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

Aly Wâne Diène & Witold Litwin
CERIA
University Paris 9 Dauphine
http://ceria.dauphine.fr
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*)…
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

- \( \text{RP}^*_N \)
  - no index
  - query multicast

- \( \text{RP}^*_C \)
  - \( \text{RP}^*_N \) + client index
  - query multicast only if the bucket address not in the index
  - forwarding by multicast

- \( \text{RP}^*_C \uparrow \)
  - variant of \( \text{RP}^*_C \) without multicast by the client

- \( \text{RP}^*_S \)
  - \( \text{RP}^*_C \) + servers' index
  - Optional multicast
  - For range queries only
**Header**
- Bucket range
- Address of the index root
- Bucket size...

**Index**
- Kind 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

Main memory

RP* Buckets

RP* Functions:
Insert, Search, Update, Delete, Forward, Split.

Execution

Request Analyze

Results

Network (TCP/IP, UDP)

Client

Client

ListenThread

SendAck

Response

Requests queue

Ack queue

W.Thread 1

W.Thread N

...
**SDDS-2000: Client Architecture**

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

### Diagram

- **Server**
- **Network (TCP/IP, UDP)**

- **Client Flow control Manager**
  - Key
  - IP Add.
  - Update

- **Requests Journal**
  - Id_Req
  - Id_App

- **Images**

- **Analysis**
  - 1
  - 2
  - 3
  - 4

- **Send Module**
  - Send Request

- **Receive Module**
  - Receive Request

- **SDDS Applications Interface**

- **Application 1**
  - ... 
  - Application N
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

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

Split times for different bucket capacity
Discussion

• 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; \(k = 1 \ldots 5\)
- Either with empty client image adjusted by IAMs or with correct image

<table>
<thead>
<tr>
<th>(k)</th>
<th>(k = 1 \ldots 5)</th>
<th>(RP^*_C)</th>
<th>(RP^*_N)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With flow control</td>
<td>Without flow control</td>
<td>With flow control</td>
</tr>
<tr>
<td></td>
<td>Empty image</td>
<td>Correct image</td>
<td>Empty image</td>
</tr>
<tr>
<td>1</td>
<td>35511</td>
<td>0.355</td>
<td>27480</td>
</tr>
<tr>
<td>2</td>
<td>27767</td>
<td>0.258</td>
<td>14440</td>
</tr>
<tr>
<td>3</td>
<td>23514</td>
<td>0.235</td>
<td>11176</td>
</tr>
<tr>
<td>4</td>
<td>22332</td>
<td>0.223</td>
<td>9213</td>
</tr>
<tr>
<td>5</td>
<td>22101</td>
<td>0.221</td>
<td>9224</td>
</tr>
</tbody>
</table>

**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**

![Graph showing performance analysis results](image-url)
Discussion

- Cost of IAMs is negligible
- Insert throughput 110 times faster than for a disk file
  - 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

<table>
<thead>
<tr>
<th>.k</th>
<th>RP*C</th>
<th>RP*N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With flow control</td>
<td>Without flow control</td>
</tr>
<tr>
<td></td>
<td>Ttl time</td>
<td>Avg time</td>
</tr>
<tr>
<td>1</td>
<td>34019</td>
<td>0.340</td>
</tr>
<tr>
<td>2</td>
<td>25767</td>
<td>0.258</td>
</tr>
<tr>
<td>3</td>
<td>21431</td>
<td>0.214</td>
</tr>
<tr>
<td>4</td>
<td>20389</td>
<td>0.204</td>
</tr>
<tr>
<td>5</td>
<td>19987</td>
<td>0.200</td>
</tr>
</tbody>
</table>

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 $\mu$s per insert

• Search throughput more than 65 times faster than that of a disk file
  – 145 $\mu$s 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

![Graph 1: Range query total time vs. Number of servers](image1)

![Graph 2: Range query time per record vs. Number of servers](image2)
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
  - 1235s for $R_{P_N}^*$
  - 1205s for $R_{P_C}^*$
- 285 splits would last 285s in total
- Inserts alone would last
  - 950s for $R_{P_N}^*$
  - 920s for $R_{P_C}^*$
### Related Works

<table>
<thead>
<tr>
<th></th>
<th>LH* Imp.</th>
<th>RP*&lt;sub&gt;N&lt;/sub&gt; Thr.</th>
<th>RP*&lt;sub&gt;N&lt;/sub&gt; Imp.</th>
<th>RP*&lt;sub&gt;C&lt;/sub&gt; Impl.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>With FC</td>
<td>No FC</td>
<td>With FC</td>
</tr>
<tr>
<td>$t_c$</td>
<td>51000</td>
<td>40250</td>
<td>69209</td>
<td>47798</td>
</tr>
<tr>
<td>$t_s$</td>
<td>0.350</td>
<td>0.186</td>
<td>0.205</td>
<td>0.145</td>
</tr>
<tr>
<td>$t_{i,c}$</td>
<td>0.340</td>
<td>0.268</td>
<td>0.461</td>
<td>0.319</td>
</tr>
<tr>
<td>$t_i$</td>
<td>0.330</td>
<td>0.161</td>
<td>0.229</td>
<td>0.095</td>
</tr>
<tr>
<td>$t_m$</td>
<td>0.16</td>
<td>0.161</td>
<td>0.037</td>
<td>0.037</td>
</tr>
<tr>
<td>$t_r$</td>
<td>0.005</td>
<td>0.010</td>
<td>0.010</td>
<td>0.010</td>
</tr>
</tbody>
</table>

$t_c$: 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 appear surprisingly globally more efficient than LH*.
  - No explanation at present.
Conclusion

- **SDDS-2000**: a prototype SDDS manager for Windows multicorputer
  - 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.