Stripe compactions

sershe, 2012-04-30, HBASE-7667

Table of Contents

Motivation

- HBase compactions, especially major compactions, can currently adversely affect HBase performance by creating a large (and prolonged) amount of IO. Because of that, they are generally managed manually.
- Major compactions are currently a necessity in HBase, due to requirement of dropping deletes; large compactions of most/all of the data are also necessary to prevent the accumulation of large number of files. At the same time, for some scenarios, such as incremental log data, there is no reason to be compacting data for certain (old) keys.

Design alternatives

The straightforward approach to spread compactions over time, and to compact only part of the key range, is splitting the key range into smaller pieces. One could create a table with large number of smaller regions (HBASE-7630), however, in current HBase, there are several problems with that; the number of regions supported is limited. Therefore, other schemes, internal to Region or Store, can be used to have smaller compactions.

We have considered the LevelDB scheme (HBASE-7519) for some time, and determined that it may cause large I/O amplification; there are also concerns about how to drop delete markers in this scheme, and lack of compatibility of current file ordering by seqNum with level files (if level files are loaded with default engine, inconsistencies are possible). None of these are showstoppers, but they suggest that a better solution might be possible.

The proposed solution is a combination of LevelDB ideas with many-regions ideas; essentially it creates multiple regions within one large region, but limits the scope to compactions only.

High-level description

Stripe is defined as a sub-range of the region key range.

For each Store (column family), the region key range is split into stripes according to some rule. After the key range is split this way, the boundaries rarely, if ever, move. The stripes always cover the entire key range of the region with no gaps. Within each stripe, there's a sequence of StoreFile-s that only have data for the corresponding sub-range of the key range. These files can be compacted together, and separately from all other files.

To avoid the creation of many separate files for each stripe during memstore flush, reduce the scope of the change (to exclude memstore), and to support bulk load, in the initial design there will also be certain files that contain keys from the full region key range. These files are called "level 0", by analogy with similar files in LevelDB (see also future improvements about pre-striping data at split time).

Thus, memstores are flushed to L0, and then several L0 files are rewritten into stripes. Stripe data is compacted according to the ratio rules similar to default compaction. The files are, in general case, never compacted together, thus one gets more of the smaller compactions; if the load is not uniform by key, the compaction will also happen earlier for keys with more data.

At the same time, we don't significantly increase the number of files used for each get request (and scans that are not overly wide), because only the files within one stripe and L0 are used for reads.

Also, we don't get nearly as much I/O amplification as level scheme, can read the files written with stripes safely via standard compaction scheme (with regard to seqNum ordering), and can drop deletes without doing major compaction.

Flavors of stripes

This discusses the approaches to splitting a region into stripes.

Count-based: "mini-regions"

The first approach is based on the number of stripes, and follows the same logic as giving the user the ability to split table into a number of regions.

Fixed number of stripes is maintained in each store; thus, it can be configured on table or column family level. Stripes are assumed to stripe the key range into segments of roughly the same size (see below about the special cases).

Size-based: sequential data

This approach is based on assuming the data keys are in roughly increasing order, so the old data is rarely (or never) updated. The variable number of stripes is maintained. As some stripes gets too big, they are split into multiple stripes of some target size; stripes containing only expired files are removed as necessary.

Maintaining/getting store files

Files for the new scheme are maintained by stripe, with end keys stored for each stripe except the last and open-ended key being assumed for the start of first, and end of last, stripe. When the compactor writes the files, stripe boundaries (determined dynamically during compaction, or supplied externally) are written to the store file metadata. The boundaries are deterministic and the same for all files in a stripe, i.e. we don't just store file key range and derive the boundaries.

Loading files on Store initialization

During loading, the stripe metadata is read and stripes are reconstructed. Files without metadata are assumed to be L0 (so, loading current store files should "just work"). If there are problems in stripes (gaps, overlaps, etc.; for example, files with

metadata from previous usage of stripes that was switched to default and then to stripes), loading code tries its best to place files into some stripes, and place subset of files into L0 for re-compactions; at worst, if stripes are too broken, everything goes to L0.

For these cases, when we choose to ignore file metadata, we store this decision in an in-memory structure by store file, so that we could continue ignoring it in future.

Adding files after memstore flush, bulk load, compactions

Initially, after memstore flush and for bulk load, files are just added to L0.

For compactions, a set of files produced by compactor replaces a set of files selected by policy. We maintain the following restrictions on the process:

- No compaction may produce L0 files (opposite makes no sense).
- No compaction may produce more than one file in any stripe (opposite makes no sense, and this simplifies implementation).
- Compaction must produce at least one file (see HBASE-6059).
- The union of stripe key ranges of files going into compaction must be contiguous (initial implementation restriction, probably makes no sense to do otherwise).
- If the stripe boundaries are changed by compaction:
    - the boundaries of the union of stripe key ranges of files going in, and boundaries of the union of stripe key ranges produced by compaction, should match (otherwise there will be overlap with neighboring stripes not affected by this compaction);
    - there should be no gaps in the union of ranges of the files produced by compaction (otherwise there will be gaps in newly creates stripes);
    - all files in the stripes being replaced should be removed by compaction (otherwise there's overlap between files from old stripes with old boundaries and files from new stripes, with different boundaries).
- No stripe can ever be left with 0 files (initial implementation restriction, may make sense to improve later to avoid empty files that can appear in some rare cases).

Getting files

To get the files from the stripes structure without copying a bunch of lists, we return a list of lists Collection implementation. Some improvements with pre-caching lists, or using other structures, can be added later if necessary.

Gets and scans

When a get or scan request arrives, we determine the stripes that might have the data based on key range, and only return store files from these stripes.

Get key before

This operation is mostly only used for META, so it might not be relevant. Still, we need to implement it.

Key before a given key can exist in a large set of stripes, half on average. We return the files in order of L0 followed by stripes from the beginning to the one containing the requisite key K in reverse order, each reverse ordered by seqNum. Every time HStore finds a candidate key Kc in some file, we update the list by removing the files for all the stripes that cannot possibly contain any keys in (Kc, K] range.

Determining split point

For obvious reasons, getting the mid-point of the biggest file no longer works for the split point if there's more than one stripe.

To get the split point, we walk the stripe list from both sides, trying to maintain balanced sides based on file size. If the resulting sides are sufficiently balanced, we return the stripe boundary between them (in future, this can also be used to improve splits since many files will only be used for one of the resulting regions). Otherwise, and if that would improve the size ratio, we return the midpoint from the larger of the bordering stripes.

Compaction selection and special cases

Stripe compaction mechanism includes several compaction types that the compaction policy is able to select. Currently, they are selected in strict order of preference (e.g. re-striping L0 compaction, if needed, will always go before single-stripe compaction). This can be improved later if needed (i.e. we can see which type is relatively more urgent).

Single-stripe compaction

Single-stripe compactions are the lowest priority, and most straightforward compactions. They operate on one stripe, applying the ratio rule similar to the default compaction algorithm, and have ratio and minimum/maximum file counts to compact. Some files are taken and replaced by one file in the same stripe and range.

In case of size-based scheme, given that we expect the last stripe to be eventually split (i.e. compacted fully), we exclude this stripe from regular compaction.

Striping the data (L0)

Striping (L0) compactions are the most important ones (after some special cases the rewrite entire data, see below), because they remove L0 files – files that have to be read for all requests. The only exception is when we want to both compact L0 and drop deletes (see below). This compaction is performed when the number of L0 files exceeds some threshold. It produces the number of files equivalent to the number of stripes (one file for each stripe, with appropriate key ranges).

When there are no stripes (new region):

- For count-based stripes, we determine the stripe boundaries via first compaction, trying to produce the requisite number of roughly equal-sized files from compaction. To get more accurate boundaries, we get more data by using an increased file count threshold for first re-striping.
- For size-based stripes, we treat L0 like a "boundary" stripe that we have to split (see "splitting off new stripe" below), and split into some number of stripes with a certain size target.

Size-based – splitting off a new stripe

In the size-based stripe scheme, occasionally we need to split a new stripe, as the amount of data increases. This is done based on the size of the stripes. When the stripe crosses the threshold (target stripe size multiplied by a multiplier) and is eligible for compaction, it is assume that it is the one receiving data, and it's compacted and split into "old" stripes with majority of data, and new stripe with smaller amount of data.

The end result of a series of such compactions is an expanding of stripes with approximately equal size in one file each (plus potentially some small files for out-of-order keys, which may cause some inline compactions).

Handling region splits (references)

For the first cut, after the split we perform the compaction of all the files, similarly to the common case. This can obviously be improved since most (or all) stripes will end up on one side of the split; see future improvements.

Dropping deletes

Without major compaction, we cannot drop deletes during compaction unless we are somehow sure that we will see all KVs for some key range, or out-of-order puts are not a concern. Therefore, we will drop deletes in the following cases:

- When we compact all files (creating new stripes, after split, etc.).
- When we compact one entire stripe (or set of stripes), and the configuration setting to assume out-of-order puts don't matter is set. Currently in HBase there's a short time window during which out of order puts can disappear and reappear in get/scan results due to major compaction/memstore interaction (or rather, lack thereof). This setting will make this window wider, but if users don't use out of order puts it doesn't matter.
- When the compaction that would compact one entire stripe (or set of stripes without changing boundaries), is possible, and L0 compaction is possible at the same time. In this case the former will be performed, and L0 files will be added to it to allow dropping deletes (since we will be compacting all data for some range). Deletes will only be dropped from the range of the former compaction, because for the data outside of this range we obviously don't have all files.

Count-based – improved splits, rebalancing stripes

There are two special cases that can arise for count-based stripes:

Number of stripes different from target due to split or configuration change

Obviously, if we go from say 6 to 3 stripes after split, it's easy to split 3 stripes back into 6; moreover, it may not even be necessary to do it immediately. However, as the splits, the main cause of this special case, currently use references (see "Handling splits"), for the first cut this special case is not smart – it simply compacts all files, just like the split handing.

Stripes are unbalanced

Due to uneven load on different keys, stripes can become unbalanced. Unbalanced stripes don't necessarily hurt us, whereas moving stripe boundary requires that 2 (or more) stripes be fully re-written together. Therefore, the threshold for unacceptable lack of balance is set very high (3 times difference between the stripe size, and the average size, as measured by KV count). In case if it's exceeded, a number of stripes (capped by a configuration setting, default 2) are compacted together, and the boundary is moved. We try to compact the minimum number of stripes that is expected to bring the imbalance within threshold.

Size-based – merging old stripes

There is one special case that can arise for size-based stripes: too many stripes. By default, in this case we assume the region needs to be split, since it has too much data to adhere to stripe target size. However, if the region contains sequential data, it is possible that some data has already expired. In that case, stripes containing only expired storefiles are merged with each other, or with next stripe, to reduce the number of stripes.

Compactor

Stripe compactor supports two types of compaction. They can probably be reused for any compaction schemes like Level/etc. with similar properties.

In each case, the range from which deletes can be dropped (if any) is supplied to compactor, and it creates StoreScanner/ScanQueryMatcher with corresponding range.

In each case, the stripe boundaries are output into file metadata.

Boundary-based compaction

Boundary-based compaction arranges data into pre-existing stripes.

Fixed row boundaries are determined by the policy. Compactor maintains one writer at a time, and outputs data to it until it reaches the next boundary. For empty ranges, empty files are created.

Size-based compaction (not to be confused with size-based stripes)

Size-based compaction arranges the data into new stripes, determining boundaries.

Two boundaries that cover all the supplied files, as well as target number of KVs and target file count are supplied. Compactor maintains one writer at a time, and outputs data to it until it has the sufficient number of KVs according to target, or indefinitely if the number of writers already created is equal to target file count. It also makes sure all rows for one row key are written to the same writer. The first row key in each new writer becomes the boundary between two writers (future stripes).

Potential future improvements

The subsections of this section could be expanded.

KVs from scanner

Compactor currently estimates the boundaries of the files that it writes under "size-based compaction" based on the number of KVs it wrote, whereas the target size given to it is based on KVs in the source. If many KVs in the source are dropped, this can cause skew. We could take KV counts from scanner to be more precise about how many KVs we have processed.

Getting rid of L0

L0 introduces x2 write amplification for any data, affecting write performance (at least). We can use custom memstore flusher to flush files directly into multiple files in the stripes. L0 will still be preserved for loading non-striped files and bulk-loaded files.

Split improvements

Split could be improved using stripe boundaries. If most or all of the files end up on a particular side of the split end up in only one region, theoretically no references or major compaction is necessary – files should be moved to their target regions, making splitting very fast.

Currently this would break snapshots though, as these rely on references.

Also, boundaries are on store level, so only biggest CF will get full benefits (other CFs will likely have one stripe that has to be split); mini-region like structure, or some other coordination on region level could improve that.

Improvement to changing number of stripes

Related to split improvements, it is not necessary to compact all files if number of stripes does not match config (due to split, or config change). Stripes could be gradually split/merged depending on size. Due to splits relying on references now it has limited practical use before split improvements (config change only).

Inferring stripes from key ranges or storing metadata outside

It is not strictly necessary to write stripes to metadata. Stripes could be inferred from key ranges of the files during load and merging of compaction results. L0 files that belong to particular range could be placed in that stripe immediately then.

Otherwise, stripe metadata (and other metadata) could be stores outside of the store file (for example, file name, or separate file). That would allow moving files about the structure without rewriting them.

Parallel compactions in CF

Parallel compactions in Store should be allowed. However, as noted in the JIRA, some throttling/coordination would be necessary in this case to keep the advantages.

"Mini-regions"

The sub-region boundaries could be added on region level, providing some additional benefits (for example for splits).