

Table of Contents

Sequential Numbering of Blocks.....	1
Renumbering existing blocks.....	1
Sequential numbering without renumbering old blocks.....	1
The original algorithmic renumbering proposal.....	2

Sequential Numbering of Blocks

The case for sequentially numbering blocks (block IDs) rests on three ideas:

- Random numbering is dorky. Doing so requires either trusting that the probability of collision is really small, or else always testing whether a block ID is in use. And HDFS has had to live threatened by the *prehistoric block* problem. If a data node has been out of service while a block ID has been released/deleted and reassigned, the data node may return to the cluster with what purports to be a new block, but is really a replica of the old, deleted block. (The difficulty is resolved with the introduction of generation stamps for blocks.) If blocks were sequentially numbered, there would be no need to test whether a block ID is already in use.
- Generation stamps could be shorter, and per file. The proposal is that generation stamps should be longs, but maybe an int (4 billion generations per file) is good enough. (A reduction of a factor of two in the space used.)
- Generation stamps could be stored in the file inode, rather than with each block, as only the *last* block really needs a generation stamp. Also a factor of two in space conservation.

It was speculated that with sequential block IDs generation stamps might be unnecessary. The append proposal requires that blocks receive a new generation stamp when the block is reopened for writing and in certain error conditions. With sequential IDs, an equivalent operation would be to rename the block. The renaming option seems to be more complex, and the ability to trace operations is lost.

Renumbering existing blocks

Originally it was thought that moving to sequential block IDs would require renumbering all existing blocks in an upgrade. If a data node rejoined a renumbered cluster late, it would have to execute the renumbering on its collection of blocks, or else, delete its blocks. The latter option is mostly unacceptable, and if the former option required maintaining a translation table for block IDs, renumbering would be judged infeasible. An algorithmic renumbering was proposed that would require no translation table. The feasibility of that option depends upon the improbability of ID collisions if the ID space is contracted. (The proposal appears below.)

But, as it happens, it is practical to change to a policy of sequential IDs without renumbering any existing blocks.

Sequential numbering without renumbering old blocks

(Special thanks to Nicholas and Konstantin!)

The key idea is that even in our big clusters, very few block IDs are in use. So few block IDs are used that there necessarily are already long sequences available for use.

There are 2^{64} available block IDs and only about 2^{26} ($67e6$) existing blocks. The gap between successive block IDs is on average 2^{38} ($274e9$). If a busy cluster allocates a block every millisecond, that would be about $3e10$ new block IDs every year. So a typical range of sequential block IDs is good for a few years. And there are lots of typical ranges, even if the first selected range was exhausted.

For a name node wanting to move to sequential allocation, in safe mode,

- Choose a sequential range.
 - ◆ Sort the existing node IDs (or a reasonable subset).
 - ◆ Select two consecutive IDs with an above-average difference.

- Record the selection to the image/journal.
 - ◆ The allocation range must remain part of the name node's persistent state.

A data node does not have to do any work, but before rejoining the cluster,

- Get from the name node the allocation range.
- Delete any existing blocks with IDs in the range.
 - ◆ These blocks necessarily were previously deleted by the name node but not yet committed by the data node.

Easy!

The original algorithmic renumbering proposal

(Rob promises to buy lunch for everybody if there is a conflict on Kryptonite before 1 May 2008. In discharge of his expected obligation, he has given \$1 to Konstantin.)

How to rename blocks 17 October 2007 Rob Chansler

Extending the file system with versions for blocks has been discussed in anticipation of new file system features (append, multiple writers). One suggestion is to combine the notions of block ID and block version into a single identification (still to be called "block ID"). This works if block IDs can be assigned as integers in a strictly increasing sequence. Present block IDs are 64-bit random integers, and so are not immediately suitable. One implementation would be to widen block IDs to 80-bit integers where new block IDs would be assigned in increasing sequence beginning with 2^{64} . Legacy blocks would retain their old IDs widened to 80-bits. This implementation has the disadvantage that it changes the the data type of block IDs to be something other than a Java primitive data type. Scattered changes in existing code would be required, and users would be inconvenienced by yet larger representations of block IDs. There would be modest increases in space and execution overhead. As an alternative, the 64-bit space of block IDs could be retained if the block IDs for existing blocks could be reassigned to a numerically segment of the space and new block IDs assigned in increasing sequence from the upper limit of the segment.

A practical reassignment procedure should let data nodes and name nodes do the reassignment within their local data stores in a predictable way without having to exchange any information on a per-block basis. One way to do this relies on the facts that a) the present utilization of the name space is very sparse (10^8 IDs in use in a space of 10^{24}), and b) a 63-bit space for block IDs is big enough (there are less than 10^{14} microseconds in a year).

1. Pause the file system.
2. The name node unregisters each data node with instructions that the data node may not reregister until it has renamed all blocks in its custody according to the rule that for each block, the new ID is exactly the old ID with the high-order bit set. Of course, only one-half of the blocks change block IDs, and as a result of this operation, the block IDs of all blocks will be negative integers. The data node must make a permanent record that it has completed the renaming. The renaming operation can be restarted if it is interrupted for some reason, but it must not be repeated if ever the data node reports success by reregistering with the name node.
3. Meanwhile the name node changes the block IDs of all blocks it knows of according to the same rule. The name node must make a permanent record that it has completed the renaming. The renaming operation can be restarted if it is interrupted for some reason, but it must not be repeated if ever the name node resumes issuing block IDs.
4. Then the name node can begin to issue new block IDs in increasing sequence starting at 1 with confidence that new block IDs will be strictly greater than the block ID of any existing block.

5. Resume normal operation.

Since the renaming rule compresses the existing ID space, there is a chance of collision where for blocks B1 and B2, if ID(B1) and ID(B2) differ only in the high-order bit. If n items are chosen from a population of d with replacement, the probability of no collision is $\prod_{k=1}^{n-1} (1 - k/d)$. Approximating e^x as $1+x$, an approximate probability of no collision is $\prod_{k=1}^{n-1} (e^{-k/d}) = e^{-(n(n-1)/2d)}$ or $e^{-(n^2/2d)}$. For the case at hand, $d=2^{63}$, $n=2^{26}$ and so the approximate probability of no collision is $e^{-(2^{26+26-1-63})} = e^{-(2^{12})} = 0.99976$. That is, the probability that there is a collision is less than 1 in a 1000. If Kryptonite has a collision, lunch is my treat!

But a user application could fix up any collisions that might exist.

1. Pause any other file system clients.
2. Read the name node's image and search for collisions.
3. If a collision is observed for block B in file F, do "mv f temp; cp temp f; rm temp". A collision involves two files; copy the smaller! It is not important that copied blocks get new-style IDs; the renaming procedure will fix up any newly issued IDs as necessary.
4. Fix any new collision step 3 introduced (unlikely, but possible!).
5. Check that there are no blocks in the custody of any data node that are not in the name node image, for example, a block deletion not yet committed by the data nodes. If a conflict is discovered, the conflicting block must really go away; there must be no chance a block report will ever report the renamed conflicting block. [Assuring that a data node has committed all deletions is tough. If the data node is online, the name node could ask, "Are you really, really sure?" But what if the data node is offline?]
6. The renaming procedure can begin without risk of collisions.

And punt the existing just-in-case-of-collision code that exists in the current new block ID method.

What if a node is not present for the grand renaming, but wants to later join the cluster? One could trust to probability and just let the node join after a local renaming. But if there were any conflicts, there is a chance that this node held a replica of one of the conflicting blocks. This could be resolved by changing the rule for copying files to copy the file with a negative block id, instead of the shorter file. Now a late renaming cannot create a new conflict. OK, but what if all replicas of the blocks in conflict are off line, and so the conflict cannot be resolved by copying? One answer is to delete one of the files, but, the name node could keep an exceptions list of renaming that should be done when a node joins the cluster late.

But there is one more case where I'm unsure I have an answer. Suppose a node goes off line, and in the mean time a block with a positive block id is deleted from the name space where the off-line node happens to have a replica. Later a new block is allocated that just happens to have the negative conjugate block id that would have conflicted if the first block had not been deleted. Now the cluster is renamed, no conflicts are observed, and the off-line node wants to rejoin the cluster. But the rejoining node missed step 5 above, and so after renaming, its block become a false replica! But don't we already have this prehistoric block problem? My only idea is that if a data node joins the cluster late, it must abandon all blocks with IDs greater than 0.