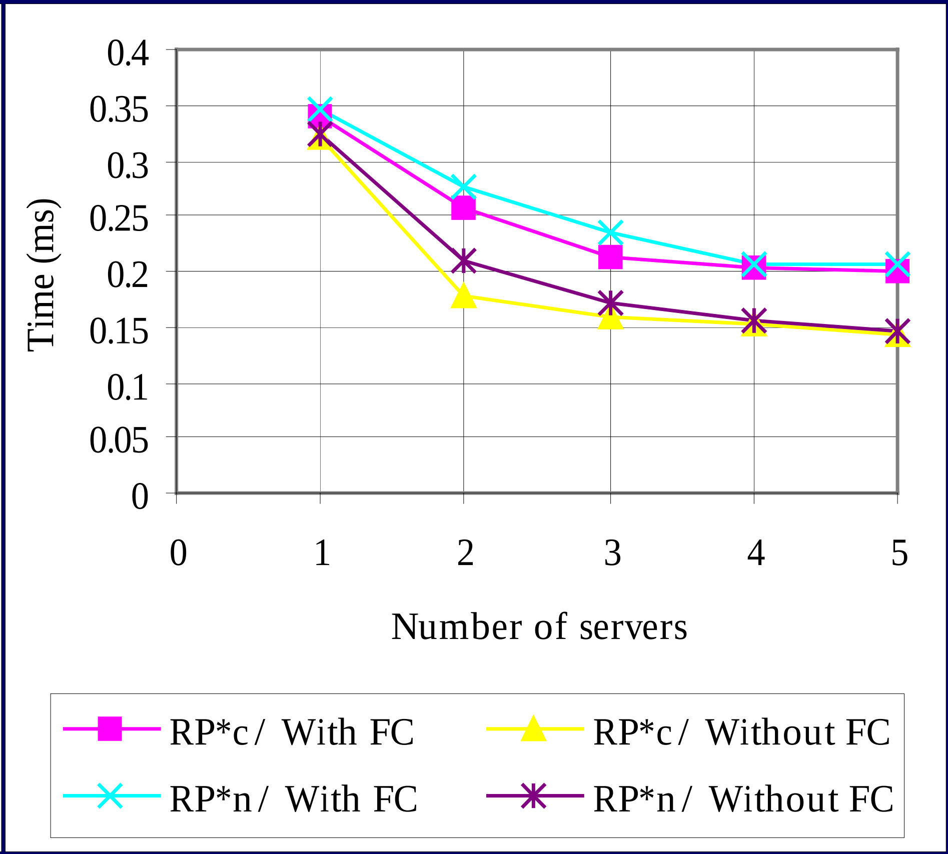
. *k*	RP^*_C				RP^*_N			
	With flow control		Without flow control		With flow control		Without flow control	
	Ttl time	Avg time	Ttl time	Avg time	Ttl time	Avg time	Ttl time	Avg time
1	34019	0.340	32086	0.321	34620	0.346	32466	0.325
2	25767	0.258	17686	0.177	27550	0.276	20850	0.209
3	21431	0.214	16002	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

Search time

Performance Analysis
Key Search



Total search time



Search time per record

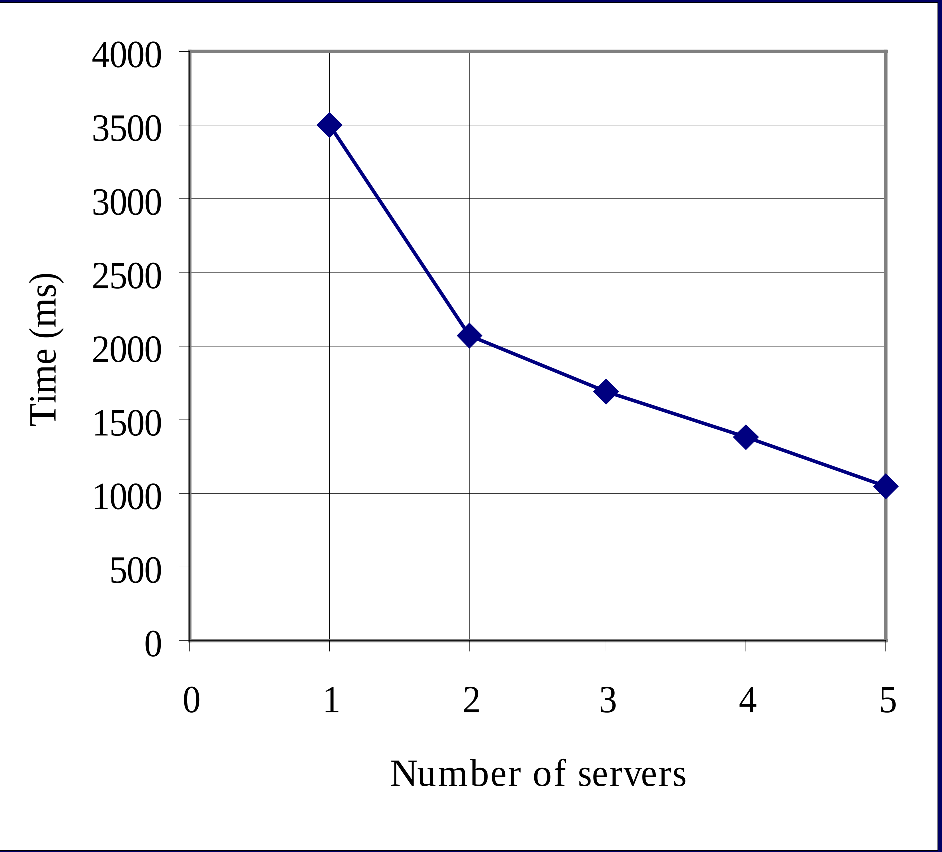
Discussion

- Single search time about 30 times faster than for a disk file
  - 350 µs per insert
- Search throughput more than 65 times faster than that of a disk file
  - 145 µs per insert
- RP*$_N$ appears again surprisingly efficient with respect RP*c for more buckets

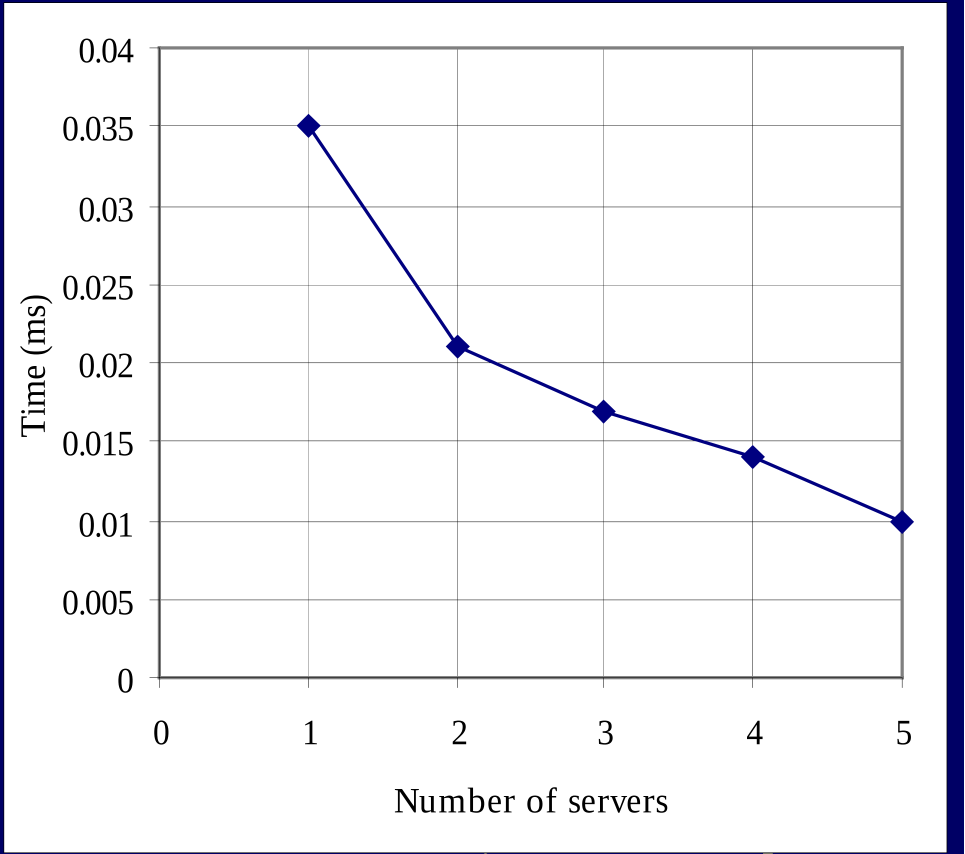
Performance Analysis

Range Query

- Deterministic termination
- Parallel scan of the entire file with all the 100.000 records sent to the client



Range query total time



Range query time per record

Discussion

- Range search appears also very efficient
  - Reaching 100 μs per record delivered
- More servers should further improve the efficiency
  - Curves do not become flat yet

Scalability Analysis

- The largest file at the current configuration
  - 64 MB buckets with *b* = 640 K
  - 448.000 records per bucket loaded at 70 % at the average.
  - 2.240.000 records in total
  - 320 MB of distributed RAM (5 servers)
  - 264 s creation time by a single $RP*_N$ client
  - 257 s creation time by a single $RP*_C$ client
  - A record could reach 300 B
    - The servers RAMs were recently upgraded to 256 MB

Scalability Analysis

- If the example file with *b* = 50.000 had scaled to 10.000.000 records
  - It would span over 286 buckets (servers)
    - There are many more machines at Paris 9
  - Creation time by random inserts would be
    - 1235 s for RP^*_N
    - 1205 s for RP^*_C
  - 285 splits would last 285 s in total
  - Inserts alone would last
    - 950 s for RP^*_N
    - 920 s for RP^*_C

Related Works

	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

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

Discussion

- The 1994 theoretical performance predictions for RP* were quite accurate
- RP* schemes at SDDS-2000 appears surprisingly globally more efficient than LH*
  - No explanation at present

Conclusion

- SDDS-2000 : a prototype SDDS manager for Windows multicomputer
  - Various SDDSs
  - Several variants of the RP*
- Performance of RP* schemes appears in line with the expectations
  - Access times in the range of a fraction of a millisecond
  - About 30 to 100 times faster than a disk file access performance
  - About ideal (linear) scalability
- Results prove also the overall efficiency of SDDS-2000 architecture

Future work

- Performance analysis
  - Larger files
- High-availability RP* schemes using RS codes
- Experimental applications

End

Work was partly supported by Microsoft Research

Earlier work on SDDS-2000 partly supported by

HP Laboratories, Palo Alto, CA,

IBM Almaden Res. Cntr., San Jose, CA.