Introduction

This specification describes a mechanism for splitting a byte stream into blocks of varying size with split boundaries based solely on the content of the input. It also describes a mechanism for organizing those blocks into a (probabilistically) balanced tree whose shape is likewise determined solely by the content of the input.

The general technique has been used by various systems such as:

- Perkeep
- Bup
- RSync
- Low-Bandwidth Network Filesystem (LBFS)
- Syncthing
- Kopia

. . . and many others. The technique permits the efficient representation of slightly different versions of the same data (e.g. successive revisions of a file in a version control system), since changes in one part of the input generally do not affect the boundaries of any but the adjacent blocks.

However, the exact functions used by these systems differ in details, and thus do not produce identical splits, making interoperability for some use cases more difficult than it should be.

The role of this specification is therefore to fully and formally describe a concrete function on which future systems may standardize, improving interoperability.

Notation

This section discusses notation used in this specification.

We define the following sets:

- U_{32}, The set of integers in the range $[0, 2^{32})$.
- U_8, The set of integers in the range $[0, 2^8)$, aka bytes.
- V_8, The set of *sequences* of bytes, i.e. sequences of U_8.
- V_v, The set of *sequences* of *sequences* of bytes, i.e. sequences of elements of V_8.
- V_{32}, The set of sequences of elements of U_{32}.

All arithmetic operations in this document are implicitly performed modulo 2^{32}. We use standard mathematical notation for addition, subtraction, multiplication, and exponentiation. Division always denotes integer division, i.e. any remainder is dropped.

Numerals staring with the prefix `0x` are hexadecimal, e.g. `0xfe` for the (decimal) number 254

We use the notation $\langle X_0, X_1, \ldots, X_k \rangle$ to denote an ordered sequence of values.

$|X|$ denotes the length of the sequence X, i.e. the number of elements it contains.

We also use the following operators and functions:

- $x \wedge y$ denotes the bitwise AND of x and y
- $x \vee y$ denotes the bitwise *inclusive* OR of x and y
- $x \oplus y$ denotes the bitwise *exclusive* OR of x and y
- $x \ll n$ denotes shifting x to the left n bits, i.e. $x \ll n = x2^n$
- $x \gg n$ denotes a *logical* right shift – it shifts x to the right by n bits, i.e. $x \gg n = x/2^n$
- $X \parallel Y$ denotes the concatenation of two sequences X and Y, i.e. if $X = \langle X_0, \ldots, X_N \rangle$ and $Y = \langle Y_0, \ldots, Y_M \rangle$ then $X \parallel Y = \langle X_0, \ldots, X_N, Y_0, \ldots, Y_M \rangle$
- $\min(x, y)$ denotes the minimum of x and y and $\max(x, y)$ denotes the maximum
- $\mathrm{ROT}_L(x, n)$ denotes the rotation of x to the left by n bits, i.e. $\mathrm{ROT}_L(x, n) = (x \ll n) \vee (x \gg (32 - n))$
- $\mathrm{Type}(x)$ denotes the type of x.

We use standard mathematical notation for summation. For example:

$\sum_{i=0}^{n} i$

denotes the sum of integers in the range $[0, n]$.

We define a similar notation for exclusive or:

$\bigoplus_{i=0}^{n} i$

denotes the bitwise exclusive or of the integers in $[0, n]$, i.e.

$\bigoplus_{i=0}^{n} i = 0 \oplus 1 \oplus \cdots \oplus n$

Finally, we define the "prefix" $\mathbb{P}_q(X)$ of a non-empty sequence X with respect to a given predicate q to be the initial subsequence X' of X up to and including the first member that makes $q(X')$ true. And we define the "remainder" $\mathbb{R}_q(X)$ to be everything left after removing the prefix.

Formally, given a sequence $X = \langle X_0, \ldots, X_{|X|-1} \rangle$ and a predicate $q \in \mathrm{Type}(X) \to \{\mathrm{true}, \mathrm{false}\}$,

$\mathbb{P}_q(X) = \langle X_0, \ldots, X_e \rangle$

for the smallest integer e such that:

- $0 \leq e < |X|$ and
- $q(\langle X_0, \ldots, X_e \rangle) = \mathrm{true}$

or $|X| - 1$ if no such integer exists. (I.e., if nothing satisfies q, the prefix is the whole sequence.) And:

$\mathbb{R}_q(X) = \langle X_b, \ldots, X_{|X|-1} \rangle$

where $b = |\mathbb{P}_q(\langle X_0, \ldots, X_{|X|-1} \rangle)|$.

Note that when $\mathbb{P}_q(X) = X$, $\mathbb{R}_q(X) = \langle\rangle$.

Splitting

The primary result of this specification is to define a family of functions:

$\text{SPLIT}_C \in V_8 \to V_v$

... which is parameterized by a *configuration* C, consisting of:

- $S_{\min} \in U_{32}$, the minimum split size
- $S_{\max} \in U_{32}$, the maximum split size
- $H \in V_8 \to U_{32}$, the hash function
- $T \in U_{32}$, the threshold

The configuration must satisfy $S_{\max} \geq S_{\min} > 0$.

Definitions

We define the constant W, which we call the "window size," to be 64.

We define the predicate $q_C(X)$ on a non-empty byte sequence X with respect to a configuration C to be:

- true if $|X| = S_{\max}$; otherwise
- true if $|X| \geq S_{\min}$ and $H(\langle X_{\max(0,|X|-W)}, \ldots, X_{|X|-1} \rangle) \mod 2^T = 0$ (i.e., the last W bytes of X hash to a value with at least T trailing zeroes); otherwise
- false.

We define $\text{SPLIT}_C(X)$ recursively, as follows:

- If $|X| = 0$, $\text{SPLIT}_C(X) = \langle\rangle$
- Otherwise, $\text{SPLIT}_C(X) = \langle \mathbb{P}_{q_C}(X) \rangle \parallel \text{SPLIT}_C(\mathbb{R}_{q_C}(X))$

Tree Construction

If sequence X and sequence Y are largely the same, SPLIT_C will produce mostly the same chunks, choosing the same locations for chunk boundaries except in the vicinity of whatever differences there are between X and Y.

This has obvious benefits for storage and bandwidth, as the same chunks can represent both X and Y with few exceptions. But while only a small number of chunks may change, the *sequence* of chunks may get totally rewritten, as when a difference exists near the beginning of X and Y and all subsequent chunks have to "shift position" to the left or right. Representing the two different sequences may therefore require space that is linear in the size of X and Y.

We can do better, requiring space that is only *logarithmic* in the size of X and Y, by organizing the chunks in a tree whose shape, like the chunk boundaries themselves, is determined by the content of the input. The trees representing two slightly different versions of the same input will differ only in the subtrees in the vicinity of the differences.

Definitions

A "chunk" is a member of the sequence produced by SPLIT_C.

The "hashval" $V_C(X)$ of a byte sequence X is:

$H(\langle X_{\max(0,|X|-W)}, \ldots, X_{|X|-1} \rangle)$

(i.e., the hash of the last W bytes of X).

A "node" $N_{h,i}$ in a hashsplit tree at non-negative "height" h is a sequence of children. The children of a node at height 0 are chunks. The children of a node at height $h+1$ are nodes at height h.

A "tier" of a hashsplit tree is a sequence of nodes $N_h = \langle N_{h,0}, \ldots, N_{h,k} \rangle$ at a given height h.

The function $\text{Rightmost}(N_{h,i})$ on a node $N_{h,i} = \langle S_0, \ldots, S_e \rangle$ produces the "rightmost leaf chunk" defined recursively as follows:

- If $h = 0$, $\text{Rightmost}(N_{h,i}) = S_e$
- If $h > 0$, $\text{Rightmost}(N_{h,i}) = \text{Rightmost}(S_e)$

The "level" $L_C(X)$ of a given chunk X is $\max(0, Q - T)$, where Q is the largest integer such that

- $Q \leq 32$ and
- $V_C(\mathbb{P}_{q_C}(X)) \mod 2^Q = 0$

(i.e., the level is the number of trailing zeroes in the hashval in excess of the threshold needed to produce the prefix chunk $\mathbb{P}_{q_C}(X)$).

The level $L_C(N)$ of a given *node* N is the level of its rightmost leaf chunk: $L_C(N) = L_C(\text{Rightmost}(N))$

The predicate $z_{C,h}(K)$ on a sequence $K = \langle K_0, \ldots, K_e \rangle$ of chunks or of nodes with respect to a height h is defined as:

- true if $L_C(K_e) > h$; otherwise

- false.

For conciseness, define

- $P_C(X) = \mathbb{P}_{z_{C,0}}(\text{SPLIT}_C(X))$ and
- $R_C(X) = \mathbb{R}_{z_{C,0}}(\text{SPLIT}_C(X))$

Algorithm

This section contains two descriptions of hashsplit trees: an algebraic description for formal reasoning, and a procedural description for practical construction.

Algebraic description

The tier N_0 of hashsplit tree nodes for a given byte sequence X is equal to

$\langle P_C(X) \rangle \| R_C(X)$

The tier N_{h+1} of hashsplit tree nodes for a given byte sequence X is equal to

$\langle \mathbb{P}_{z_{C,h+1}}(N_h) \rangle \| \mathbb{R}_{z_{C,h+1}}(N_h)$

(I.e., each node in the tree has as its children a sequence of chunks or lower-tier nodes, as appropriate, up to and including the first one whose "level" is greater than the node's height.)

The root of the hashsplit tree is $N_{h',0}$ for the smallest value of h' such that $|N_{h'}| = 1$

Procedural description

For this description we use N_h to denote a single node at height h. The algorithm must keep track of the "rightmost" such node for each tier in the tree.

To compute a hashsplit tree from a byte sequence X, compute its "root node" as follows.

1. Let N_0 be $\langle\rangle$ (i.e., a node at height 0 with no children).
2. If $|X| = 0$, then:
    a. Let h be the largest height such that N_h exists.
    b. If $|N_0| > 0$, then:
        i. For each integer i in $[0..h]$, "close" N_i (see below).
        ii. Set $h \leftarrow h + 1$.
    c. [pruning] While $h > 0$ and $|N_h| = 1$, set $h \leftarrow h - 1$ (i.e., traverse from the prospective tree root downward until there is a node with more than one child).
    d. **Terminate** with N_h as the root node.

3. Otherwise, set $N_0 \leftarrow N_0 \parallel \langle P_C(X) \rangle$ (i.e., add $P_C(X)$ to the list of children in N_0).
4. For each integer i in $[0..L_C(X))$, "close" the node N_i (see below).
5. Set $X \leftarrow R_C(X)$.
6. Go to step 2.

To "close" a node N_i:

1. If no N_{i+1} exists yet, let N_{i+1} be $\langle\rangle$ (i.e., a node at height $i+1$ with no children).
2. Set $N_{i+1} \leftarrow N_{i+1} \parallel \langle N_i \rangle$ (i.e., add N_i as a child to N_{i+1}).
3. Let N_i be $\langle\rangle$ (i.e., new node at height i with no children).

Rolling Hash Functions

CP32

The `cp32` hash function is based on cyclic polynomials. The family of related functions is sometimes also called "buzhash." `cp32` is the recommended hash function for use with hashsplit; use it unless you have clear reasons for doing otherwise.

Definition

We define the function $\mathrm{CP32} \in V_8 \rightarrow U_{32}$ as:

$\mathrm{CP32}(X) = \bigoplus_{i=0}^{|X|-1} \mathrm{ROT}_L(g(X_i), |X| - i + 1)$

Where $g(n) = G_n$ and the sequence $G \in V_{32}$ is defined in the appendix.

The sequence G was chosen at random. Note that $|G| = 256$, so $g(n)$ is always defined.

Implementation

Rolling

CP32 can be computed in a rolling fashion; for sequences

$X = \langle X_0, \ldots, X_N \rangle$

and

$Y = \langle X_1, \ldots, X_N, y \rangle$

Given $\mathrm{CP32}(X)$, X_0 and y, we can compute $\mathrm{CP32}(Y)$ as:

$\text{CP32}(Y) = \text{ROT}_L(\text{CP32}(X), 1) \oplus \text{ROT}_L(g(X_0), |X| \mod 32) \oplus g(y)$.

Note that the splitting algorithm only computes hashes on sequences of size $W = 64$, and since 64 is a multiple of 32 this means that for the purposes of splitting, the above can be simplified to:

$\text{CP32}(Y) = \text{ROT}_L(\text{CP32}(X), 1) \oplus g(X_0) \oplus g(y)$.

The RRS Rolling Checksums

The `rrs` family of checksums is based on an algorithm first used in rsync, and later adapted for use in bup and perkeep. `rrs` was originally inspired by the adler-32 checksum. The name `rrs` was chosen for this specification, and stands for `rsync rolling sum`.

Definition

A concrete `rrs` checksum is defined by the parameters:

- M, the modulus
- c, the character offset

Given a sequence of bytes $\langle X_0, X_1, \ldots, X_N \rangle$ and a choice of M and c, the `rrs` hash of the sub-sequence $\langle X_k, \ldots, X_l \rangle$ is $s(k, l)$, where:

$a(k, l) = (\sum_{i=k}^{l}(X_i + c)) \mod M$

$b(k, l) = (\sum_{i=k}^{l}(l - i + 1)(X_i + c)) \mod M$

$s(k, l) = b(k, l) + 2^{16}a(k, l)$

RRS1

The concrete hash called `rrs1` uses the values:

- $M = 2^{16}$
- $c = 31$

`rrs1` is used by both Bup and Perkeep, and implemented by the Go package `go4.org/rollsum`.

Implementation

Rolling

`rrs` is a family of *rolling* hashes. We can compute hashes in a rolling fashion by taking advantage of the fact that, for $l \geq k \geq 0$:

$$a(k+1,l+1) = (a(k,l) - (X_k + c) + (X_{l+1} + c)) \mod M$$

$$b(k+1,l+1) = (b(k,l) - (l-k+1)(X_k + c) + a(k+1,l+1)) \mod M$$

So, a typical implementation will work like this:

- Keep $\langle X_k, \ldots, X_l \rangle$ in a ring buffer.
- Also store $a(k,l)$ and $b(k,l)$.
- When X_{l+1} is added to the hash:
- Dequeue X_k from the ring buffer, and enqueue X_{l+1}.
- Use X_k, X_{l+1}, and the stored $a(k,l)$ and $b(k,l)$ to compute $a(k+1,l+1)$ and $b(k+1,l+1)$. Then use those values to compute $s(k+1,l+1)$ and also store them for future use.

In all cases the ring buffer should initially contain all zero bytes, reflecting the use of $X_i = 0$ for $i < 0$ in "Splitting", above.

Choice of M

Choosing $M = 2^{16}$ has the advantages of simplicity and efficiency, as it allows $s(k,l)$ to be computed using only shifts and bitwise operators:

$$s(k,l) = b(k,l) \vee (a(k,l) \ll 16)$$

Appendix

The definition of G as used by CP32 is:

⟨

```
0x6b326ac4, 0x13f8e1bd, 0x1d61066f, 0x87733fc7, 0x37145391, 0x1c115e40,
0xd2ea17a3, 0x8650e4b1, 0xe892bb09, 0x408a0c3a, 0x3c40b72c, 0x2a988fb0,
0xf691d0f8, 0xb22072d9, 0x6fa8b705, 0x72bd6386, 0xdd905ac3, 0x7fcba0ba,
0x4f84a51c, 0x1dd8477e, 0x6f972f2c, 0xaccd018e, 0xe2964f13, 0x7a7d2388,
0xebf42ca7, 0xa8e2a0a2, 0x8eb726d3, 0xccd169b6, 0x5444f61e, 0xe178ad7a,
0xd556a18d, 0xbac80ef4, 0x34cb8a87, 0x7740a1a9, 0x62640fe1, 0xb1e64472,
0xdee2d6c8, 0x27849114, 0xb6333f4b, 0xbb0b5c1d, 0x57e53652, 0xfde51999,
0xef773313, 0x1bbaf941, 0x2e9aa084, 0x37587ab8, 0xa61e7c54, 0xb779be61,
0xd8795bfd, 0x1707c1f6, 0x50fe9c54, 0x32ff3685, 0x94f55c22, 0x2a32ce1a,
0x0b9076ab, 0x14363079, 0xae994b2c, 0x4a8da881, 0x4770b9c4, 0xf4d143dd,
0x70a90c0b, 0xa094582a, 0x4b254d10, 0x2454325e, 0x1725a589, 0x9a3380da,
0x948eeade, 0x79f88224, 0x7b8dc378, 0xc2090db6, 0x41f7a7ac, 0xd4d9528c,
0x7f0bace7, 0xd3157814, 0xd7757bc4, 0xb428db06, 0x2e2b1d02, 0x0499bcf5,
0x310f963e, 0xe5f31a83, 0xe0cd600f, 0x8b48af14, 0x568eb23a, 0x01d1150b,
0x33f54023, 0xa0e59fdf, 0x8d17c2dd, 0xfb7bd347, 0x4d8cd432, 0x664db8de,
0xd48f2a6c, 0x16c3412d, 0x873a32fc, 0x10796a21, 0xed40f0f8, 0x5ca8e9b2,
0x0f70d259, 0x0df532c2, 0x016d73aa, 0x45761aa5, 0x189b45a7, 0x4accd733,
```

```
0x641f90e3, 0x592ed9ee, 0x4b1d72ad, 0x42ff2cd4, 0x0654b609, 0x799012c0,
0x595f36a4, 0x082bdbd6, 0x0375ddd3, 0xc16c1fb5, 0x57492df8, 0xa2d56a98,
0xdfb2aa28, 0x3728f35f, 0xdc49ea71, 0x9aee8377, 0xd62de2ab, 0x2c3aa155,
0x407d9eed, 0xbc5b3832, 0x42961924, 0x1498172a, 0xc7126716, 0x95494b56,
0xd40442fb, 0xb22a3ed1, 0x0ad3e0ae, 0x77a6136a, 0xfb1bc3f0, 0x1a715c38,
0xccbbd21d, 0x061ff037, 0x85d700cb, 0x8a8fb396, 0x956bbe48, 0xf2556ed8,
0x3319c88b, 0xe0d6d3e9, 0x4783b316, 0x03a73543, 0x253be5ed, 0x41322aea,
0xdfc00c7a, 0x972b9413, 0xccca42f5, 0x0a1cdf35, 0xa2dc31b8, 0xf48397eb,
0xbe3f2b3e, 0xd2950b9f, 0xccd269cf, 0x51a64ca9, 0xea46d96e, 0xcaec892e,
0x3fae3a62, 0xf12e53db, 0x3753464c, 0x214fbd91, 0x609ce2f7, 0x6158b44c,
0xa74b8027, 0x79f36912, 0x16cac162, 0x5e76df4f, 0xbc4184fb, 0x912cac7d,
0xf97e5704, 0x664dd25f, 0x7d837805, 0x5386cfe0, 0x4e585d77, 0xa0fa527e,
0xeb5c8401, 0xa186cc51, 0x05ef3f1f, 0xc1efc774, 0x38730c2c, 0xad9c5539,
0x27cd4938, 0x7317b4f2, 0x852c186f, 0xa4c9b0f4, 0xf592f010, 0xf6fe86f3,
0xb14ba86c, 0x07109a27, 0x0d00568d, 0xd92ee49f, 0xdc643eb3, 0x8d81c333,
0xcd1d7bbd, 0x87ff9cda, 0x80fa4285, 0x25258d5b, 0xd9e4065a, 0x78955c18,
0x84874c2a, 0xfdae136b, 0x48eeb3d3, 0xc2623958, 0x5a74f96d, 0x0bcb49f5,
0x3041cefc, 0xa5b0a1a8, 0x2d29bae6, 0x916ace93, 0x0e70564d, 0xa24894ae,
0x9897044d, 0xcba97c2a, 0x52a313b1, 0x318ec481, 0xc4729ec1, 0xd90ad78a,
0x55eb9f90, 0x4f159fda, 0xa90fbd44, 0xd0ca6208, 0x5c597269, 0xe05a471e,
0x26a5e224, 0x97144944, 0xece2c486, 0xf65c9a9e, 0x82a3fbbb, 0x925d1a62,
0xd6c4c29b, 0x61b9292d, 0x161529c9, 0x37713240, 0x68ec933b, 0xed80a4e5,
0x02b2db41, 0x47cfd676, 0xbfe26b41, 0x5e8468bb, 0x6e0d15a4, 0x40383ef4,
0x81e622fb, 0x194b378c, 0x0c503af5, 0x8e0033a7, 0x003aaa5e, 0x9d7b6723,
0x0702e877, 0x34b75166, 0xd1ba98d8, 0x9b9f1794, 0xe8961c84, 0x9d773b17,
0xf9783ee9, 0xdff11758, 0x49bea2cf, 0xa0e0887f
```

$\rangle$