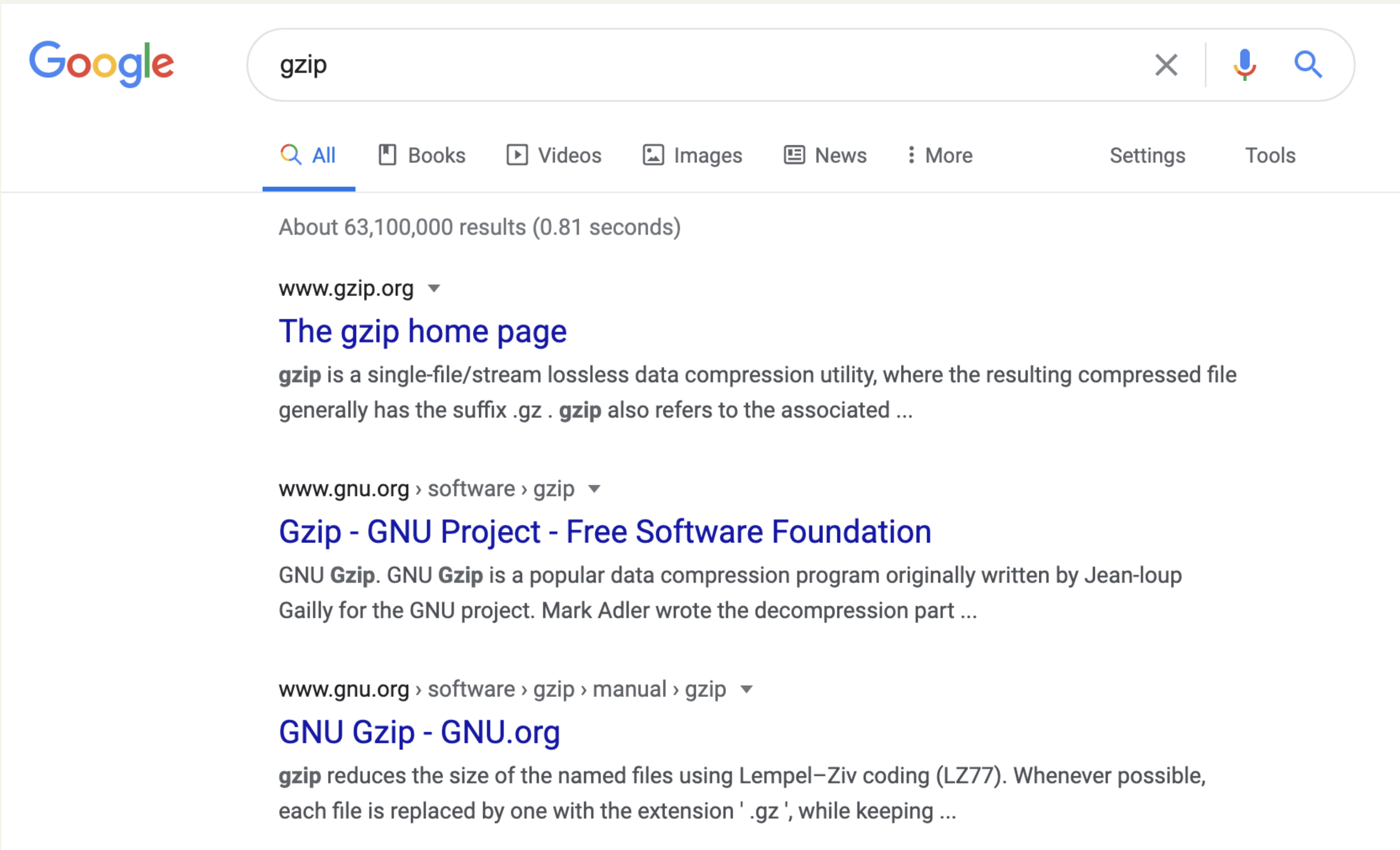
Compression of documents

Paolo Ferragina
Dipartimento di Informatica
Università di Pisa

Raw docs are needed



snappy

A fast compressor/decompressor

Project Information

+76 Recommend this on Google

Project feeds

Code license
New BSD License

Labels
Google, Compression

Members
se...@google.com
1 committer

Featured

Downloads
snappy-1.0.5.tar.gz
Show all »

Links

Groups
General Snappy discussion

Snappy is a compression/decompression library. It does not aim for maximum compression, or compatibility with any other compression library; instead, it aims for very high speeds and reasonable compression. For instance, compared to the fastest mode of zlib, Snappy is an order of magnitude faster for most inputs, but the resulting compressed files are anywhere from 20% to 100% bigger. On a single core of a Core i7 processor in 64-bit mode, Snappy compresses at about 250 MB/sec or more and decompresses at about 500 MB/sec or more.

Snappy is widely used inside Google, in everything from BigTable and MapReduce to our internal RPC systems. (Snappy has previously been referred to as "Zippy" in some presentations and the likes.)

For more information, please see the README. Benchmarks against a few other compression libraries (zlib, LZO, LZF, FastLZ, and QuickLZ) are included in the source code distribution. The source code also contains a formal format specification, as well as a specification for a framing format useful for higher-level framing and encapsulation of Snappy data, e.g. for transporting Snappy-compressed data across HTTP in a streaming fashion. Note that there is currently no known code implementing the latter.

Snappy is written in C++, but C bindings are included, and several bindings to other languages are maintained by third parties:

- C89 port
- Common Lisp
- Erlang: esnappy, snappy-erlang-nif
- Go
- Haskell
- Java: JNI wrapper, native reimplementation
- Node.js
- Perl
- PHP
- Python
- Ruby

If you know of more, do not hesitate to let us know. The easiest way to get in touch is via the Snappy discussion mailing list.

Introducing Brotli: a new compression algorithm for the internet

Posted: Tuesday, September 22, 2015

At Google, we think that internet users' time is valuable, and that they shouldn't have to wait long for a web page to load. Because fast is better than slow, two years ago we published the Zopfli compression algorithm. This received such positive feedback in the industry that it has been integrated into many compression solutions, ranging from PNG optimizers to preprocessing web content. Based on its use and other modern compression needs, such as web font compression, today we are excited to announce that we have developed and open sourced a new algorithm, the Brotli compression algorithm.

While Zopfli is Deflate-compatible, Brotli is a whole new data format. This new format allows us to get 20–26% higher compression ratios over Zopfli. In our study '*Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms*' we show that Brotli is roughly as fast as zlib's Deflate implementation. At the same time, it compresses slightly more densely than LZMA and bzip2 on the Canterbury corpus. The higher data density is achieved by a 2nd order context modeling, re-use of entropy codes, larger memory window of past data and joint distribution codes. Just like Zopfli, the new algorithm is named after Swiss bakery products. Brötli means 'small bread' in Swiss German.

The smaller compressed size allows for better space utilization and faster page loads. We hope that this format will be supported by major browsers in the near future, as the smaller compressed size would give additional benefits to mobile users, such as lower data transfer fees and reduced battery use.

By Zoltan Szabadka, Software Engineer, Compression Team

Labels

Archive

Feed

Apple Open-Sources its New Compression Algorithm LZFSE

by Sergio De Simone on Jul 02, 2016 | Discuss

Apple has open-sourced its new lossless compression algorithm, LZFSE, introduced last year with iOS 9 and OS X 10.10. According to Apple, LZFE provides the same compression gain as ZLib level 5 while being 2x–3x faster and with higher energy efficiency.

LZFSE is based on Lempel-Ziv and uses Finite State Entropy coding, based on Jarek Duda's work on Asymmetric Numeral Systems (ANS) for entropy coding. Shortly, ANS aims to "end the trade-off between speed and rate" and can be used both for precise coding and very fast encoding, with support for data encryption. LZFSE is one of a growing number of compression libraries that use ANS in place of the more traditional Huffman and arithmetic coding.

Admittedly, LZFSE does not aim to be the best or fastest algorithm out there. In fact, Apple states that LZ4 is faster than LZFSE while LZMA provides a higher compression ratio, albeit at the cost of being an order of magnitude slower than other options available in Apple SDKs. LZFSE is Apple's suggested option when compression and speed are more or less equally important and you want reduce energy consumption.

LZFSE reference implementation is available on GitHub. Building on macOS is as easy as executing:

```
$ xcodebuild install DSTROOT=/tmp/lzfse.dst
```

If you want to build LZFSE for a current iOS device, you can execute:

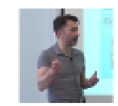
```
xcodebuild -configuration "Release" -arch armv7 install DSTROOT=/tmp/lzfse.dst
```

RELATED CONTENT

Facebook Open-Sources New Compression Algorithm Outperforming Zlib Sep 02, 2016

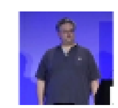
Swift 3 is Out Sep 20, 2016

Image Processing for iOS Jul 05, 2016



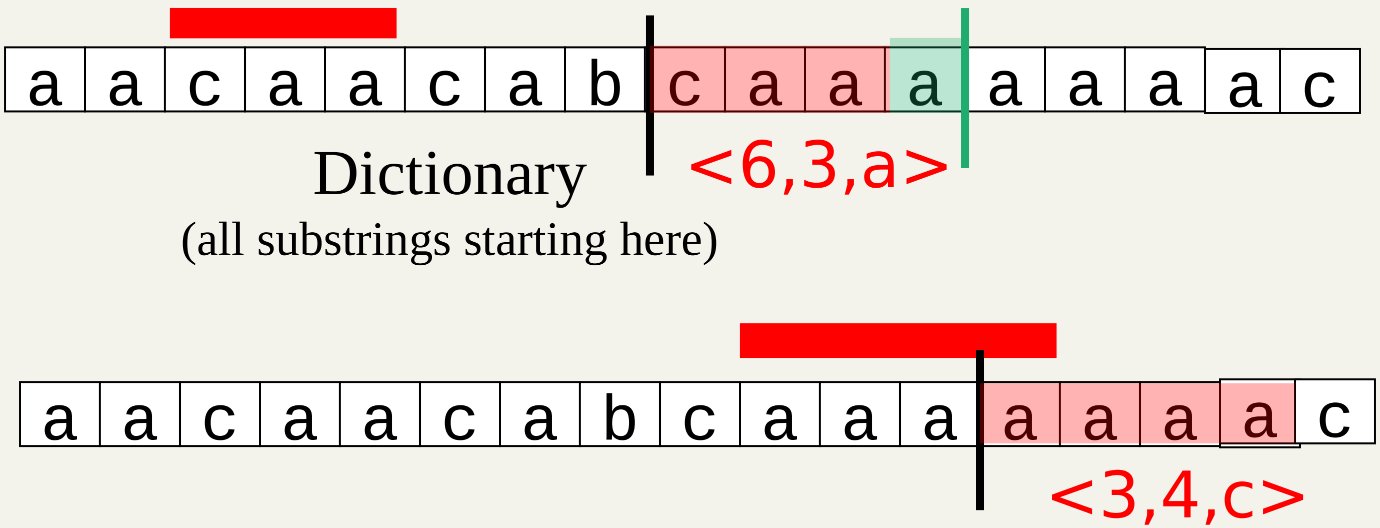
Oracle Gives NetBeans to the Apache Foundation Sep 19, 2016

Open Source Swift Under the Hood Apr 09, 2016



IBM's Swift on the Server Apr 09, 2016

LZ77



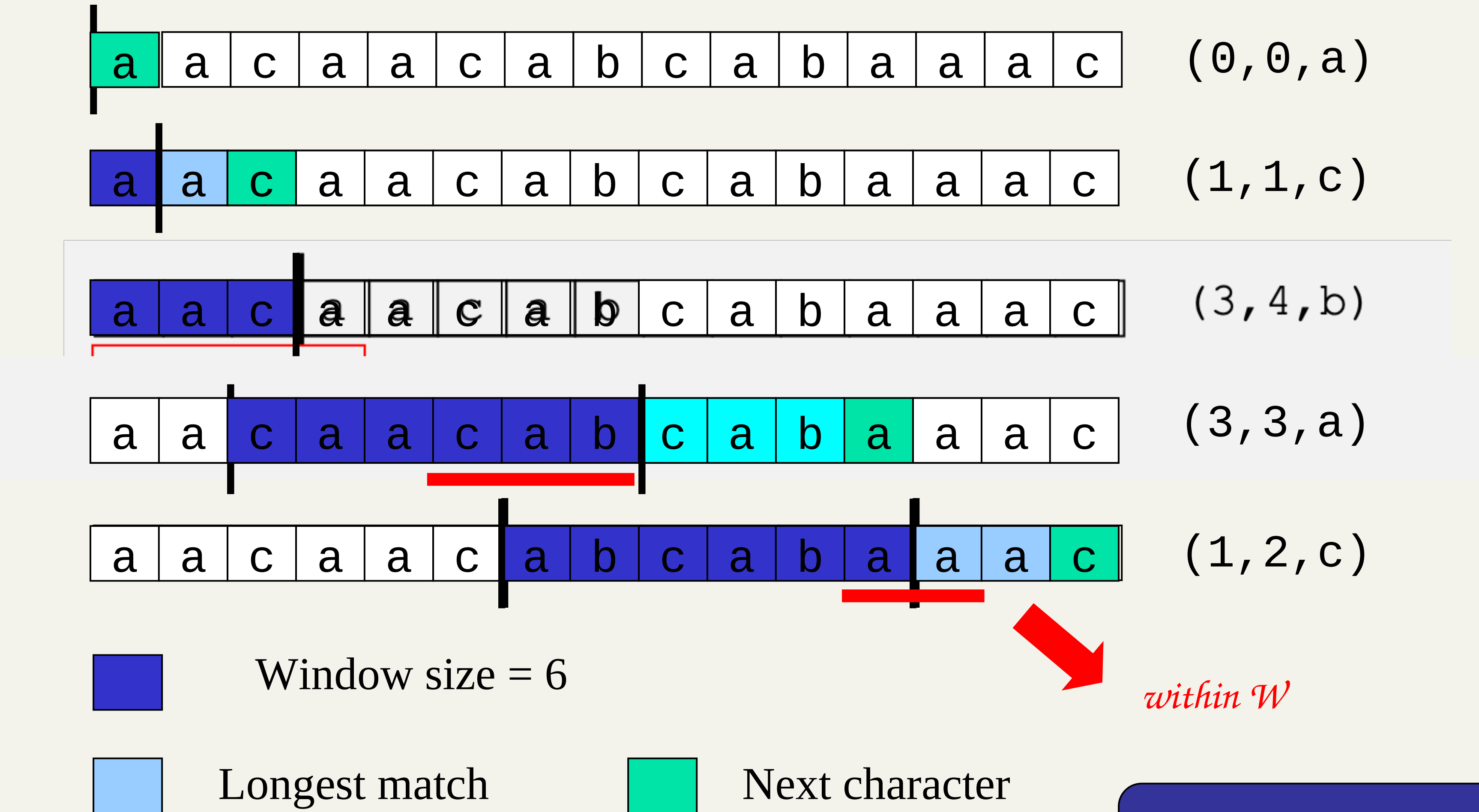
Algorithm’s step:

- Output <dist, len, next-char>
- Advance by len + 1

A buffer “window” has fixed length and moves

<0,0,a><1,1,c><3,4,b>…

Example: LZ77 with window



Gzip -1…-9

Which is faster in decompression among gzip -1…-9 ?

LZ77 Decoding

Decoder keeps same dictionary window as encoder.

- Finds substring <len,dist,char> in previously decoded text
- Inserts a copy of it

What if len > dist ? (overlap with text to be compressed)

- E.g. seen = abcd, next codeword is <2,**9**,e>

```
for (i = 0; i < len; i++)
  out[cursor+i] = out[cursor-d+i]
```

- Output is correct: abcdcdcdcdcdce

You find this at: www.gzip.org/zlib/

Squash Compression Benchmark

The Squash library is an abstraction layer for compression algorithms, making it trivial to switch between them… or write a benchmark which tries them all, which is what you see here!

The Squash Compression Benchmark currently consists of 28 datasets, each of which is tested against 29 plugins containing 46 codecs at every compression level they offer—the number varies by codec, but there are 235 in total, yielding 6,580 different settings. The benchmark is currently run on 9 different machines for a current grand total of 59,220 configurations, and growing.

SKIP TO RESULTS (PRETTY PICTURES!) LEARN MORE ABOUT SQUASH

Configuration

Choose a dataset

Different codecs can behave *very* differently with different data. Some are great at compressing text but horrible with binary data, some excel with more repetitive data like logs. Many have long initialization times but are fast once they get started, while others can compress/decompress small buffers almost instantly.

This benchmark is run against many standard datasets. Hopefully one of them is interesting for you, but if not don't worry—you can use Squash to easily run your own benchmark with whatever data you want. That said, if you think you have a somewhat common use case, please let us know—we may be interested in adding the data to this benchmark.

> **Note**
>
> The default dataset is selected randomly.

Name	Source	Description	Size
alice29.txt	Canterbury Corpus	English text	148.52 KiB
asyoulik.txt	Canterbury Corpus	Shakespeare	122.25 KiB
cp.html	Canterbury Corpus	HTML source	24.03 KiB
dickens	Silesia Corpus	Collected works of Charles Dickens	9.72 MiB
enwik8	Large Text Compression Benchmark	The first 10^8 bytes of the English Wikipedia dump on Mar. 3, 2006	95.37 MiB
fields.c	Canterbury Corpus	C source	10.89 KiB
fireworks.jpeg	Snappy	A JPEG image	120.21 KiB
geo.protodata	Snappy	A set of Protocol Buffer data	115.81 KiB

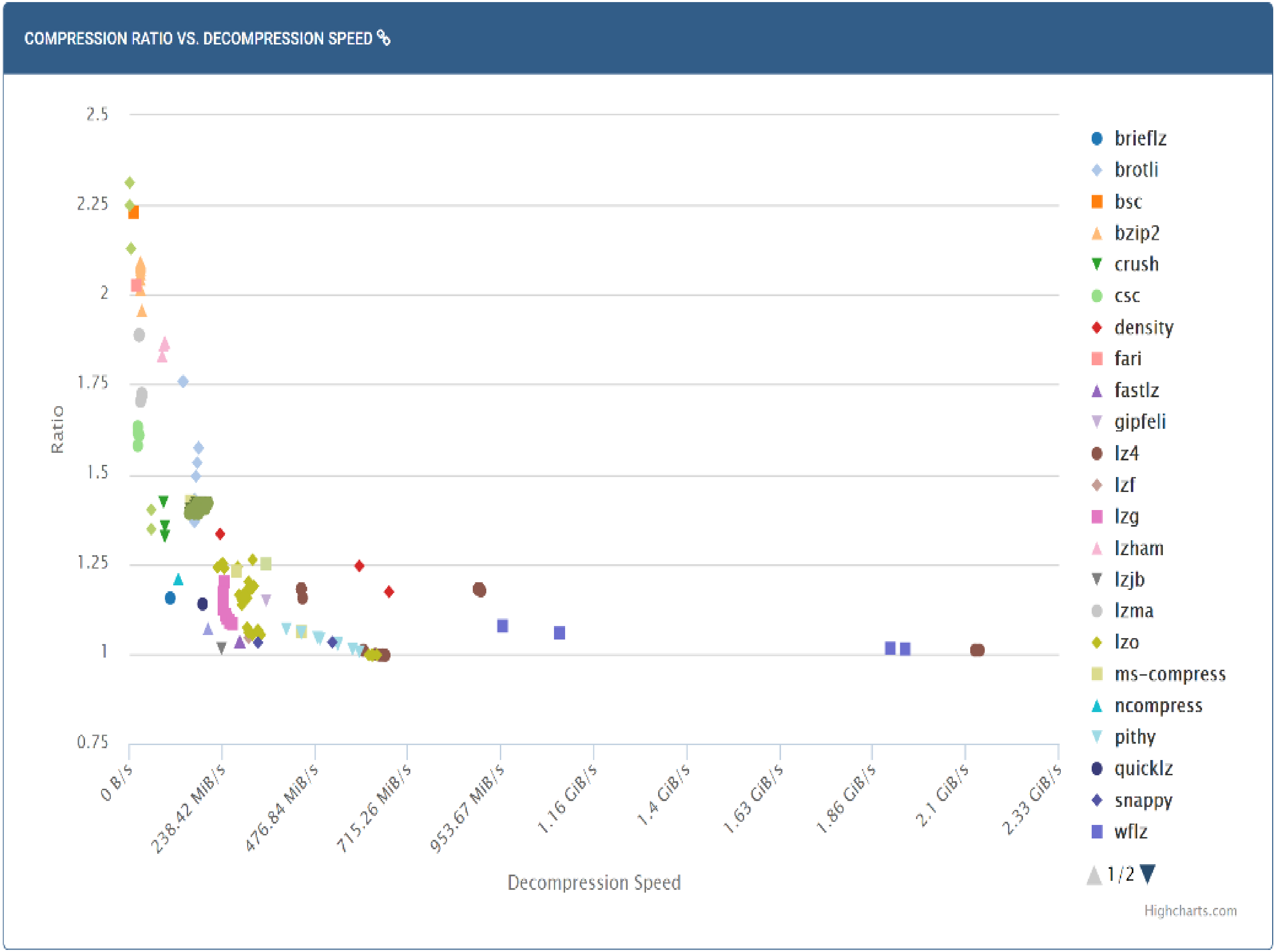
Choose a machine

If you think certain algorithms are always faster, you've got another thing coming! Different CPUs can behave very differently with the same data.

The Squash benchmark is currently run on many of the machines I have access to—this happens to be fairly recent Intel CPUs, and a mix of ARM SBCs. There is an entry in the FAQ with more details.

> **Note**
>
> The default machine is selected randomly.

	Name	Status	CPU/SoC	Architecture	Clock Speed	Memory	Platform	Distro	Kernel	Compiler	CSV
○	**beagleboard-xm**	✔	Texas Instruments DM3730	armv7l	1 GHz	512 MiB	BeagleBoard-xM revision B	Ubuntu 15.04	4.1.5	gcc-4.9.2	
○	**e-desktop**	✔	Intel® Core™ i3-2105	x86_64	3.1 GHz	8 GiB	Asus P8H61-H	Fedora 22	4.1.4	gcc-5.1.1	
○	**hoplite**	✔	Intel® Core™ i7-2630QM	x86_64	2 GHz	6 GiB	Toshiba Satellite A660-X	Fedora 22	4.1.4	gcc-5.1.1	
○	**odroid-c1**	✔	Amlogic S805	armv7l	1.5 GHz	1 GiB	ODROID-C1	Ubuntu 14.04.4	3.10.80	gcc-4.9.2	
◉	**peltast**	✔	Intel® Xeon® Processor E3-1225 v3	x86_64	3.2 GHz	20 GiB	Lenovo ThinkServer TS140	Fedora 22	4.1.6	gcc-5.1.1	
○	**phalanx**	✔	Intel® Atom™ D525	x86_64	1.8 GHz	4 GiB	Asus AT5NM10T-I	Fedora 22	4.1.4	gcc-5.1.1	
○	**raspberry-pi-2**	✔	Broadcom BCM2709	armv7l	900 MHz	1 GiB	Raspberry Pi 2 Model B	Raspbian Jessie	4.1.6	gcc-4.9.2	
○	**s-desktop**	✔	Intel® Core™ i5-2400	x86_64	3.1 GHz	4 GiB	Asus P8Z68-V	Fedora 22	4.1.4	gcc-5.1.1	
○	**satellite-a205**	✔	Intel® Celeron® Processor 540	x86_64	1.86 GHz	1 GiB	Toshiba Satellite A205-S5805	Fedora 21	4.1.6	gcc-4.9.2	



COMPRESSION RATIO VS. DECOMPRESSION SPEED
Ratio
2.5
2.25
2
1.75
1.5
1.25
1
0.75
0 B/s
238.42 MiB/s
476.84 MiB/s
715.26 MiB/s
953.67 MiB/s
1.16 GiB/s
1.4 GiB/s
1.63 GiB/s
1.86 GiB/s
2.1 GiB/s
2.33 GiB/s
Decompression Speed
brieflz
brotli
bsc
bzip2
crush
csc
density
fari
fastlz
gipfeli
lz4
lzf
lzg
lzham
lzjb
lzma
lzo
ms-compress
ncompress
pithy
quicklz
snappy
wflz
1/2
Highcharts.com

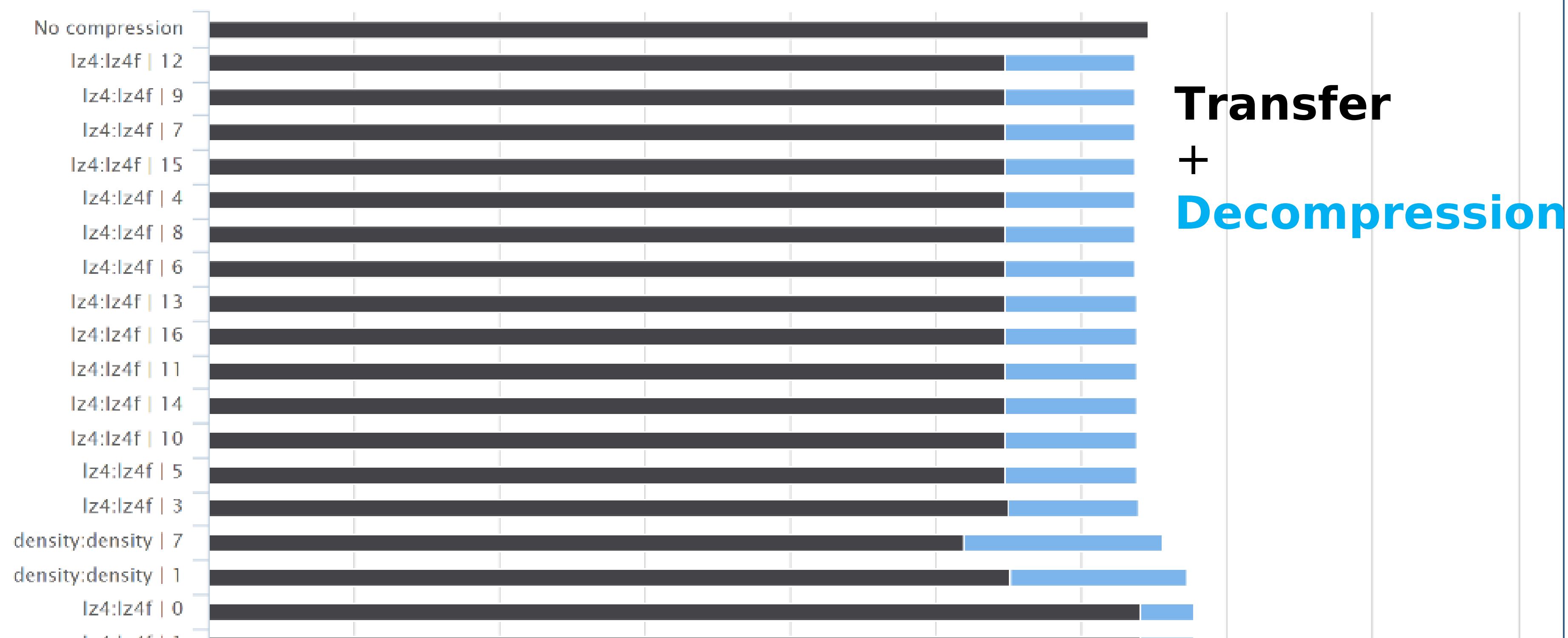
Transfer Speed PRESETS 125 MIB/S

Visible Items Within 25% of no compression

Sort Total time Name

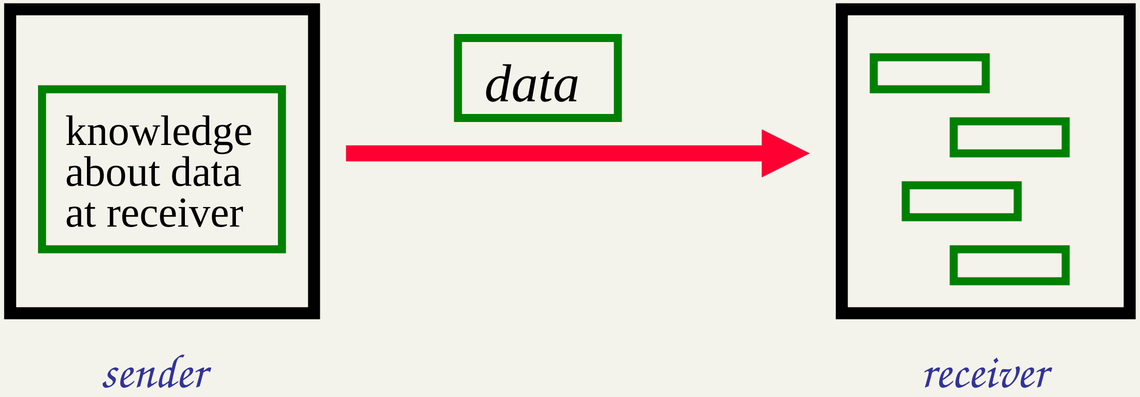
Direction Decompression Compression Both

DRAW



Compression & Networking

Background



- network links are getting faster and faster but
  - many clients still connected by fairly slow links (mobile?)
  - people wish to send more and more data
  - battery life is a king problem

how can we make this transparent to the user?

Two standard techniques

- caching: *“avoid sending the same object again”*
  - Done on the basis of “atomic” objects
  - Thus only works if objects are unchanged
  - How about objects that are slightly changed?

- compression: *“remove redundancy in transmitted data”*
  - avoid repeated substrings in transmitted data
  - can be extended to history of past transmissions (overhead)
  - How if the sender has never seen data at receiver ?

Types of Techniques

- Common knowledge between sender & receiver
  - Unstructured file: delta compression

- “Partial” knowledge between sender & receiver
  - Unstructured files: file synchronization
  - Record-based data: set reconciliation

Formalization

- **Delta compression** [*diff*, *zdelta*, *REBL*,…]
  - Compress file *f* deploying **known** file *f’*
  - Compress a group of files
  - Speed-up web access by sending differences between the requested page and the ones available in cache

- **File synchronization** [*rsynch*, *zsync*]
  - Client updates **its** old file f_{old} with **new** file f_{new} **available** on a server
  - Mirroring, Shared Crawling, Content Distribution Network

Z-delta compression (one-to-one)

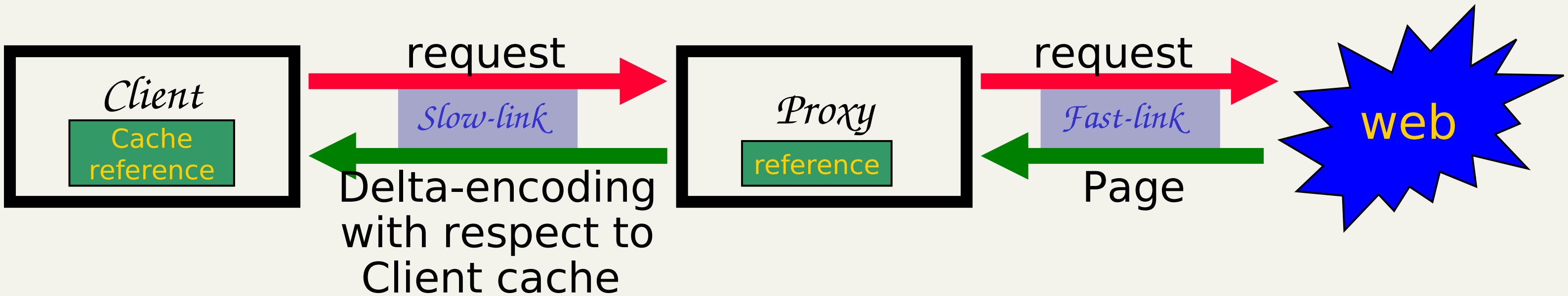
Problem: We have two files f_{known} (known to both parties) and f_{new} (is known only to the sender) and the goal is to compute a file f_d of minimum size such that f_{new} can be derived by the receiver from f_{known} and f_d

- Assume that block **moves** and **copies** are allowed
- **Find an optimal covering set of f_{new} based on f_{known}**

- **LZ77-scheme** provides and efficient, optimal solution
  - f_{known} is "previously encoded text", compress $f_{known}f_{new}$ starting from f_{new}
- **zdelta** is one of the best implementations
- Uses e.g. in Version control, Backups, and Transmission.

Efficient Web Access

Dual proxy architecture: pair of proxies (client cache + proxy) located on each side of the slow link use a proprietary protocol to increase performance



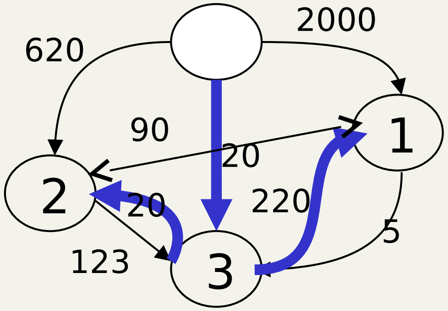
- Use *zdelta* to reduce traffic:
  - Old version is available at both proxies (one on client cache, and one on proxy)
  - Restricted to pages already visited (30% cache hits), or UR

Small cache

Cluster-based delta compression

Problem: We wish to compress a group of files F

- Useful on a dynamic collection of web pages, back-ups, …
- Apply *pairwise* **zdelta**: find a *good reference* for each $f \in F$
- Reduction to the Min Branching problem on DAGs
  - Build a (**complete?**) weighted graph G_F, nodes=files, weights= **zdelta**-size
  - Insert a dummy node connected to all, and weights are **gzip**-coding
  - Compute the directed spanning tree of min tot cost, covering G's nodes.



		space	time
uncompr		30Mb	---
tgz		20%	linear
THIS		8%	quadratic

Improvement (group of files)

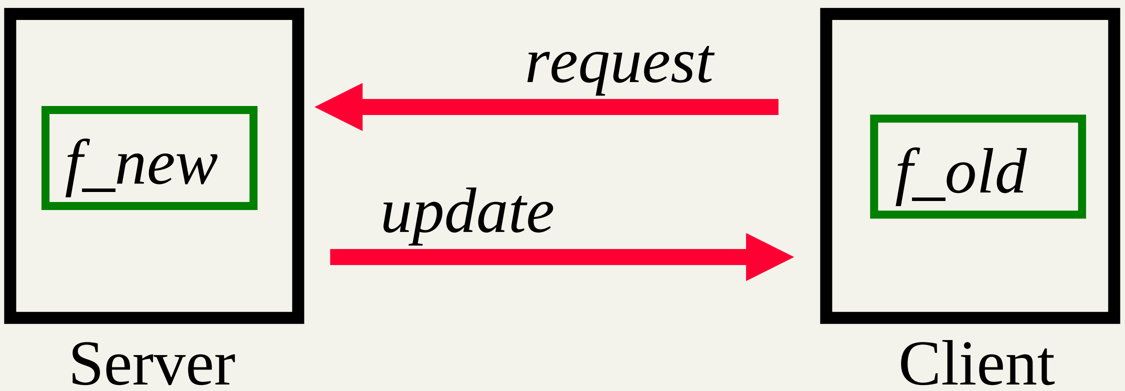
Problem: Constructing G is very costly, n^2 edge calculations (*zdelta exec*)

- We wish to exploit some pruning approach
  - **Collection analysis**: Cluster the files that appear similar and thus good candidates for zdelta-compression. Build a sparse weighted graph G'_F containing only edges between pairs of files in the same cluster
  - **Assign weights**: Estimate appropriate edge weights for G'_F thus saving zdelta execution. Nonetheless, strict n^2 time

		space	time
uncompr		260Mb	---
tgz		12%	2 mins
THIS		8%	16 mins

File Synchronization

File synch: The problem

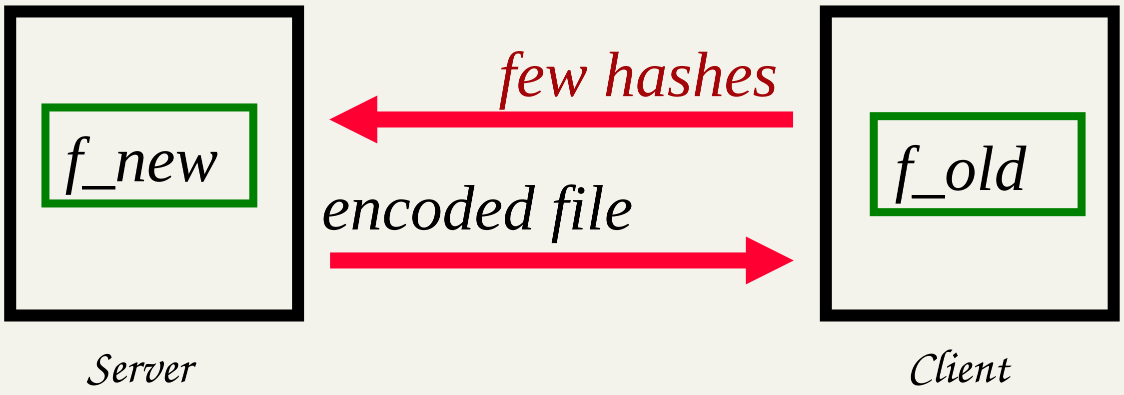


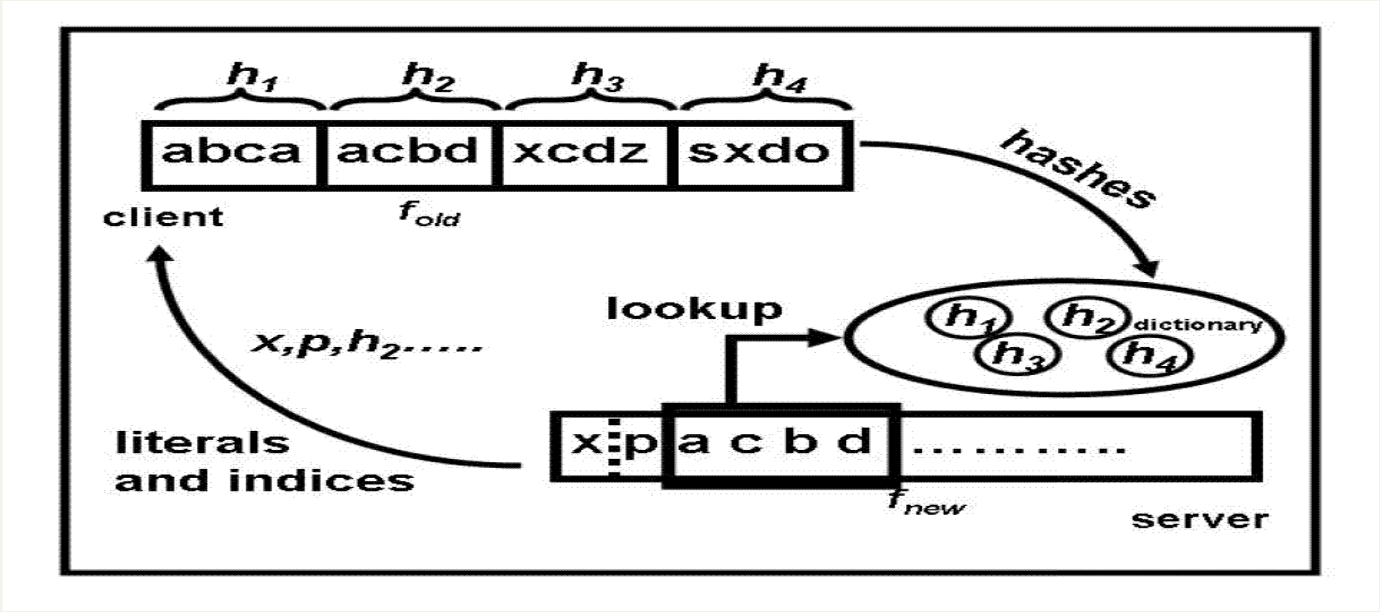
- **client**
  - *request* to update an old file
  - Sends a sketch of the old one
- **server**
  - has new file but <u>**does not know**</u> the old file
  - Sends an update of *f_old* given its received sketch and *f_new*
- *rsync*: file synch tool, distributed with Linux

Delta compression is a sort of local synch

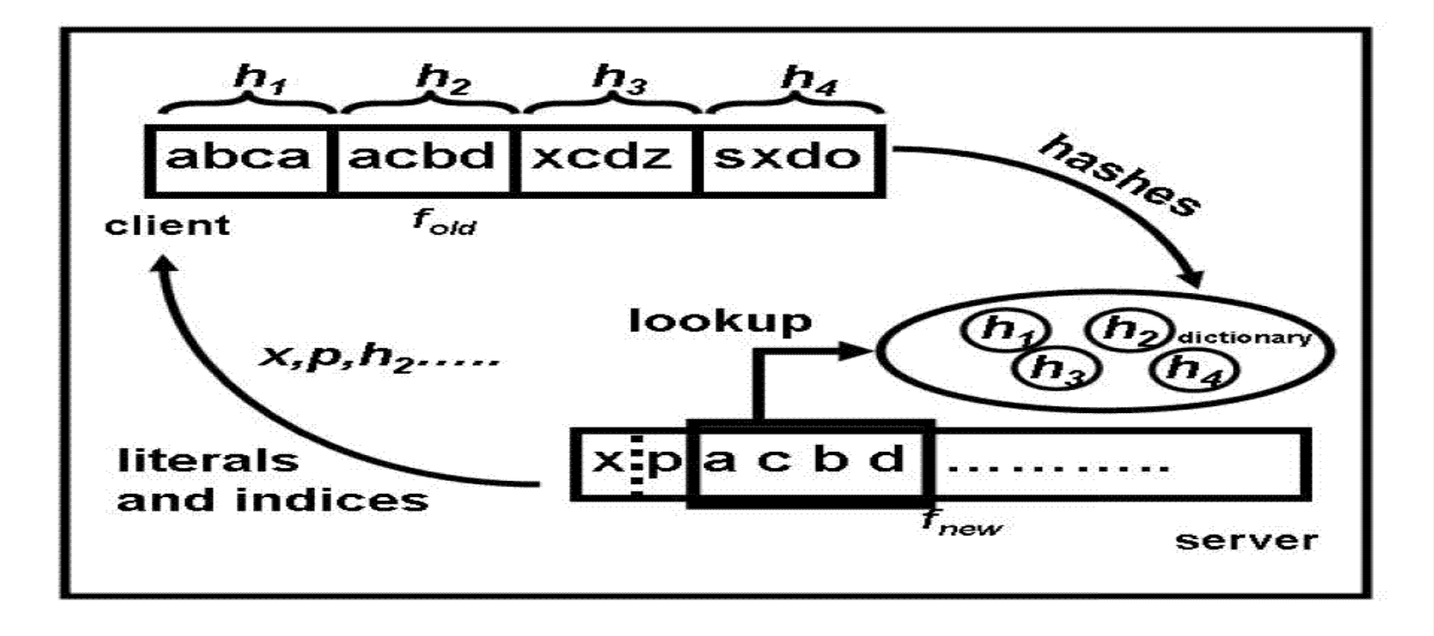
Since the server knows both files

The rsync algorithm



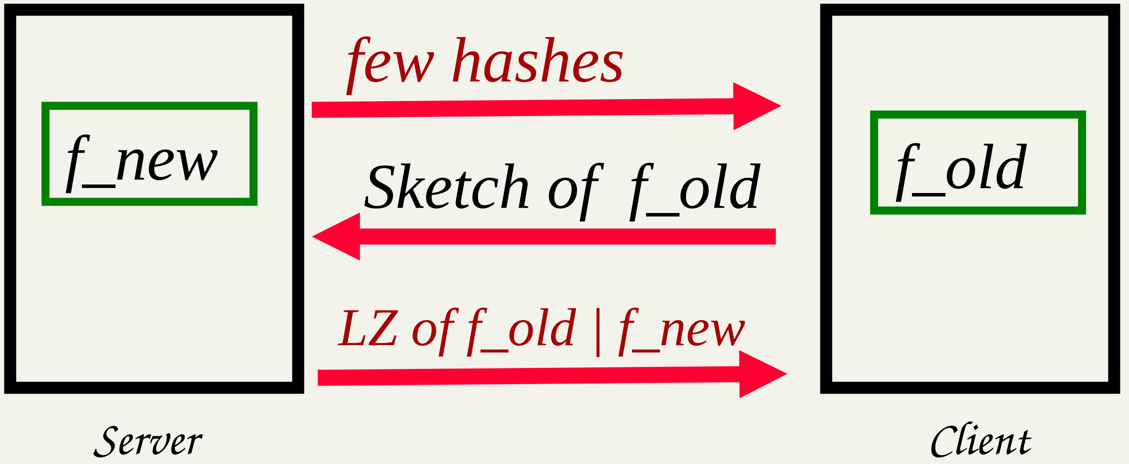


The rsync algorithm (contd)



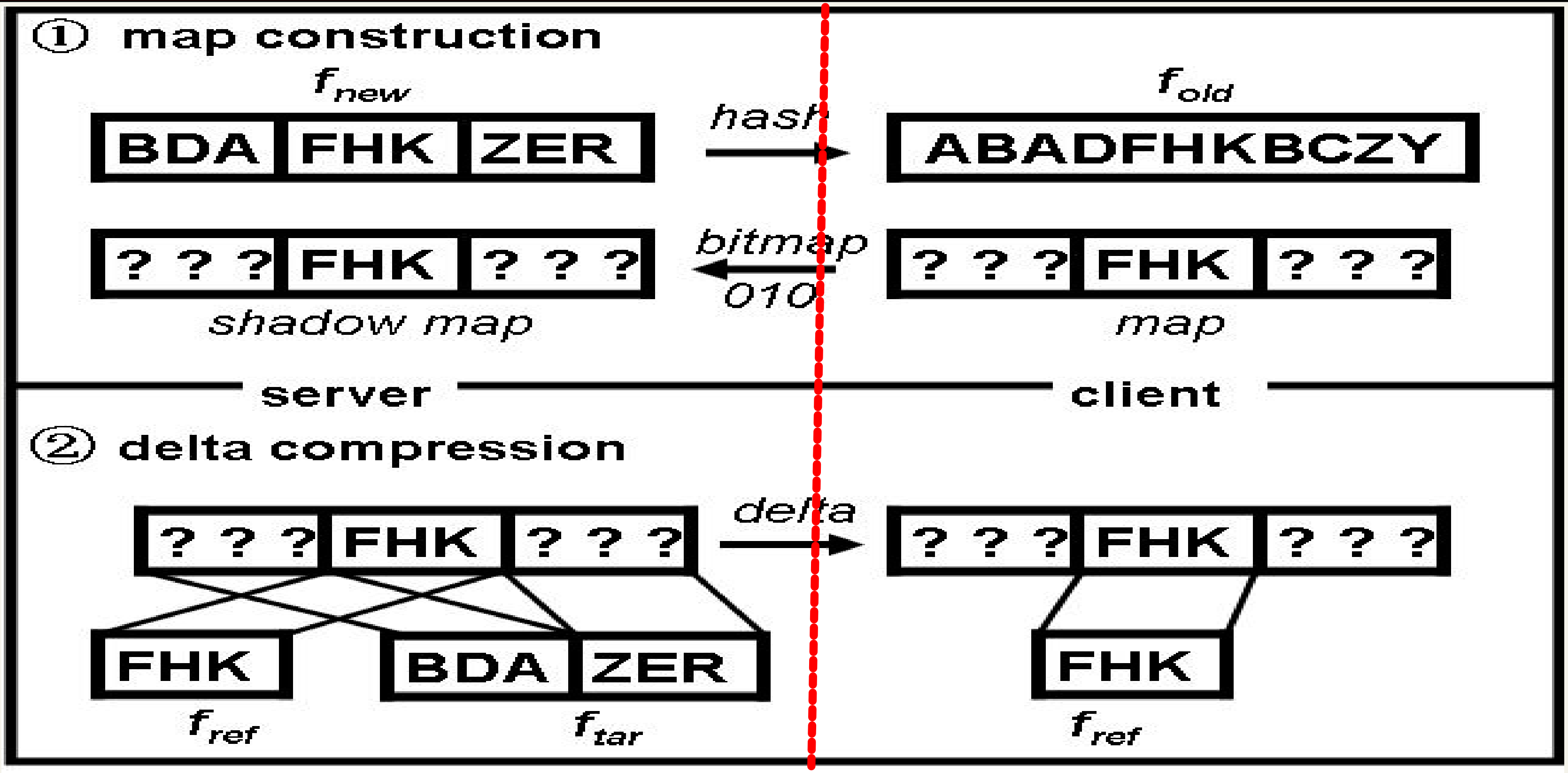
- simple, widely used, **single** roundtrip
- optimizations: 4-byte rolling hash + 2-byte MD5, *gzip* for literals
- choice of block size problematic (*default*: max{700, √n} bytes)
- not good in theory: granularity of changes may disrupt use of blocks
- There is a high load on the server

Minimize server load: zsync



- Server starts
- Client is assumed to be “good at CPU”: it makes the scan
- Three communication steps
- Hashes of f_new precalculated, and stored in .zsync
- Z-delta is used to send f_new, given a sketch of f_old

Minimize server load: zsync



The hashes of the blocks can be precalculated and stored in .zsync file
Server should be not overloaded. So it sends .zsync, clients checks them
It is better suited to distribute multiple-files through network, given one .zsync

Bitmap = 3 bits because of #blocks in f_new are three