Journal File Systems

(Modern File System)

Juan I. Santos Florido

Linux Gazette

Issue 55

July 2000

http://www.linuxgazette.com/issue55/florido.html

Introduction (1 of 2)

- Linux increasingly heterogeneous, so taking on features to satisfy other environments
  - micro-controllers, routers, 3-D hardware speedup, multi-head Xfree, games, window managers…
- Huge step forward for Linux server needs
  - Getting most important commercial UNIX and large server's features

→ Support for server-quality file systems

Introduction (2 of 2)

- Linux servers must …
  - Deal with large hard-disk partitions
  - Scale up easily with thousands of files
  - Recover quickly from crash
  - Increase I/O performance
  - Behave well with both small and large files
  - Decrease internal and external fragmentation
- This article introduces basics of *Journal File Systems:*
  - Examples: JFS, XFS, Ext3FS, and ReiserFS

Outline

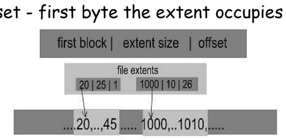
- Introduction (done)
- Glossary (next)
- Problems
  - System crashes
  - Scalability
  - Dir entries
  - Free blocks
  - Large files
- Other Enhancements
- Summary

Glossary

- Internal Fragmentation
  - Allocated but unused
  - (Same as in memory)
- External Fragmentation
  - Spread out blocks create slowdown
  - (How is this different than in memory?)
- Extents (next)
- B+ Tree (next, next)

Extents

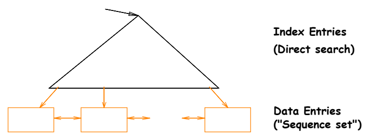
- Sets of contiguous logical blocks
  - Beginning - block addr where extent begins
  - Size - size in blocks
  - Offset - first byte the extent occupies



- Benefits
  - Enhance spatial locality, reducing external frag, having better scan times, since more blocks kept spatially together
  - Improve multi-sector transfer chances and reduce hard disk cache misses

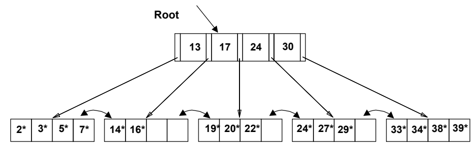
B+Tree

- Heavily used in databases for data indexing
- Insert, delete, search all $O(\log_F N)$
  - F = fanout, N = # leaves
  - tree is *height-balanced.*
- Minimum 50% occupancy (except for root). Each node contains d <= *m* <= 2d entries
  - d is called the *order* of the tree
  - Typically d = (½ pagesize) / (entry size)



Example B+Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5* or 15* ...



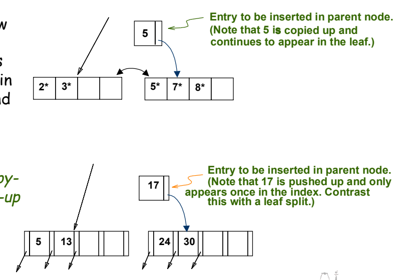
✎*Based on the search for 15*, we know it is not in the tree!*

Inserting a Data Entry into a B+Tree

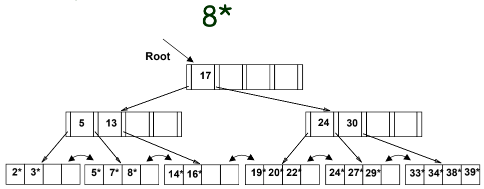
- Find correct leaf *L.*
- Put data entry onto *L.*
  - If *L* has enough space, *done*!
  - Else, must split *L (into L and a new node L2)*
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Inserting 8* into Example B+Tree

- Observe how minimum occupancy is guaranteed in both leaf and index splits.
- Note difference between *copy-up* and *push-up*



Example B+Tree After Inserting 8*

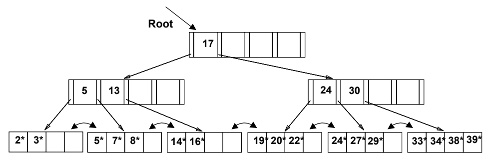


- Notice root split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Deleting a Data Entry from a B+Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L).*
    - If re-distribution fails, *merge L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L.*
- Merge could propagate to root, decreasing height.

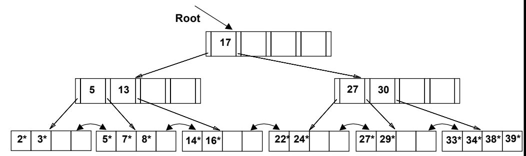
Deleting 19* and then 20*



Deletion of 19* → leaf node is not below the minimum number of entries after the deletion of 19*. No re-adjustments needed.

Deletion of 20* →leaf node falls below minimum number of entries
- re-distribute entries
- copy-up low key value of the second node

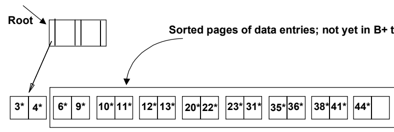
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...



- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

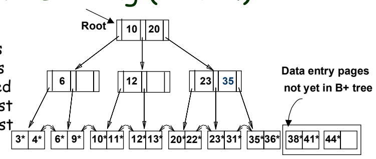
Bulk Loading of a B+ Tree

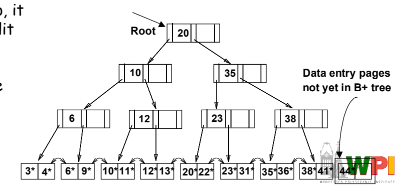
- Large collection of records, and we want to create a B+ tree
  - could do repeated insert, but slow
- *Bulk Loading* more efficient
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)





Summary of Bulk Loading

- Option 1: *multiple inserts*
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control "fill factor" on pages.

B+ Trees in Practice (db)

- Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 records
  - Height 4: 133^4 = 312,900,700 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 MByte
  - Level 3 = 17,689 pages = 133 MBytes

Outline

- Introduction (done)
- Glossary (done)
- Problems
  - System crashes (next)
  - Scalability
  - Dir entries
  - Free blocks
  - Large files
- Other Enhancements
- Summary

Problem : System Crashes

- Memory cache to improve disk performance
- System crash causes inconsistent state
  - Example: block added to i-node but not flushed to disk
- Upon reboot, must repair whole file system
- Problematic for systems with 100 Gigabytes or even Terabytes
- Solution? → Journaling

The Journal : How it Works

- *Atomicity* - All operations in transaction are:
  - *completed* without errors, or …
  - *cancelled*, producing no changes
- Log every operation to log file
  - Operation name
  - Before and after values
- Every transaction has commit operation
  - Write buffers to disk
- System crash?
  - Trace log to previous commit statement
  - Writing values back to disk
- Note, unlike databases, file systems tend to log metadata only
  - i-nodes, free block maps, i-nodes maps, etc.

Transaction : Example

- Record action plus old and new values

	Log	Log	Log

x = 0;
y = 0;
BEGIN_TRANSACTION;
x = x + 1;
y = y + 2
x = y * y;
END_TRANSACTION;
	[x = 0 / 1]	[x = 0 / 1]
		[y = 0/2]
			[x = 0 / 1]
			[y = 0/2]
			[x = 1/4]

(a)	(b)	(c)	(d)

a) A transaction

b) - d) Record to log before each statement is executed

- If transaction commits, nothing to do
- If transaction is aborted, use log to *rollback*

Problem : Scalability

- UNIX File Systems (ext2fs) for smaller hard disks
- Disks growing in capacity
  - Leads to bigger files, directories and partitions
- File system structures have fixed bits to store file size info, logical block number, etc.
  - Thus, file sizes, partition sizes and the number of directory entries are limited
- Even if can manage sizes, performance suffers

Solution : Scalability
Solving the Size Limitations

	Max filesys size	Block sizes	Max. file size
XFS	18 thousand petabytes	512 bytes to 64KB	9 thousand petabytes
JFS	512 bytes blocks / 4 petabytes	512, 1024, 2048, 4096 bytes	512 Tb with 512 bytes blocks
	4KB blocks / 32 petabytes		4 petabytes with 4KB blocks
Reiser	4GB of blocks, 16 Tb	Up to 64KB Currently fixed 4KB	4GB, 2^{10} petabytes in ReiserFS (3.6.xx)
Ext3	4Tb	1KB-4KB	2GB

Problem : Obtaining Free Blocks

- UFS and ext2fs use bitmap,
  - As file system grows, bitmap grows
  - Sequential scan for free blocks results in performance decrease (O(num_blocks))
    - (Notice not bad for moderate size file system!)
- **Solution?** → Use extents and/or B+Trees

Solution : Obtaining Free Blocks

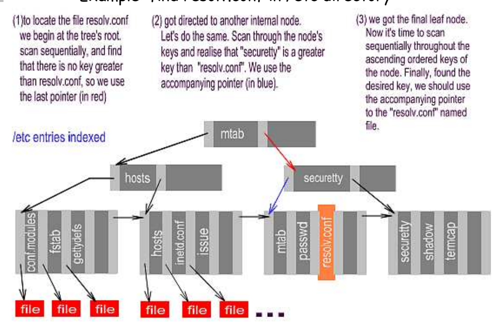
- Extents
  - Locate several free blocks at once, avoiding multiple searches
  - Reduce the structure's size, since more logical blocks are tracked with less than a bit for each block
  - Free block's structure size no longer depends on the file system size (structure size would depend on the number of extents maintained)
- B+Trees
  - Organize free blocks in *B+tree* instead of lists
- B+Trees **and** Extents
  - Organize extents in B+tree
  - Indexing by extent *size* also by extent *position*

Problem: Large Number of Directory Entries

- Directories entries are pairs (i-node, file name).
- To find file, traverse the directory entries directory into a list
  - Sequential scan for free blocks results in performance decrease (O(num_entries))
    - (Notice not bad for moderate size file system!)
- Solution? → B+Trees for directory entries
  - Some have B+Trees for each dir, while others have B+Tree for the whole file system directory tree.

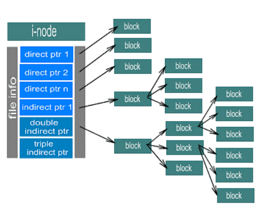
Example: B+Tree for Dir Entries

Example: find resolv.conf in /etc directory



Problem : Large Files

- Ext2fs and UFS were designed with the idea that the file systems would contain small files mainly.
  - Large files use more indirect pointers, so a higher more disk
- Solution? → B+Trees and modified i-nodes



Solution : Large Files

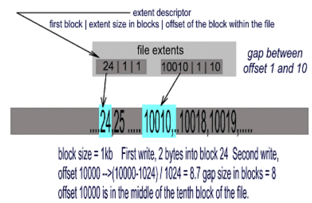
- i-node use for small files
  - Direct block pointers
  - Or even data in i-node (good for symbolic links)
- B+Trees to organize the file blocks for larger files
  - indexed by the offset within the file; then, when a certain offset within the file is requested, the file system routines would traverse the B+Tree to locate the block required.

Outline

- Introduction (done)
- Glossary (done)
- Problems
  - System crashes (done)
  - Scalability (done)
  - Dir entries (done)
  - Free blocks (done)
  - Large files (done)
- Other Enhancements (next)
- Summary

Other Enhancements : Sparse Files

- Support for sparse files
  - New file, write 2 bytes, then write offset 10000. Would need blocks to cover gap.
  - Solution? → Extents



Return "null" if read between extents

Other Enhancements : Internal Fragmentation Solution

- Large blocks == large int. fragmentation
- Small blocks == more disk I/O
- Solution? → Leaf nodes of B+Tree can have data itself
  - Allow multiple file tails to be allocated together
  - However, can increase external fragmentation
  - Made an option for system administrators

Other Enhancements : Dynamic I-node Allocation

- Typically on UFS, fixed number of i-nodes
  - Created during disk format
  - Can run out, even if disk space left!
- Solution? → Dynamic allocation of i-nodes
  - Allocate i-nodes as needed
  - Need data structures to keep track of
    - Store allocated i-nodes in B+Tree
  - Access a bit slower, since no direct table
- Overall, dynamic i-nodes more complex and time consuming, but help broaden the file system limits

Modern File System Summary

- Built to operate on today's large disks
- Journaling to reduce costs of "fixing" disks upon system crash
- B+Trees and Extents to improve performance for large file systems
- Misc other system features in some
  - Sparse file support
  - Combat internal fragmentation
  - Dynamic i-nodes

Future Work

- Performance
  - Read/Write
  - Search operations
  - Directoriy operations
- Robustness