B-trees, shadowing, and clones

Article *in* ACM Transactions on Storage · February 2008

DOI: 10.1145/1326542.1326544 · Source: DBLP

Ohad Rodeh
Ultima Genomics

B-trees, Shadowing, and Clones

Ohad Rodeh

Talk outline

- Preface
- Basics of getting b-trees to work with shadowing
- Performance results
- Algorithms for cloning (writable-snapshots)

Introduction

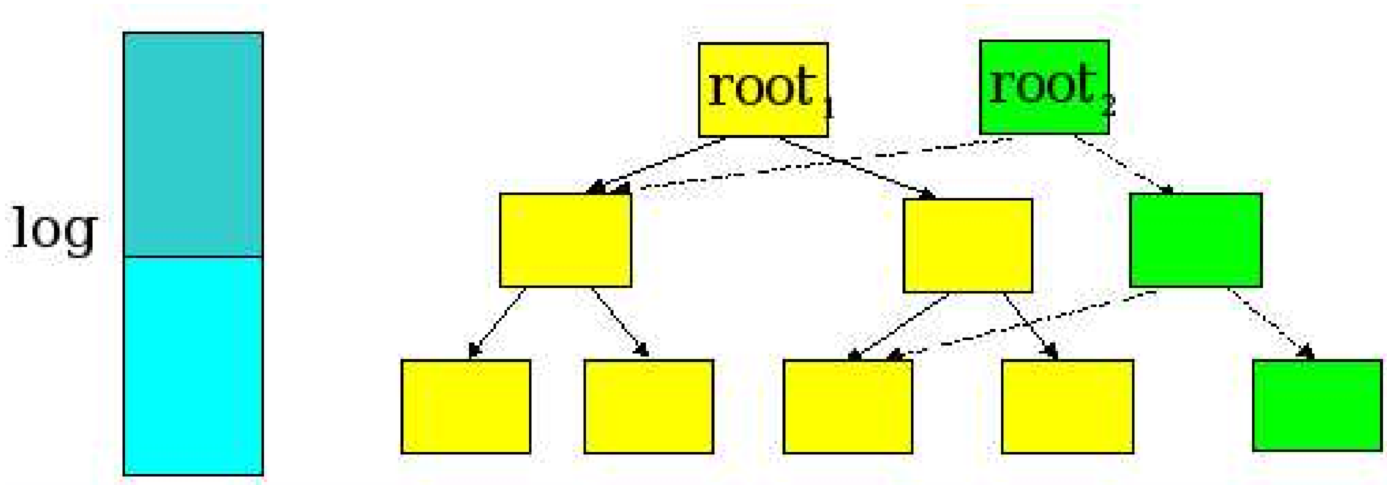
- The talk is about a free technique useful for file-systems like ZFS and WAFL
- It is appropriate for this forum due to the talked about port of ZFS to Linux
- The ideas described here were used in a research prototype of an object-disk.
- A b-tree was used for the OSD catalog (directory), an extent-tree was used for objects (files).

Shadowing

- Some file-systems use shadowing: WAFL, ZFS, . . .
- Basic idea:
  1. File-system is a tree of fixed-sized pages
  2. Never overwrite a page
- For every command:
  1. Write a short logical log entry
  2. Perform the operation on pages written off to the side
  3. Perform a checkpoint once in a while

Shadowing II

- In case of crash: revert to previous stable checkpoint and replay the log
- Shadowing is used for: Snapshots, crash-recovery, write-batching, RAID



Shadowing III

- Important optimizations
  1. Once a page is shadowed, it does not need to be shadowed again until the next checkpoint
  2. Batch dirty-pages and write them sequentially to disk

Snapshots

- Taking snapshots is easy with shadowing
- In order to create a snapshot:
  1. The file-system allows more than a single root
  2. A checkpoint is taken but not erased

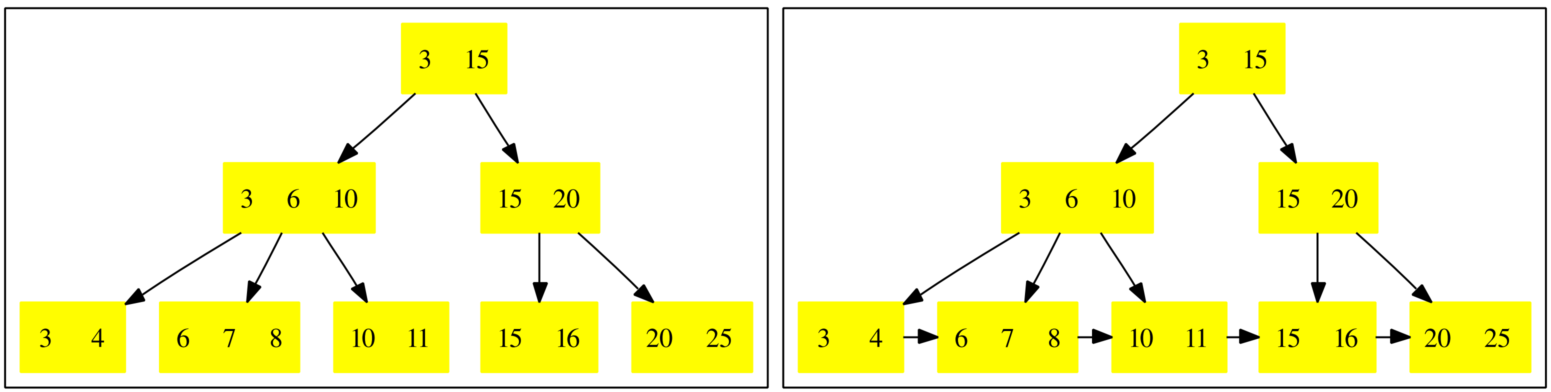
B-trees

- B-trees are used by many file-systems to represent files and directories: XFS, JFS, ReiserFS, SAN.FS
- They guarantee logarithmic-time key-search, insert, remove
- The main questions:
  1. *Can we use B-trees to represent files and directories in conjunction with shadowing?*
  2. *Can we get good concurrency?*
  3. *Can we supports lots of clones efficiently?*

Challenges

- Challenge to multi-threading: changes propagate up to the root. The root is a contention point.
- In a regular b-tree the leaves can be linked to their neighbors.

  Not possible in conjunction with shadowing
- Throws out a lot of the existing b-tree literature



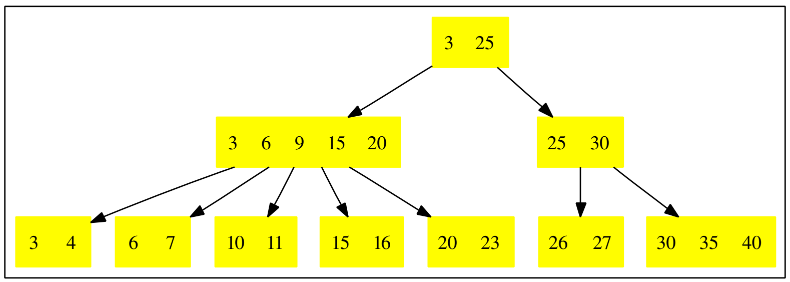
Write-in-place b-tree

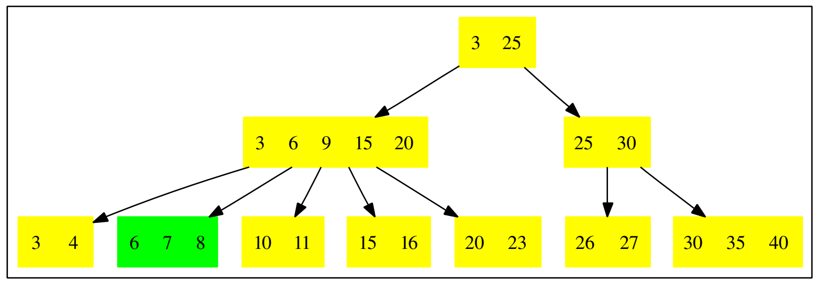
- Used by DB2 and SAN.FS
- Updates b-trees in place; no shadowing
- Uses a bottom up update procedure

Write-in-place example

- Insert-element
  1. Walk down the tree until reaching the leaf L
  2. If there is room: insert in L
  3. If there isn't, split and propagate upward

  Note: tree nodes contain between 2 and 5 elements





Alternate shadowing approach

- Used in many databases, for example, Microsoft SQL server.
- Pages have virtual addresses
- There is a table that maps virtual addresses to physical addresses
- In order to modify page P at address L_1
  1. Copy P to another physical address L_2
  2. Modify the mapping table, $P \rightarrow L_2$
  3. Modify the page at the L_2

Alternate shadowing approach II

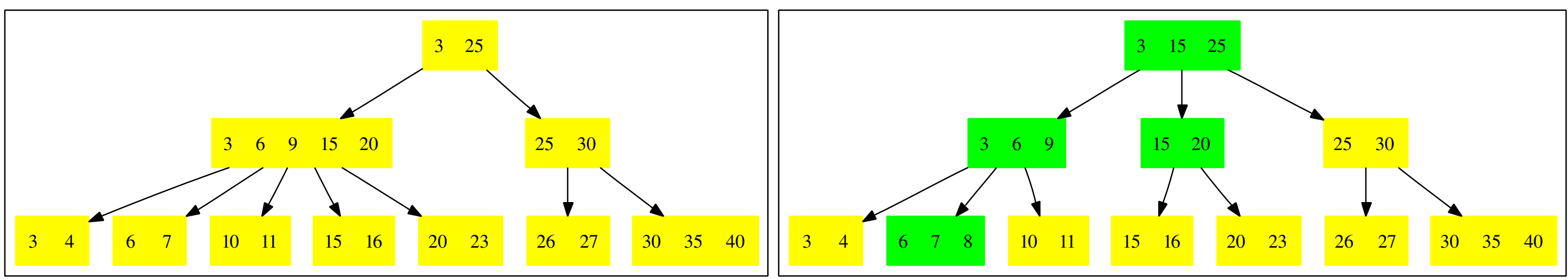
- Pros
  - Avoids the ripple effect of shadowing
  - Uses b-link trees, very good concurrency
- Cons
  - Requires an additional persistent data structure
  - Performance of accessing the map is crucial to good behavior

Requirements from shadowed b-tree

- The b-tree has to:
  1. Have good concurrency
  2. Work well with shadowing
  3. Use deadlock avoidance
  4. Have guarantied bounds on space and memory usage per operation
- Tree has to be balanced

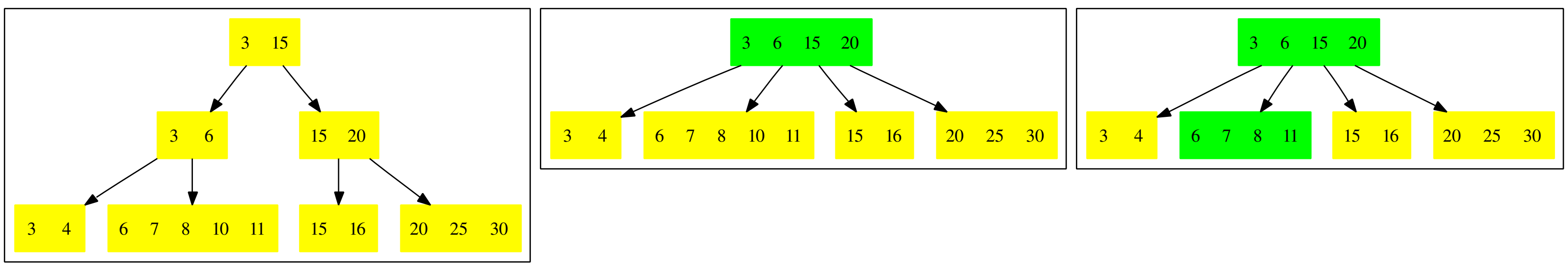
The solution: insert-key

- Top-down
- Lock-coupling for concurrency
- Proactive split
- Shadowing on the way down
- Insert element 8
  1. Causes a split to node [3,6,9,15,20]
  2. Inserts into [6,7]



Remove-key

- Top-down
- Lock-coupling for concurrency
- Proactive merge/shuffle
- Shadowing on the way down
- Example: remove element 10



Analysis for Insert/Remove-key

- Always hold two/three locks
- At most three pages held at any time
- Modify a single path in the tree

Pros/Cons

- Cons:
  - Effectively lose two keys per node due to proactive split/merge policy
  - Need loose bounds on number of entries per node $(b \ldots 3b)$
- Pros:
  - No deadlocks, no need for deadlock detection/avoidance
  - Relatively simple algorithms, adaptable for controllers

Cloning

- To clone a b-tree means to create a writable copy of it that allows all operations: lookup, insert, remove, and delete.
- A cloning algorithm has several desirable properties

Cloning properties

- Assume p is a b-tree and q is a clone of p, then:
  1. Space efficiency: p and q should, as much as possible, share common pages
  2. Speed: creating q from p should take little time and overhead
  3. Number of clones: it should be possible to clone p many times
  4. Clones as first class citizens: it should be possible to clone q

Cloning, a naive solution

- A trivial algorithm for cloning a tree is copying it wholesale.
- This does not have the above four properties.

WAFL free-space

- There are deficiencies in the classic WAFL free space
- A bit is used to represent each clone
- With a map of 32-bits per data block we get 32 clones
- To support 256 clones, 32 bytes are needed per data-block.
- In order to clone a volume or delete a clone we need to make a pass on the entire free-space

Challenges

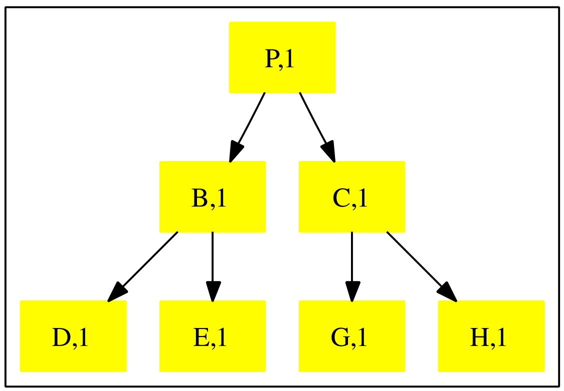
- *How do we support a million clones without a huge free-space map?*
- *How do we avoid making a pass on the entire free-space?*

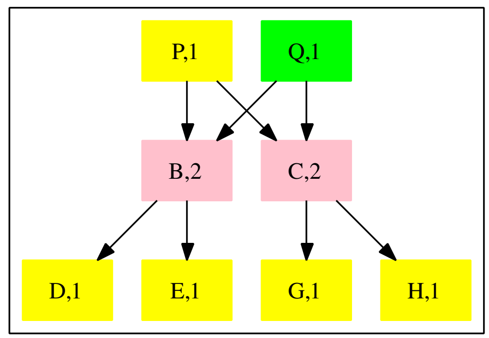
Main idea

- Modify the free space so it will keep a reference count (ref-count) per block
- Ref-count records how many times a page is pointed to
- A zero ref-count means that a block is free
- Essentially, instead of copying a tree, the ref-counts of all its nodes are incremented by one
- This means that all nodes belong to two trees instead of one; they are all shared
- Instead of making a pass on the entire tree and incrementing the counters during the clone operation, this is done in a lazy fashion.

Cloning a tree

1. Copy the root-node of p into a new root
2. Increment the free-space counters for each of the children of the root by one

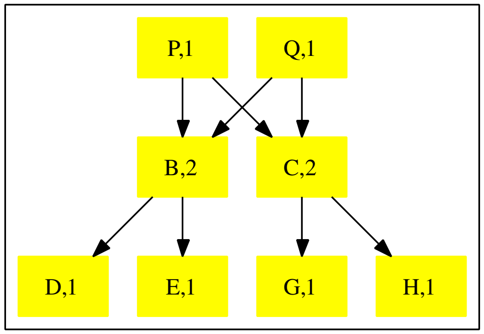




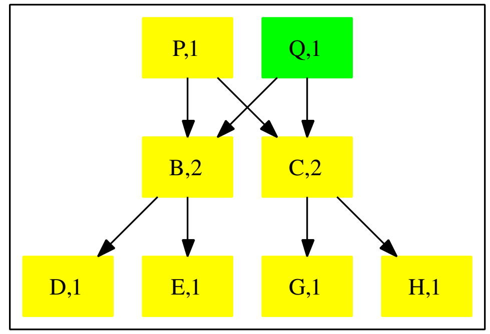
Mark-dirty, without clones

- Before modifying page N, it is “marked-dirty”
  1. Informs the run-time system that N is about to be modified
  2. Gives it a chance to shadow the page if necessary
- If ref-count == 1: page can be modified in place
- If ref-count > 1, and N is relocated from address a_1 to address a_2
  1. the ref-count for a_1 is decremented
  2. the ref-count for a_2 is made 1
  3. The ref-count of N’s children is incremented by 1

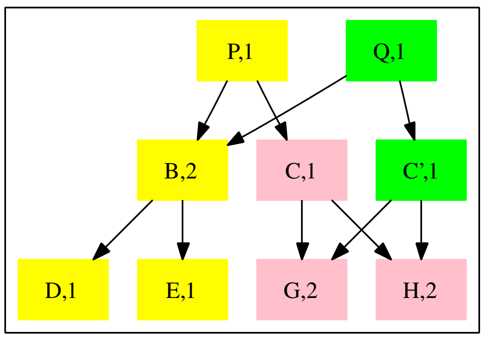
Example, insert-key into leaf H,tree q



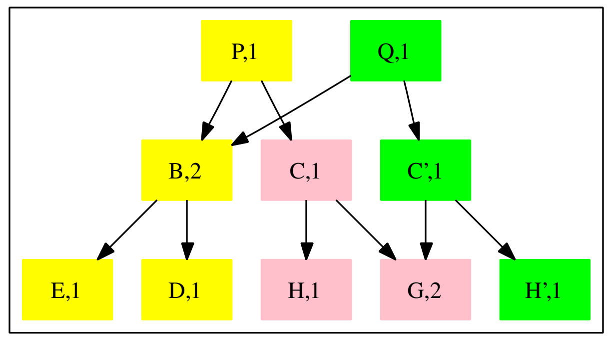
(I) Initial trees, T_p and T_q



(II) Shadow Q



(III) shadow C

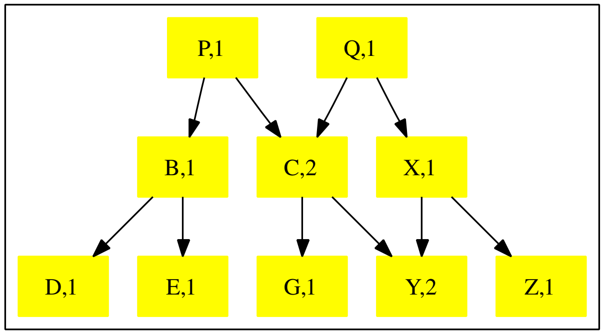


(IV) shadow H

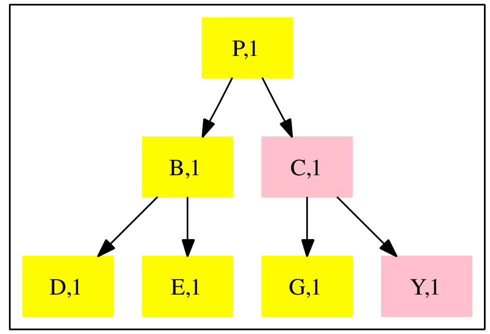
Deleting a tree

- ref-count(N) > 1: Decrement the ref-count and stop downward traversal. The node is shared with other trees.
- ref-count(N) == 1 : It belongs only to q. Continue downward traversal and on the way back up de-allocate N.

Delete example



(I) Initial trees T_p and T_q



(II) After deleting T_q

Comparison to WAFL free-space

- Clone:
  - Ref-counts: increase ref-counts for children of root
  - WAFL: make a pass on the entire free-space and set bits
- Delete clone p
  - Ref-counts: traverse the nodes that belong only to p and decrement ref-counters
  - WAFL: make a pass on the entire free-space and set snapshot bit to zero

Comparison to WAFL free-space

- During normal runtime
  - Ref-counts: additional cost of incrementing ref-counts while performing modifications
  - WAFL: none
- Space taken by free-space map
  - Ref-counts: two bytes per block allow 64K clones
  - WAFL: two bytes allow 16 clones

Resource and performance analysis

- The addition of ref-counts does not add b-tree node accesses. Worst-case estimate on memory-pages and disk-blocks used per operation is unchanged
- Concurrency remains unaffected by ref-counts
- Sharing on any node that requires modification is quickly broken and each clone gets its own version

FS counters

- The number of free-space accesses increases. Potential of significantly impacting performance.
- Several observations make this unlikely:

1. Once sharing is broken for a page and it belong to a single tree, there are no additional ref-count costs associated with it.
2. The vast majority of b-tree pages are leaves. Leaves have no children and therefore do not incur additional overhead.

FS counters II

- The experimental test-bed uses in-memory free-space maps
  1. Precludes serious investigation of this issue
  2. Remains for future work

Summary

- The b-trees described here:
  - Are recoverable
  - Have good concurrency
  - Are efficient
  - Have good bounds on resource usage
  - Have a good cloning strategy

Backup slides

Performance

- In theory, top-down b-trees have a bottle-neck at the top
- In practice, this does not happen because the top nodes are cached
- In the experiments
  1. Entries are 16bytes: key=8bytes, data=8bytes
  2. A 4KB node contains 70-235 entries

Test-bed

- Single machine connected to a DS4500 through Fiber-Channel.
- Machine: dual-CPU Xeon 2.4Ghz with 2GB of memory.
- Operating System Linux-2.6.9
- The b-tree on a virtual LUN
- The LUN is a RAID1 in a 2+2 configuration
- Strip width is 64K, full stripe=512KB
- Read and write caching is turned off

Basic disk performance

- IO-size=4KB
- Disk-area=1GB

#threads	op.	time per op.(ms)	ops per second
10	read	N/A	1217
	write	N/A	640
	R+W	N/A	408
1	read	3.9	256
	write	16.8	59
	R+W	16.9	59

Test methodology

- The ratio of cache-size to number of b-tree pages
  1. Is fixed at initialization time
  2. This ratio is called the in-memory percentage
- Various trees were used, with the same results. The experiments reported here are for tree T_{235} .

Tree T_{235}

- Maximal fanout: 235
- Legal #entries: 78 .. 235
- Contains 9.5 million keys and 65500 nodes
  1. 65000 leaves
  2. 500 index-nodes
- Tree depth is: 4
- Average node capacity   150 keys

Test methodology II

- Create a large tree using random operations
- For each test
  1. Clone the tree
  2. Age the clone by doing 1000 random insert-key/remove-key operations
  3. Perform $10^4 – 10^8$ measurements against the clone with random keys
  4. Delete the clone
- Perform each measurement 5 times, and average.
- The standard deviation was less than 1% in all tests.

Latency measurements

- Four operations whose latency was measured: lookup-key, insert-key, remove-key, append-key.
- Latency measured in milliseconds

Lookup	Insert	Remove	Append
3.43	16.76	16.46	0.007

Different in-memory ratios

- Workload: 100% lookup workload

% in-memory	1 thread	10 threads	ideal
100	14237	19805	∞
50	391	1842	2434
25	321	1508	1622
10	268	1290	1352
5	254	1210	1281
2	250	1145	1241

Throughput

- Four workloads were used:
  1. Search-100: 100% lookup
  2. Search-80: 80% lookup, 10% insert, 10% remove
  3. Modify: 20% lookup, 40% insert, 40% remove
  4. Insert: 100% insert
- Metric: operations per second
- Since there isn't much difference between 2% in memory and 50%, the rest of the experiments were done using 5%.
- Allows putting all index nodes in memory.

Throughput II

Tree	#threads	Src-100	Src-80	Modify	Insert
T_{235}	10	1227	748	455	400
	1	272	144	75	62
Ideal		1281			429

Workload with some locality

- Workload: randomly choose a key
  - With 80% probably read the next 100 keys after it
  - With 10% probability, insert/overwrite the next 100 keys
  - With 10% probability, remove the next 100 keys

#threads	semi-local
10	16634
1	3848

Clone performance

- Two clones are made of base tree T_{235}
- Aging is performed
  1. 12000 operations are performed
  2. 6000 against each clone

	Src-100	Src-80	Modify	Insert
2 clones	1221	663	393	343
base	1227	748	455	400

Clone performance at 100% in-memory

	Src-100	Src-80	Modify	Insert
2 threads	20395	18524	16907	16505
1 thread	13910	12670	11452	11112

Performance of checkpointing

- A checkpoint is taken during the throughput test
- Performance degrades
  1. A dirty page has to be destaged prior to being modified
  2. Caching of dirty-pages is effected

Performance of checkpointing II

Tree T_{235}	Src-100	Src-80	Modify	Insert
checkpoint	1205	689	403	353
base	1227	748	455	400