Filesystems, Snapshots, and Balanced Search

October 19, 2014

1 What is a filesystem really?

Filesystems keep track of files. Let's take a look at what happens when you run this line of code:

```
int fd = open("cat.jpg", O_RDONLY);
```

- C compiler → AST → IR → assembly:

```
.LC0:
.string "cat.jpg"
main:
movl    $.LC0, %edi
movl    $0, %eax
call    open
```

- Assembly → machine code object file → linker → loader → `libc`:

```
asm volatile (
    movl $2,    %eax   ; System call code.
    movq $.LC0, %rdi   ; Load the file path.
    movl $0,    %esi   ; Load the file mode.
    syscall            ; Software interrupt.
    movq %rax,  ...    ; Store the return code.
);
```

- `libc` → software interrupt → hardware interrupt controller → interrupt descriptor table → kernel interrupt service routine → system call table → `linux/fs/open.c:992`:

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;

        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

- System call handler → kernel virtual filesystem switch → `ext4` → `lookup()` → directory entry cache → inode cache → ACLs and permission checking → `ext4_file_open` → journal device → buffer cache → disk quota handler → inode space reservation → process file descriptor table → scheduler → context switch → program:

```
    fd = 3;
```

  Once you have a file descriptor you can `read`, `write`, `link`, `unlink`, `rename`, `stat`, `close` files (etc.) in a similar manner. The filesystem lets you do this safely and efficiently.

  We skipped steps, but that's the general idea. You may be wondering how the hardware knows to call the kernel when it experiences a software interrupt. Well, the kernel loads its own interrupt descriptor table (and an accompanying set of interrupt service routines) using the programmable interrupt controller during boot time. See the `lidt` instruction (`linux/arch/x86/realmode/rm/trampoline_64.S:67`).

2 What are snapshots?

Over time, filesystems undergo changes. A snapshot captures the state of a filesystem at a particular moment in time. *Writeable snapshots* are analogous to git branches: you branch the state of your disk (`git branch`), update files (`git add, commit`), and go back in time (`git checkout`) as you like. Atomic filesystem snapshots are useful for the same reason git branches are useful: they provide safety (you can always roll back to a consistent state).

Some filesystems support *continuous snapshotting* (e.g `nilfs2`). Instead of explicitly taking snapshots (branching your repository), every filesystem update is appended to a log (every character you type is stored in a list of deltas). It turns out that this can be fast, space-economical, and SSD-friendly. This is done by replacing the common {journal, in-place tree} combination with a log: the filesystem *is* the log.

3 How do you implement snapshots?

Let's take a simplified view of a filesystem as a map: File → Metadata [1]. A snapshot is just a copy of this map. Continuing the git analogy, you start off with a `master` branch when you create a filesystem (repository). All filesystem updates (commits) are applied to the current snapshot map (working tree).

When you create a snapshot, you don't need to copy the entire File → Metadata map from the parent snapshot. In fact, you don't need to copy anything at all. You only need to store the *changes* between current snapshot and the parent snapshot in the current snapshot map. This is called *copy-on-write* [2].

So in order to implement snapshots, we need a fast, space-efficient, and highly-concurrent associative container which supports copy-on-write.

4 Balanced search structures

When evaluating search structures, keep in mind that they operate on *block devices*. RAM has consistently fast byte-addressing (modulo cache and alignment effects). Block devices don't: they're block-addressed and slow.

- AVL trees (no one). $\Theta(\log_2 n)$ search, insertion, and deletion. Balancing via L-R and R-L rotations. Linear space requirement. Pros: simple. Cons: does not minimize disk accesses. If $n = 2^{30}$ (small), that's awful.

- B+ trees (SQLite, NTFS, XFS). Contains up to B keys per node, up to $B + 1$ children per internal node (binary search each node to walk down the tree). Internal nodes contain only keys. Leaf nodes contain keys, values, and forward pointers for range queries. Balancing via requiring at least $\lceil b/2 \rceil$ children for internal nodes and at least $\lfloor b/2 \rfloor$ values for leaves.

  Assume a top-down rebalancing scheme. Roughly, at level i, after inserting B^i items we rebalance each node once. The rebalance cost is $\Theta(B)$ per node, and the cost of not rebalancing is $\Theta(1)$. Therefore search, insertion, and deletion is $\Theta(\log_B n)$ amortized. Linear space requirement.

  Pros: provides fan-out, tunable for disk block sizes, supports efficient range queries and point queries. Cons: complicated rebalancing, complicated locking for concurrent updates, not write-optimal.

- Log-structured merge trees (LevelDB, Cassandra, HBase). Contains one in-memory level (C0, capacity 2^l) and several on-disk levels (C1, C2, ...). C0 is usually some logarithmic-time container [3]. C1 has 2 slots, each with capacity 2^l. The slot sizes of each subsequent on-disk component double.

  Insertions and deletions walk through the levels until a non-full level is reached. Searches walk through the levels until the key is found. Levels are periodically merged: there are minor compactions (background optimization) and major compactions (forced by an update). When compacting two slots, you will frequently encounter large, contiguous, mutually disjoint subsets of (k, v) tuples. Instead of reading and then re-writing these ranges, we just add *pointers* to these ranges in the merged slot: this optimization is called *stitching*.

  As specified, searches must binary search each slot in each level until the key is found. If it isn't found, we have to scan the entire tree (prohibitively slow). We can attach approximate membership query structures to each slot to speed searches up: these structures have no false negatives, but they do sometimes have false positives. They are small enough to fit in RAM.

[1] 'Metadata' includes information like the location of the file on disk, permissions, modification times, etc.
[2] We're ignoring important considerations such as designing the space-allocator, block references, fragmentation, etc.
[3] Typically, C0 is implemented as a skiplist. Skiplists support range searches and can be made lock-free.

5 Quotient filters

Bloom filters are a popular AMQ structure. They call for k, m-bit hash functions: $H_i : Z \to Z \pmod{2^m}$. You also have a bit-array A of size 2^m. To insert a key x into the filter, you set $A[H_i(x)] = 1$ for all $1 \leq i \leq k$. To check if x is in the filter, just check that $A[H_i(x)] = 1$ for all i. Lookups produce no false negatives, but there are false positives. It's impossible to delete keys from a Bloom filter without introducing false negatives. It's impossible to merge two Bloom filters together without causing the false positive rate to explode, or rehashing all of the keys.

Quotient filters fix a few of the problems Bloom filters have. They support deletions and merging (without rehashing), require fewer hash functions, and can sometimes perform better (both in terms of speed and the false positive rate). Attaching Bloom filters to slots in an LSM tree would make compaction expensive, since every key must be fetched from disk and rehashed. With quotient filters, we can merge filters directly without any rehashing: this significantly reduces the amortized I/O cost of LSM trees.

Quotient filters are parameterized by q and r, the size in bits of the quotient and remainder of the key hashes (respectively). There are 2^q $(r+3)$-bit entries in the quotient filter. To understand how it works, imagine a chained hash table with q-bit quotients as keys and r-bit remainders as values. The insertion, deletion, and lookup semantics of quotient filters follow from this picture: simply compact the structure into as few bits as possible. The important thing to keep in mind is that inserting into a quotient filter can shift existing runs of elements. While the average case insertion time is low (if the filter load factor is low), the worst case insertion time is linear in the size of the filter. Bloom filters do not have this problem because insertion is guaranteed to be constant time.

6 A first attempt at analyzing LSM trees

Let n be the total number of searches. Assume our quotient filters have false positive rate ϵ and that the overall probability of finding a key in a slot with capacity c is c/n. In reality the probability may be much lower than c/n because not all keys are guaranteed to be in the tree. At any rate, the search cost for an LSM tree with C0 and C1 is then $\Theta(1) + 2l[\frac{2^l}{n} + (1 - \frac{2^l}{n})\epsilon]$. For a k-level tree, we have:

$$\Theta(1) + O\left(\sum_{i=0}^{k-1} 2(l+i)\left[\frac{2^{l+i}}{n} + \left(1 - \frac{2^{l+i}}{n}\right)\epsilon\right]\right)$$

I'm not sure how to estimate insertion / deletion costs, since I don't have a good way to model the effects of compaction and stitching. Thoughts?

7 Further filesystem-related topics

1. Reducing write-amplification.
2. Block-level data deduplication.
3. Block-level compression.
4. Hashing, Merkle-trees, checksumming.
5. RAID: redundancy and speed.
6. Defragmentation, or garbage collection.
7. Copy-on-write files.
8. Full-disk encryption.