Scalable Distributed Data Structures

State-of-the-art

Part 1

Witold Litwin

Paris 9

litwin@dauphine.fr

Plan

- What are SDDSs ?
- Why they are needed ?
- Where are we in 1996 ?
  - Existing SDDSs
  - Gaps & On-going work
- Conclusion
  - Future work

What is an SDDS

- A new type of data structure
  - Specifically for multicomputers
- Designed for high-performance files :
  - horizontal scalability to very large sizes
    - larger than any single-site file
  - parallel and distributed processing
    - especially in (distributed) RAM
      - access time better than for any disk file
        - 200 µs under NT (100 Mb/s net, 1KB records)
  - distributed autonomous clients

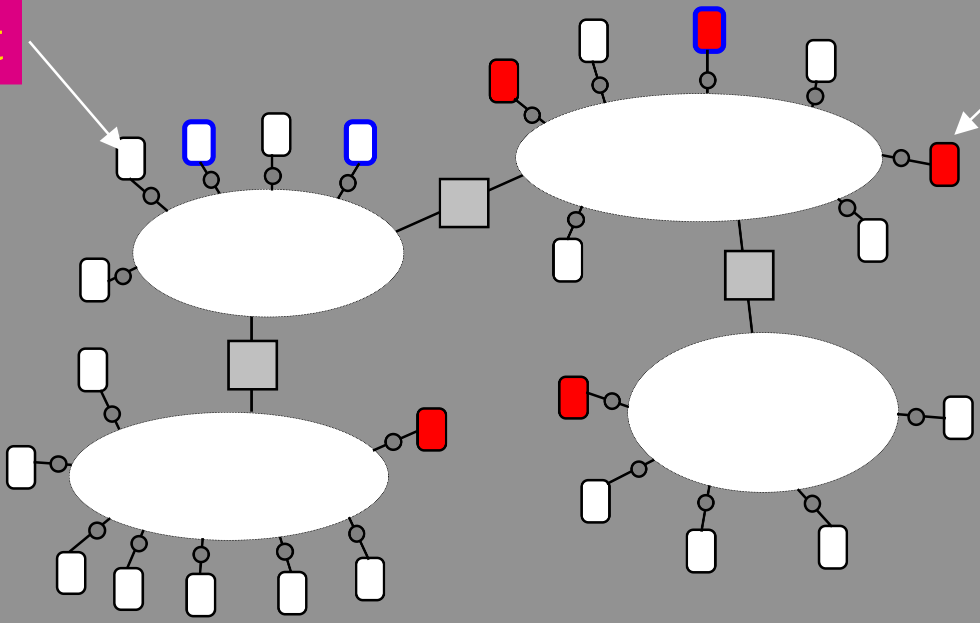
Killer apps

- Storage servers
  - software & hardware scalable & HA servers
  - commodity component based
    - Do-It-Yourself-RAID
- Object storage servers
- Object-relational databases
- WEB servers
  - like Inktomi
- Video servers
- Real-time systems
- HP Scientific data processing

Multicomputers

- A collection of loosely coupled computers
  - common and/or preexisting hardware
  - share nothing architecture
  - message passing through high-speed net (≥ 10 Mb/s)
    - Network multicomputers
      - use general purpose nets
        - LANs: Ethernet, Token Ring, Fast Ethernet, SCI, FDDI...
        - WANs: ATM...
    - Switched multicomputers
      - use a bus, or a switch
        - e.g., IBM-SP2, Parsytec

Network multicomputer



Client

Server

Why multicomputers ?

- Potentially unbeatable price-performance ratio
  - Much cheaper and more powerful than supercomputers
    - 1500 WSs at HPL with 500+ GB of RAM & TBs of disks
- Potential computing power
  - file size
  - access and processing time
  - throughput
- For more pro & cons :
  - ***Bill Gates at Microsoft Scalability Day***
  - *NOW project (UC Berkeley)*
  - *Tanenbaum: "Distributed Operating Systems", Prentice Hall, 1995*
  - *www.microoft.com White Papers from Business Syst. Div.*

Why SDDSs

- Multicomputers need data structures and file systems
- Trivial extensions of traditional structures are not best
  - hot-spots
  - scalability
  - parallel queries
  - distributed and autonomous clients
  - distributed RAM & distance to data

Distance to data

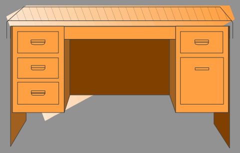
(Jim Gray)

10 msec	local disk
100 µsec	distant RAM (Ethernet)
1 µsec	distant RAM (gigabit net)
100 ns	RAM

Distance to data

10 msec	local disk
100 µsec	distant RAM (Ethernet)
1 µsec	distant RAM (gigabit net)
100 nsec	RAM

1 min

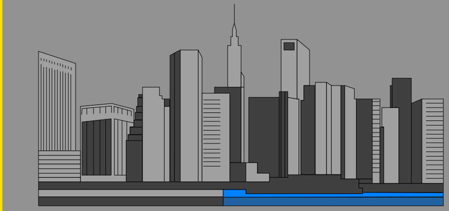


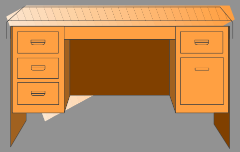
Distance to data

10 msec	local disk	
100 µsec	distant RAM (Ethernet)	
1 µsec	distant RAM (gigabit net)	**10 min**
100 ns	RAM	**1 min**

Distance to data

10 msec	local disk	
100 µsec	distant RAM (Ethernet)	**2 hours**
1 µsec	distant RAM (gigabit net)	**10 min**
100 ns	RAM	**1 min**

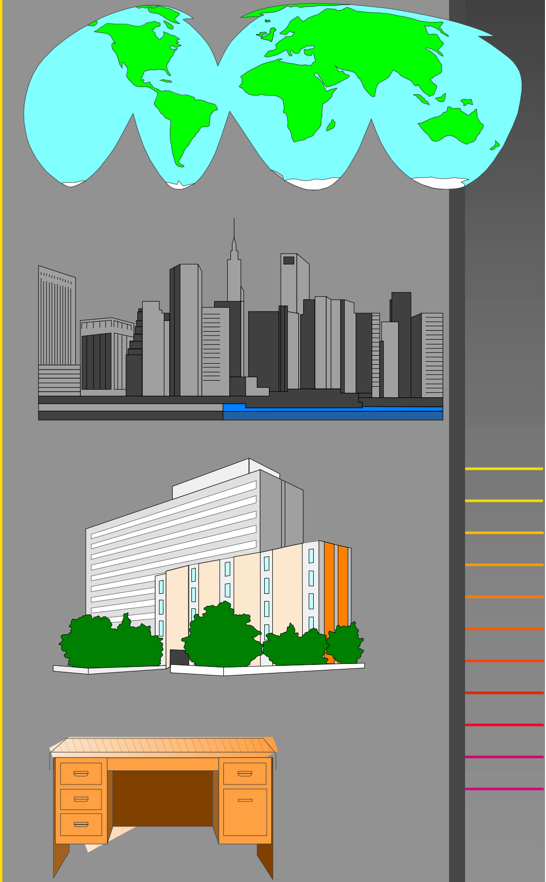




Distance to data

lune

10 msec	local disk	**8 days**
100 µsec	distant RAM (Ethernet)	**2 hours**
1 µsec	distant RAM (gigabit net)	**10 min**
100 ns	RAM	**1 min**



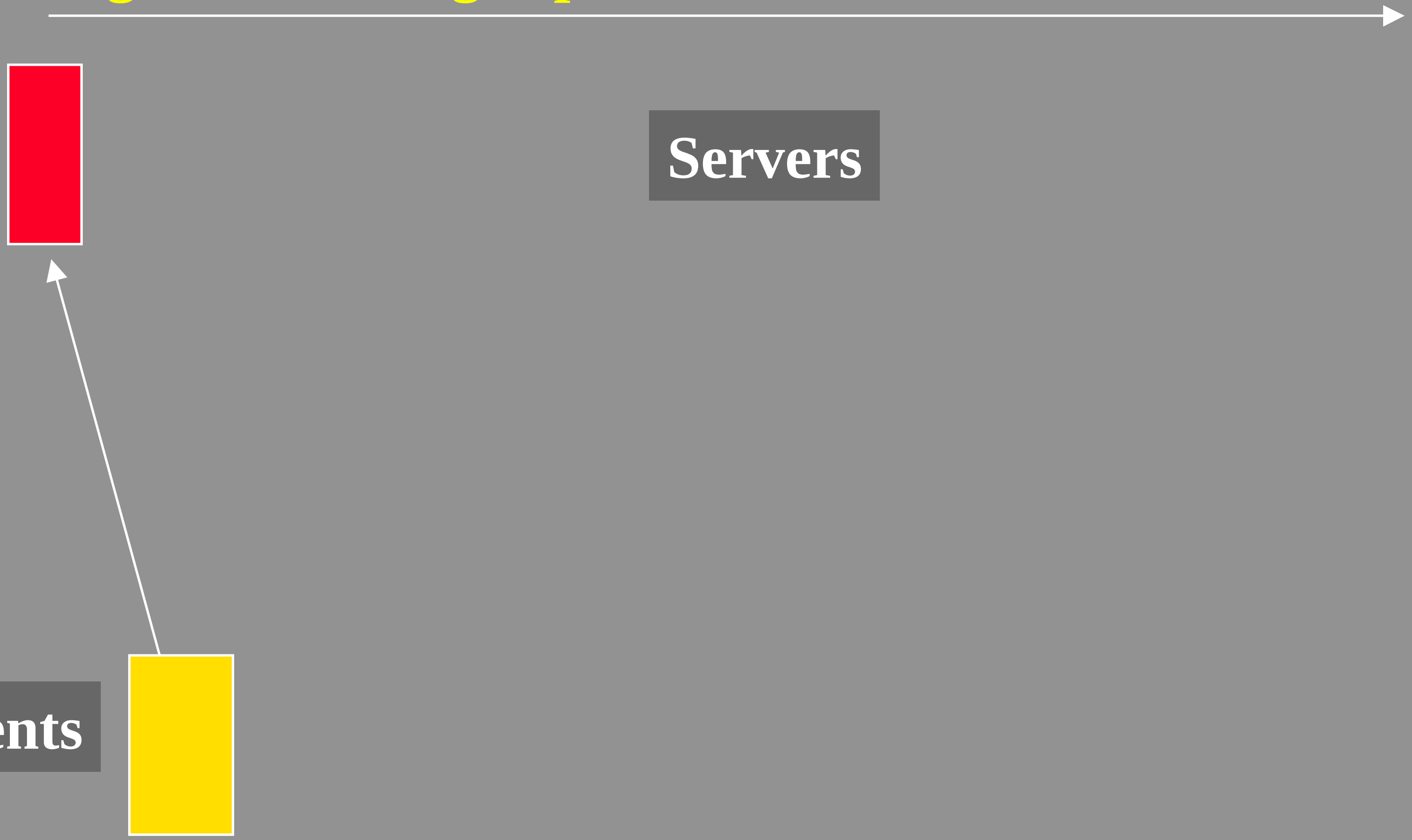
Economy etc.

- Price of RAM storage dropped in 1996 almost 10 times !
  - $10 for 16 MB (production price)
  - ~~$30-40 for 16 MB RAM (end user price)~~
    - $47 for 32 MB (Fry's price, Aug. 1997)
  - $1000 for 1GB
- RAM storage is eternal (no mech. parts)
- RAM storage can grow incrementally
- NT plans for 64b addressing for VLM
- MS plans for VLM-DBMS

What is an SDDS

- A **scalable** data structure where:
- ☞ Data are on servers
  - – always available for access
- ☞ Queries come from autonomous clients
  - – available for access only on their initiative
- ☞ There is no centralized directory
- ☞ Clients may make addressing errors
  - » Clients have less or more adequate image of the actual file structure
- ✌ Servers are able to forward the queries to the correct address
  - – perhaps in several messages
- ☞ Servers may send Image Adjustment Messages
  - » Clients do not make same error twice

An SDDS

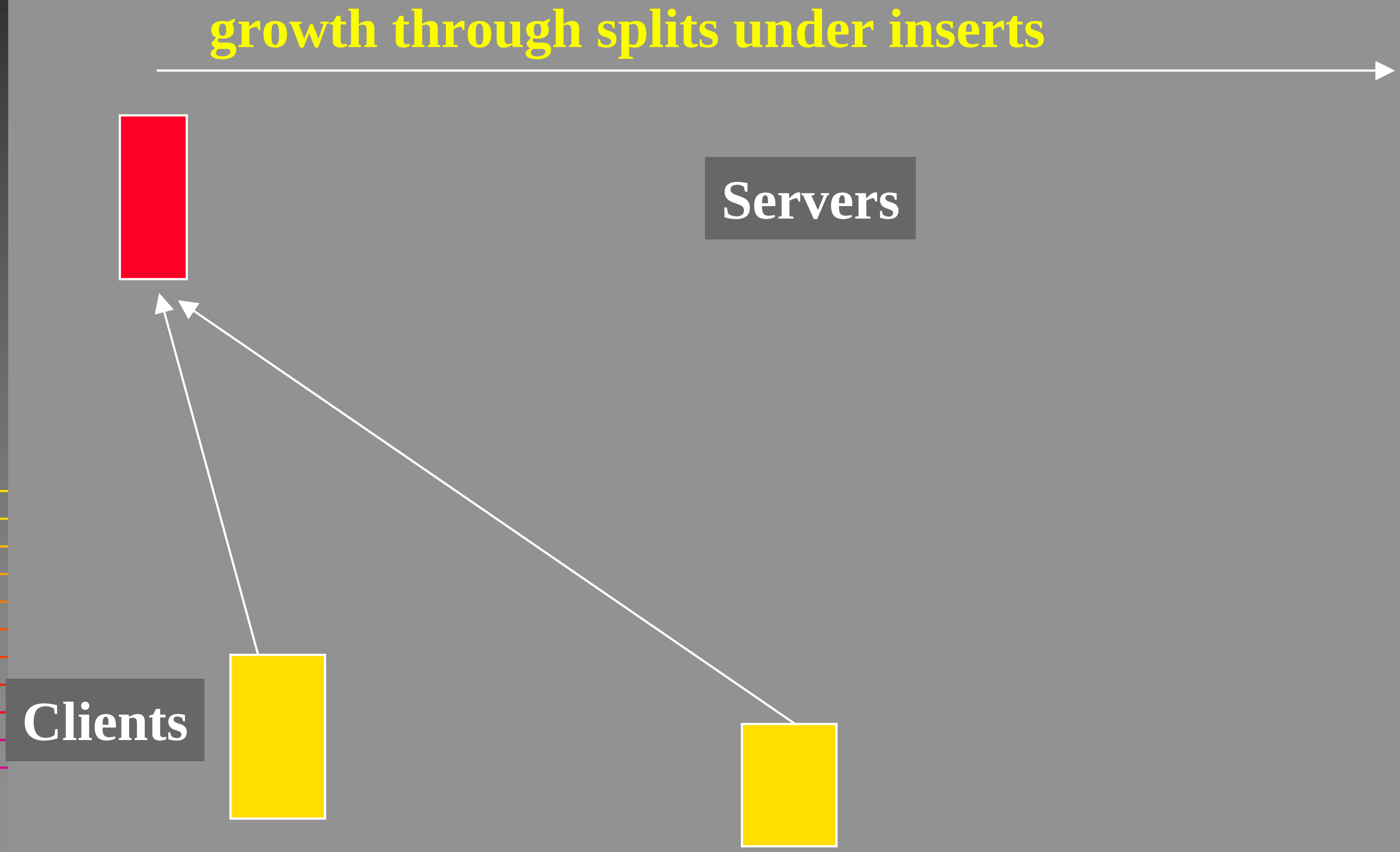


growth through splits under inserts

Servers

Clients

An SDDS

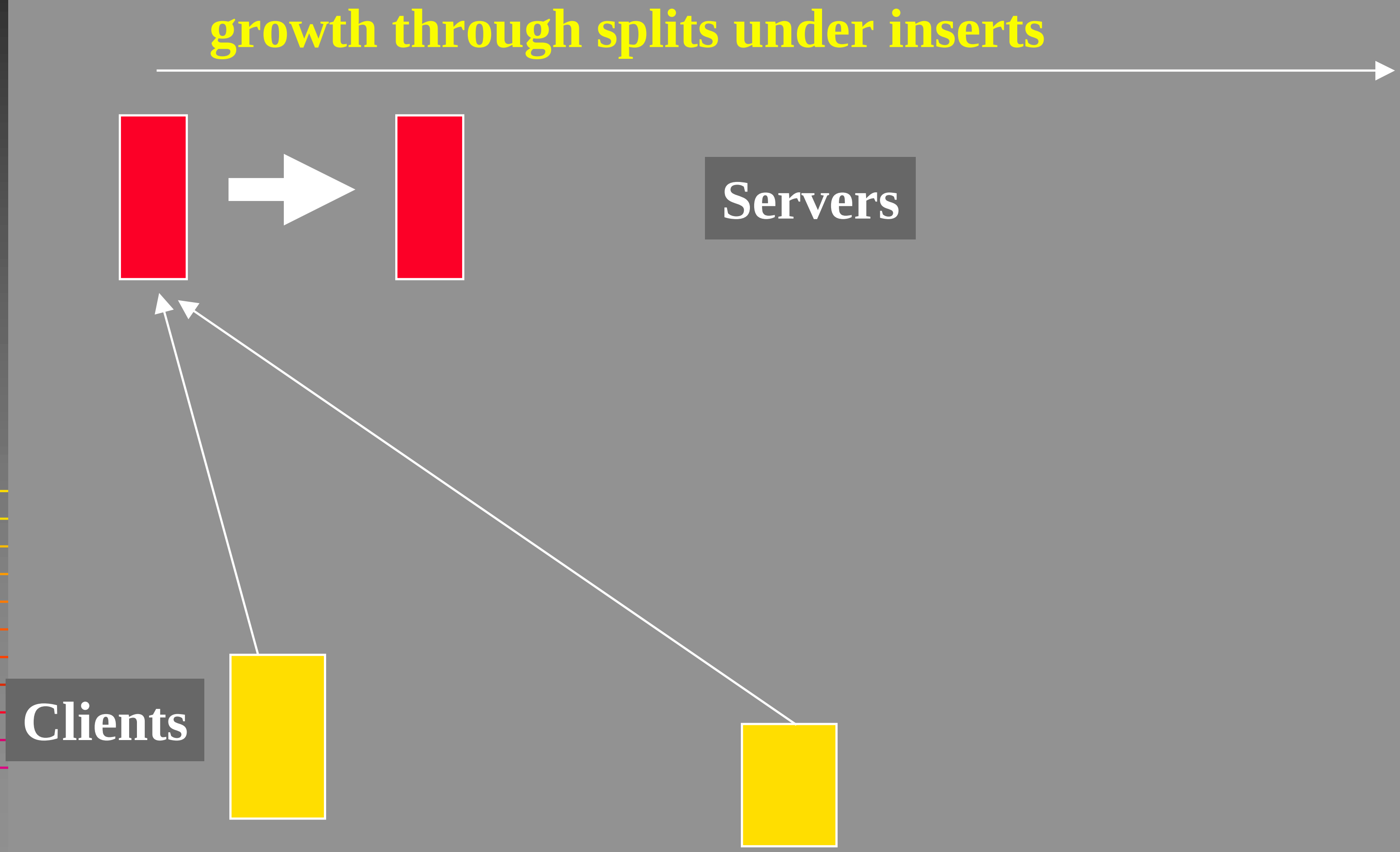


growth through splits under inserts

Servers

Clients

An SDDS

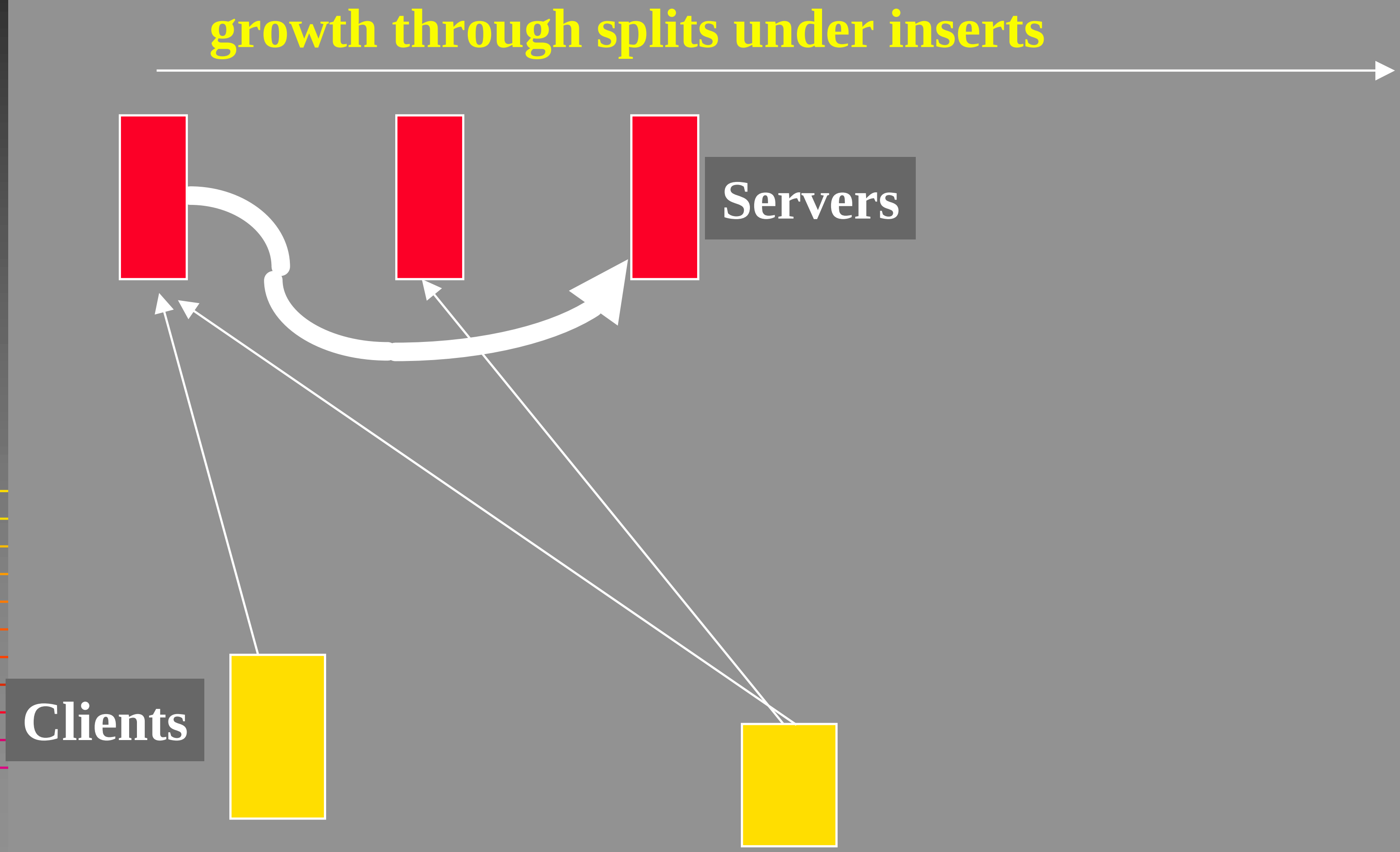


growth through splits under inserts

Servers

Clients

An SDDS

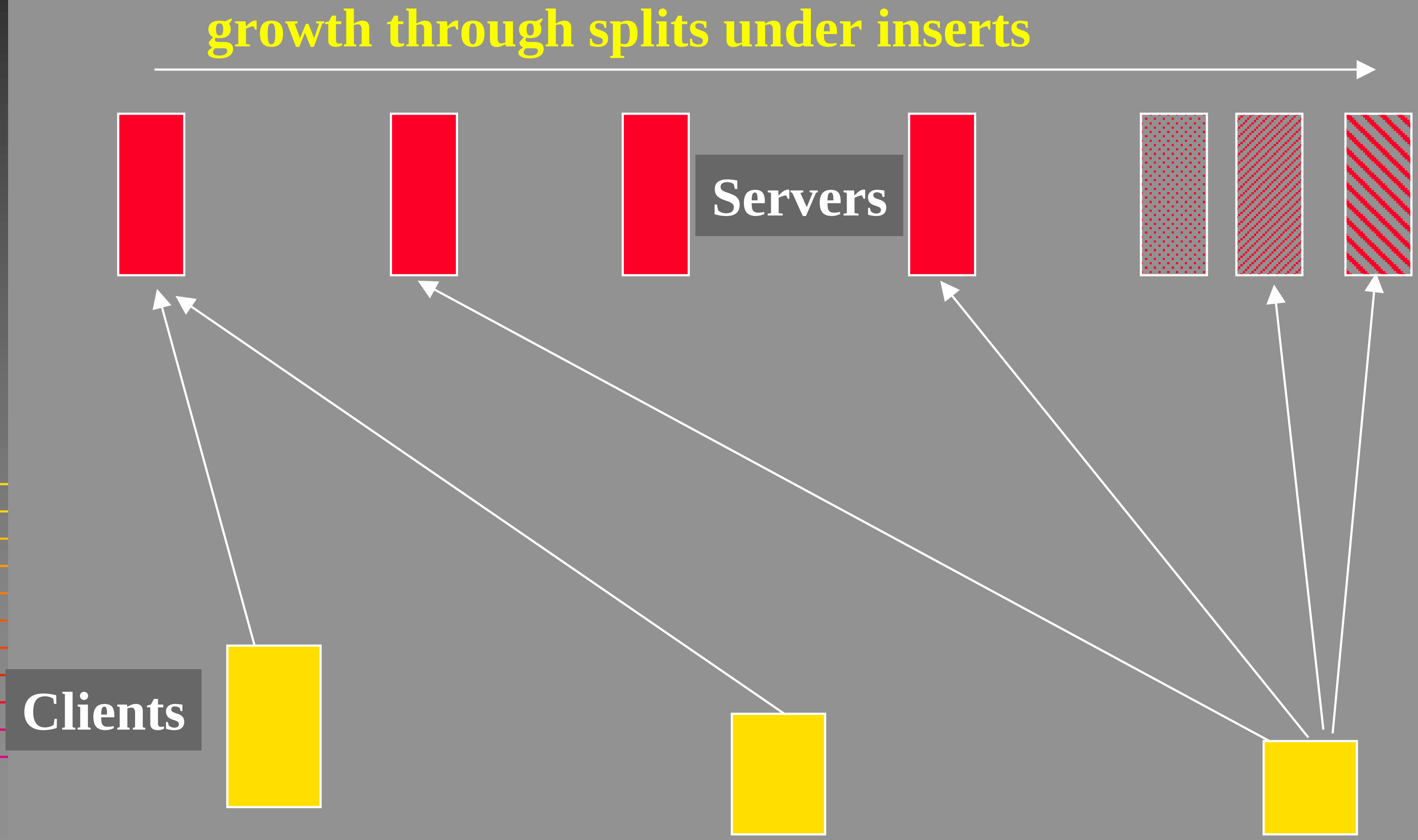


growth through splits under inserts

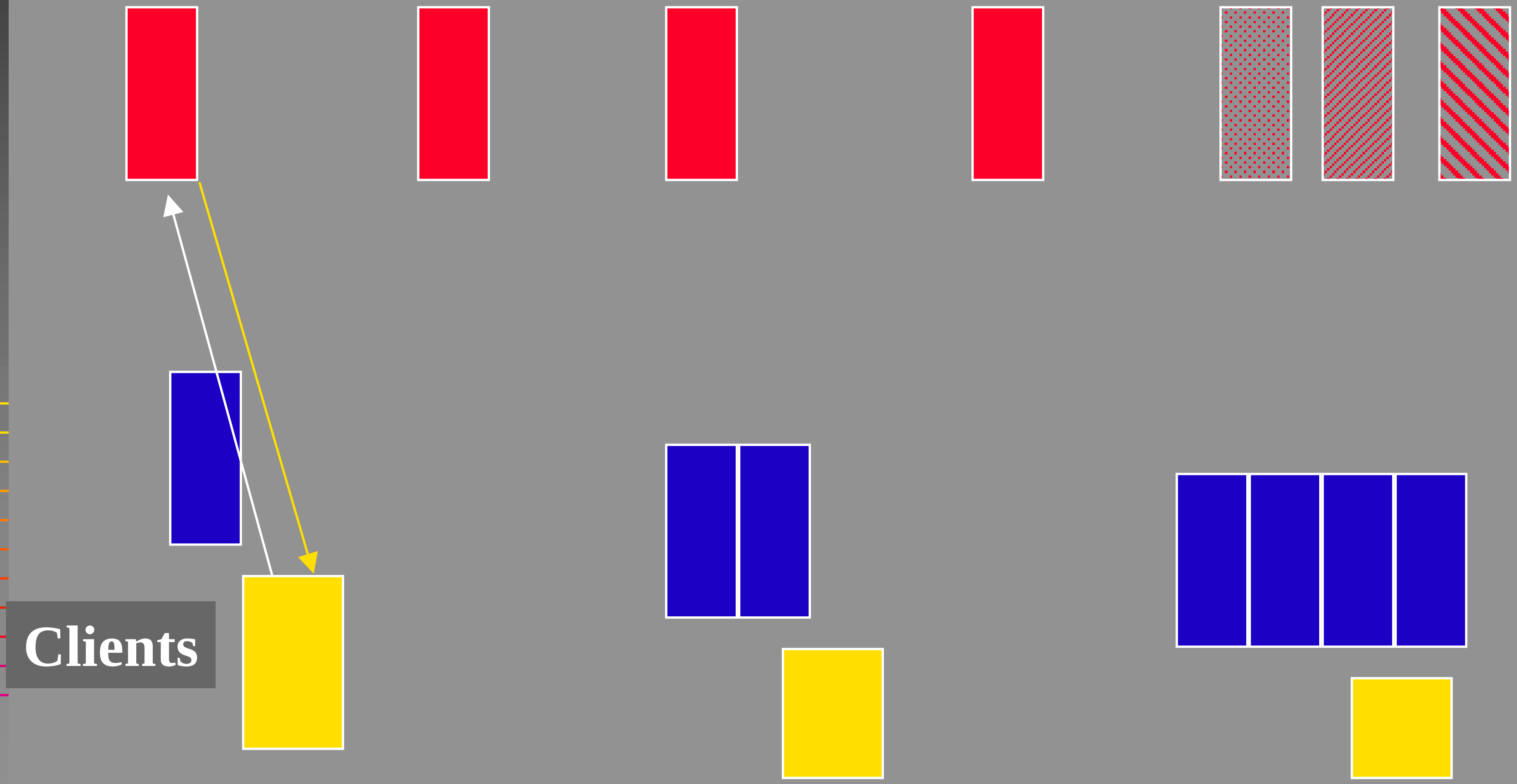
Servers

Clients

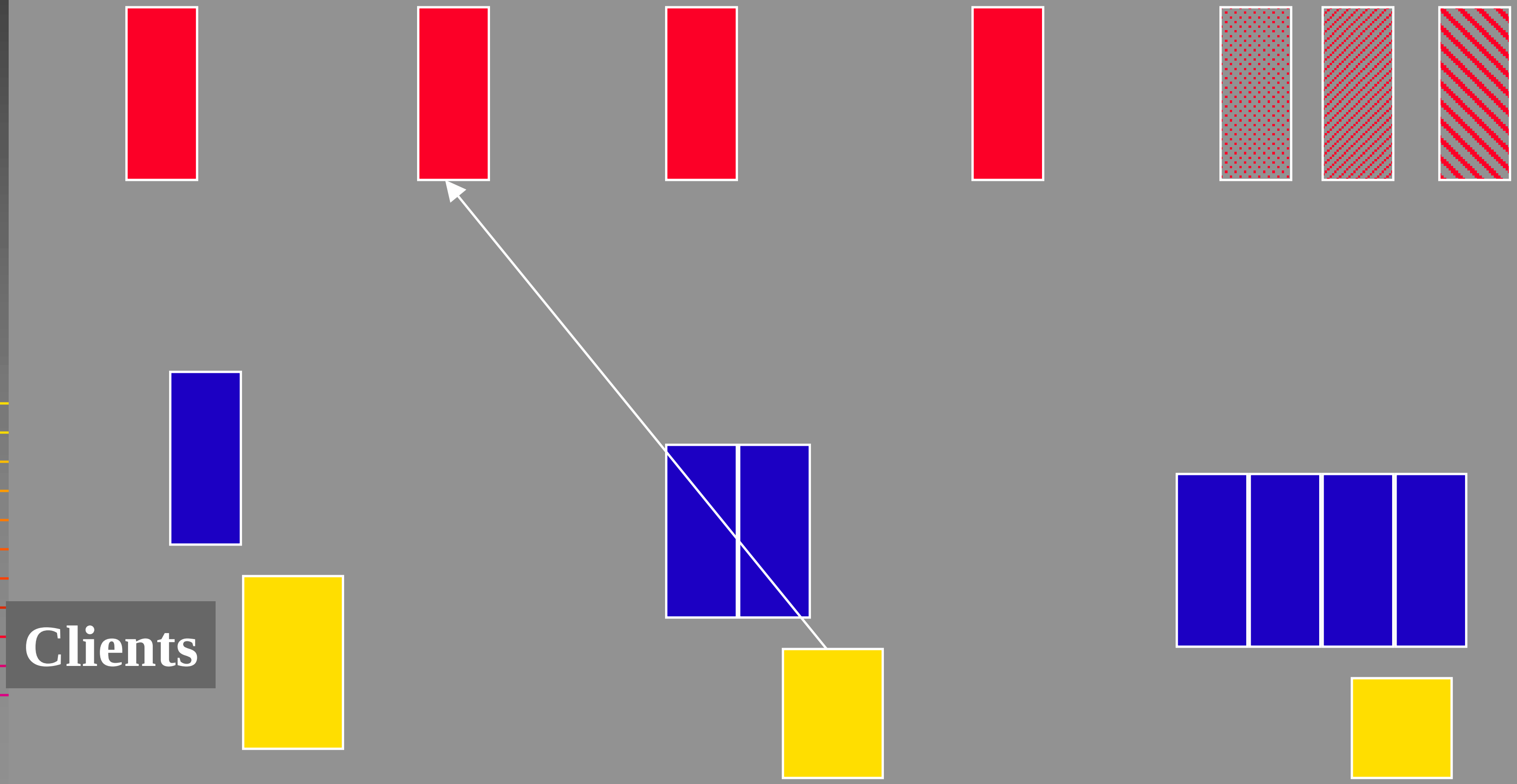
An SDDS



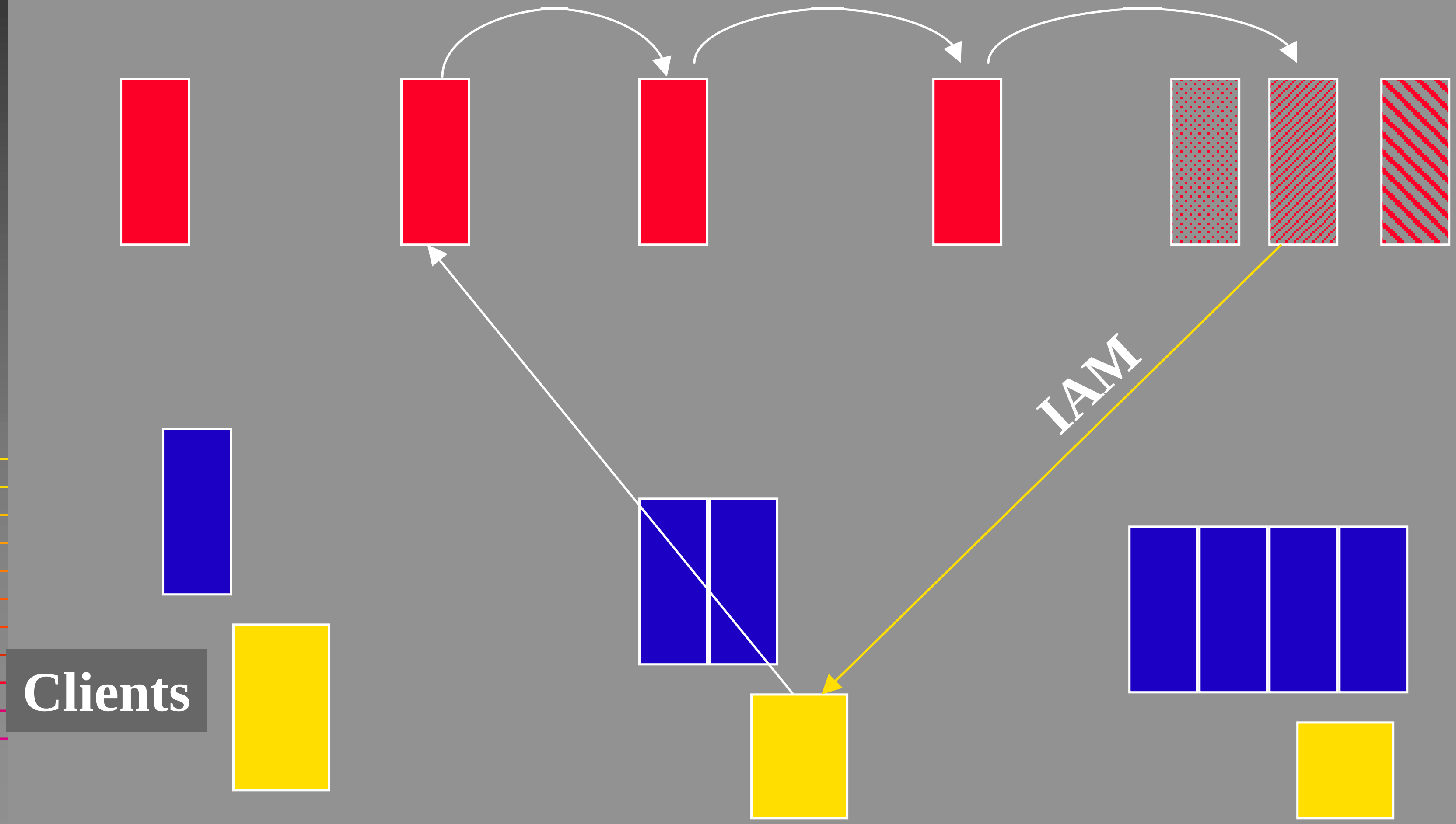
An SDDS



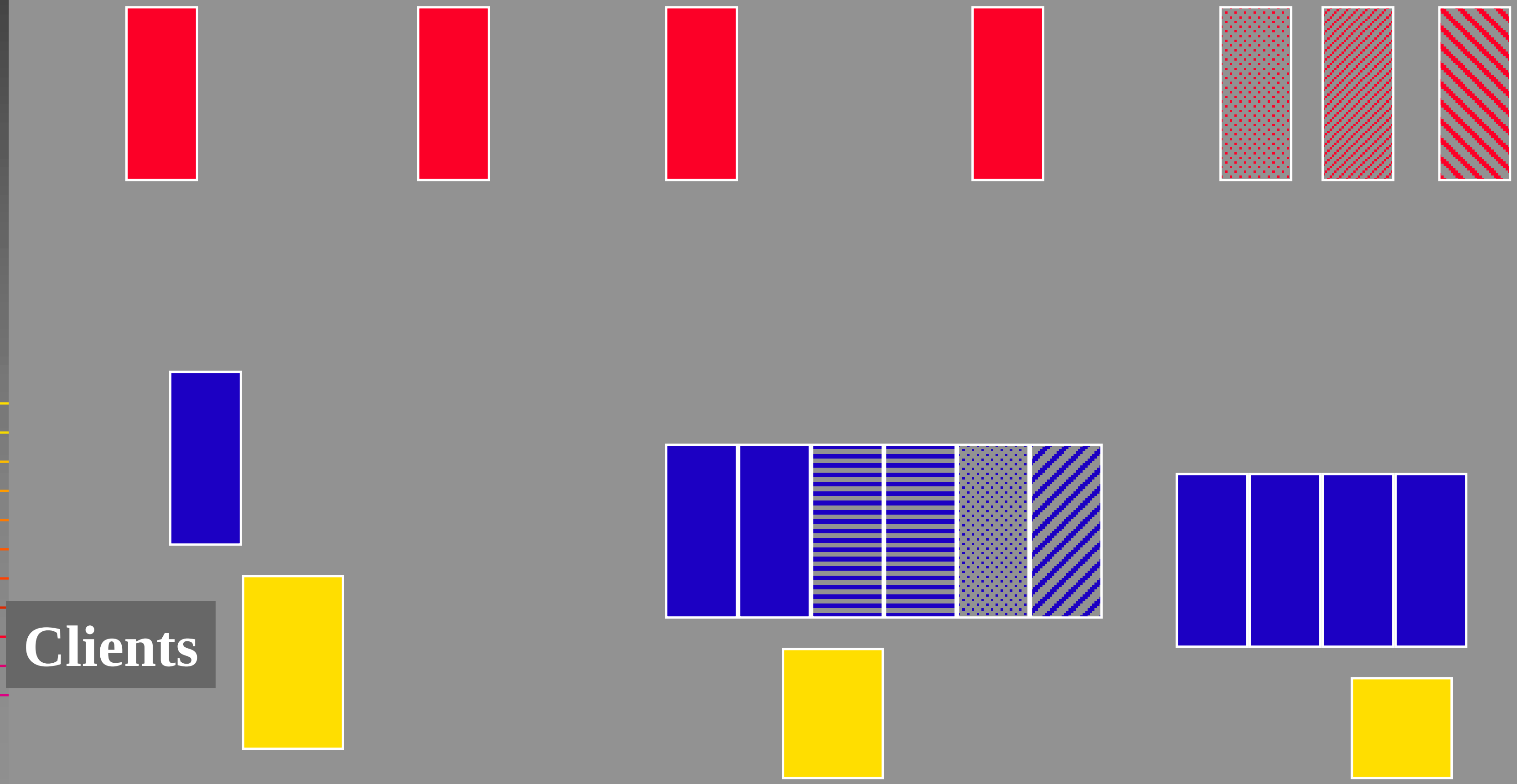
An SDDS



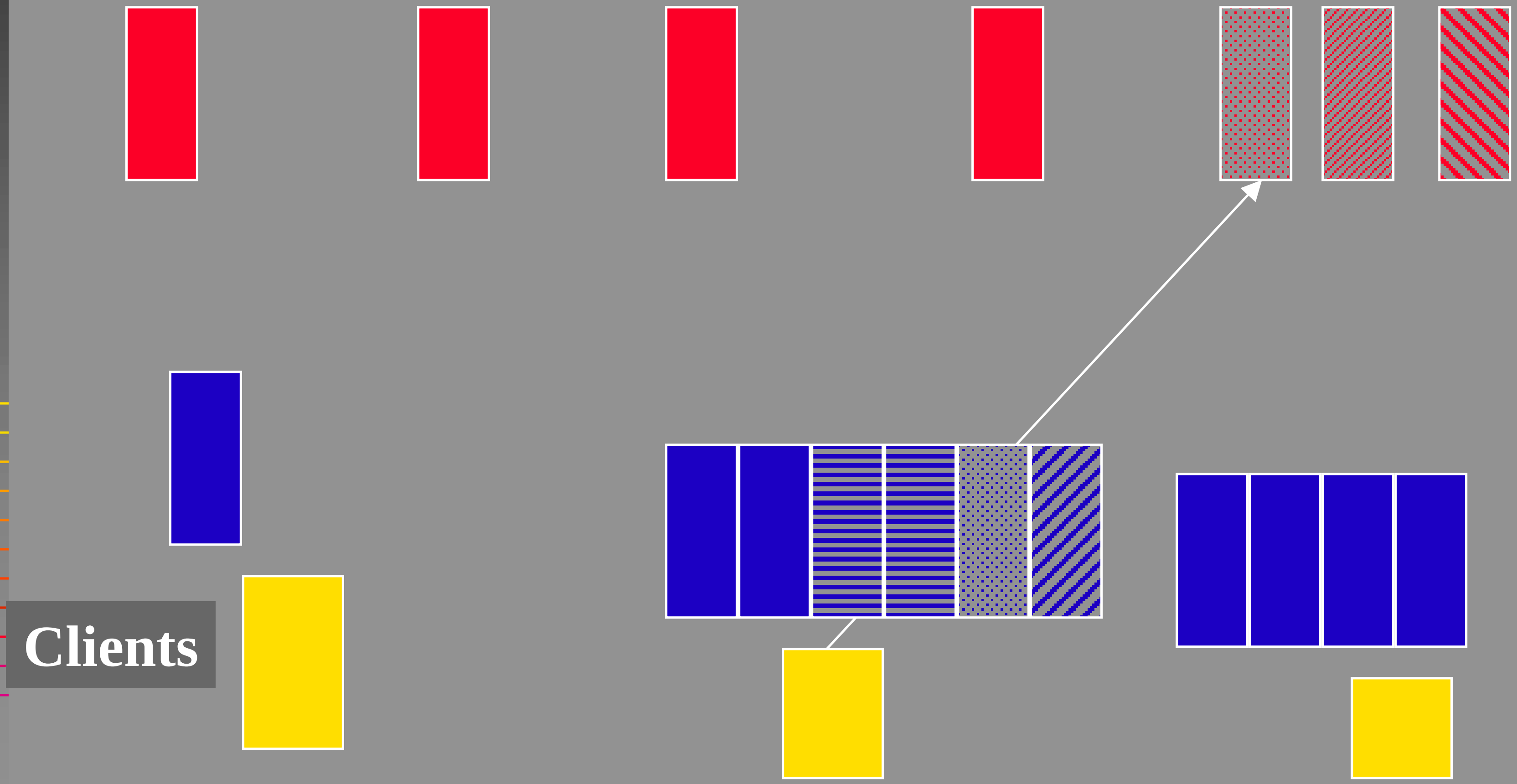
An SDDS



An SDDS



An SDDS



Performance measures

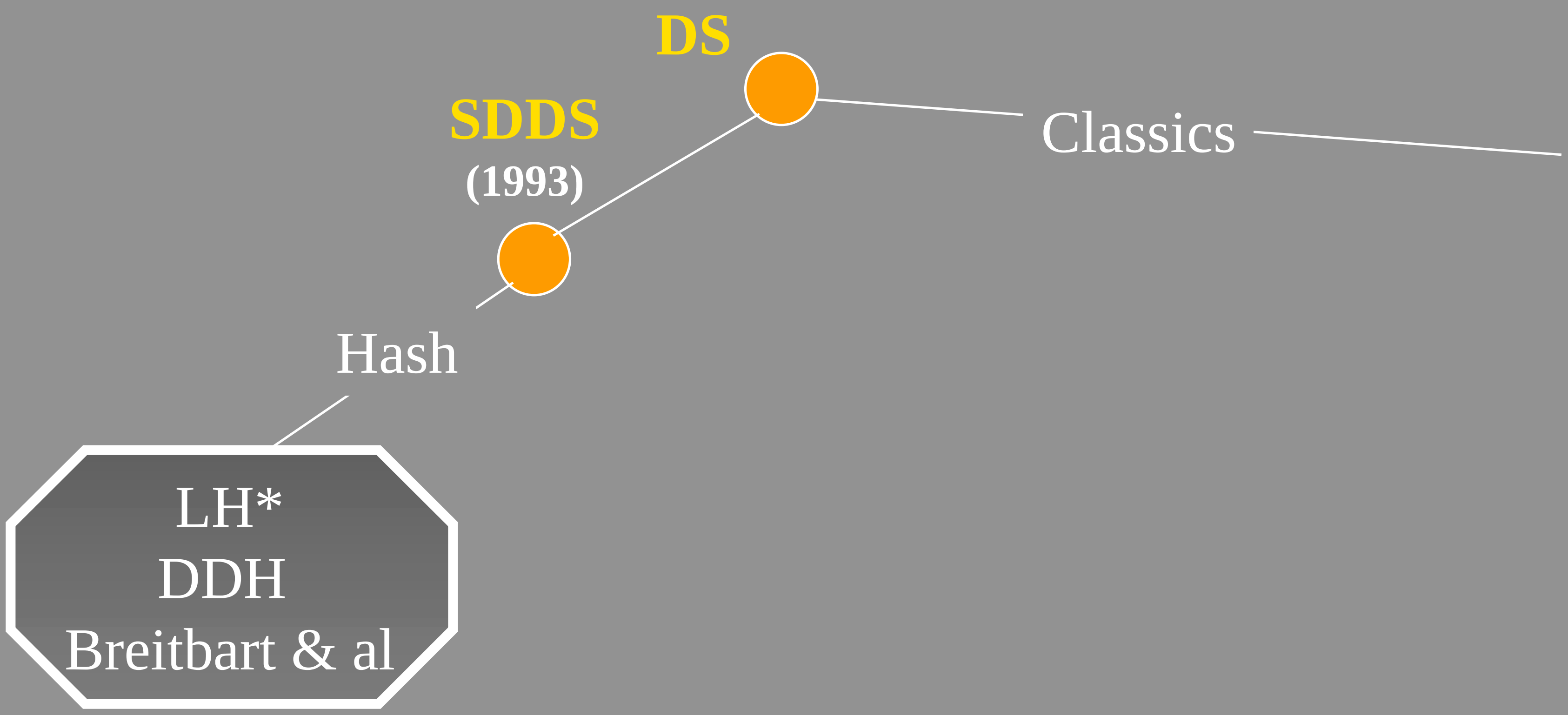
- Storage cost
  - load factor
    - same definitions as for the traditional DSs
- Access cost
- ☎ messaging
  - number of messages (rounds)
    - network independent
  - access time

Access performance measures

- Query cost
  - key search
    - forwarding cost
  - insert
    - split cost
  - delete
    - merge cost
  - Parallel search, range search, partial match search, bulk insert...
- Average & worst-case costs
- Client image convergence cost
- New or less active client costs

Known SDDSs

Known SDDSs



DS

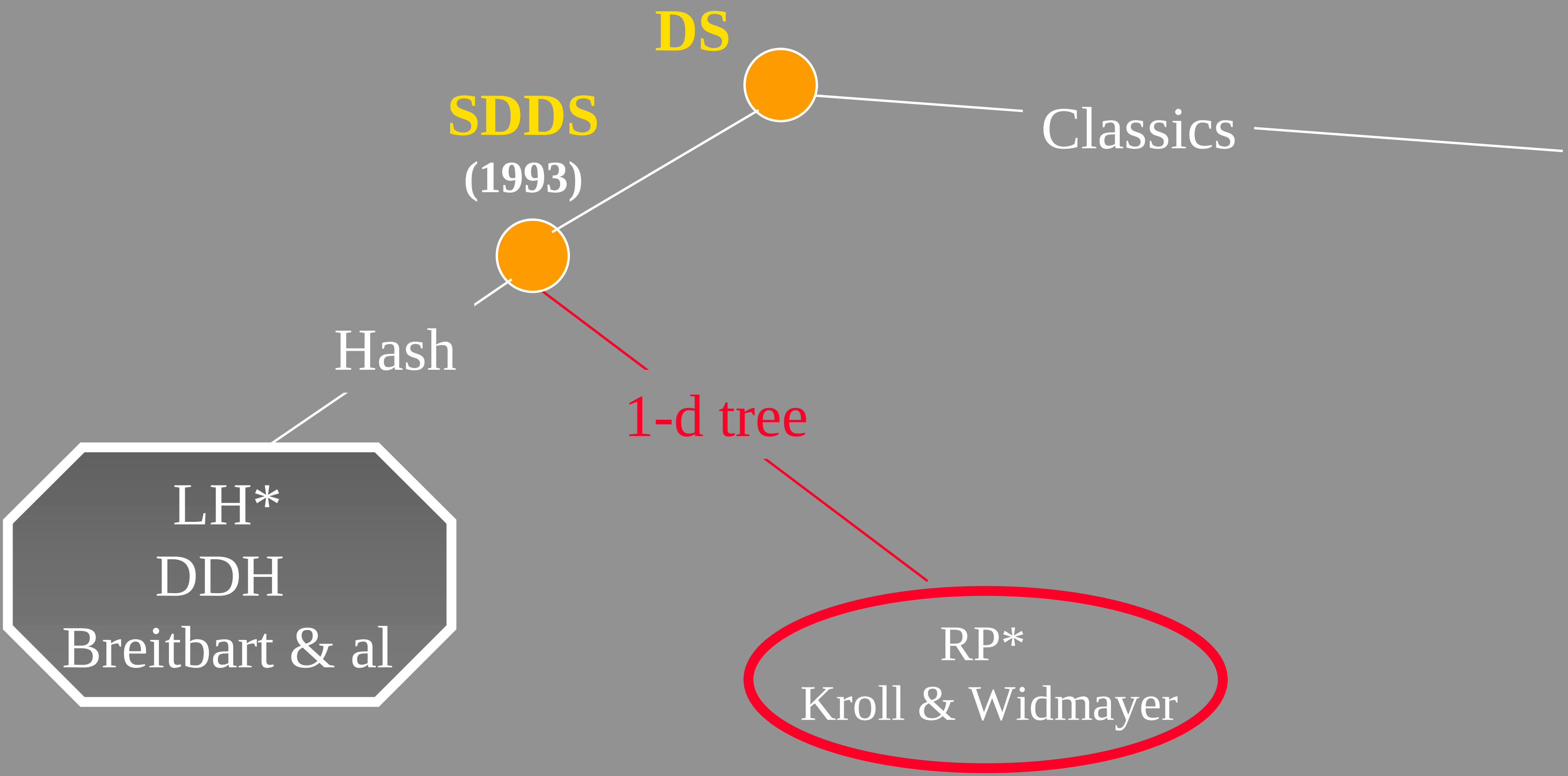
SDDS
(1993)

Classics

Hash

LH*
DDH
Breitbart & al

Known SDDSs



DS

SDDS

(1993)

Classics

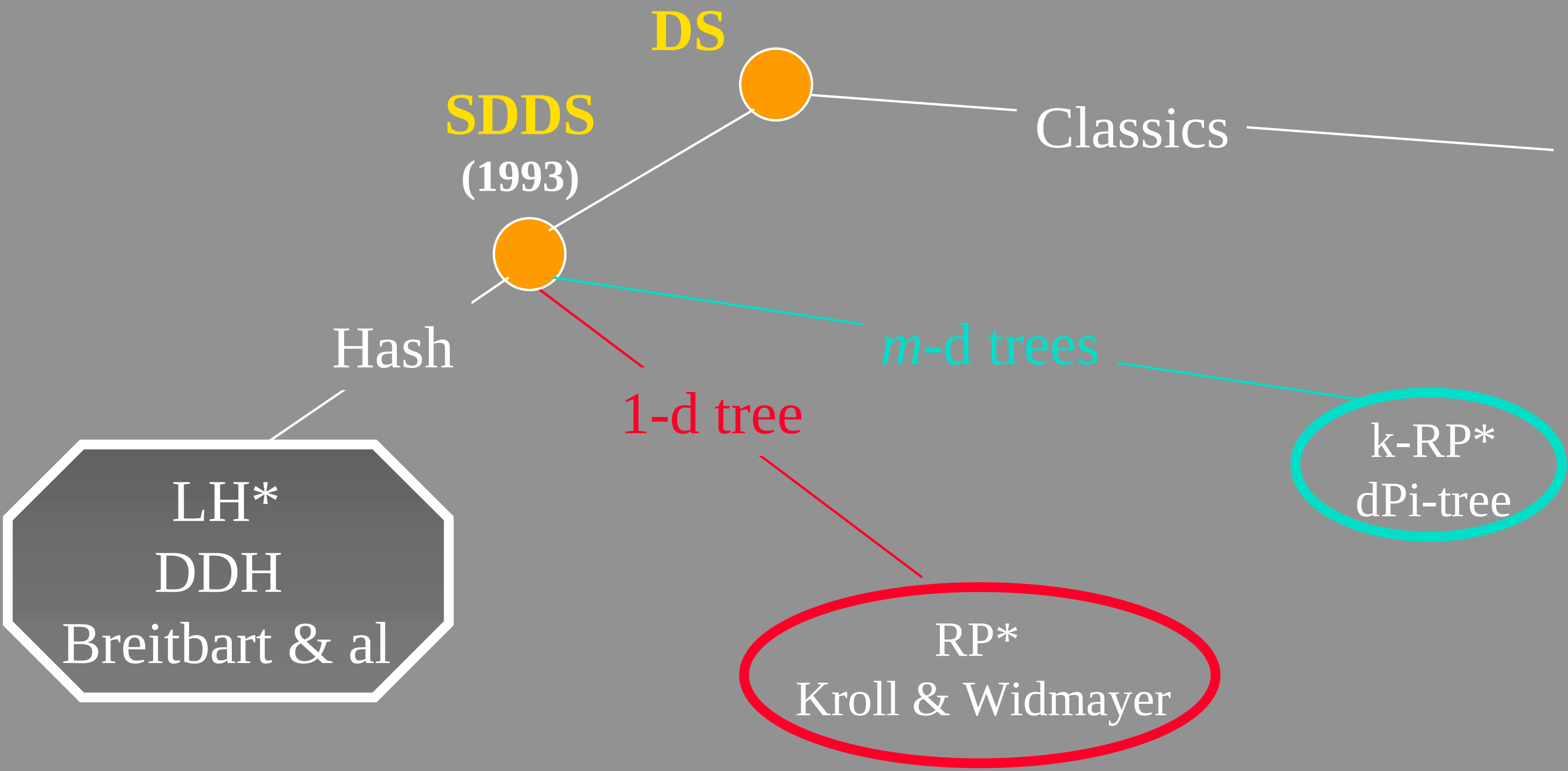
Hash

1-d tree

LH*
DDH
Breitbart & al

RP*
Kroll & Widmayer

Known SDDSs



DS

SDDS

(1993)

Classics

Hash

1-d tree

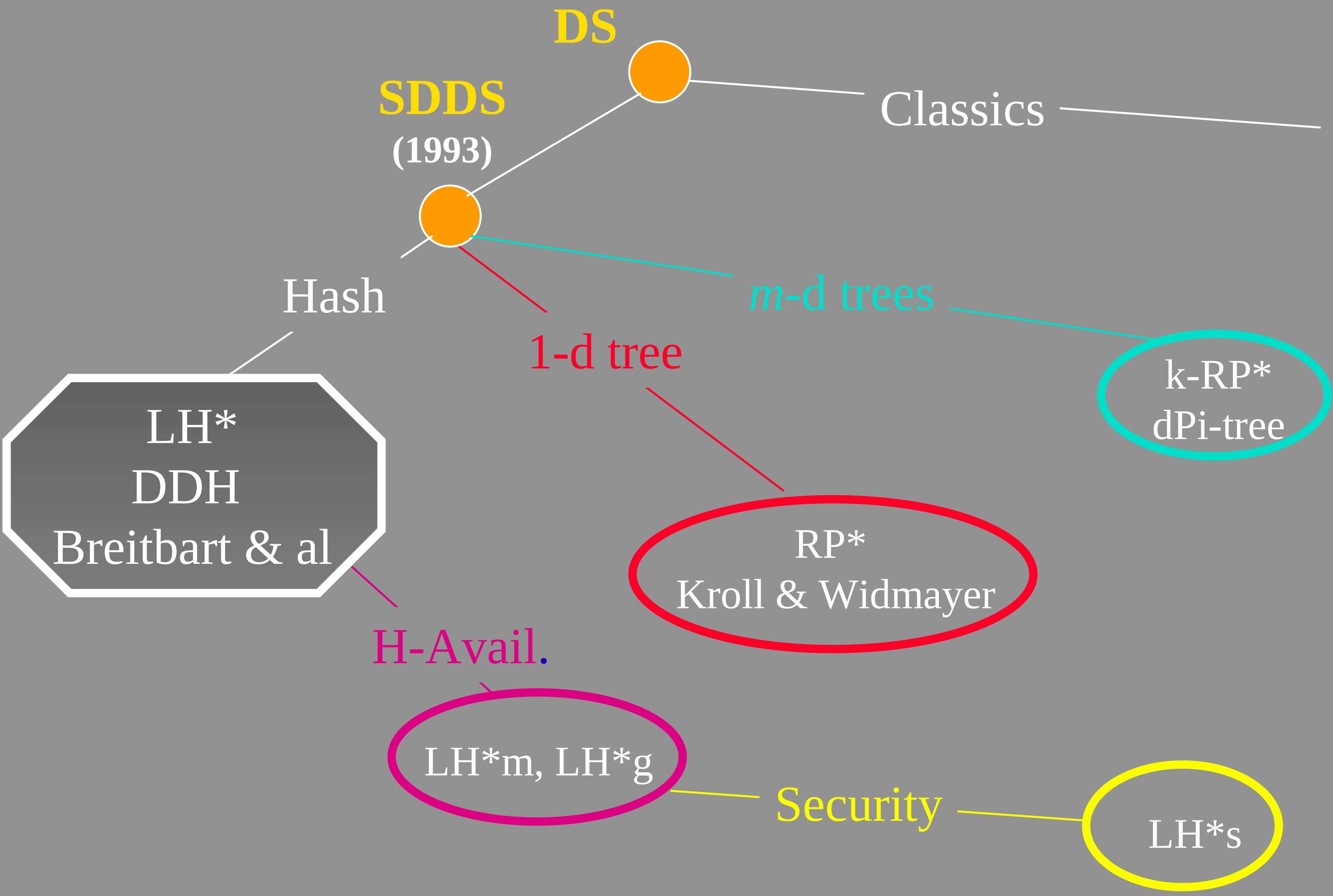
m-d trees

LH*
DDH
Breitbart & al

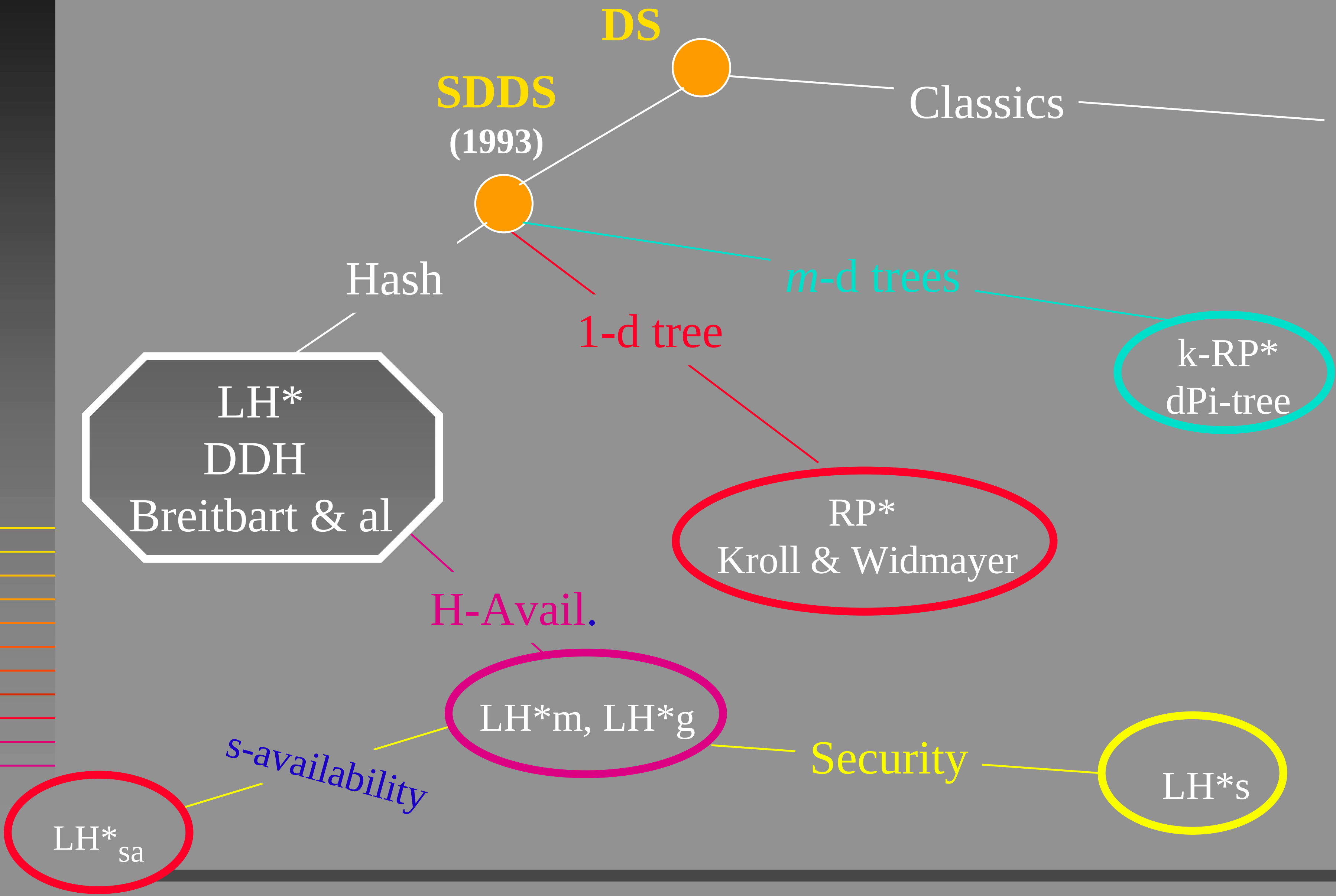
RP*
Kroll & Widmayer

k-RP*
dPi-tree

Known SDDSs


DS
SDDS
(1993)
Classics
Hash
1-d tree
m-d trees
k-RP*
dPi-tree
LH*
DDH
Breitbart & al
RP*
Kroll & Widmayer
H-Avail.
LH*m, LH*g
Security
LH*s

Known SDDSs



DS

SDDS (1993)

Classics

Hash

1-d tree

m-d trees

k-RP*
dPi-tree

LH*
DDH
Breitbart & al

RP*
Kroll & Widmayer

H-Avail.

LH*m, LH*g

s-availability

Security

LH*s

LH*$_{sa}$

LH* (A classic)

- Allows for the primary key (OID) based hash files
  - generalizes the LH addressing schema
- Load factor 70 - 90 %
- At most 2 forwarding messages
  - regardless of the size of the file
- In practice, 1 m/insert and 2 m/search on the average
- 4 messages in the worst case
- Search time of 1 ms (10 Mb/s net), of 150 ms (100 Mb/s net) and of 30 us (Gb/s net)

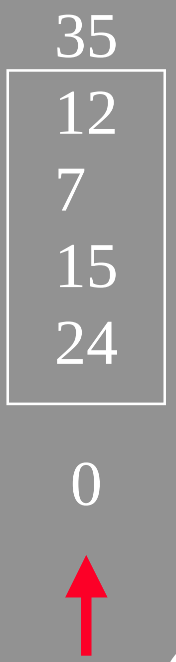
Overview of LH

- Extensible hash algorithm
  - used, e.g.,
    - Netscape browser (100M copies)
    - LH-Server by AR (700K copies sold)
  - tought in most DB and DS classes
  - address space expands
    - to avoid overflows & access performance deterioration
- the file has buckets with capacity *b* >> 1
- Hash by division $h_i : c \rightarrow c \bmod 2^i N$ provides the address *h (c)* of key *c*.
- Buckets split through the replacement of h_i with h_{i+1} ; $i = 0,1,..$
- On the average, *b/2* keys move towards new bucket

Overview of LH

- Basically, a split occurs when some bucket *m* overflows
- One splits bucket *n*, pointed by pointer *n*.
  - usually $m \neq n$
- *n* évolue : 0, 0,1, 0,1,..,2, 0,1..,3, 0,..,7, 0,..,$2^i N$, 0..
- One consequence => no index
  - characteristic of other EH schemes

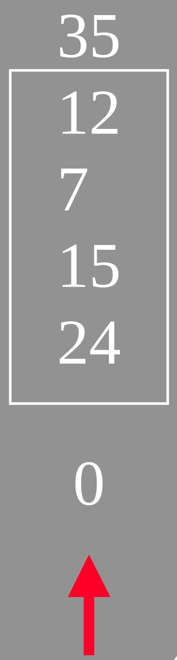
LH File Evolution



$N = 1$
$b = 4$
$i = 0$
$h_0 : c \text{ -> } 2^0$

$\boldsymbol{h_0 \;\; ; \;\; n = 0}$

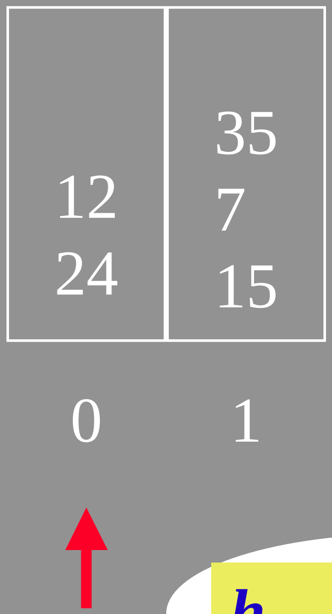
LH File Evolution



$N = 1$
$b = 4$
$i = 0$
$h_1 : c \to 2^1$

h_1 ; $n = 0$

LH File Evolution



$N = 1$

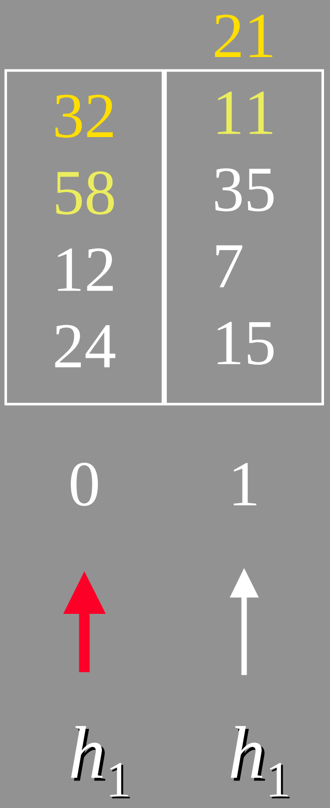
$b = 4$

$i = 1$

$h_1 : c \to 2^1$

$h_1 \; ; \; n = 0$

LH File Evolution



21

32	11
58	35
12	7
24	15
0	1

h_1 h_1

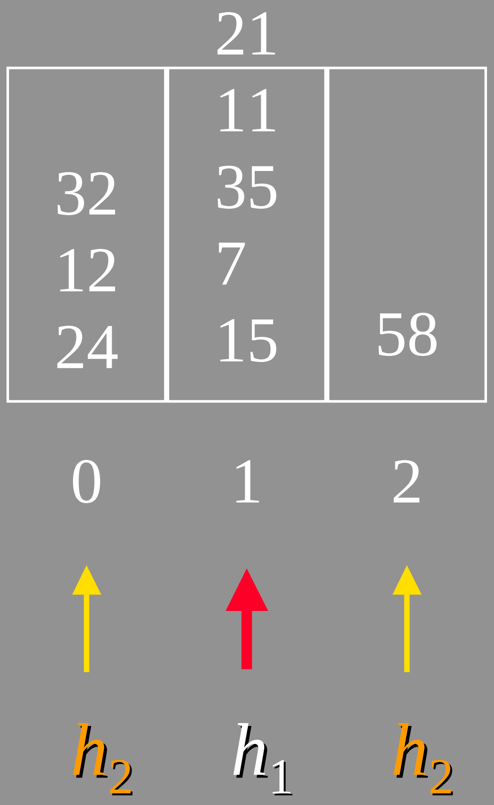
$N = 1$

$b = 4$

$i = 1$

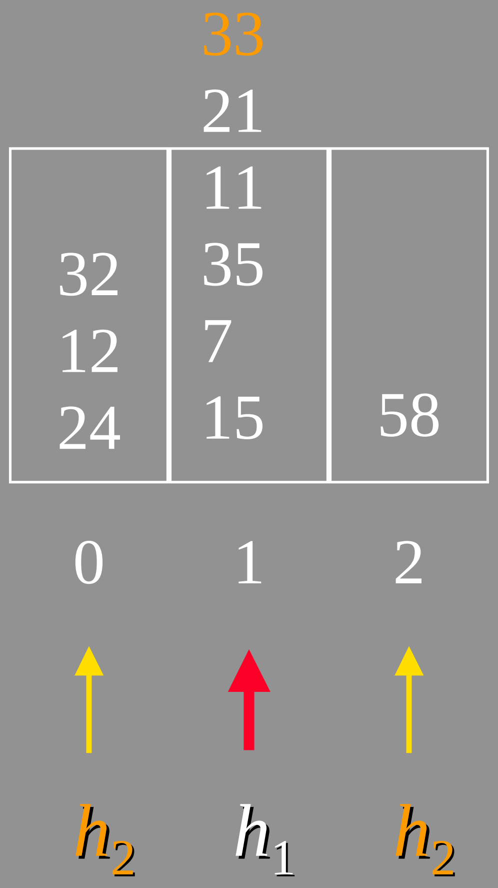
$h_1 : c \text{ -> } 2^1$

LH File Evolution



$N = 1$
$b = 4$
$i = 1$
$h_2 : c \to 2^2$

LH File Evolution



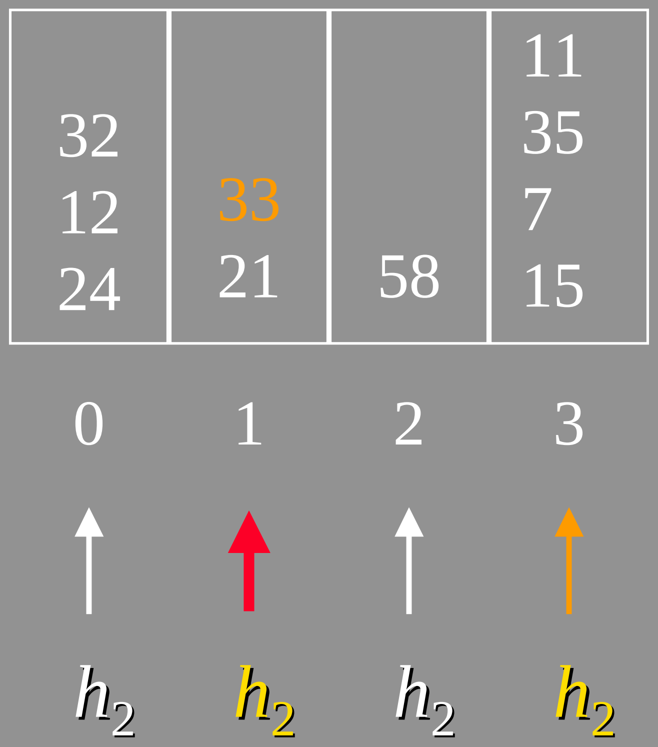
33
21

0	1	2
32 12 24	11 35 7 15	58

h_2 h_1 h_2

$N = 1$
$b = 4$
$i = 1$
$h_2 : c \text{ -> } 2^2$

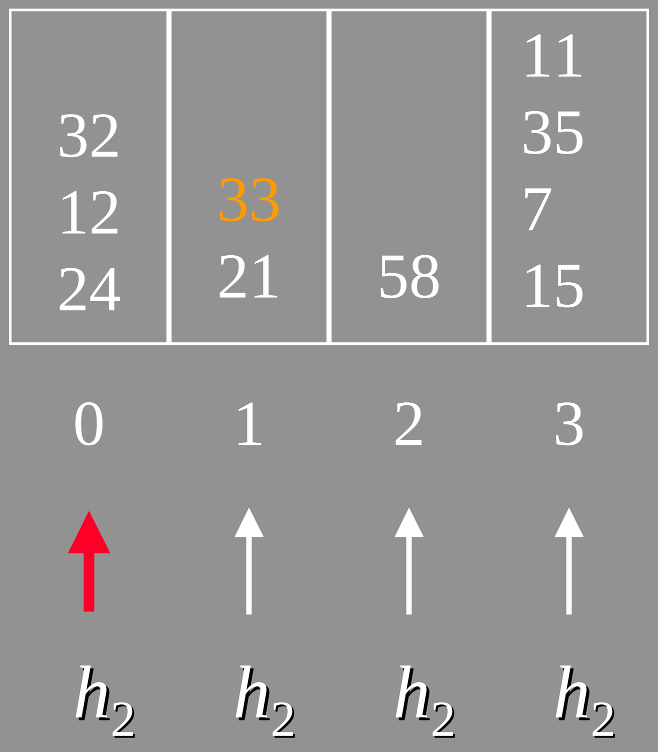
LH File Evolution



0	1	2	3
			11
32			35
12	33		7
24	21	58	15
h_2	h_2	h_2	h_2

$N = 1$
$b = 4$
$i = 1$
$h_2 : c \rightarrow 2^2$

LH File Evolution



0	1	2	3
32 12 24	33 21	58	11 35 7 15
h_2	h_2	h_2	h_2

$N = 1$

$b = 4$

$i = 2$

$h_2 : c \text{ -> } 2^2$

LH File Evolution

- Etc
  - One starts h_3 then h_4 ...
- **The file can expand as much as needed**
  - without too many overflows ever

Addressing Algorithm

```
a <- h (i, c)
if $n$ = 0 alors exit
else
    if  a < n then a <- h (i+1, c) ;
end
```

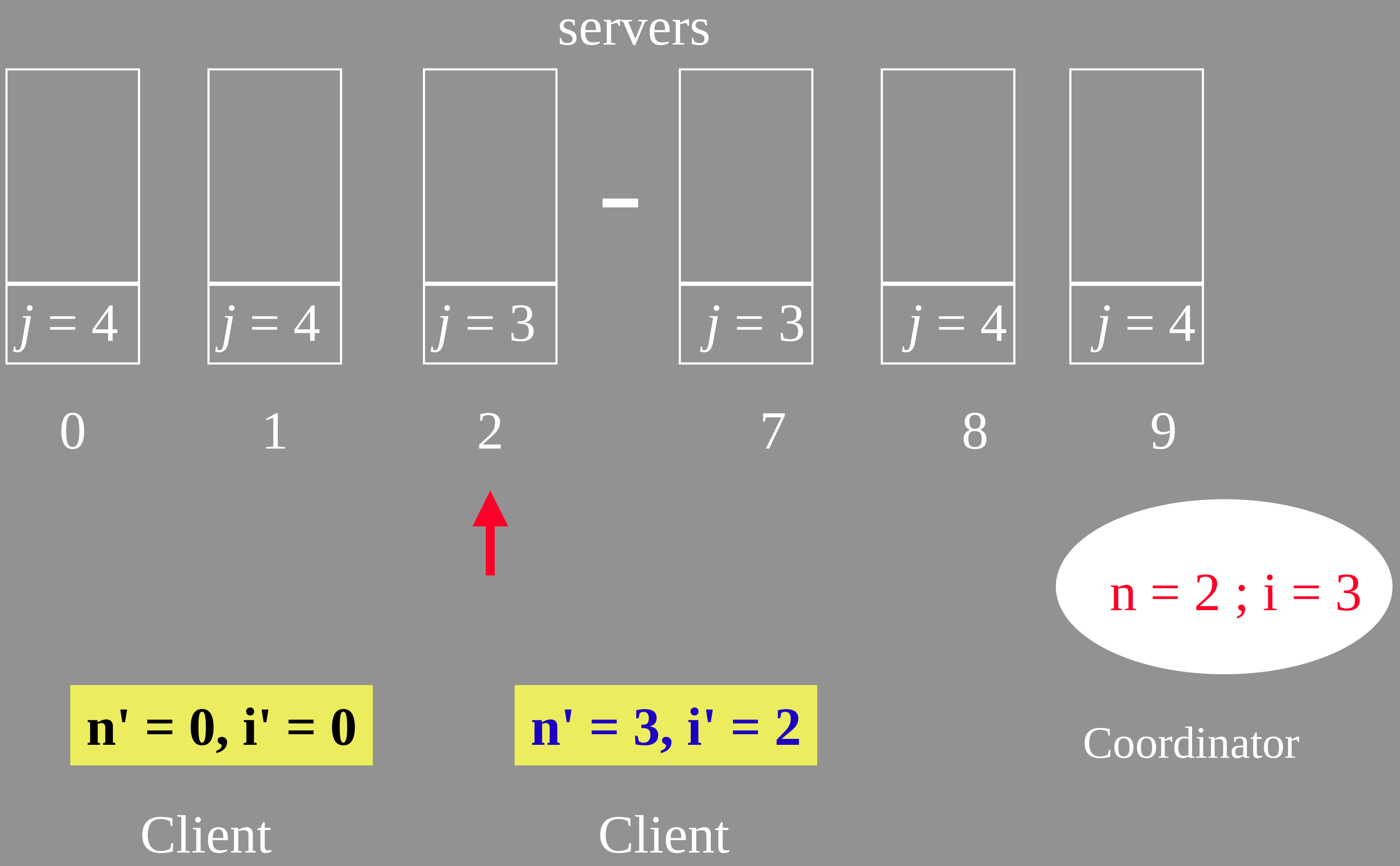
LH*

- Property of LH :
  - Given $j = i$ or $j = i + 1$, key c is in bucket m iff

    $h_j(c) = m \; ; \; j = i$ **ou** $j = i + 1$

    - » Verify yourself
- Ideas for LH* :
  - LH addresing rule = global rule for LH* file
  - every bucket at a server
  - bucket level j in the header
  - Check the LH property when the key comes form a client

LH* : file structure



servers

j = 4

j = 4

j = 3

j = 3

j = 4

j = 4

0

1

2

7

8

9

n = 2 ; i = 3

Coordinator

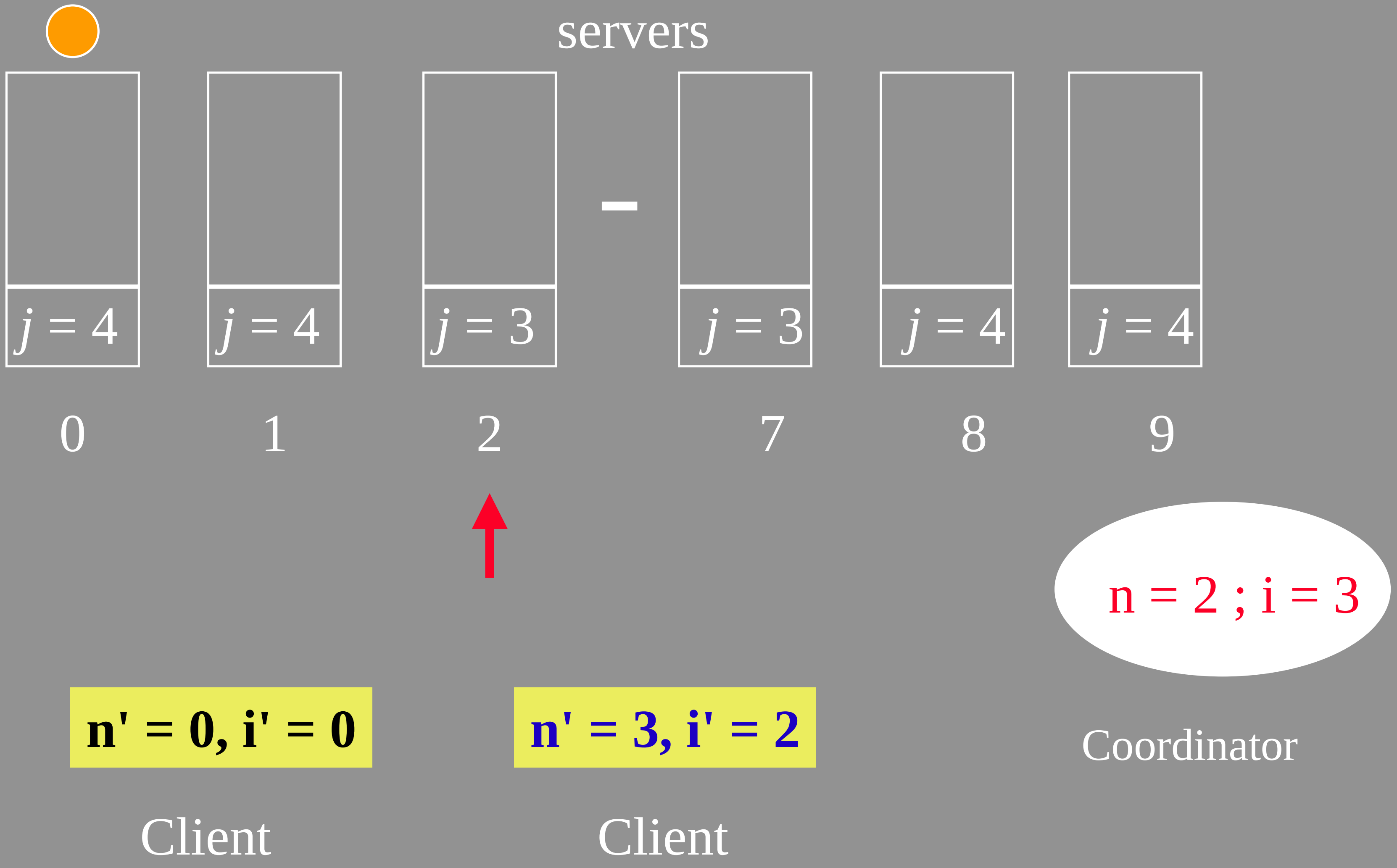
n' = 0, i' = 0

Client

n' = 3, i' = 2

Client

LH* : file structure



servers

j = 4

j = 4

j = 3

-

j = 3

j = 4

j = 4

0

1

2

7

8

9

n = 2 ; i = 3

n' = 0, i' = 0

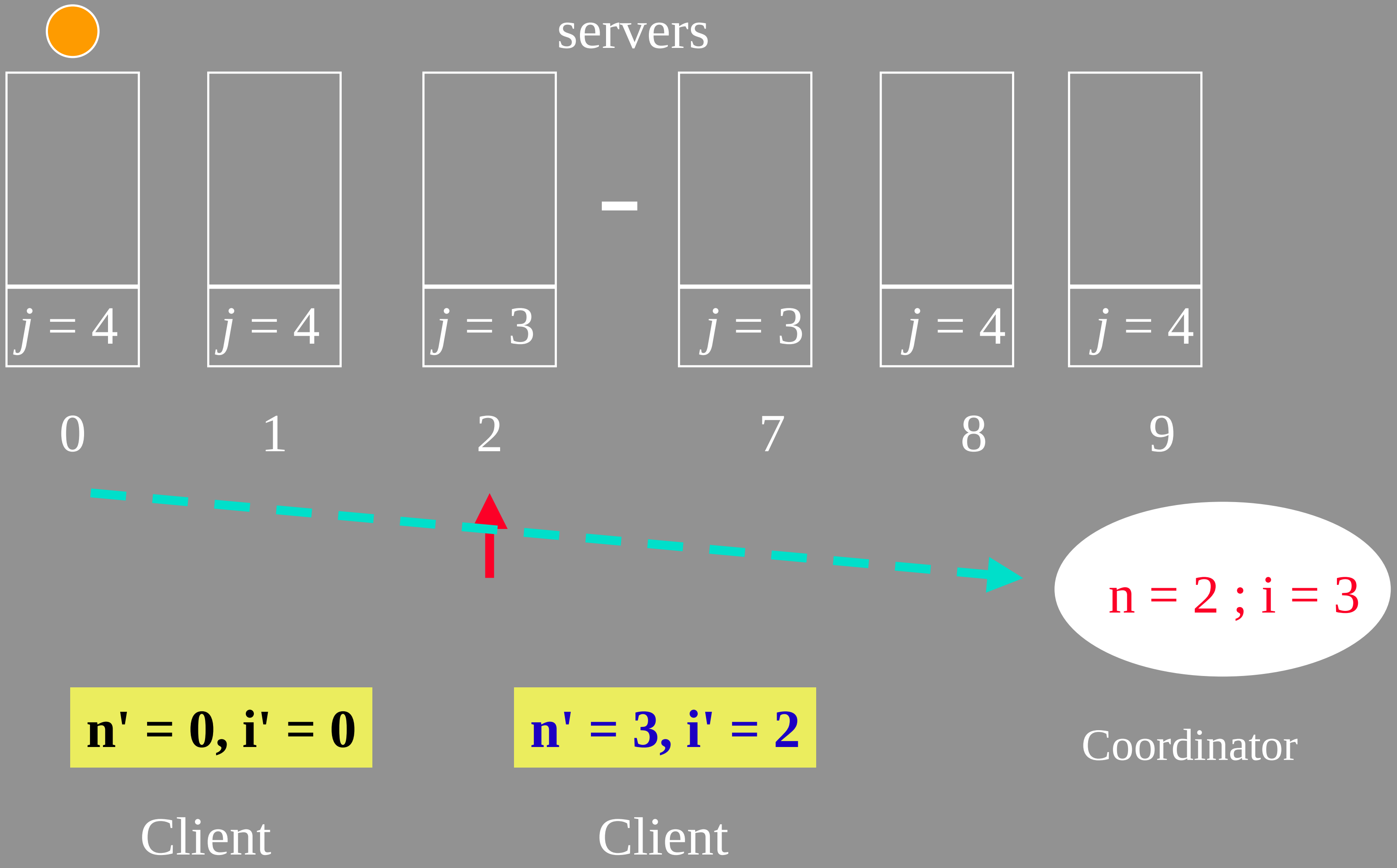
n' = 3, i' = 2

Coordinator

Client

Client

LH* : split



servers

$j = 4$ $j = 4$ $j = 3$ – $j = 3$ $j = 4$ $j = 4$

0 1 2 7 8 9

n = 2 ; i = 3

Coordinator

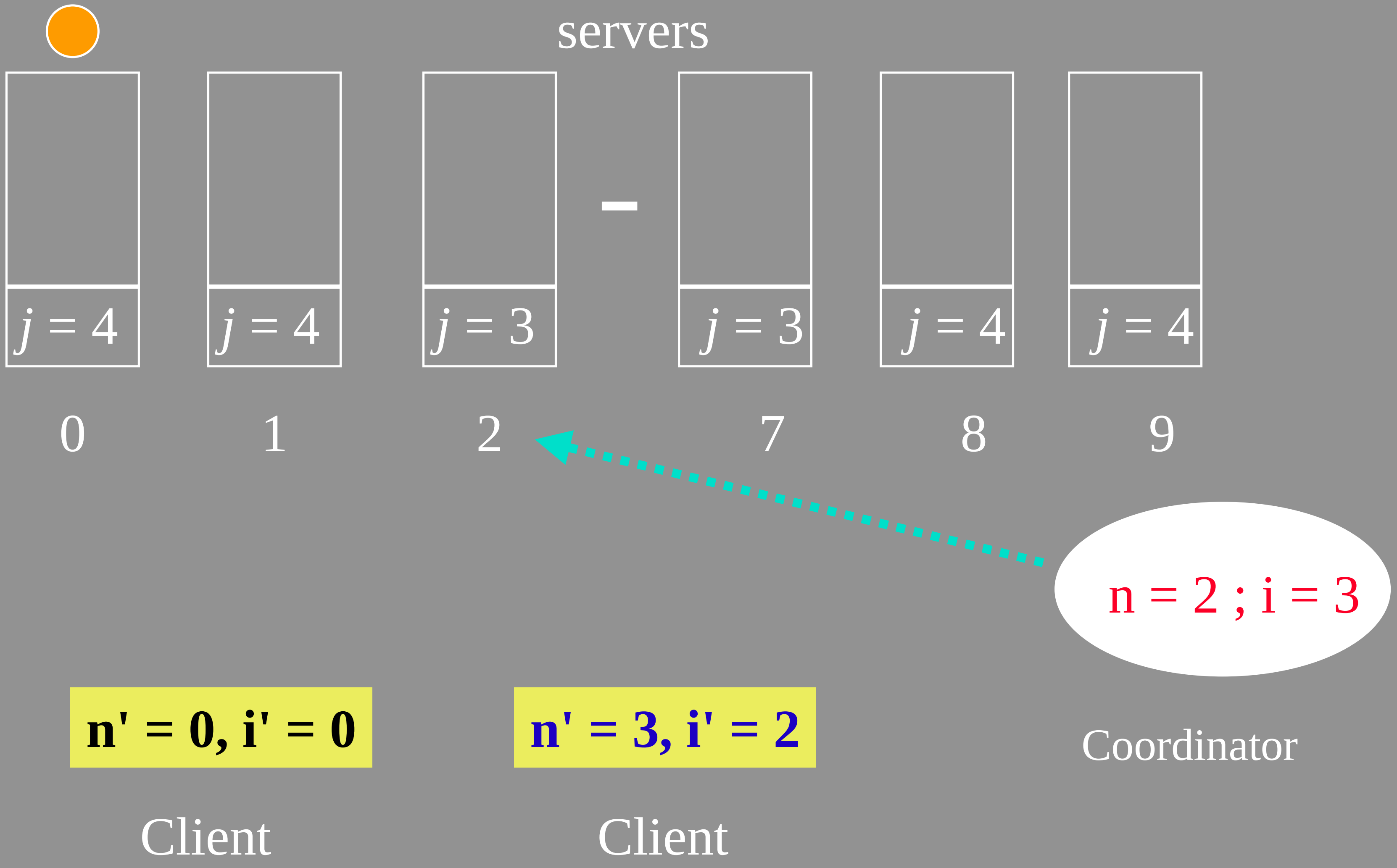
n' = 0, i' = 0

Client

n' = 3, i' = 2

Client

LH* : split



servers

$j = 4$ $j = 4$ $j = 3$ - $j = 3$ $j = 4$ $j = 4$

0 1 2 7 8 9

n = 2 ; i = 3

Coordinator

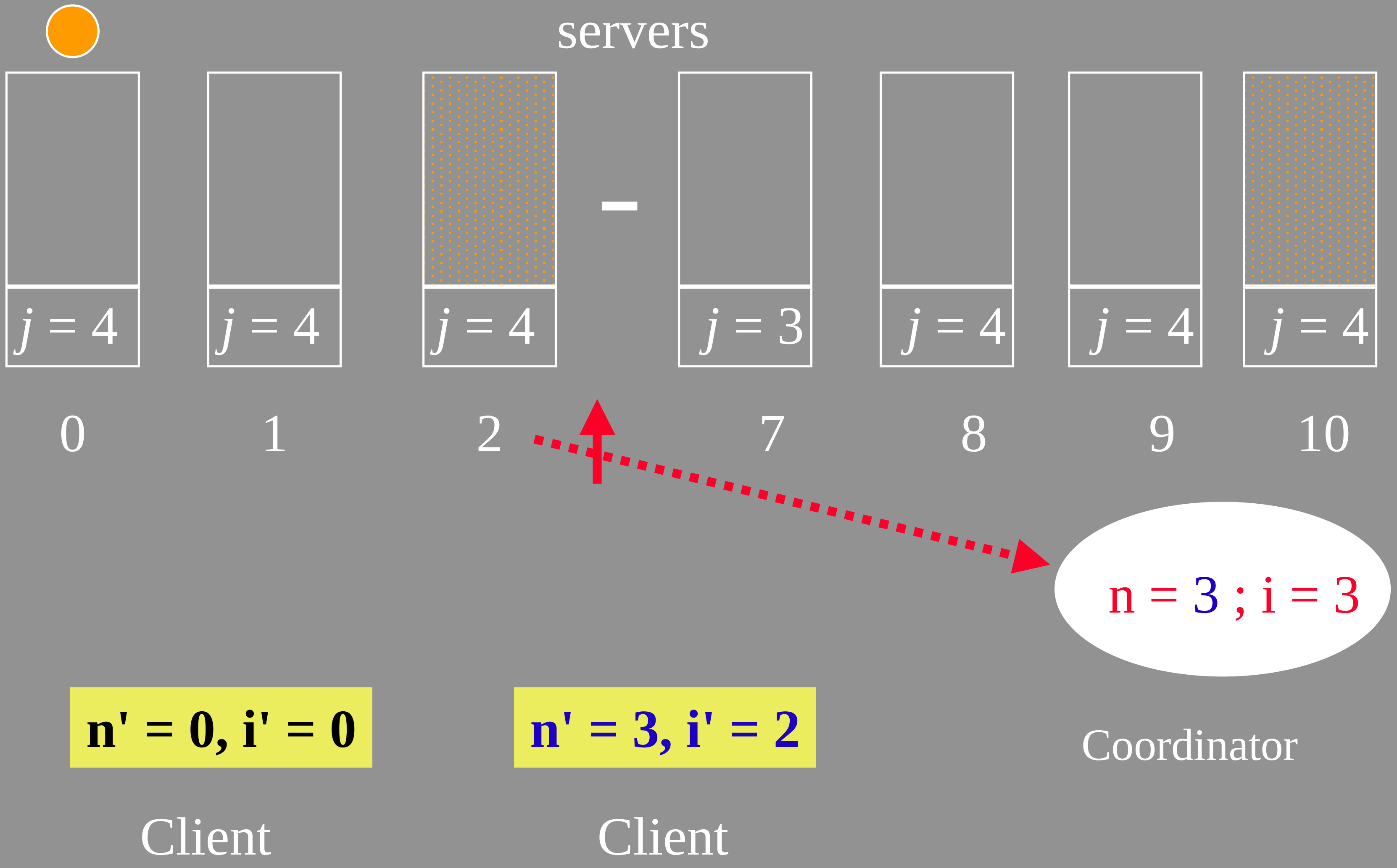
n' = 0, i' = 0

Client

n' = 3, i' = 2

Client

LH* : split



servers
$j = 4$
$j = 4$
$j = 4$
$j = 3$
$j = 4$
$j = 4$
$j = 4$
0
1
2
7
8
9
10
n = 3 ; i = 3
n' = 0, i' = 0
n' = 3, i' = 2
Coordinator
Client
Client

LH* Addressing Schema

- Client
  - computes the LH address m of c using its image,
  - send c to bucket m
- Server
  - Server a getting key c, $a = m$ in particular, computes :

```
$a' := h_j (c)$ ;
if $a' = a$ then accept $c$ ;
else $a'' := h_{j-1} (c)$ ;
    if $a'' > a$ and $a'' < a'$ then $a' := a''$ ;
send $c$ to bucket $a'$ ;
```

LH* Addressing Schema

- Client
  - computes the LH address m of c using its image,
  - send c to bucket m
- Server
  - Server a getting key c, $a = m$ in particular, computes :

```
$a' := h_j(c)$ ;
if $a' = a$ then accept $c$ ;
else $a'' := h_{j-1}(c)$ ;
    if $a'' > a$ and $a'' < a'$ then $a' := a''$ ;
send $c$ to bucket $a'$ ;
```

Simple ?

- **See [LNS93] for the (long) proof**

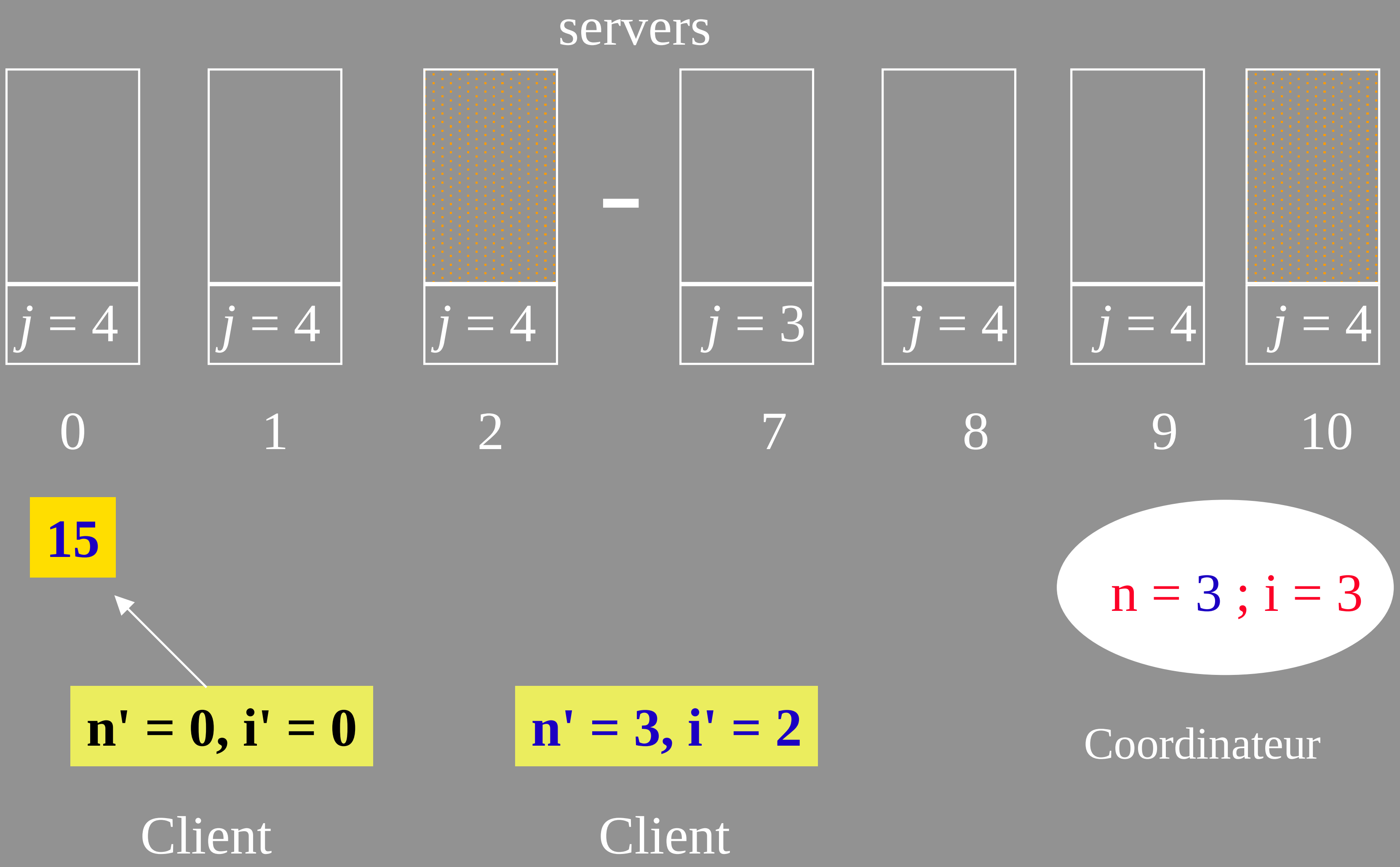
Client Image Adjustement

- The IAM consists of address *a* where the client sent *c* and of *j* (*a*)
  - *i'* is presumed *i* in client's image.
  - *n'* is preumed value of pointer *n* in client's image.
  - initially, $i' = n' = 0$.

```
if  $j > i'$  then  $i' := j - 1$, $n' := a + 1$ ;
if $n' \geq 2^{i'}$  then  $n' = 0$,  $i' := i' + 1$ ;
```

- The algo. garantees that client image is within the file [LNS93]
  - if there is no file contractions (merge)

LH* : addressing



servers

j = 4
j = 4
j = 4
j = 3
j = 4
j = 4
j = 4

0
1
2
7
8
9
10

15

n' = 0, i' = 0

Client

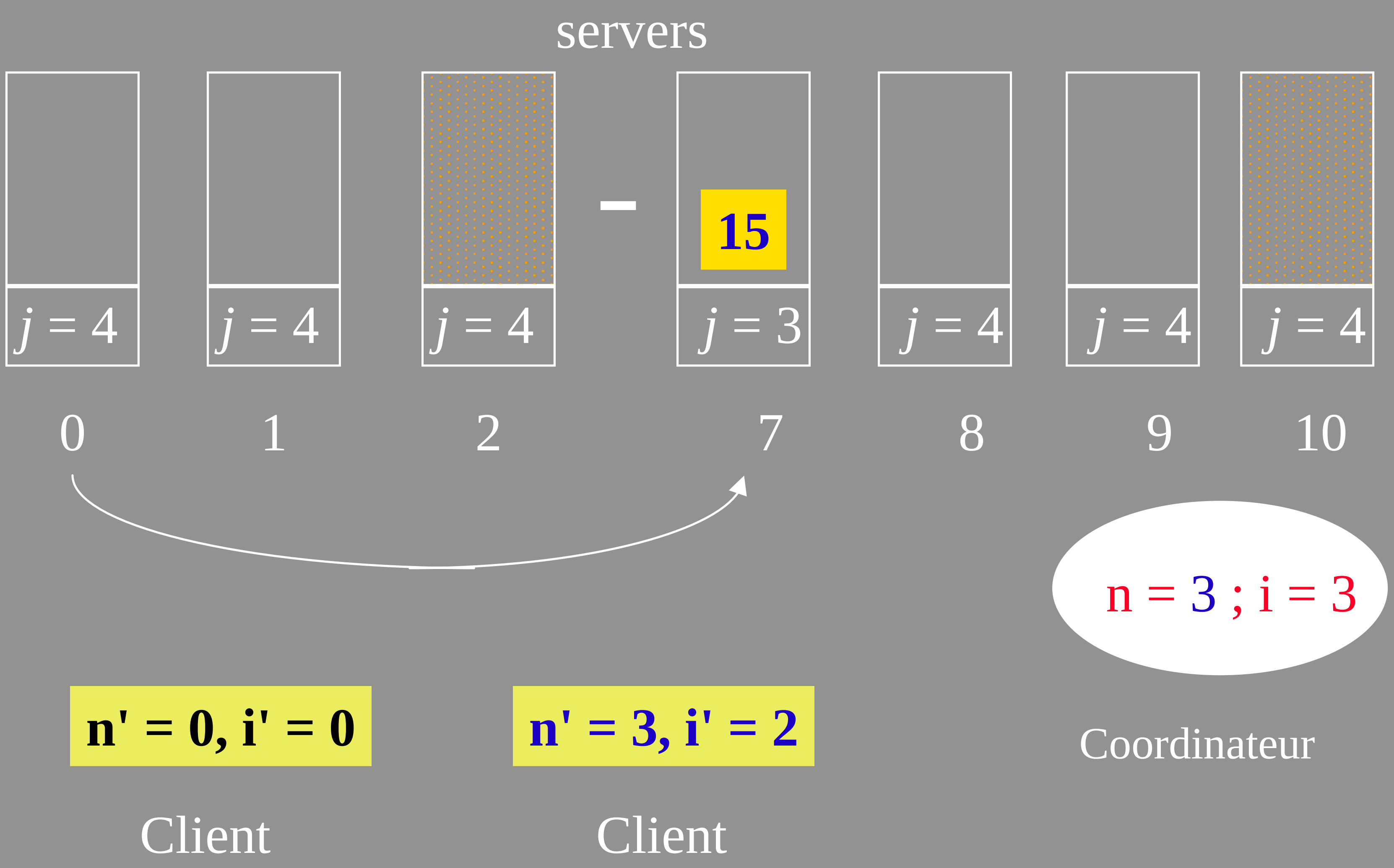
n' = 3, i' = 2

Client

n = 3 ; i = 3

Coordinateur

LH* : addressing



servers

j = 4 | j = 4 | j = 4 | - | 15 j = 3 | j = 4 | j = 4 | j = 4

0 1 2 7 8 9 10

n = 3 ; i = 3

Coordinateur

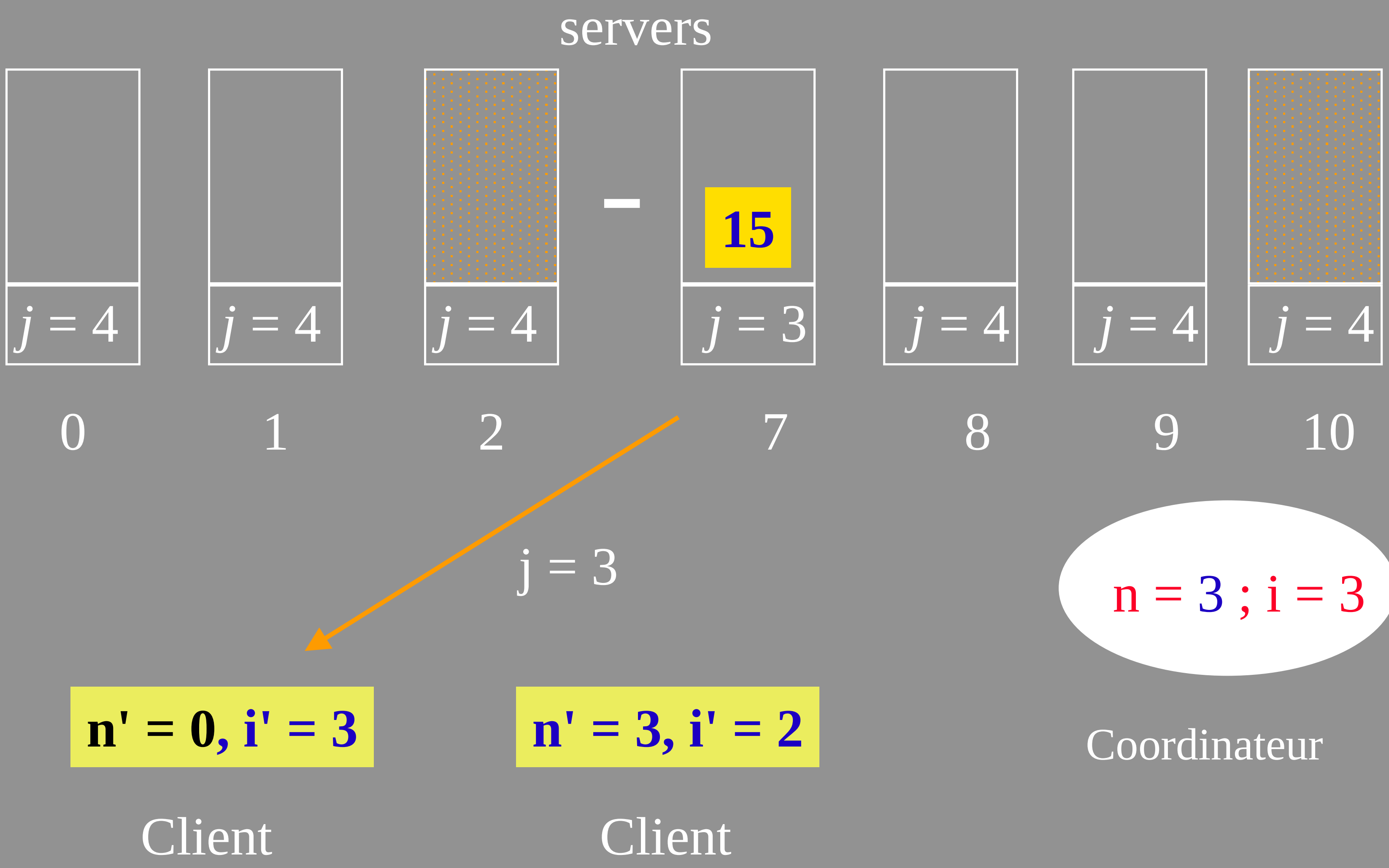
n' = 0, i' = 0

Client

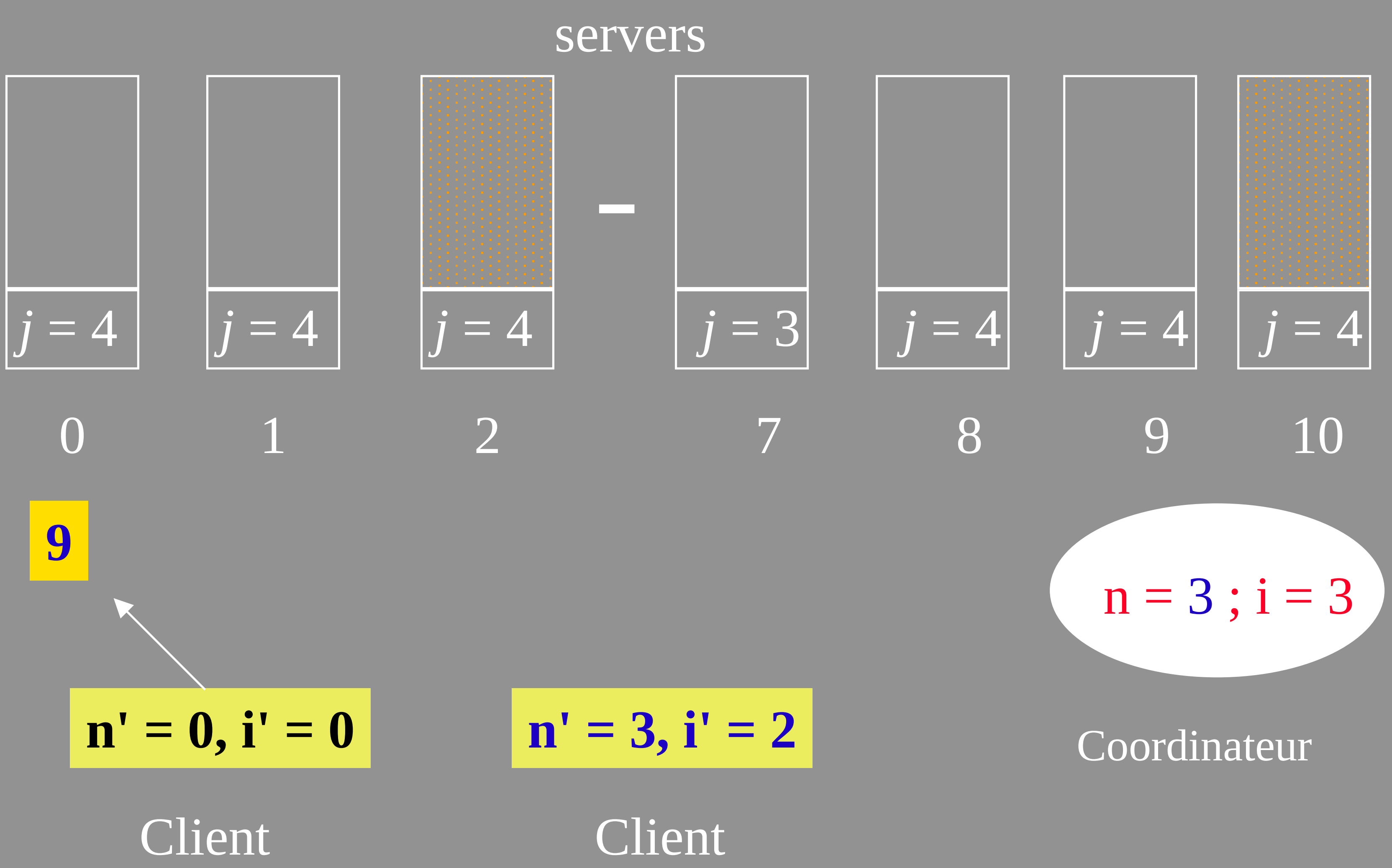
n' = 3, i' = 2

Client

LH* : addressing



LH* : addressing



servers

0	1	2		7	8	9	10
j = 4	*j* = 4	*j* = 4	–	*j* = 3	*j* = 4	*j* = 4	*j* = 4

9

n' = 0, i' = 0

Client

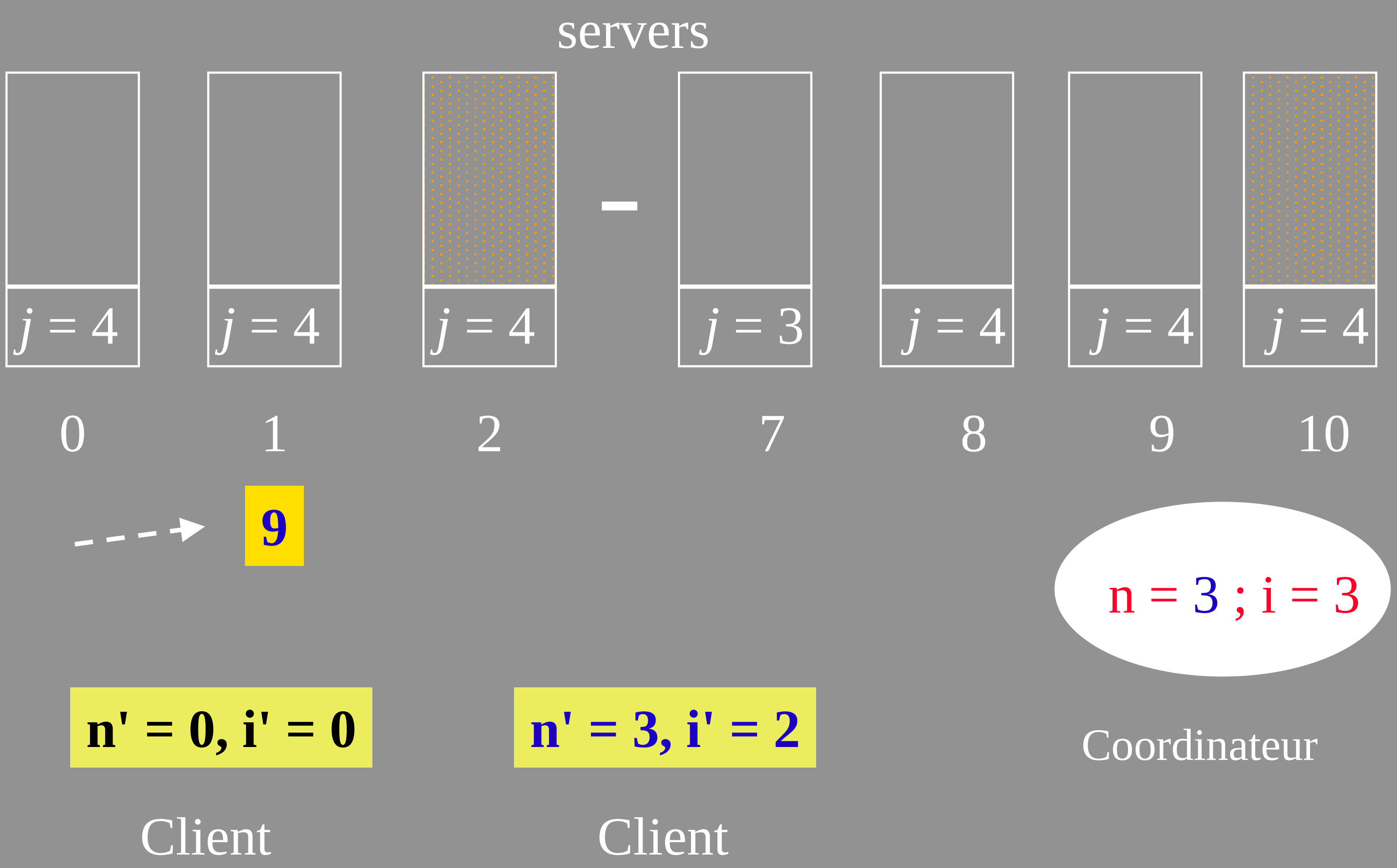
n' = 3, i' = 2

Client

n = 3 ; i = 3

Coordinateur

LH* : addressing



servers

j = 4 *j* = 4 *j* = 4 – *j* = 3 *j* = 4 *j* = 4 *j* = 4

0 1 2 7 8 9 10

9

n = 3 ; i = 3

n' = 0, i' = 0

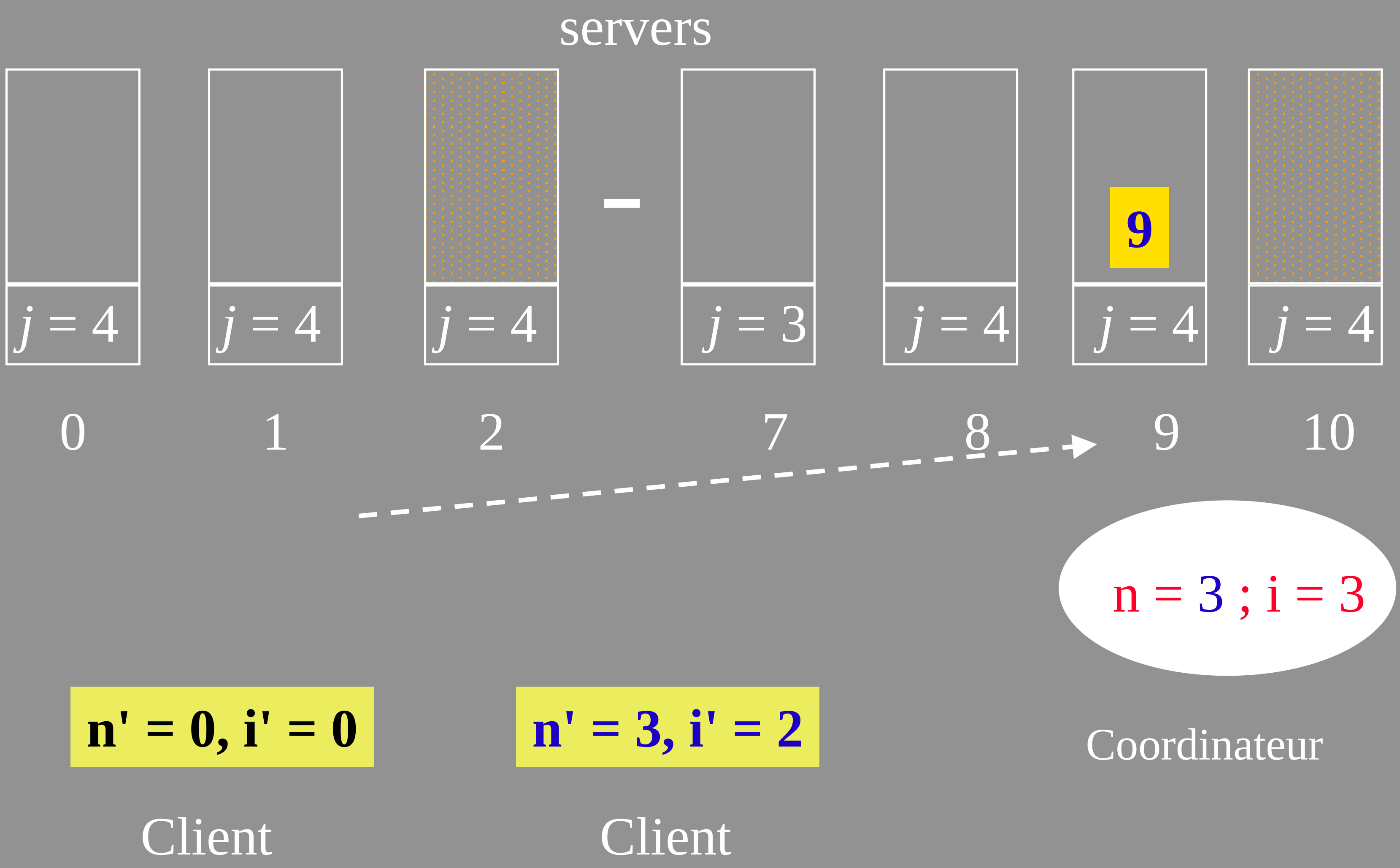
n' = 3, i' = 2

Coordinateur

Client

Client

LH* : addressing



servers

9

$j = 4$ $j = 4$ $j = 4$ - $j = 3$ $j = 4$ $j = 4$ $j = 4$

0 1 2 7 8 9 10

n = 3 ; i = 3

n' = 0, i' = 0

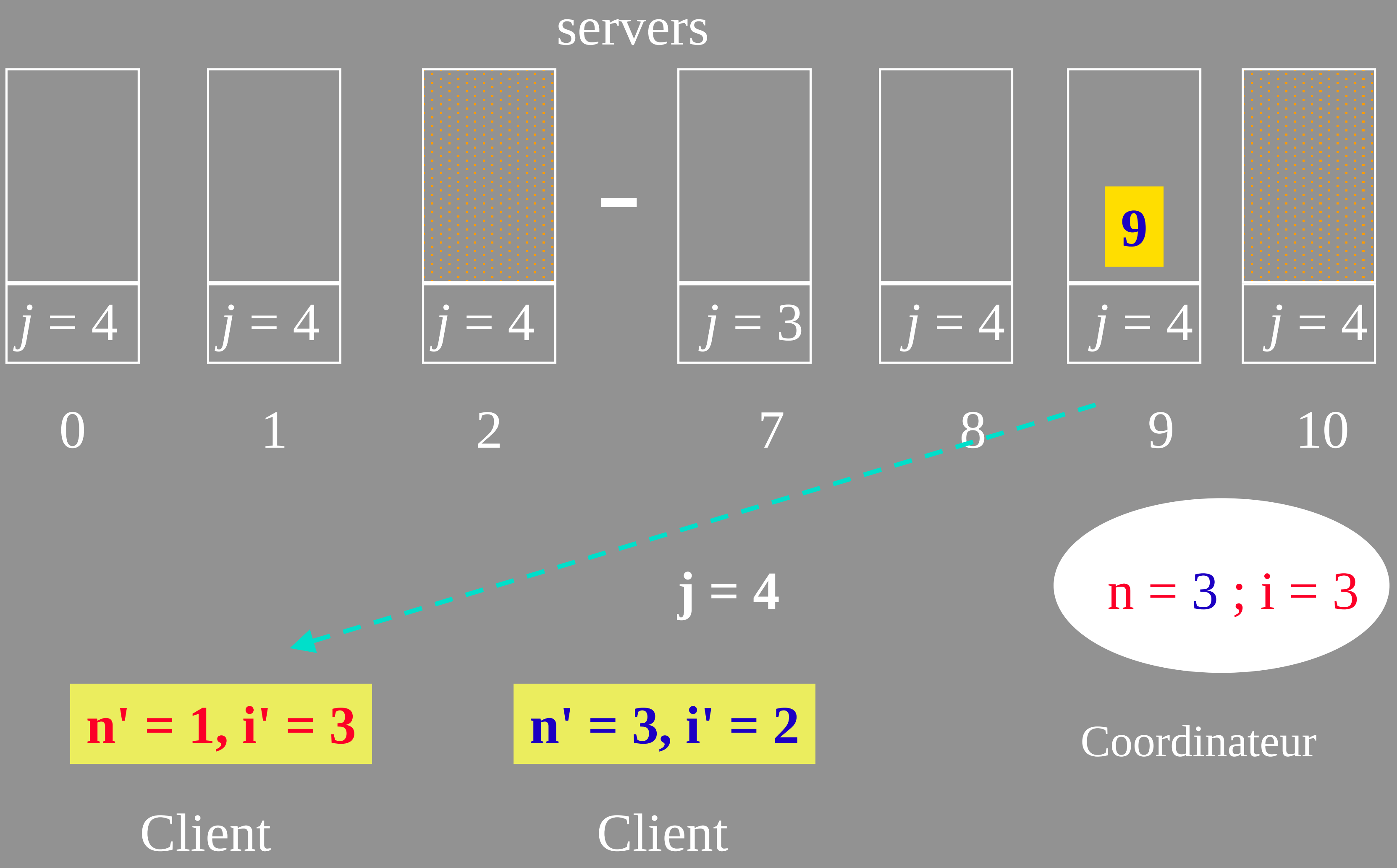
n' = 3, i' = 2

Coordinateur

Client

Client

LH* : addressing



servers

j = 4 | *j* = 4 | *j* = 4 | – | *j* = 3 | *j* = 4 | *j* = 4 | *j* = 4

9

0 1 2 7 8 9 10

j = 4

n = 3 ; i = 3

n' = 1, i' = 3

n' = 3, i' = 2

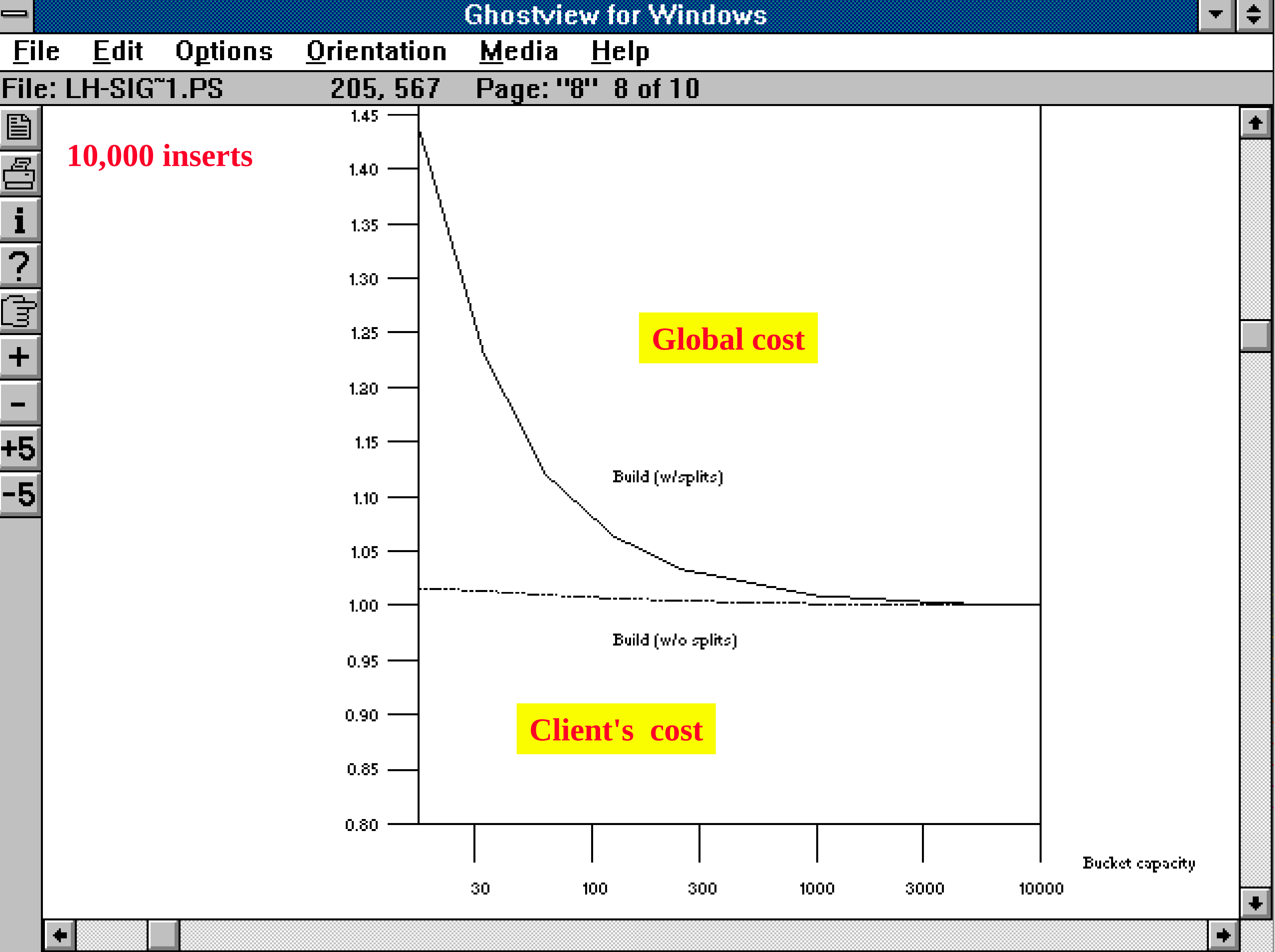
Coordinateur

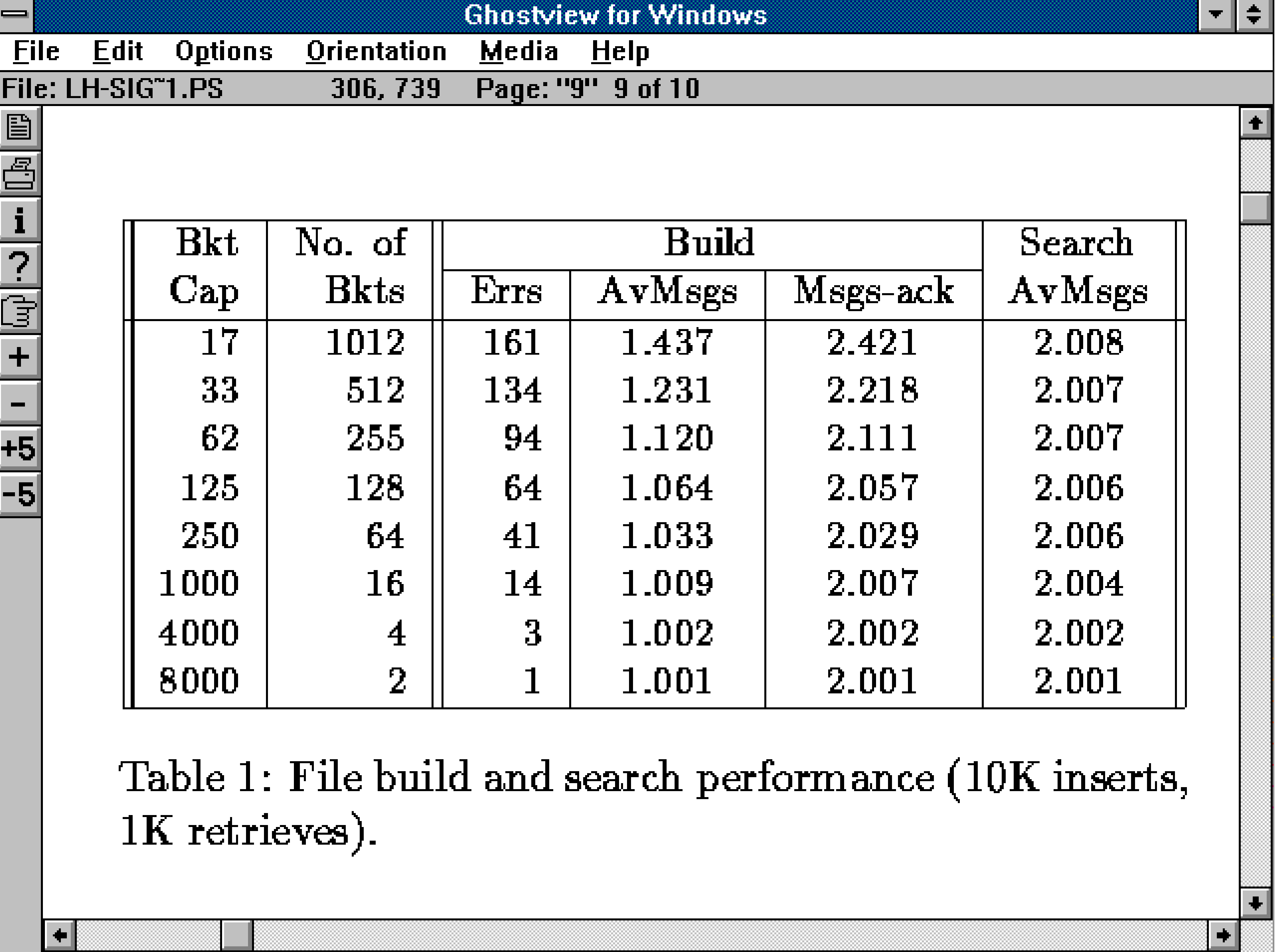
Client

Client

Result

- The distributed file can grow to even whole Internet so that :
  - every insert and search are done in four messages (IAM included)
  - in general an insert is done in one message and search in two messages
  - proof in [LNS 93]

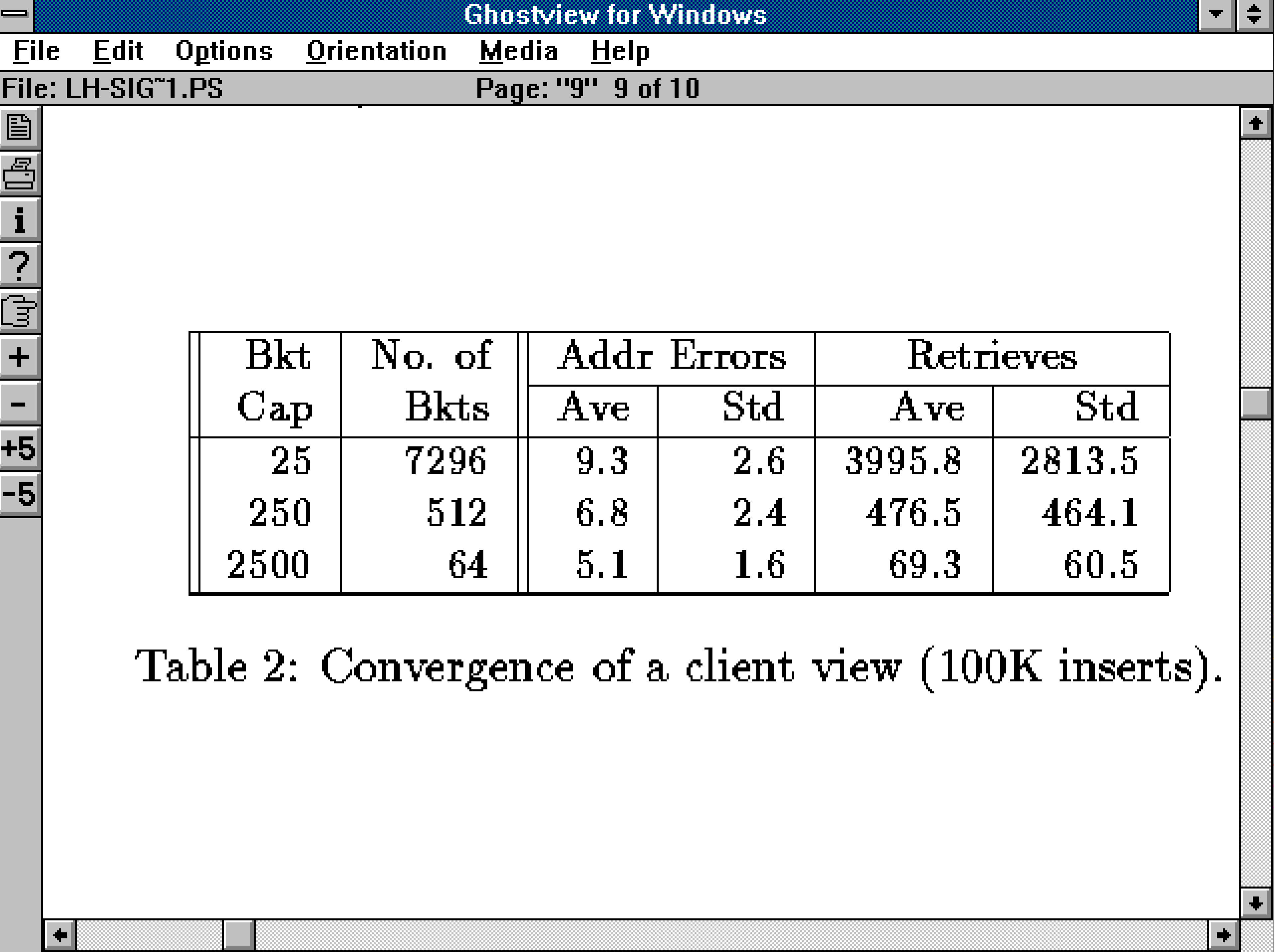

Ghostview for Windows
File
Edit
Options
Orientation
Media
Help
File: LH-SIG~1.PS
205, 567
Page: "8" 8 of 10
10,000 inserts
Global cost
Client's cost
Build (w/splits)
Build (w/o splits)
1.45
1.40
1.35
1.30
1.25
1.20
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0.80
30
100
300
1000
3000
10000
Bucket capacity
+5
-5



File: LH-SIG~1.PS 306, 739 Page: "9" 9 of 10

Bkt Cap	No. of Bkts	Build			Search AvMsgs
		Errs	AvMsgs	Msgs-ack	
17	1012	161	1.437	2.421	2.008
33	512	134	1.231	2.218	2.007
62	255	94	1.120	2.111	2.007
125	128	64	1.064	2.057	2.006
250	64	41	1.033	2.029	2.006
1000	16	14	1.009	2.007	2.004
4000	4	3	1.002	2.002	2.002
8000	2	1	1.001	2.001	2.001

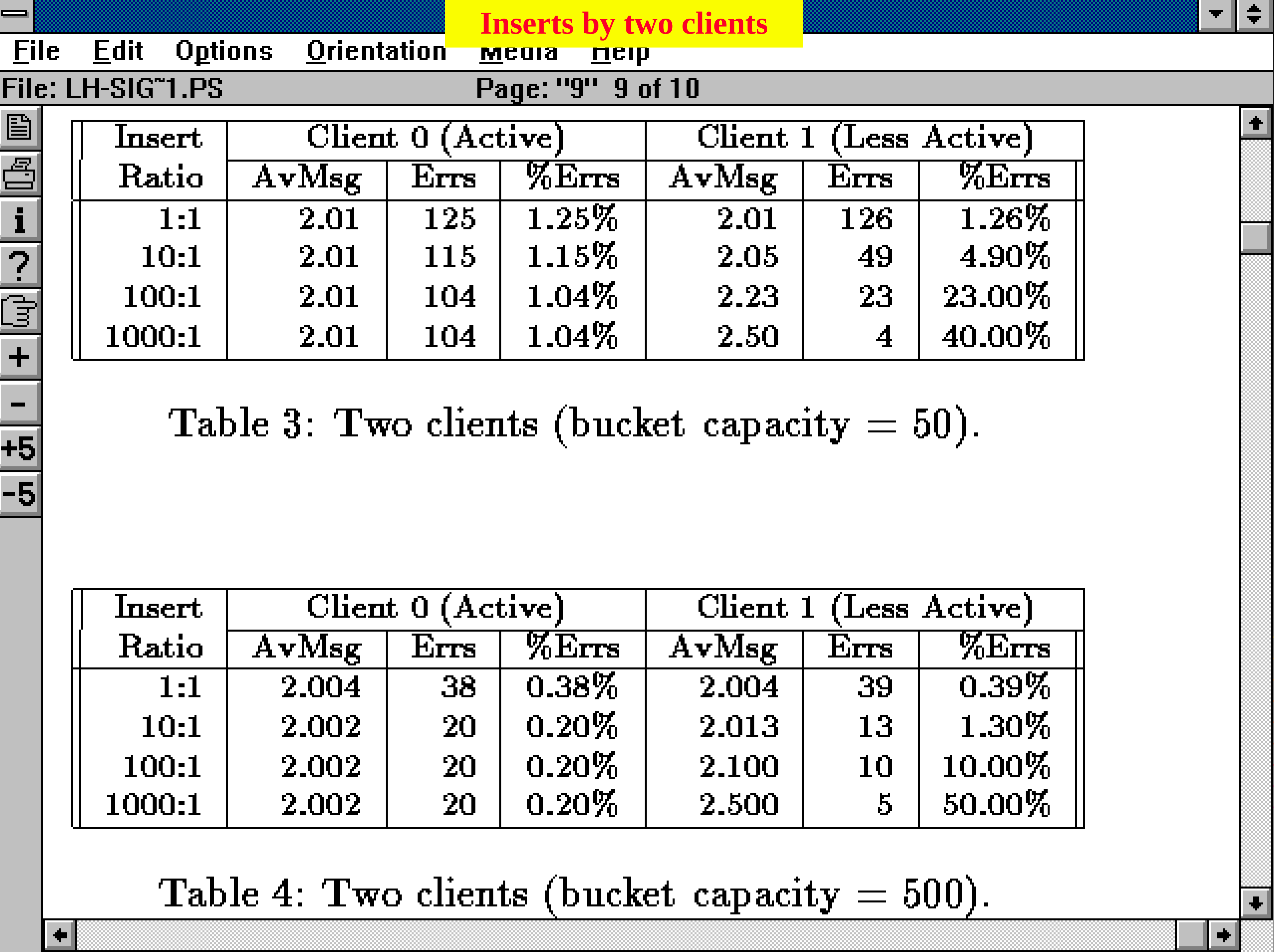
Table 1: File build and search performance (10K inserts, 1K retrieves).



Bkt Cap	No. of Bkts	Addr Errors		Retrieves	
		Ave	Std	Ave	Std
25	7296	9.3	2.6	3995.8	2813.5
250	512	6.8	2.4	476.5	464.1
2500	64	5.1	1.6	69.3	60.5

Table 2: Convergence of a client view (100K inserts).

Inserts by two clients



Insert Ratio	Client 0 (Active)			Client 1 (Less Active)		
	AvMsg	Errs	%Errs	AvMsg	Errs	%Errs
1:1	2.01	125	1.25%	2.01	126	1.26%
10:1	2.01	115	1.15%	2.05	49	4.90%
100:1	2.01	104	1.04%	2.23	23	23.00%
1000:1	2.01	104	1.04%	2.50	4	40.00%

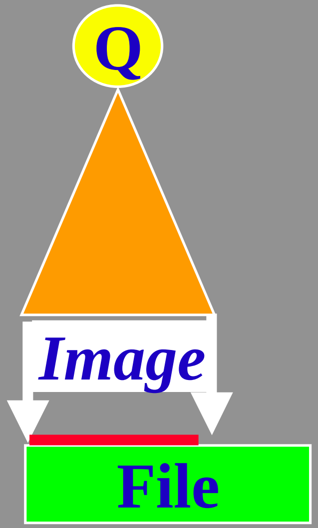
Table 3: Two clients (bucket capacity = 50).

Insert Ratio	Client 0 (Active)			Client 1 (Less Active)		
	AvMsg	Errs	%Errs	AvMsg	Errs	%Errs
1:1	2.004	38	0.38%	2.004	39	0.39%
10:1	2.002	20	0.20%	2.013	13	1.30%
100:1	2.002	20	0.20%	2.100	10	10.00%
1000:1	2.002	20	0.20%	2.500	5	50.00%

Table 4: Two clients (bucket capacity = 500).

Parallel Queries

- A query *Q* for all buckets of file ***F*** with independent local executions
  - every buckets should get *Q* exactly once
- The basis for function shipping
  - fundamental for high-perf. DBMS appl.
- Send Mode :
  - multicast
    - » not always possible or convenient
  - unicast
    - » client may not know all the servers
    - » severs have to forward the query
      - – how ??



LH* Algorithm for Parallel Queries

(unicast)

- Client sends ***Q*** to every bucket *a* in the image
- The message with ***Q*** has the *message level* j' :
  - initialy $j' = i'$ if $n' \leq \alpha < 2^{i'}$ else $j' = i' + 1$
  - bucket *a* (of level j) copies Q to all its children using the alg. :

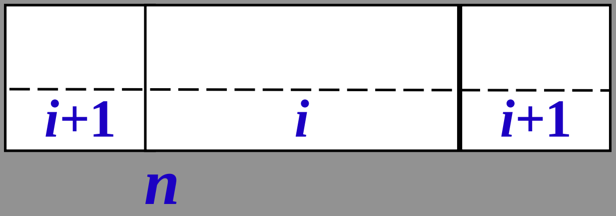
```
while $j' < j$ do
    $j' := j' + 1$
    forward ($Q$, $j'$ ) à case $a + 2^{j' - 1}$ ;
endwhile
```

- **Prove it !**

Termination of Parallel Query

(multicast or unicast)

- How client *C* knows that last reply came ?
- Deterministic Solution (expensive)
  - Every bucket sends its *j*, *m* and selected records if any
    - » *m* is its (logical) address
  - The client terminates when it has every *m* fullfiling the condition ;
    - » $m = 0,1..., 2^{i} + n$ where
      - $i = \min(j)$ and $n = \min(m)$ where $j = i$



Termination of Parallel Query

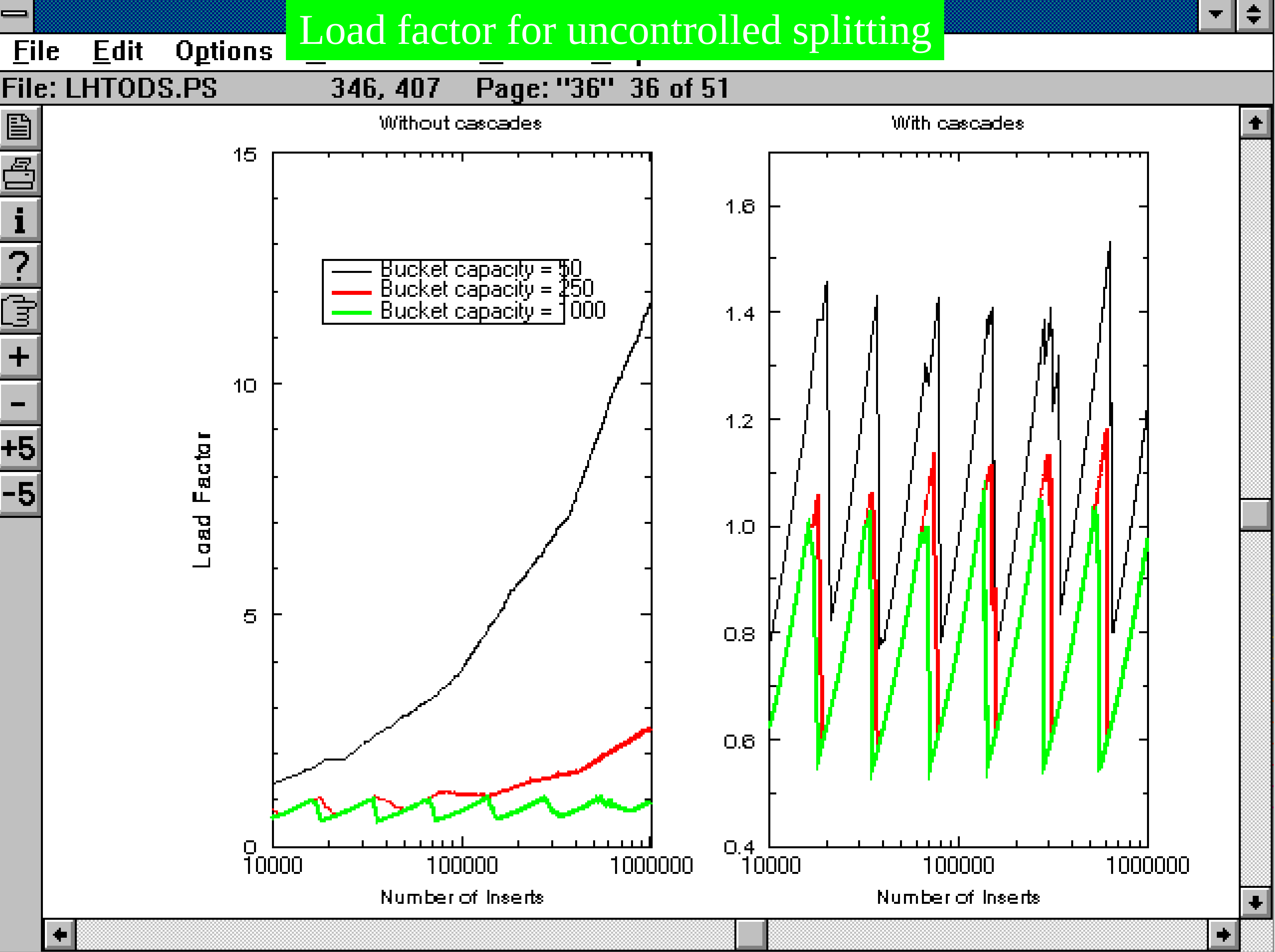
(multicast or unicast)

- Probabilistic Termination (may need less messaging)
  - all and only buckets with selected records reply
  - after each reply *C* reinitialises a time-out *T*
  - *C* terminates when *T* expires
- Practical choice of *T* *is* network and query dependent
  - ex. 5 times Ethernet everage retry time
    - » 1-2 msec ?
      - – experiments needed
- Which termination is finally more useful in practice ?
  - an open problem

LH* variants

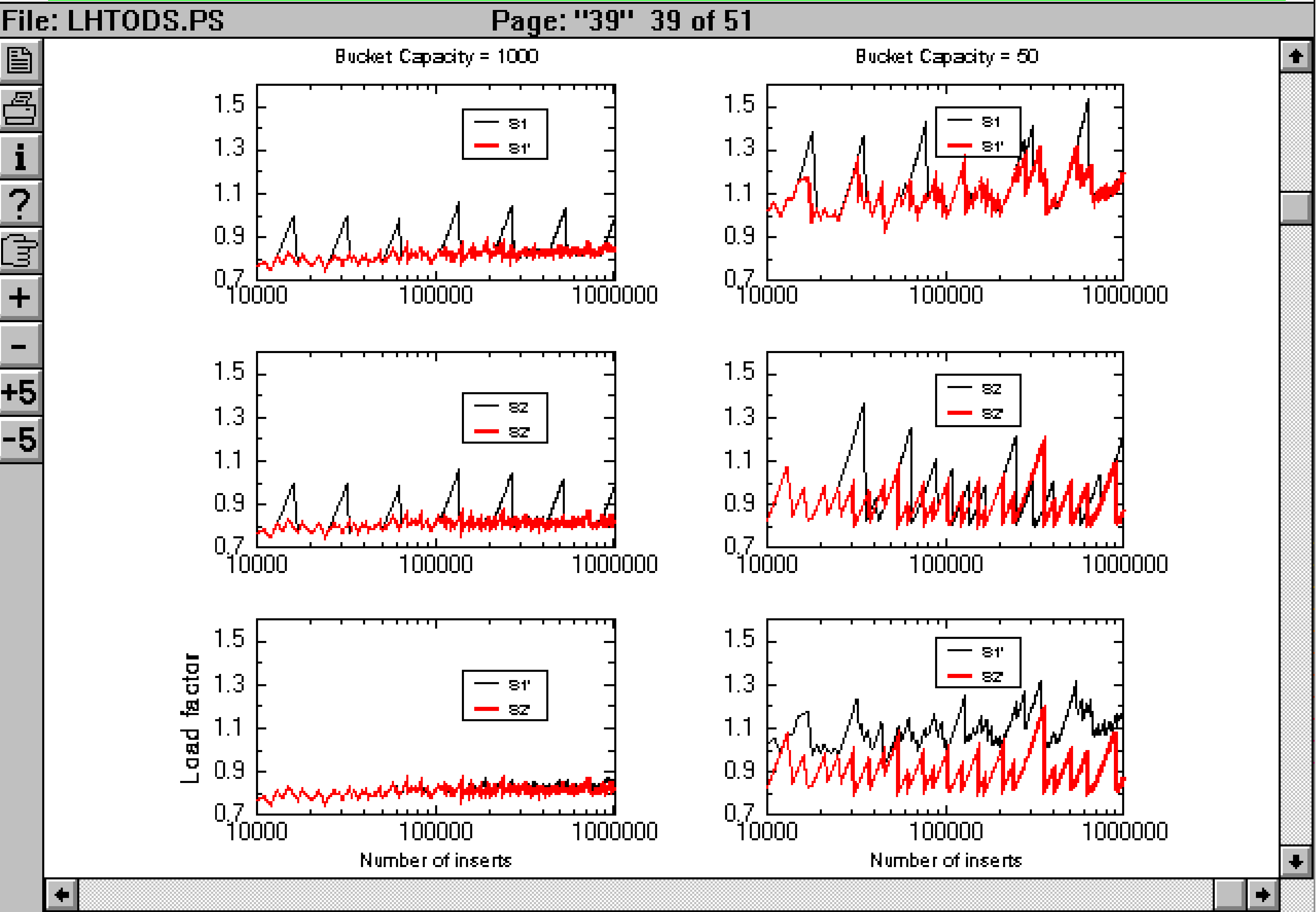
- With/without load (factor) control
- With/without the (split) coordinator
  - the former one was discussed
  - the latter one is a token-passing schema
    - bucket with the token is next to split
      - if an insert occurs, and file overload is guessed
        - several algs. for the decision
        - use cascading splits

Load factor for uncontrolled splitting



File: LHTODS.PS 346, 407 Page: "36" 36 of 51

Load factor for different load control strategies and threshold $t = 0.8$



File: LHTODS.PS Page: "39" 39 of 51

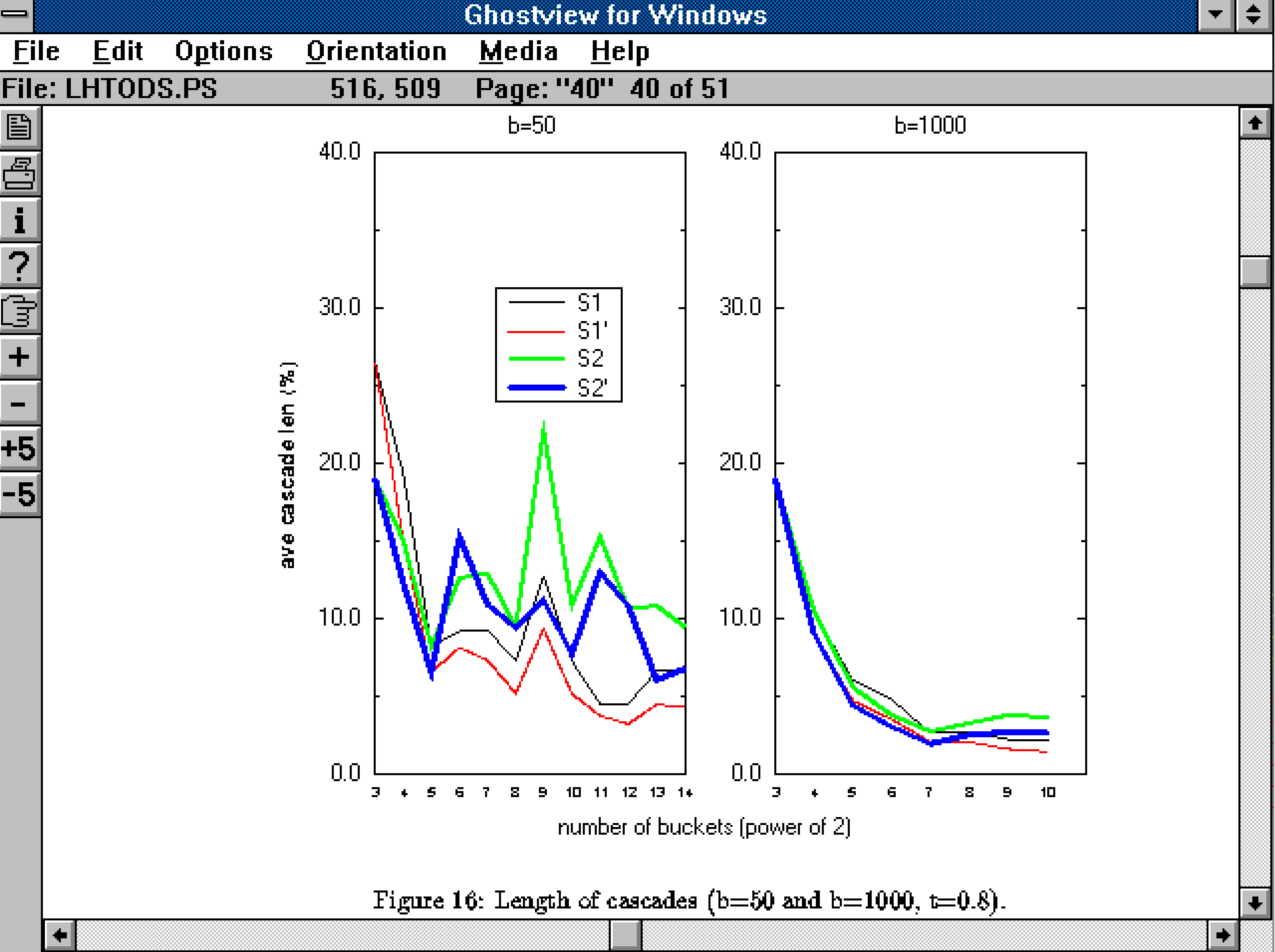


Figure 16: Length of cascades (b=50 and b=1000, t=0.8).

LH* for switched multicomputers

- LH^*_{LH}
  - implemented on Parsytec machine
    - » 32 Power PCs
    - » 2 GB of RAM (128 GB / CPU)
  - uses
    - » LH for the bucket management
    - » conurrent LH* splitting (described later on)
  - access times : < 1 ms
- Presented at EDBT-96

LH* with presplitting

- (Pre)splits are done "internally" immediately when an overflow occurs
- Become visible to clients, only when LH* split should be normally performed
- **Advantages**
  - less overflows on sites
  - parallel splits
- **Drawbacks**
  - Load factor
  - Possibly longer forwardings
- **Analysis remains to be done**

LH* with concurrent splitting

- Inserts and searches can be done concurrently with the splitting in progress
  - used by LH*$_{LH}$
- **Advantages**
  - obvious
  - and see EDBT-96
- **Drawbacks**
  - \+ alg. complexity

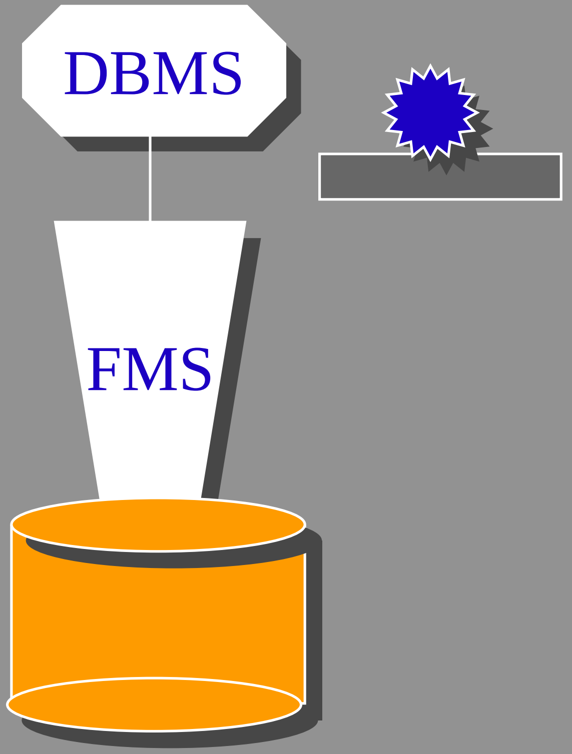
Research Frontier

- Actual implementation
  - the SDDS protocols
    - Reuse the MS CFIS protocol
    - + record types, forwarding, splitting, IAMs...
  - system architecture
    - client, server, sockets, UDP, TCP/IP, NT, Unix...
    - Threads
- Actual performance
    - 250 us per search
      - 1 KB records, 100 mb AnyLan Ethernet
      - **40 times faster than a disk**
      - **e.g. response time of a join improves from 1m to 1.5 s.**

Research Frontier

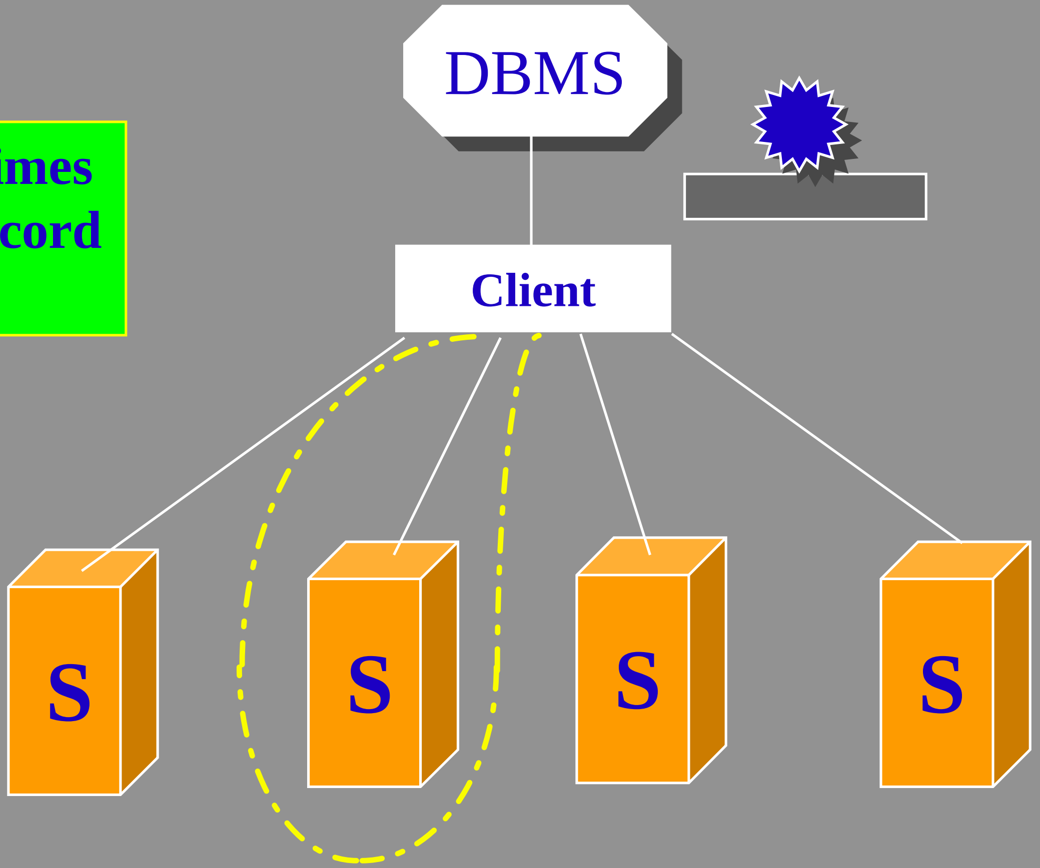
- Use within a DBMS
        - » **scalable AMOS, DB2 Parallel, Access**
    - – replace the traditional disk access methods
        - » DBMS is the single SDDS client
            - – LH* and perhaps other SDDSs
    - – use function shipping
    - – use from multiple distributed SDDS clients
        - » concurrency, transactions, recovery...
- Other applications
    - – A scalable WEB server (like INKTOMI)

Traditional



SDDS 1st stage

40 - 80 times faster record access

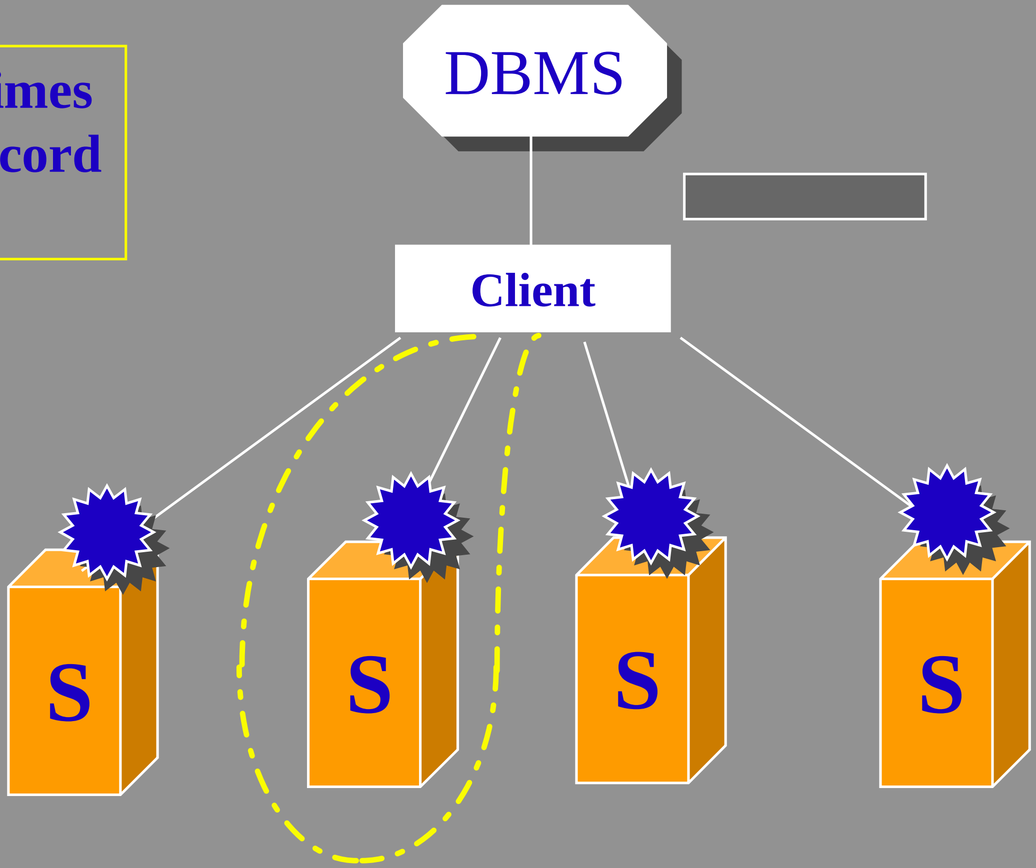


Memory
mapped
files

SDDS 2nd stage

40 - 80 times faster record access

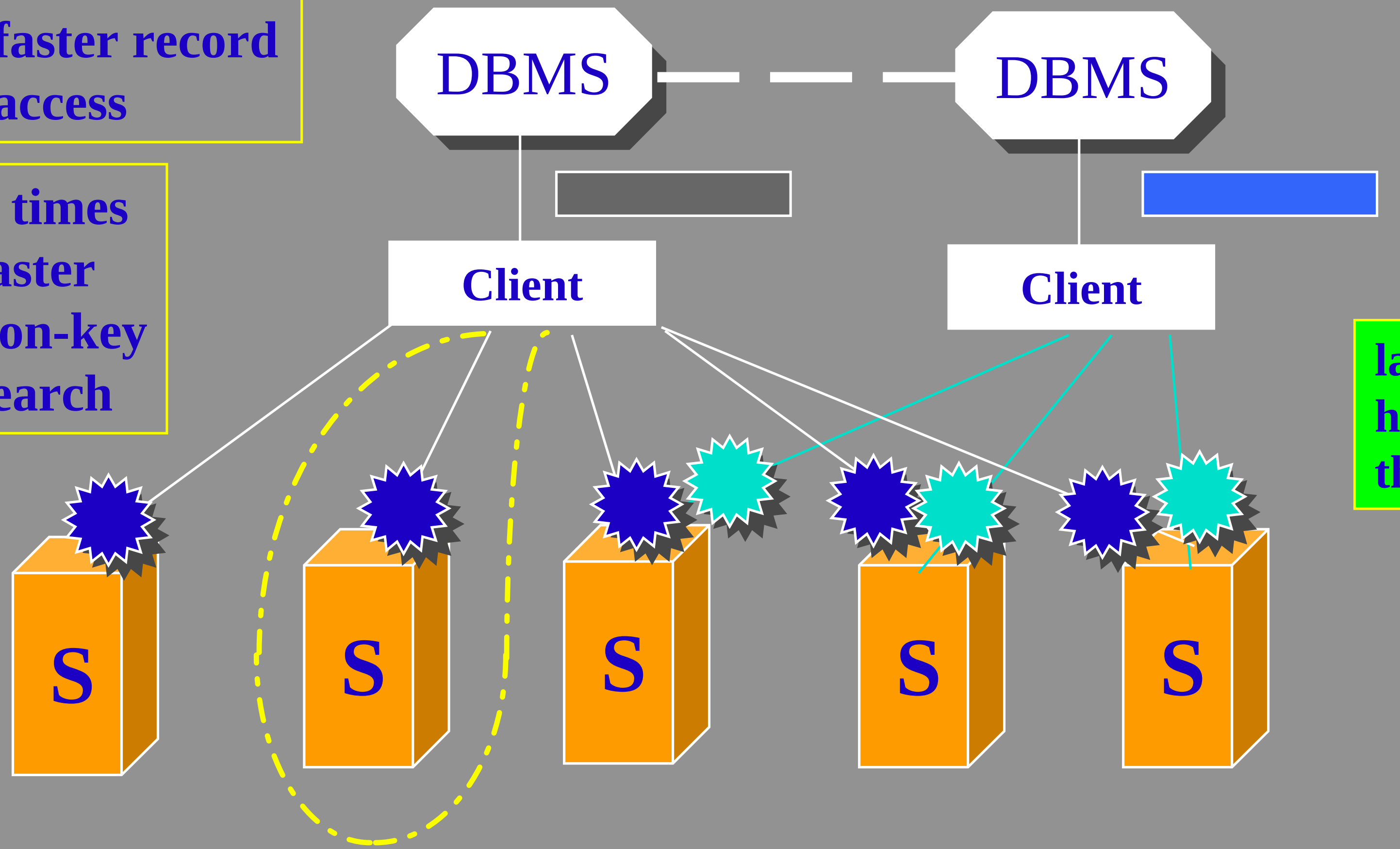
***n* times faster non-key search**



SDDS 3rd stage

40 - 80 times
faster record
access

n times
faster
non-key
search



larger files
higher
throughput

Conclusion

- Since their inception, in 1993, SDDS were subject to important research effort
- In a few years, several schemes appeared
  - with the basic functions of the traditional files
    - hash, primary key ordered, multi-attribute k-d access
  - providing for much faster and larger files
  - confirming inital expectations

Future work

- Deeper analysis
  - formal methods, simulations & experiments
- Prototype implementation
  - SDDS protocol (on-going in Paris 9)
- New schemes
  - High-Availability & Security
  - R* - trees ?
- Killer apps
  - large storage server & object servers
  - object-relational databases
    - Schneider, D & al (COMAD-94)
  - video servers
  - real-time
  - HP scientific data processing

END

(Part 1)

Thank you for your attention

Witold Litwin
litwin@dauphine.fr
wlitwin@cs.berkeley.edu

Scalable Distributed Data Structures

Part 2

Witold Litwin

Paris 9

litwin@cid5.etud.dauphine.fr

High-availability LH* schemes

- In a large multicomputer, it is unlikely that all servers are up
- Consider the probability that a bucket is up is 99 %
  - bucket is unavailable 3 days per year
- One stores every key in 1 bucket
  - case of typical SDDSs, LH* included
- Probability that *n*-bucket file is entirely up is
    - » 37 % for $n = 100$
    - » 0 % for $n = 1000$

High-availability LH* schemes

- Using 2 buckets to store a key, one may expect :
  - 99 % for *n* = 100
  - 91 % for *n* = 1000
- High availability SDDS
  - make sense
  - are the only way to reliable large SDDS files

High-availability LH* schemes

- High-availability LH* schemes keep data available despite server failures
  - any single server failure
  - most of two server failures
  - some catastrophic failures
- Three types of schemes are currently known
  - mirroring
  - striping
  - grouping

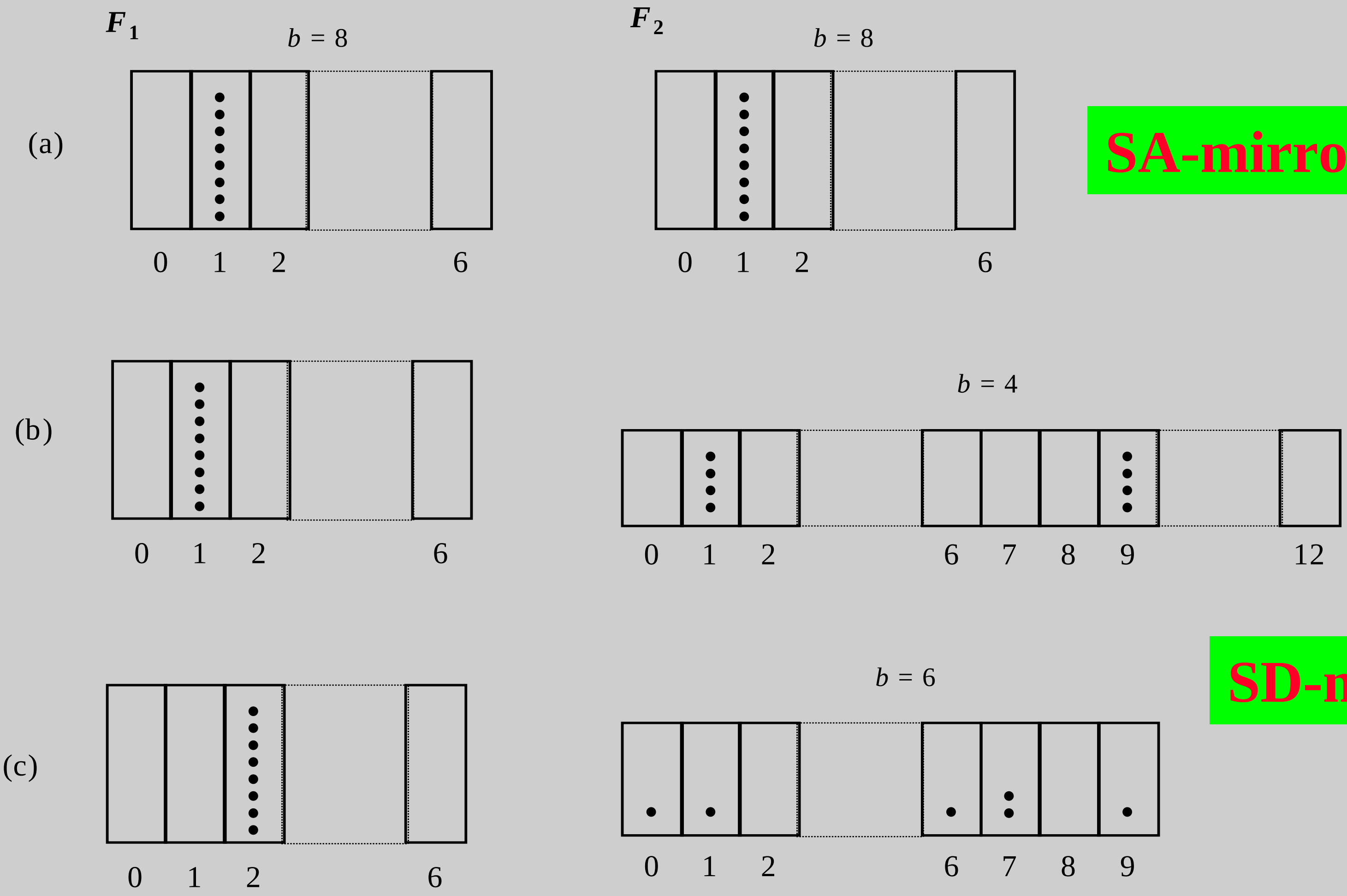
High-availability LH* schemes

- There are two files called mirrors
- Every insert propagates to both
  - splits are nevertheless autonomous
- Every search is directed towards one of the mirrors
  - the primary mirror for the corresponding client
- If a bucket failure is detected, the spare is produced **instantly** at some site
  - the storage for failed bucket is reclaimed
  - it is allocated to another bucket when again available

High-availability LH* schemes

- Two types of LH* schemes with mirroring appear
- Structurally-alike (SA) mirrors
  - same file parameters
    - keys are presumably at the same buckets
- Structurally-dissimilar (SD) mirrors
    - keys are presumably at different buckets
  - loosely coupled = same LH-functions h_i
  - minimally coupled = different LH-functions h_i

LH* with mirroring



SA-mirrors

SD-mirrors

LH* with mirroring

- SA-mirrors
  - most efficient for access and spare production
  - but max loss in the case of two-bucket failure
- Loosely-coupled SD-mirrors
  - less efficient for access and spare production
  - lesser loss of data for a two-bucket failure
- Minimally-coupled SD-mirrors
  - least efficient for access and spare production
  - min. loss for a two-bucket failure

Some design issues

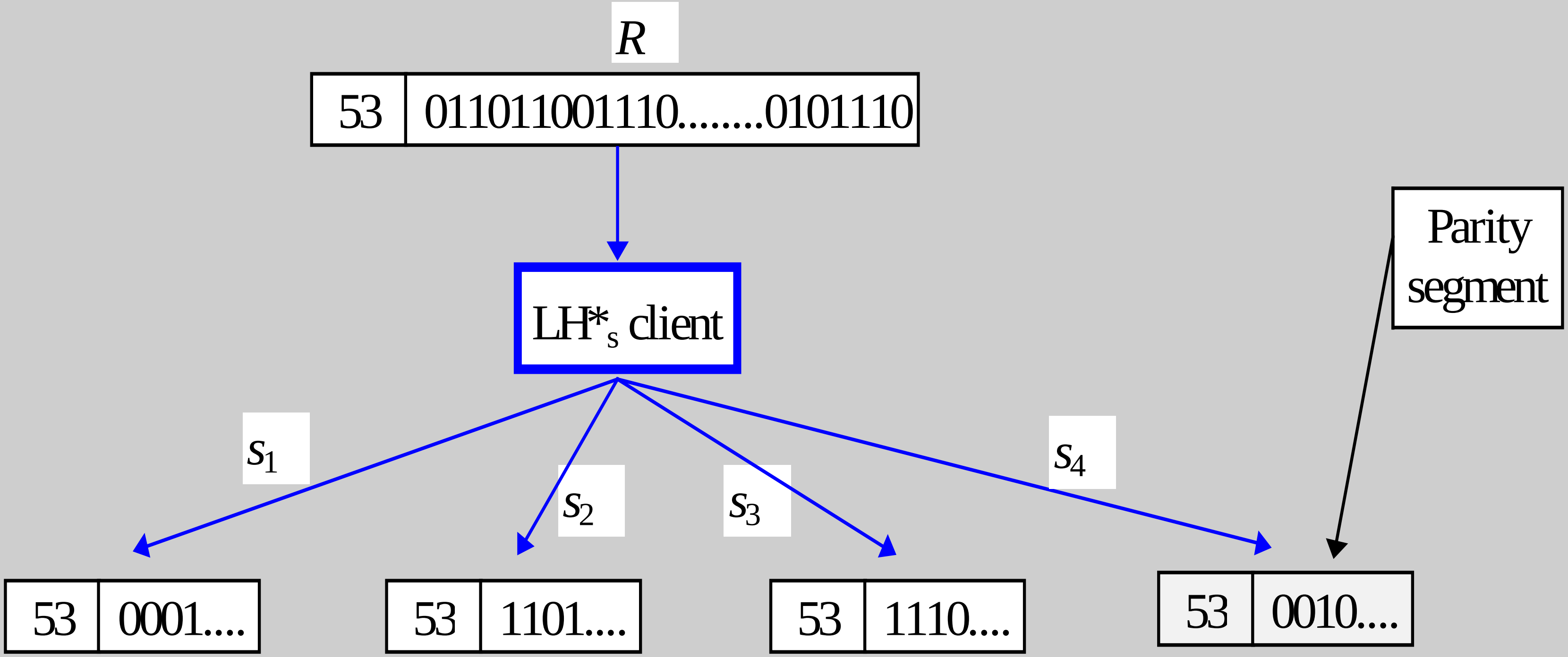
- Spare production algorithm
- Propagating the spare address to the client
- Forwarding in the presence of failure
- Discussion in :
  - *High-Availability LH* Schemes with Mirroring. W. Litwin, M.-A. Neimat.* *COOPIS-96, Brussels*

LH* with striping

(LH* arrays)

- High-availability through striping of a record among several LH* files
  - as for RAID (disk array) schemes
  - but scalable to as many sites as needed
- Less storage than for LH* with mirroring
- But less efficient for insert and search
  - more messaging
    - » although using shorter messages

LH*$_S$



R
53
011011001110........0101110
LH*$_s$ client
Parity segment
s_1
s_2
s_3
s_4
53
0001....
53
1101....
53
1110....
53
0010....

LH*$_S$

- Spare segments are produced when failures are detected
- Segment file schemes are as for LH* with mirroring
  - SA-segment files
    - + perf. pour l'adressage, mais - perf. pour la sécurité
  - SD-segment files
- Performance analysis in detail remains to be done

Variantes

- Segmentation level
  - bit
    - **best security**
      - meaningless single-site data
      - meaningless content of a message
  - block
    - less CPU time for the segmentation
  - attribute
    - selective access to segments becomes possible
    - fastest non-key search

LH*g

- Avoids striping
  - to improve non-key search time
  - keeping about the same storage requirements for the file
- Uses *grouping* of *k* records instead
  - group members remain always in different buckets
    - despite the splits and file growth
- Allows for high-availability on demand
  - without restructuring the original LH* file

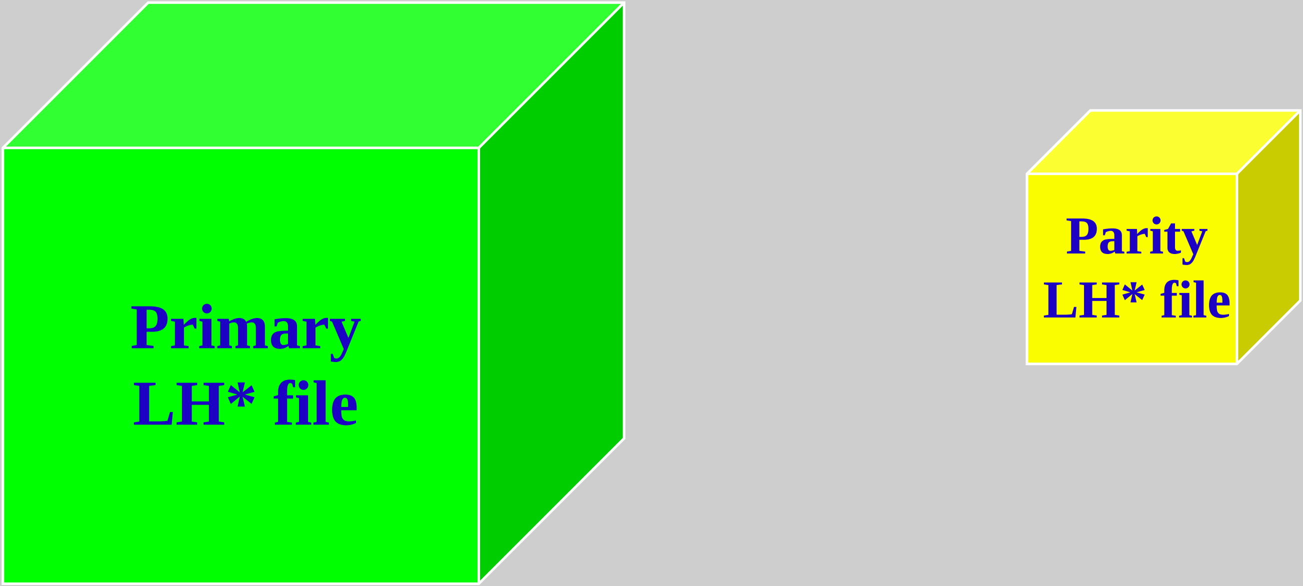
LH*g

Primary record

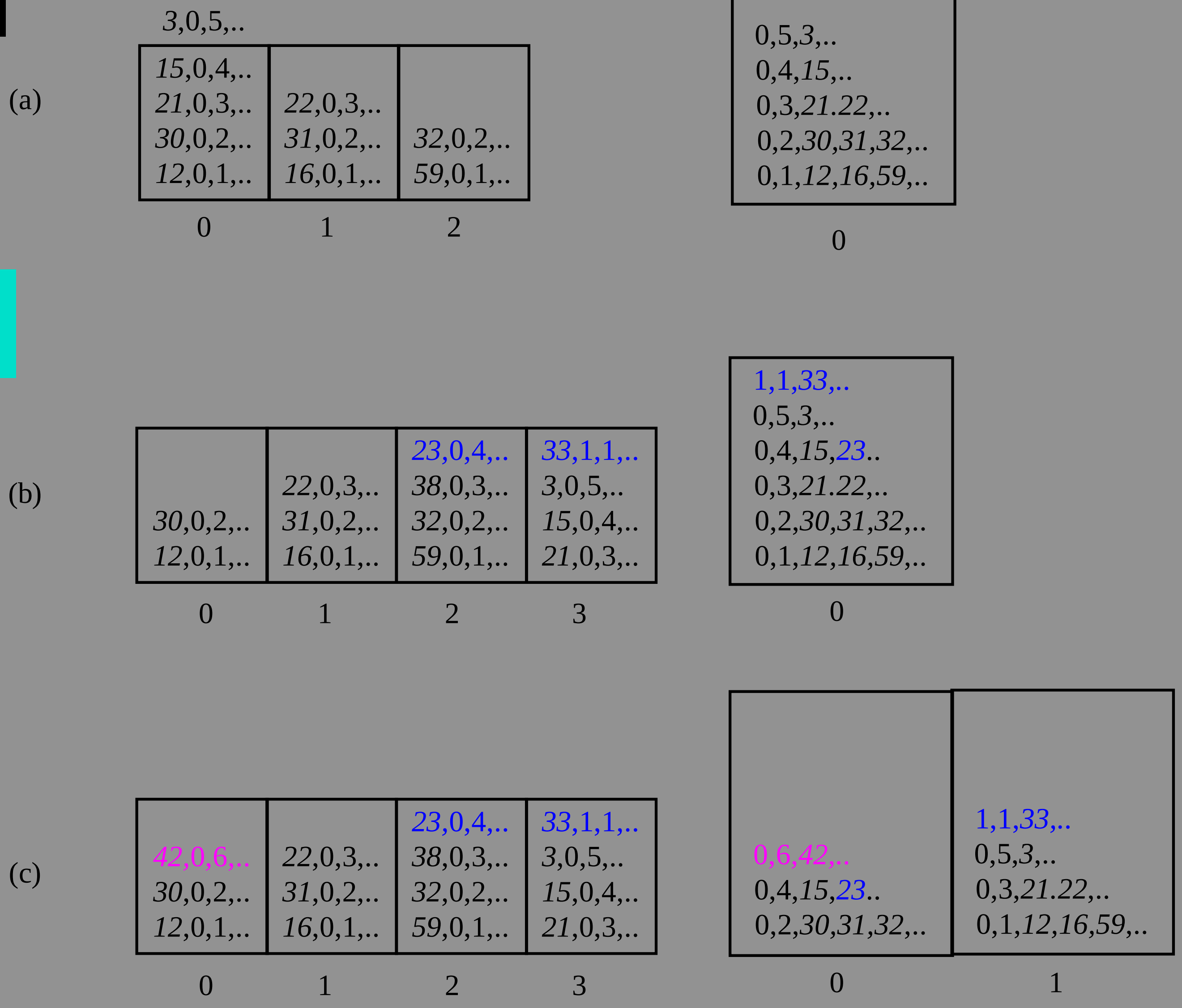
c	*g*	non-key data

Parity record

g	c_1	…	c_k	parity bits



LH*g



(a)

3,0,5,..

0	1	2
15,0,4,.. *21*,0,3,.. *30*,0,2,.. *12*,0,1,..	*22*,0,3,.. *31*,0,2,.. *16*,0,1,..	*32*,0,2,.. *59*,0,1,..

0
0,5,*3*,.. 0,4,*15*,.. 0,3,*21.22*,.. 0,2,*30*,*31*,*32*,.. 0,1,*12*,*16*,*59*,..

(b)

0	1	2	3
30,0,2,.. *12*,0,1,..	*22*,0,3,.. *31*,0,2,.. *16*,0,1,..	*23*,0,4,.. *38*,0,3,.. *32*,0,2,.. *59*,0,1,..	*33*,1,1,.. *3*,0,5,.. *15*,0,4,.. *21*,0,3,..

0
1,1,*33*,.. 0,5,*3*,.. 0,4,*15*,*23*.. 0,3,*21.22*,.. 0,2,*30*,*31*,*32*,.. 0,1,*12*,*16*,*59*,..

(c)

0	1	2	3
42,0,6,.. *30*,0,2,.. *12*,0,1,..	*22*,0,3,.. *31*,0,2,.. *16*,0,1,..	*23*,0,4,.. *38*,0,3,.. *32*,0,2,.. *59*,0,1,..	*33*,1,1,.. *3*,0,5,.. *15*,0,4,.. *21*,0,3,..

0	1
0,6,*42*,.. 0,4,*15*,*23*.. 0,2,*30*,*31*,*32*,..	1,1,*33*,.. 0,5,*3*,.. 0,3,*21.22*,.. 0,1,*12*,*16*,*59*,..

Group size
k = 3

Evolution of an LH*g file before 1st split (a), and after a few more inserts, (b), (c).

LH*g

- If a primary or parity bucket fails
  - the hot-spare can always be produced from the group members that are still alive
- If more than one group member fails
  - then there is data loss
- Unless the parity file has more extensive data
  - e.g. Hamming codes

Other hash SDDSs

- DDH (B. Devine, FODO-94)
  - uses Extensible Hash as the kernel
  - clients have EH images
  - **less overflows**
  - **more forwardings**
- Breitbart & al (ACM-Sigmod-94)
  - **less overflows & better load**
  - **more forwardings**

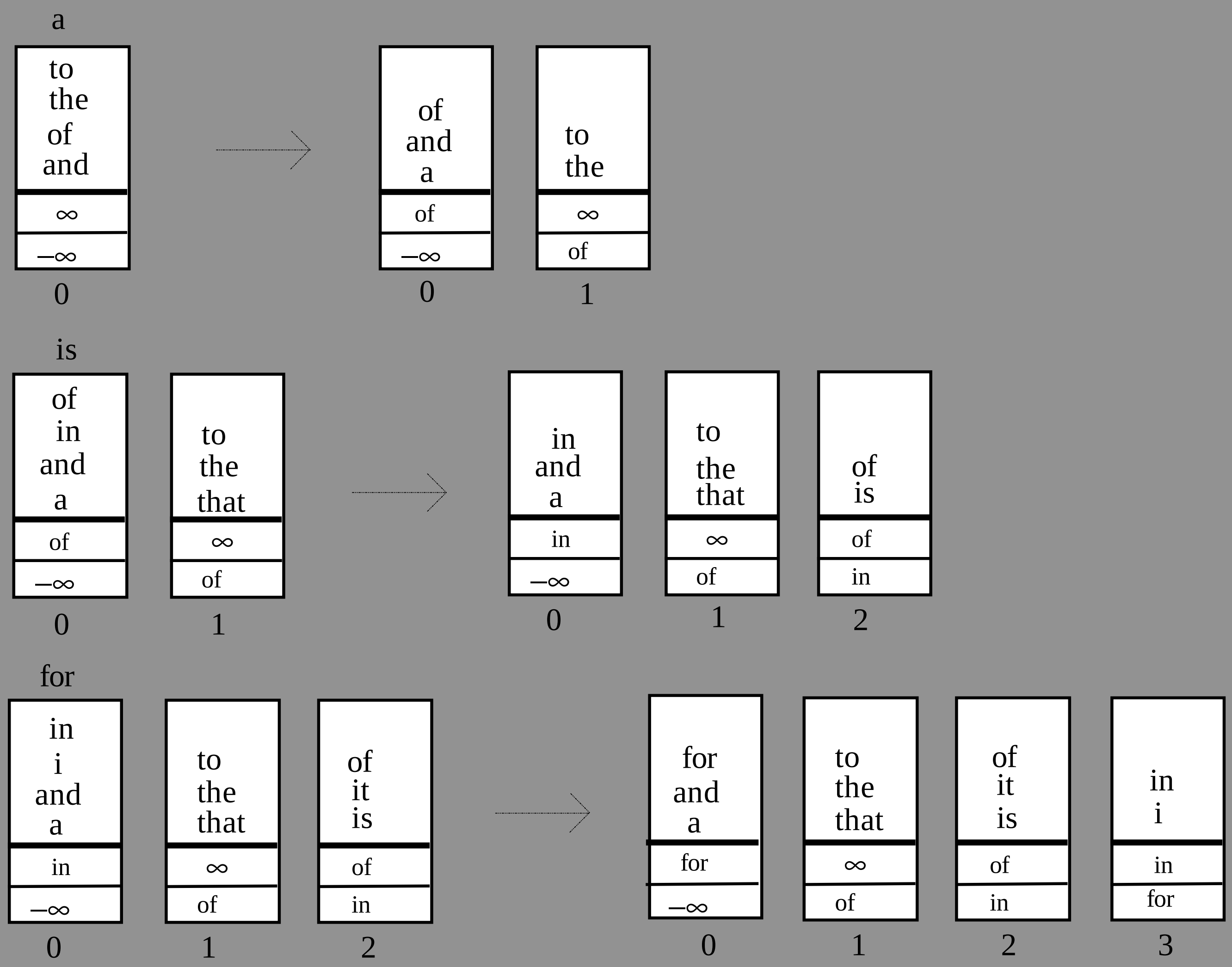
RP* schemes

- Produce 1-d ordered files
  - for range search
- Uses m-ary trees
  - like a B-tree
- Efficiently supports range queries
  - LH* also supports range queries
    - but less efficiently
- Consists of the family of three schemes
  - $RP*_N$ $RP*_C$ and $RP*_S$

RP* schemes

RP*$_S$	+ servers index	optional multicast
RP*$_C$	+ client index	limited multicast
RP*$_N$	No index	all multicast

RP* design trade-offs

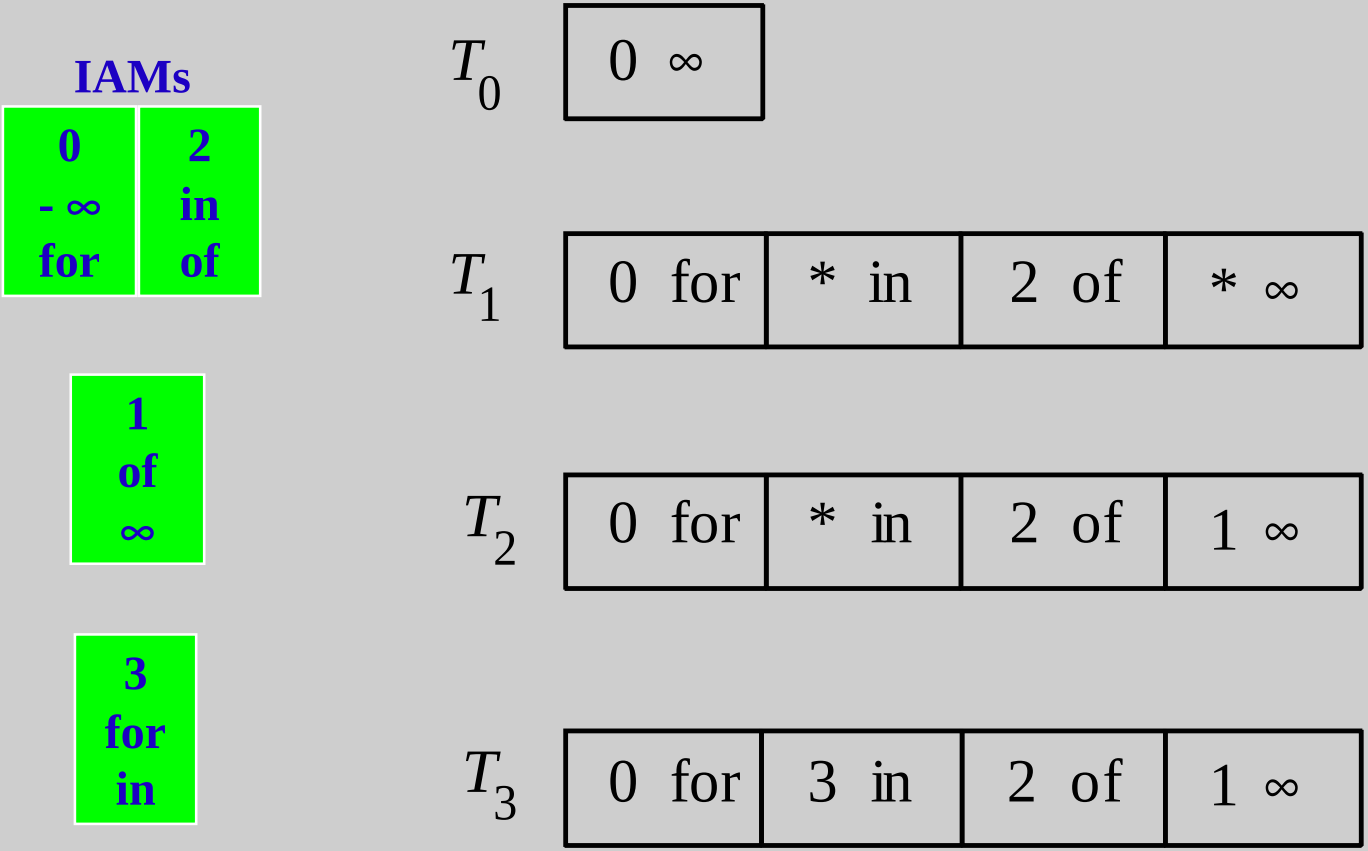


RP* file expansion

RP* Range Query Termination

- Time-out
- Deterministic
  - Each server addressed by *Q* sends back at least its current range
  - The client performs the union *U* of all results
  - It terminates when *U* covers *Q*

RP*c client image



IAMs

0 - ∞ for | **2 in of**

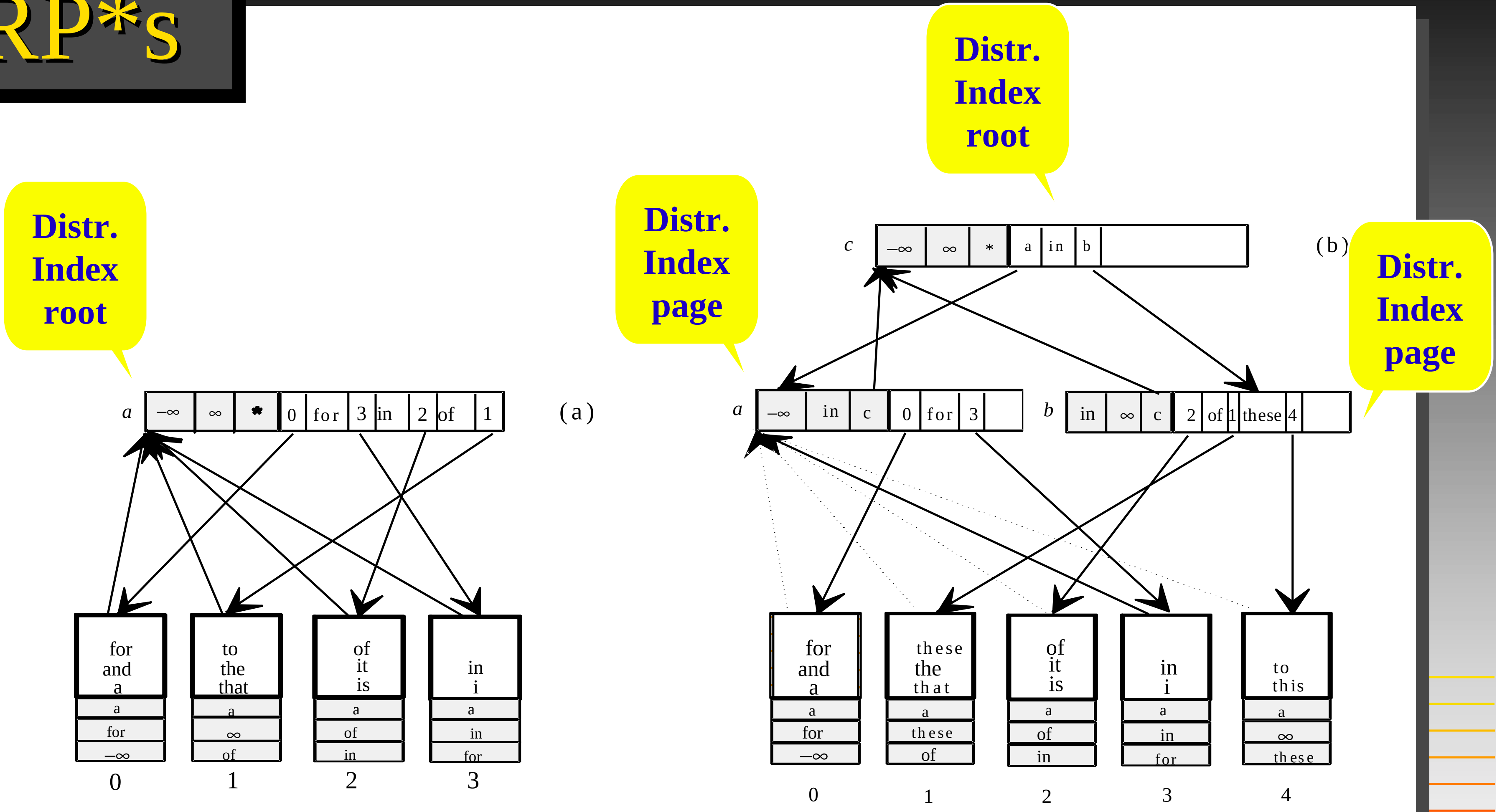
1 of ∞

3 for in

T_0	0 ∞			
T_1	0 for	* in	2 of	* ∞
T_2	0 for	* in	2 of	1 ∞
T_3	0 for	3 in	2 of	1 ∞

Evolution of RP*$_c$ client image after searches for keys **it, that, in**

RP*s



Distr. Index root

Distr. Index page

Distr. Index root

Distr. Index page

IAM = traversed pages

An RP^*_s file with (a) 2-level kernel, and (b) 3-level kernel

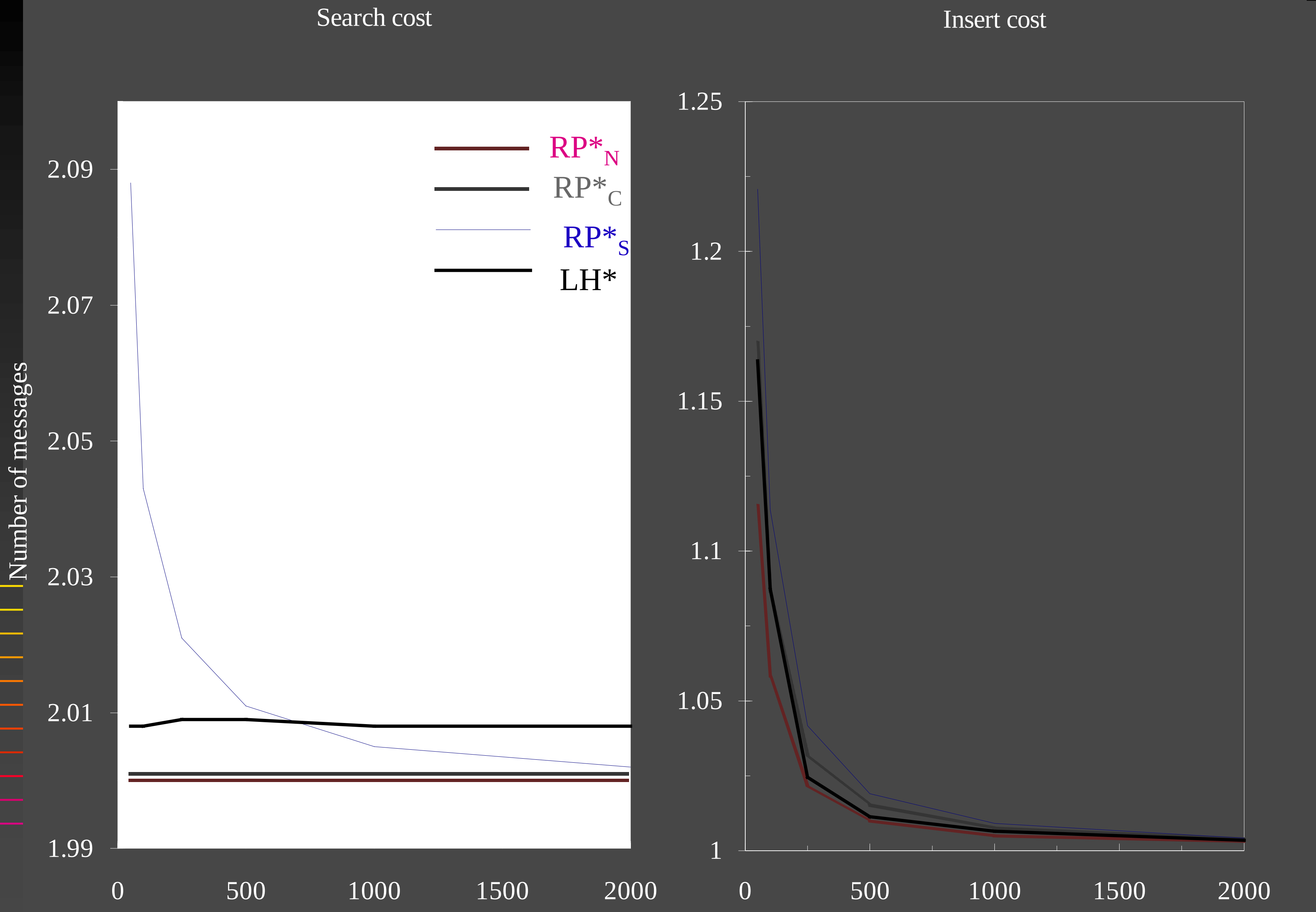
RP*$_N$

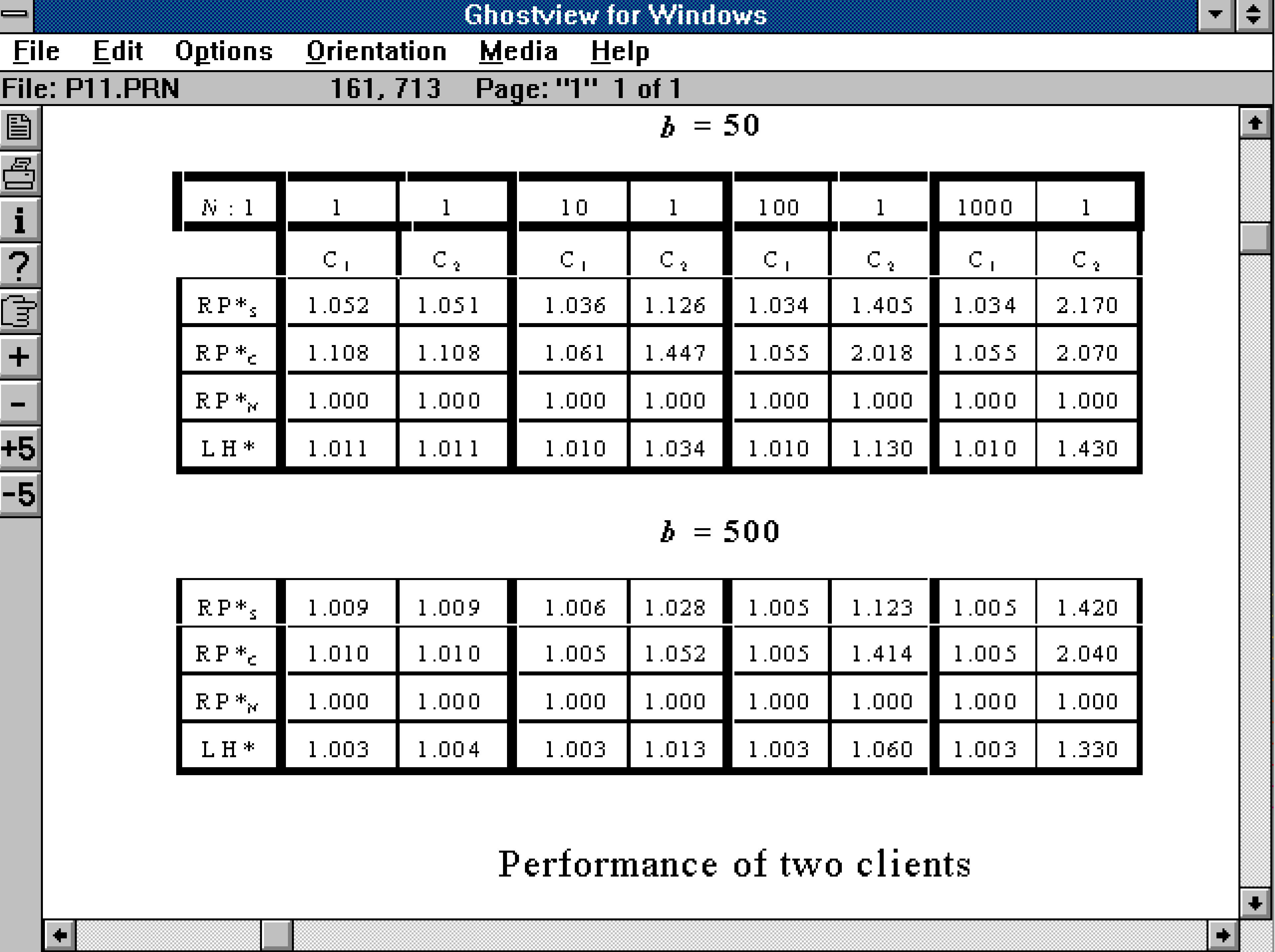
	m-net	h-net	g-net
	10 Mb/s	100 Mb/s	1 Gb/s
t_i	1.061 ms	161 µs	71 µs
t_s	1.176 ms	186 µs	87 µs
t_r	10.141 ms	1.061 ms	152 µs
t_g	15.585 ms	1.555 ms	585 µs
$t_{b\text{-}i}$	1010 ms	100.06 ms	10.07 ms
$t_{i,\,t}$	1.010 ms	110 µs	20 µs
$t_{s,\,t}$	1.120 ms	130 µs	31µs

s_i	965 o/s	7352 o/s	21739 o/s
$s_{i,\,t}$	990 o/s	9991 o/s	50000 o/s
$\%_{CPU}$	3 %	19 %	57 %
s_s	872 o/s	6410 o/s	17544 o/s
$s_{s,\,t}$	893 o/s	7692 o/s	32258 o/s
$\%_{CPU}$	2 %	17 %	45 %

drawback
of
multicasting

Elapsed times and throughputs


Search cost
Insert cost
Number of messages
RP*$_N$
RP*$_C$
RP*$_S$
LH*
2.09
2.07
2.05
2.03
2.01
1.99
0
500
1000
1500
2000
1.25
1.2
1.15
1.1
1.05
1
0
500
1000
1500
2000



$b = 50$

$N : 1$	1	1	10	1	100	1	1000	1
	C_1	C_2	C_1	C_2	C_1	C_2	C_1	C_2
RP^*_s	1.052	1.051	1.036	1.126	1.034	1.405	1.034	2.170
RP^*_c	1.108	1.108	1.061	1.447	1.055	2.018	1.055	2.070
RP^*_N	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
LH^*	1.011	1.011	1.010	1.034	1.010	1.130	1.010	1.430

$b = 500$

RP^*_s	1.009	1.009	1.006	1.028	1.005	1.123	1.005	1.420
RP^*_c	1.010	1.010	1.005	1.052	1.005	1.414	1.005	2.040
RP^*_N	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
LH^*	1.003	1.004	1.003	1.013	1.003	1.060	1.003	1.330

Performance of two clients

b	RP^*_C	RP^*_S	LH*
50	2867	22.9	8.9
100	1438	11.4	8.2
250	543	5.9	6.8
500	258	3.1	6.4
1000	127	1.5	5.7
2000	63	1.0	5.2

Number of IAMs until image convergence

Research Frontier

High Availability RP* schemes

High Availability RP* schemes

?

Kroll & Widmayer schema

(ACM-Sigmod 94)

- Provides for 1-d ordered files
  - practical alternative to RP* schemes
- Efficiently supports range queries
- Uses a paged distributed binary tree
  - can get unbalanced

k-RP*

- Provides for multiattribute (k-d) search
  - key search
  - partial match and range search
  - candidate key search
    - Orders of magnitude better search perf. than traditional ones
- Uses
  - a paged distributed k-d tree index on the servers
  - partial k-d trees on the clients

Access performance

(case study)

- Three queries to a 400 MB, 4GB and a 40 GB file
  - Q1 - A range query, which selects 1% of the file
  - Q2 - Query Q1 and an additional predicate on non-key attributes selecting 0.5% of the records selected by Q1
  - Q3 - A partial match x0 = c0 successful search in a 3-d file, where x0 is a candidate key
- Response time is computed for:
  - a traditional disk file
  - a k-RP* file on a 10 Mb/s net
  - a k-RP* file on a 1 Gb/s net
- Factor *S* is the corresponding speed-up
  - **reaches 7 orders of magnitude**

F [GB]	$Q_{1,t}$ [s]		$Q_{2,t}$ [s]		$Q_{3,t}$ [s]	
0.4	15		15		84	
4	150		150		474	
40	1,500		1,500		2667	
	$Q_{1,e}$ [s]	S_1	$Q_{2,e}$ [s]	S_2	$Q_{3,e,d}$ [ms]	S_3
0.4	2.24	7	0.01	1,674	1.17	29,612
4	22.40	7	0.09	1,674	3.25	61,386
40	224.00	7	0.90	1,674	14.95	79,844
				$Q_{3,e,t}$ [ms]	1	696,238
	$Q_{1,g}$ [s]	S_1	$Q_{2,g}$ [ms]	S_2	$Q_{3,g,d}$ [µs]	S_3
0.4	0.02	669	0.11	136,861	31.70	1,045,393
4	0.22	670	0.92	163,755	52.50	3,375,681
40	2.24	670	8.98	163,755	169.46	6,494,760
				$Q_{3,g,t}$ [µs]	31	22,459,301

Access times to a k-RP$^*_s$ file, and to a traditional file of the same size

dPi-tree

- Side pointers between the leaves
  - traversed when an addressing error occurs
    - a limit can be set-up to guard against the worst case
- Base index at some server
- Client images tree built path-by-path
  - from base index or through IAMs
    - called correction messages
- Basic performance similar to k-RP* ?
  - Analysis pending

Conclusion

- Since their inception, in 1993, SDDS were subject to important research effort
- In a few years, several schemes appeared
  - with the basic functions of the traditional files
    - hash, primary key ordered, multi-attribute k-d access
  - providing for much faster and larger files
  - confirming inital expectations

Future work

- Deeper analysis
  - formal methods, simulations & experiments
- Prototype implementation
  - SDDS protocol (on-going in Paris 9)
- New schemes
  - High-Availability & Security
  - R* - trees ?
- Killer apps
  - large storage server & object servers
  - object-relational databases
    - Schneider, D & al (COMAD-94)
  - video servers
  - real-time
  - HP scientific data processing

END

Thank you for your attention

Witold Litwin
litwin@dauphine.fr
wlitwin@cs.berkeley.edu