Pluggable compactions

Table of Contents

Motivation

In HBase, compaction is the process that aims to reduce the number of files used to store data. HBase writes new data into a sequence of files (separate for each region of key space of each column family), never changing existing files; thus, over time the number of files increases, making reads less efficient, and adding overhead. Compaction solves this problem by rewriting several files from a sequence into a single file. This, however, consumes a lot of resources, especially I/O, causing performance and stability issues.

There's a balance between compaction efficiency (as we rewrite the files again and again, I/O is multiplied), the number of files that are read for gets/scan, and avoiding disruption caused by large compactions. It is hard to achieve and may not be the same for all scenarios and datasets. Thus, a lot of ideas are proposed with regard to compaction design.

We want to enable experimentation in this area, and enable HBase users to choose algorithms that are good for their data. Thus, compactions should be pluggable.

At least for the initial cut, until we have fleshed out the interfaces and have 2-3 compaction algorithms implemented inside HBase, this extension point will not be fully public - it would be usable by people providing implementations outside HBase code base but not supported in the same way as, for example, coprocessors.

Existing code layout

As described, compaction is essentially the process of choosing some files, rewriting them and putting the result back. Currently, one compaction proceeds as follows:

- HStore orchestrates the compaction process when requested;
- HStore also manages store files. HStore is a class with many responsibilities; the default store file management is simple, so it's embedded there;
- CompactionPolicy receives a pre-filtered set of files from HStore and selects the files to compact;
- Compactor compacts the files, and they go back into HStore.

How we will make things pluggable

An important note

Changing StoreFile format is out of the scope of this feature. Metadata may be added to files; however making store file format an extension point is a maintainability nightmare.

Proposed code layout changes

As a result of recent refactoring-s, CompactionPolicy and Compactor are already at a relatively good level of abstraction. However, there's still some stuff remaining in HStore that should properly be a part of policy (such as figuring out the valid compaction based on already-compacting files, etc.; **HBASE-7784**). This will need to move into policy.

Then, there's a question of handling additional metadata that some compaction schemes may use, and logic changes outside of compactions (for example, the split point based on the biggest file doesn't make sense for level or stripe compactions). It's possible to retain the list of files inside HStore, with stores like stripe and level using it directly, and deriving the custom structure dynamically from it; however this would make them hacky and easy to break by making some assumptions about the file list in other HStore code; it can also preclude optimizations that could be achieved by alternative data structures. Thus, we should refactor store file management out of HStore to be behind an interface (**StoreFileManager, aka SFM; HBASE-7603**). The implementation would manage the structure between multiple StoreFile, not their internals or physical location. The default implementation will just maintain the list, as HStore does now.

Finally, there's a question of how to make it all pluggable. The default approach would be to make each class pluggable via reflection and configuration. However, in the schemes like stripe and level, file management, policy and compactor only work together, e.g. level store with default policy will not work. Thus, all 3 need to be created in one place. On the other hand, for default store file management, it makes sense to have compactor and policy in various combinations (e.g. tiered policy with rest default, or improved compactor with rest default). Thus, we don't want any of the three creating the others. Thus, we will have a separate class to create all three (**StoreEngine; HBASE-7678)**. To make it easy to combine things, we want to keep it very simple – just a factory. StoreEngine is pluggable and contains the methods to create StoreFileManager, CompactionPolicy and Compactor.

Technical details of targeting compaction implementations

StoreEngine class will be pluggable based on a configuration setting and reflection, as is the tradition. Given that it's a factory it can be created in HStore initialization, used and discarded. If we want more flexibility we may have a default StoreEngine that separately allows pluggable policy, for example. This can be added trivially when needed.

To be able to configure custom compactions per table/CF, per table/CF configuration can be used. This is a recently added feature - see **HBASE-7571**.

In all cases, changing the setting for an existing table or column family will require restarting all region servers, until we have dynamic configuration sorted out in general.

New interface details

StoreEngine has 3 logical operations that create 3 respective objects. The ctor for implementations will accept the objects necessary for their creation (at this point, HStore, Configuration and KVComparator).

SFM has the following logical operations:

1. Initialize - add the initial set of files loaded by HStore for the region+CF.
2. Add new files to the structure, after memstore flush/bulk add.
3. Remove some files and add other files, after compaction.
4. De-initialize - remove and return all the files.
5. Get all files.
6. Get the files to be read for a particular get or scan request.
   Different from (4) for schemes such as stripe and level.
7. Get the files to be read for a particular "key before" request, optionally with some existing candidate. Different from (4) for schemes such as stripe and level.
8. Get the approximate size-weighted middle of the key range (split point).
   Alternatively, we could do a universal implementation of this in HStore; for that, StoreFileManager-s would have to produce size function over key range in some general form. We will not go there for the time being.

Migration between different schemes

In the cases where store file management is not affected (e.g. tier-based), there will be no migration as such; the default SFM will just load files. Otherwise, the following will apply.

From default to custom: the custom SFM is supposed to know how to convert the default files into its structure (e.g. for stripe/level – add to L0). If some future implementation cannot do it, it should produce an error causing the region open to fail (and document it).

From custom to default: Some of the currently envisioned SFMs (e.g. stripe) maintain the file relations that can be ignored without repercussions, and thus the files can be loaded by the default SFM without problems (other than temporary perf hit). In other cases (e.g. level) the files cannot be loaded by the default SFM without breaking some assumptions. To solve this problem, we can add "breaking StoreEngine class name" to file metadata. Then, either HStore reads this field and fails to open region with a different StoreEngine, so the conversion back to default is impossible; or HStore creates both StoreEngine-s, orders the old Compactor to compact all data into one file/compatible files (Compactor will have to handle this special case correctly), and gives the files to new one. It is only necessary for some schemes (e.g. level), so **HBASE-TBD.**

Between schemes: if the old scheme is compatible with default scheme - same as "default to custom"; otherwise, same as "custom to default".

Examples

Default scheme will use all the default implementations. Other existing ideas/examples are:

- **Tier-based** compaction only changes the selection process, keeping other things intact; thus, it will have StoreEngine that returns default store file manager, default compactor and custom policy.
- **Stripe and level** compaction require custom SFMs that maintain the files according to stripes/levels; compactor that can output files with specific boundaries; and policy that chooses the files based on specific structure. Thus, StoreEngine for it will return all custom components. Internally, policy will be able to make assumptions about having the correct file manager, for example, and fail if the wrong one is provided.