HBASE-10382

Problem statement

In the current design, every scanner creates a StoreScanner on the Stores associated with the table. During the course of a scan, if any new HFile is added to the store due to flush, compaction and bulk load, it is immediately added to the scanner's heap and the current scan tries to use the new files also in its scan. In order to support this, we have all the scanners created against this store registered with the HStore. Whenever any new file is added to the store, all the current ongoing scanners are notified about this and using a lock we reset the scanners' heap so that all the current ongoing scans can make use of the newly added files. As we can see that there is a need for a lock to be acquired to do this updation, every read operation like next(), seek() and reseek() tries to acquire the lock before proceeding with the operation.

This leads to a lot of lock acquiring and releasing step thus leading to a performance impact.

Solution

One solution to solve this – (illustrated by Lar's patch) where we try to move the lock from the StoreScanner to the RegionScanner level and pass down this RegionScanner level lock to these StoreScanners such that we don't need to acquire and release the lock every time. However still the problem suggested above still exists because when there is an updation to the files in the store happens we need to synchronize on the RegionScanner lock and do the necessary updations before proceeding with the scans. Also passing down this region lock to the StoreScanners require a change in the way the CPs are allowed to create their own Storescanners since the lock has to be passed to the CPs to make use of the region level lock.

The other solution is to have a **versioned Storefile approach**. What we mean by versioned approach is that the current ongoing scans will use the existing files for its scan to complete. Take the case of a compaction, suppose a scan is going through the 3 files that were involved in the compaction, the scan would still use those 3 files for the scan to be completed though in the back ground the 3 files have been compacted to a new file. Any new scanner that starts after the compaction is completed will only make use of the new file created out of compaction.

Doing this will avoid any locks in the StoreScanner and the need for any notification to all the registered scanners. This also ensures that the current ongoing scanners blocks that are in the block cache can effectively be used till the scanners are completed and there will be no need for any eviction of the blocks of the compacted files and reading the blocks of the newly compacted files from hdfs for the ongoing scan.

Implementation

For implementing this versioned approach, we introduce a ref counting mechanism on all the Storefile readers. Whenever a scanner is created on a store file we increment the ref count by 1. When a scan completes using the current file we decrement the ref count. Along with the ref count, we also have CompactionStatus (DISCARDED and ACTIVE) added to these store file readers. Any file by default is in ACTIVE state. Once a compaction completes we update the state to DISCARDED.

When a scanner tries to create a scanner on these storefiles, we check for the status of the file whether they are in DISCARDED or ACTIVE state. If the state is ACTIVE, we allow the scanner to be created on this file and also increment the ref count.

Any file which is already in the DISCARDED state is not allowed to be part of the scan because when the state is changed to DISCARDED it means that already the current file has been compacted and its content are copied to a new file which is in the ACTIVE state and that scan will be using that new file instead of the files in the DICARDED state.

StoreFile Management

In the current design whenever a file is compacted, the compacted files are removed from the storefile list maintained by the Storefile manager. Thus at any point of time asking for a list of store files in the current store will only give the list of files that are not compacted. The compacted files are moved to archive directory.

In order to implement the versioned structure, we now introduce a new list to maintain the list of compacted files. Once a set of files are compacted, the compacted files are removed from the store files list and moved to the list of compacted files. Initially thought of having one list and ensure that the compacted files are also maintained in the store file list and the cleanup happens as part of the cleaner, but the problem was that lot of decisions like the flushes to wait for compactions to complete, the store related metrics and how split works based on the presence of reference files in the list of store files. Hence holding these compacted files in a new list helps us in avoiding all the above problems.

On completion of a compaction, mark the storefile as DISCARDED and do not archive the compacted store files. Ensure that a background thread runs periodically runs that scans the list of store files and checks for the state as DISCARDED and that there are no active readers on that file (ref count on those files should be 0), select all such files and remove them from the list of compacted files and move those files to the archive directory.

The compacted file cleaner thread is configured to run every 2 mins and the same can be adjusted using the config knob **'hbase.hfile.compactions.cleaner.interval'.** This chore is configured per region so that each cleaner thread is able to clean only those store files associated with that region.

Flushes and ongoing scans

Any ongoing flush will create a snapshot and the memstore scanners are built in such a way that the scan happens on the memstore and any snapshot that is currently active.

Once the flush has happened to a new file the snapshot is cleared and then the active store scanners are notified to include the store file recently flushed. In the current way of things, before the scanners could be notified to update and reset the heap, if any ongoing scanner is holding the store scanner lock the notification has to wait and though the flush has cleared the current snapshot and created a new one, the ongoing scan will still use the older snapshot and thus prevent the GC from clearing the snapshot reference.

We try to tweak this logic a bit such a way that just after clearing the snapshot we indicate all the scanners that a flush has happened. The storescanners before entering any of its API check if there is a flush that has happened. IF so it clears and resets the scanner heap and proceeds with the ongoing scan. If suppose the current scan misses this notification still the scan can continue on the older snapshot that

is not yet cleared by GC because this scan holds on to its reference. This way there are no other threads trying to update the scanner heap which in turn helps us to avoid the use of locks.

The other thing to note here is that the same scanner thread resetting its heap without lock is fine because StoreScanners are always single threaded once a scanner starts.

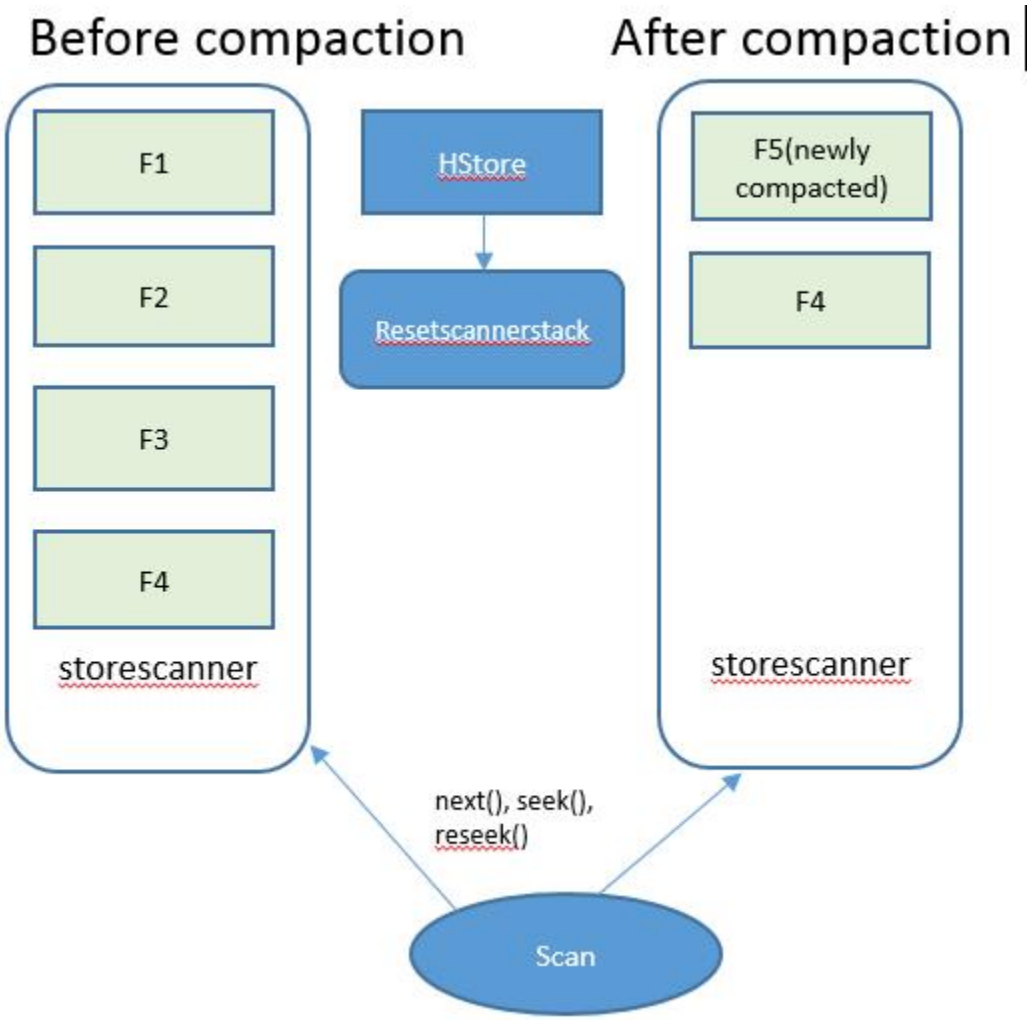
Points to be noted

- ➔ The state of the file if DISCARDED or ACTIVE is not persisted. On doing some initial cluster testing one problem that was seen is that, if suppose the region server crashes before the compacted files are actually archived, on restart those files may again be included in the new region server when the region is opened once again. The new compaction that happens on this new region server should ensure that the compaction once again happens and moves the files to the archive region.

  But do remember that the problem stated above could also happen in the current code when an archiving operation fails or a server crashes just after the compaction is successful and before moving the file to archive directory. So may not be too much of a worry.

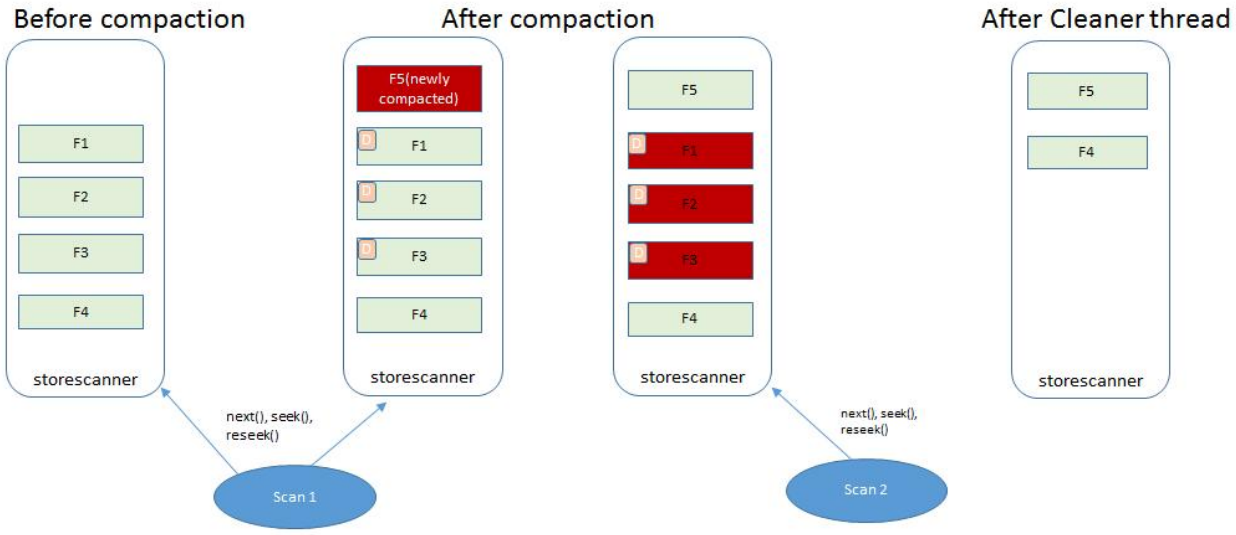
- ➔ In case of secondary replica regions – as per the current code if any newly added files are refreshed and loaded in the secondary region the older files are considered to be compacted and we close the reader after notifying the scanners active in the secondary region (Scanners created when client specifies TIMELINE consistency). But now since we go with the ref counting mechanism we cannot directly close the readers of the files that are to be removed from the secondary region and hence we have two active readers on the same file (one from the primary region and other from the secondary region). If the compacted hfile chore tries to archive the file from the primary region – it cannot do it successfully till the secondary region has closed the reader which happens again part of its compacted hfile chore. (HDFS does not allow to rename if there are any active readers).
- ➔ We should also ensure that the cleaner thread does running as part of the secondary region should not archive the DISCARDED files where as it should be done by the active region only.

Current approach



As shown in the above diagram, a scan is operating on 4 files F1, F2, F3 and F4, while a compaction happens on the files F1, F2 and F3 which creates a new file F5, we can see that the HStore does a scanner reset such that the ongoing scan now uses the files F5 and F4. This operation happens under the lock held by the StoreScanner. Since the files F1 to F3 are compacted they are evicted from the block cache and the new blocks of F5 needs to be fetched from the HDFS.

Versioned store file approach

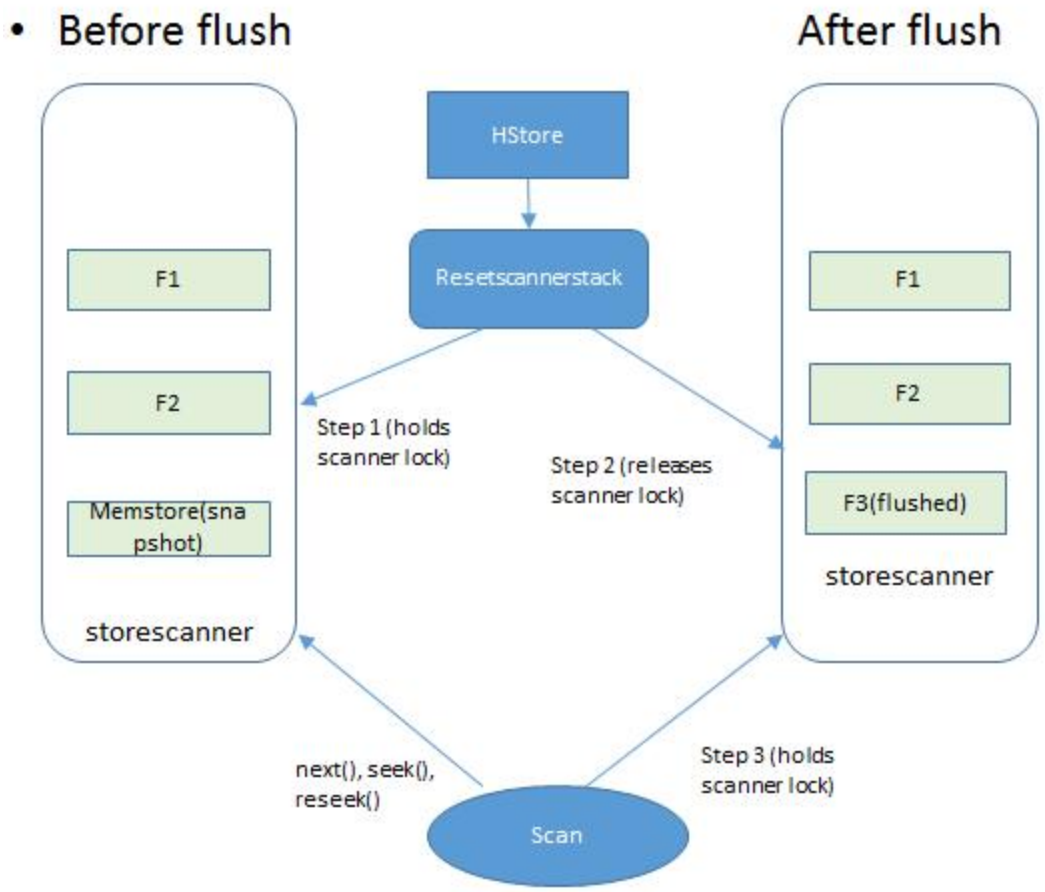


In the above diagram, it can be seen that a scan operation on the files F1, F2, F3 and F4 before compaction, still continues to operate on the same files F1, F2, F3 and F4 though F5 is a newly compacted file but the scanner does not make use of that new file F5. When a new scan is created ‘scan2’, it can be seen that only the files F4 and F5 are used whereas the files F1 to F3 are not included since they are already compacted.

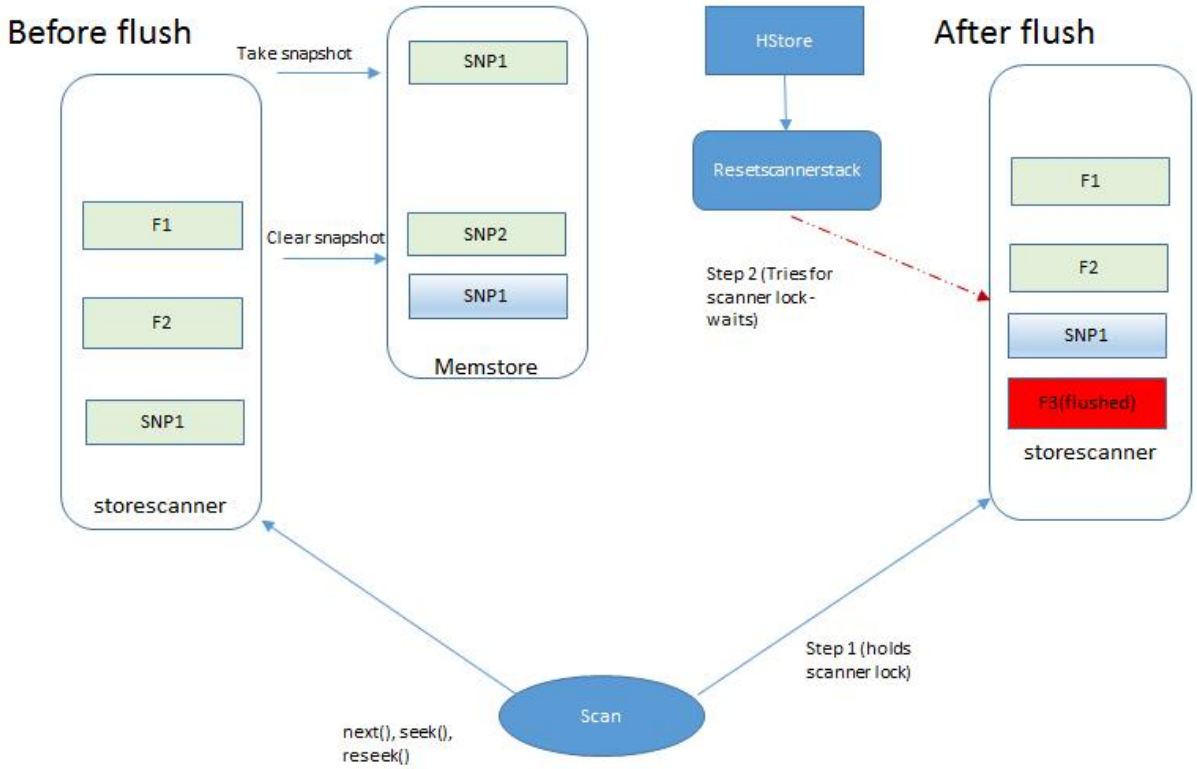
Once the Cleaner thread runs in the back ground it can be seen that the compacted files from F1 to F3 are removed from the store files list itself.

Note : The D indicates the DISCARDED state on those files that are compacted

Current approach for flushing if reset happens before scanner lock



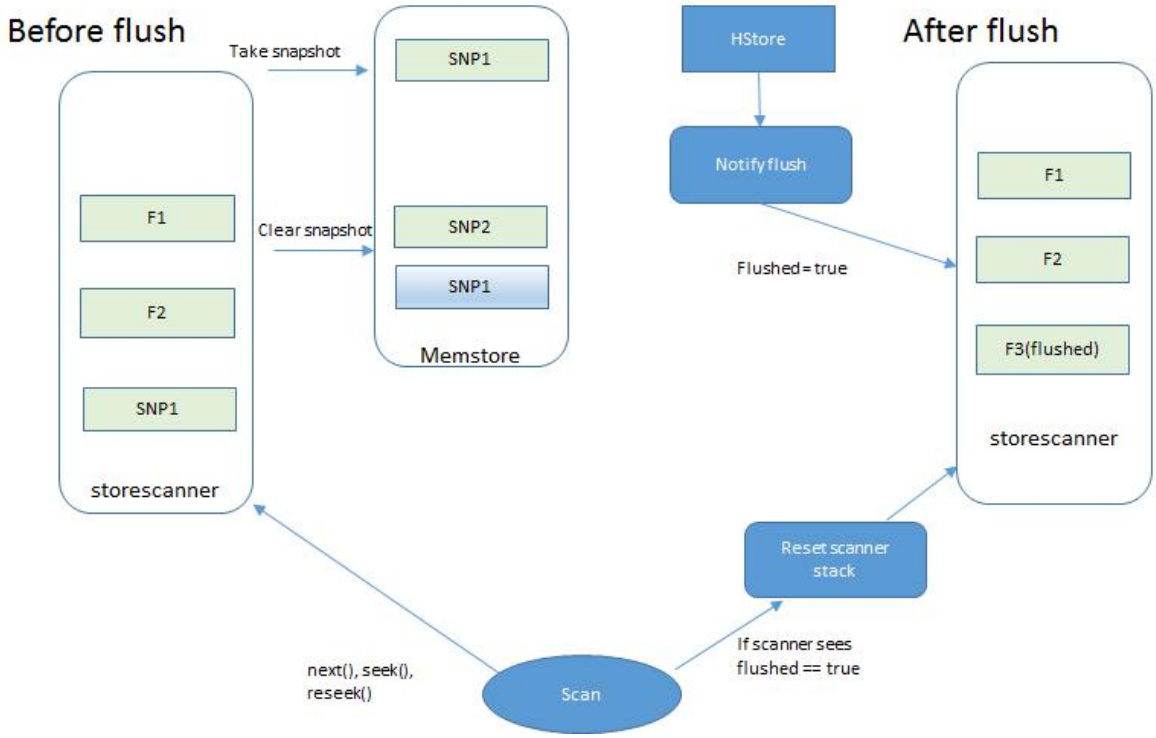
Current approach for flushing if scanner locks before reset happens



Here you can see that the SNAPSHOT SNP1 though it got cleared and a new snapshot was created SNP2, still the ongoing scanner that is holding the lock will still go on with SNP1 snapshot till the reset scanner stack happens for which the current scanner should release the lock. The GC cannot release the reference of SNP1 because of the ongoing scanner.

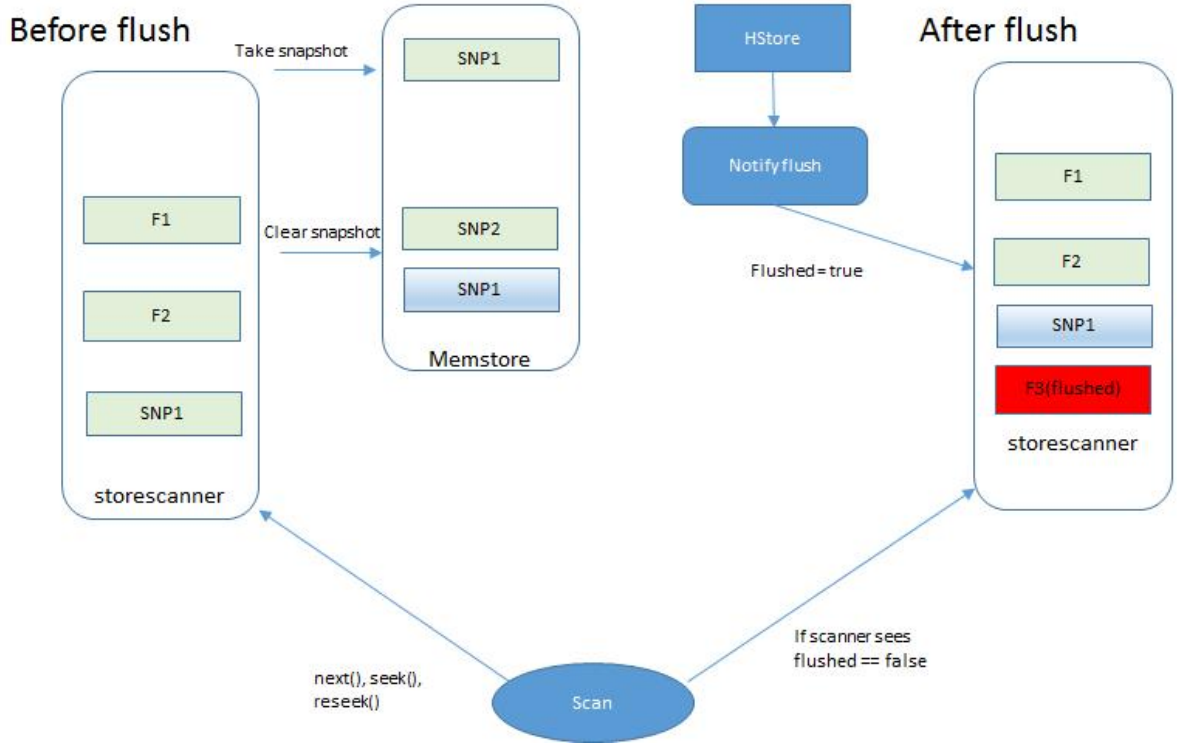
Versioned approach

If scanner notices that a flush has happened



Here you can see that the notifier tries to set a flag saying that the flush has happened. If the current scan before entering its APIs like next(), reseek(), seek() checks if the flag is set, if so the scanner itself resets the scanner stack and goes ahead with the scan with the new set of files created out of flush. So there is no external thread clearing off the stack the scan is currently using.

If scanner does not notice that a flush has happened



So this is some what similar to what happens in the current approach when the notifier is not able to reset the scanner stack. Here again the scan goes with the SNP1 snapshot that is still referenced by the GC and only after the current operation is completed the current scan could see that the flush has happened and reset its scanner stack.