Sub Chunk Delta Update Implements Discussion

Author: E*ternal* C*rystal*

Definition

Let ${}^{\dagger}s[i]$ represents a sub chunk who at time i. For each i, $s[i]$ represents the same sub chunk, but for different time.

Note for $j > i$, $s[j]$ is newer than $s[i]$.

Then, let $s[i][j]$ represents a block in position j. This block is in this sub chunk, and the time of this sub chunk is i.

${}^{\dagger}s[i]$ is an integer array who have 4096 elements.

Difference Array

Consider do difference operation once between $s[i-1]$ and $s[i]$ $(i \geq 1)$, then we get a difference array d that for each k $(0 \leq k \leq 4095)$ we have $s[i-1][k] + d[k] = s[i][k]$.

Do the same operator for all i $(i \geq 1)$ between $s[i-1]$ and $s[i]$, then we get a two-dimension difference array du that for each i, k $(i \geq 1, 0 \leq k \leq 4095)$ we have ${}^{\dagger}s[i-1][k] + du[i][k] = s[i][k]$.

Specially, let $s[0][k] = du[0][k] = 0$. Therefore, $du[1][k] = s[1][k]$.

Note that, it could be easy to prove the following expression is correct.

$$s[n][k] = \sum_{i=1}^{n} du[i][k]$$

That means, if we get the difference array, then we need $O(n)$ to get a block who at position k and the time of this block is n^{th}.

†Could use gzip to make each $s[i]$ as small as possible.

Append element

To append a newer sub chunk $ns = [i_1, i_2, i_3, \ldots, i_{4096}]$ to s who size is n, then that means create $s[n]$ and let $s[n] = ns$.

Note that for a single sub chunk ns, the size of ns is 4096 and have 4096 integers.

If we always save the data of $m = s[n-1]$, then $du[n] = s[n] - s[n-1] = ns - m$. Therefore, when need append a newer sub chunk to the timeline of current sub chunk, create $du[n]$ and make $du[n] = ns - m$, and then let $m = s[n]$.

Note that n must meet $n \geq 1$. And, for $n = 1$, we have $m = s[0] = [0,0,0,\ldots,0]$ and size of $s[0]$ is 4096.

Limit the length of timeline

It's impossible to let the size s become bigger and bigger. Therefore, it's necessary to limit the size of s.

Given a number g, we consider the size of s is len, and it must meet $len \leq g$.

Consider $len = g$ and now we will append a new element to s, and now we should delete $s[1]$. However, delete $s[1]$ make all i $(i > 1)$ in $s[i]$ will move, and this require time complexity $O(len)$ and is bad.

Consider we're modify a key/value database, so it's safe to maintain two number l and r $(r \geq l)$, and ensure for all i such $l \leq i \leq r$, $s[i]$ is valid and make for all i such $0 \leq i < r$ or $i > l$, $s[i]$ is not exist.

Specially, for all i such $0 \leq i < r$, although $s[i]$ is not exist, we can consider them as $[0,0,0,\ldots,0]$ and the size of this array is 4096.

Therefore, when $l - r + 1 = g$, and need to append a new element this time, we just need [†]delete $s[l]$ and let $l = l + 1$. After finish this operation, we can do our append operation as normal.

[†]Note that although $s[l]$ was deleted, we can consider $s[l]$ is an integer array that holds 4096 zeros.

Block palette

For each block in this sub chunk, no matter when the time it is, we always save just one block palette that could share by all the sub chunks on this timeline.

That means, we can't maintain the block palette very well when we increase l or append new sub chunk to the timeline, due to we don't know which blocks are still on using, and which blocks are not used.

It's suggested to add a function named *compact* that could make the block palette

minimize, but it also means we need to do the prefix sum operation for all element in s. Therefore, the time complexity is $O(4096 \times len)$ and result in take some time to finish.

All in all, increase l is no need to change the block palette. The time we need to change block palette is when a newer sub chunk is coming to s, and this time we need to compute newer block and add them to block palette.

Additionally, for block $b = s[i][k]$, b is the index of block palette. That means, if block palette is $bp = [\text{glass}, \text{grass}, \text{chain}]$, and $b = s[i][k] = 2$, then this block is $bp[b] = bp[s[i][k]] = bp[2] = \text{grass}$.

Delta update for block NBT

The previous part only discuss how to do delta update for blocks, so here let's turn our attention to NBT.

Unfortunately, NBT is bytes represents and can't process like a difference array.

A basic solution is firstly making each NBT stable. That means, when we convert NBT to bytes, we must ensure the key of TAG_Compound is sorted (Stable sorting).

Then, we can make each NBT object is stable, and the next step is use bsdiff to compute the changes between older and newer NBT object.

Note that if NBT is deleted, then just record which blocks (a number to represent the block position, from 0 to 4095) are deleted. And, if NBT is added, give the origin NBT bytes without do bsdiff. That means, only do bsdiff for changed (modified) NBT.

Lastly, we combine deleted, added and changed (modified) bytes together, and do gzip operation to make the binary result as small as possible.

For append, limit and query operation, it is same as blocks, and I will not discuss it again.

Recommendation

Consider all players moved for 100,000 blocks in Overworld at total every day (It is approximately a straight-line travel), and they set their chunk rendering radius r is 8, then it will have $(100000 \gg 4) \times (8 \times 2 + 1) = 106250$ chunks to be loaded.

Then, we consider we do delta update every 24 hours, then after 7 days, and each sub chunk delta update payload size is 512 Bytes, then the size of underlying database will be:

$$\frac{106250 \times (512 \times 24) \times 7}{1024 \times 1024 \times 1024} \approx 8 \text{ GB}$$

It is acceptable for a lot of users, so the standard recommendation is do delta update every day, and let $g = 7$.

In addition, consider the worst case, which $r = 32$, then 7 days will consume:

$$\frac{(100000 \gg 4) \times (32 \times 2 + 1) \times (512 \times 24) \times 7}{1024 \times 1024 \times 1024} \approx 32.5 \text{ GB}$$