bup: the git-based backup system

Avery Pennarun

2011 04 30

The Challenge

- Back up entire filesystems (> 1TB)
- Including huge VM disk images (files >100GB)
- Lots of separate files (500k or more)
- Calculate/store incrementals efficiently
- Create backups in O(n), where n = number of changed bytes
- Incremental backup direct to a remote computer (no local copy)
- ...and don't forget metadata

The Result

- bup is extremely fast - **~80 megs/sec** in python
- Sub-file incrementals are very space efficient
  - **>5x better than rsnapshot** in real-life use
- VMs compress **smaller and faster** than gzip
- Dedupe between **different client machines**
- O(log N) seek times to any part of any file
- You can mount your backup history as a filesystem or browse it as a web page

The Design

Why git?

- Easily handles file renames
- Easily deduplicates between identical files and trees
- Debug my code using git commands
- Someone already thought about the repo format (packs, idxes)
- Three-level “work tree” vs “index” vs “commit”

Problem 1: Large files

- 'git gc' explodes badly on large files; totally unusable
- git bigfiles fork “solves” the problem by just never deltifying large objects: lame
- zlib window size is very small: lousy compression on VM images --

Digression: zlib window size

- gzip only does two things:
  - backref: copy some bytes from the preceding 64k window
  - huffman code: a dictionary of common words
- That 64k window is a serious problem!
- Duplicated data >64k apart can't be compressed
- cat file.tar file.tar | gzip -c | wc -c
  - surprisingly, twice the size of a single tar.gz

bupsplit

- Uses a rolling checksum to --

Digression: rolling checksums

- Popularized by rsync (Andrew Tridgell, the Samba guy)
- He wrote a (readable) Ph.D. thesis about it
- bup uses a variant of the rsync algorithm to --

Double Digression: rsync algorithm

- First player:
  - Divide the file into fixed-size chunks
  - Send the list of all chunk checksums
- Second player:
  - Look through existing files for any blocks that have those checksums
  - But *any* n-byte subsequence might be the match
  - Searching naively is about O(n^2) ... ouch.
  - So we use a rolling checksum instead

Digression: rolling checksums

- Calculate the checksum of bytes 0..n
- Remove byte 0, add byte n+1, to get the checksum from 1..n+1
  - And so on
- Searching is now more like O(n)... vastly faster
- Requires a special “rollable” checksum (adler32)

Digression: gzip --rsyncable

- You can't rsync gzipped files efficiently
- Changing a byte early in a file changes the compression dictionary, so the rest of the file is different
- --rsyncable resets the compression dictionary whenever low bits of adler32 == 0
- Fraction of a percent overhead on file size
- But now your gzip files are rsyncable!

bupsplit

- Based on gzip --rsyncable
- Instead of a compression dictionary, we break the file into blocks on adler32 boundaries
  - If low 13 checksum bits are 1, end this chunk
  - Average chunk: 2**13 = 8192
- Now we have a list of chunks and their sums
- **Inserting/deleting bytes changes at most two chunks!**

bupsplit trees

- Inspired by “Tiger tree” hashing used in some P2P systems
- Arrange chunk list into trees:
  - If low 17 checksum bits are 1, end a superchunk
  - If low 21 bits are 1, end a superduperchunk
  - and so on.
- Superchunk boundary is also a chunk boundary
- **Inserting/deleting bytes changes at most 2*log(n) chunks!**

Advantages of bupsplit

- Never loads the whole file into RAM
- Compresses most VM images more (and faster) than gzip
- Works well on binary *and* text files
- Don't need to teach it about file formats
- Diff huge files in about O(log n) time
- Seek to any offset in a file in O(log n) time

Problem 2: Millions of objects

- Plain git format:
  - 1TB of data / 8k per chunk: 122 million chunks
  - x 20 bytes per SHA1: 2.4GB
  - Divided into 2GB packs:
    500 .idx files of 5MB each
  - 8-bit prefix lookup table

- Adding a new chunk means searching
  500 * (log(5MB)-8) hops
  = 500 * 14 hops
  = 500 * 7 pages = 3500 pages

Millions of objects (cont'd)

- bup .midx files: merge all .idx into a single .midx
- Larger initial lookup table to immediately narrow search to the last 7 bits
- log(2.4GB) = 31 bits
- 24-bit lookup table * 4 bytes per entry: 64 MB
- **Adding a new chunk *always* only touches two pages**

Bloom Filters

- Problems with .midx:
  - Have to rewrite the whole file to merge a new .idx
  - Storing 20 bytes per hash is wasteful; even 48 bits would be unique
  - Paging in two pages per chunk is maybe too much
- Solution: Bloom filters
  - Idea borrowed from Plan9 WORM fs
  - Create a big hash table
  - Hash each block several ways
  - Gives false positives, never false negatives
  - Can update incrementally

bup command line: indexed mode

- Saving:
  - bup index -vux /
  - bup save -vn mybackup /
- Restoring:
  - bup restore (extract multiple files)
  - bup ftp (command-line browsing)
  - bup web
  - bup fuse (mount backups as a filesystem)

Not implemented yet

- Pruning old backups
  - 'git gc' is far too simple minded to handle a 1TB backup set
- Metadata
  - Almost done: 'bup meta' and the .bupmeta files

Crazy Ideas

- Encryption
- 'bup restore' that updates the index like 'git checkout' does
- Distributed filesystem: bittorrent with a smarter data structure

Questions?