The Key is further decomposed as:

- rowlength
- row (i.e., the rowkey)
- columnfamilylength
- columnfamily
- columnqualifier
- timestamp
- keytype (e.g., Put, Delete, DeleteColumn, DeleteFamily)

KeyValue instances are *not* split across blocks. For example, if there is an 8 MB KeyValue, even if the block-size is 64kb this KeyValue will be read in as a coherent block. For more information, see the KeyValue source code.

9.7.6.6.1. Example

To emphasize the points above, examine what happens with two Puts for two different columns for the same row:

- Put #1: `rowkey=row1, cf:attr1=value1`
- Put #2: `rowkey=row1, cf:attr2=value2`

Even though these are for the same row, a KeyValue is created for each column:

Key portion for Put #1:

- rowlength `------------> 4`
- row `----------------> row1`
- columnfamilylength `---> 2`
- columnfamily `--------> cf`
- columnqualifier `------> attr1`
- timestamp `-----------> server time of Put`
- keytype `-------------> Put`

Key portion for Put #2:

- rowlength `------------> 4`
- row `----------------> row1`
- columnfamilylength `---> 2`
- columnfamily `--------> cf`
- columnqualifier `------> attr2`
- timestamp `-----------> server time of Put`
- keytype `-------------> Put`

It is critical to understand that the rowkey, ColumnFamily, and column (aka columnqualifier) are embedded within the KeyValue instance. The longer these identifiers are, the bigger the KeyValue is.

9.7.6.7. Compaction

When the MemStore reaches a given size (`hbase.hregion.memstore.flush.size`), it flushes its contents to a StoreFile. The number of StoreFiles in a Store increases over time. *Compaction* is an operation which reduces the number of StoreFiles, by merging them together, in order to increase performance on read operations. Compactions can be resource-intensive to perform, and can either help or hinder performance depending on many factors.

Compactions fall into two categories: minor and major. Minor and major compactions differ in the following ways.

Minor compactions usually select a small number of small, adjacent StoreFiles and rewrite them as a single StoreFile. Minor compactions do not drop (filter out) deletes or expired versions, because of potential side effects. See Compaction and Deletions and Compaction and Versions for information on how deletes and versions are handled in relation to compactions. The end result of a minor compaction is fewer, larger StoreFiles for a given Store.

The end result of a *major compaction* is a single StoreFile per Store. Major compactions also process delete markers and max versions. See Compaction and Deletions and Compaction and Versions for information on how deletes and versions are handled in relation to compactions.

Compaction and Deletions. When an explicit deletion occurs in HBase, the data is not actually deleted. Instead, a *tombstone* marker is written. The tombstone marker prevents the data from being returned with queries. During a major compaction, the data is actually deleted, and the tombstone marker is removed from the StoreFile. If the deletion happens because of an expired TTL, no tombstone is created. Instead, the expired data is filtered out and is not written back to the compacted StoreFile.

Compaction and Versions. When you create a Column Family, you can specify the maximum number of versions to keep, by specifying `HColumnDescriptor.setMaxVersions(int versions)`. The default value is `3`. If more versions than the specified maximum exist, the excess versions are filtered out and not written back to the compacted StoreFile.

Major Compactions Can Impact Query Results

In some situations, older versions can be inadvertently resurrected if a newer version is explicitly deleted. See Section 5.9.2.2, "Major compactions change query results" for a more in-depth explanation. This situation is only possible before the compaction finishes.

In theory, major compactions improve performance. However, on a highly loaded system, major compactions can require an inappropriate number of resources and adversely affect performance. In a default configuration, major compactions are scheduled automatically to run once in a 7-day period. This is sometimes inappropriate for systems in production. You can manage major compactions manually. See Section 2.5.2.8, "Managed Compactions".

Compactions do not perform region merges. See Section 17.2.2, "Merge" for more information on region merging.

9.7.6.7.1. Compaction Policy - HBase 0.96.x and newer

Compacting large StoreFiles, or too many StoreFiles at once, can cause more IO load than your cluster is able to handle without causing performance problems. The method by which HBase selects which StoreFiles to include in a compaction (and whether the compaction is a minor or major compaction) is called the *compaction policy*.

Prior to HBase 0.96.x, there was only one compaction policy. That original compaction policy is still available as `RatioBasedCompactionPolicy` The new compaction default policy, called `ExploringCompactionPolicy`, was subsequently backported to HBase 0.94 and HBase 0.95, and is the default in HBase 0.96 and newer. It was implemented in HBASE-7842. In short, `ExploringCompactionPolicy` attempts to select the best possible set of StoreFiles to compact with the least amount of work, while the `RatioBasedCompactionPolicy` selects the first set that meets the criteria.

Regardless of the compaction policy used, file selection is controlled by several configurable parameters and happens in a multi-step approach. These parameters will be explained in context, and then will be given in a table which shows their descriptions, defaults, and implications of changing them.

9.7.6.7.1.1. Being Stuck

When the MemStore gets too large, it needs to flush its contents to a StoreFile. However, a Store can only have `hbase.hstore.blockingStoreFiles` files, so the MemStore needs to wait for the number of StoreFiles to be reduced by one or more compactions. However, if the MemStore grows larger than `hbase.hregion.memstore.flush.size`, it is not able to flush its contents to a StoreFile. If the MemStore is too large and the number of StpreFo;es is also too high, the algorithm is said to be "stuck". The compaction algorithm checks for this "stuck" situation and provides mechanisms to alleviate it.

9.7.6.7.1.2. The ExploringCompactionPolicy Algorithm

The ExploringCompactionPolicy algorithm considers each possible set of adjacent StoreFiles before choosing the set where compaction will have the most benefit.

One situation where the ExploringCompactionPolicy works especially well is when you are bulk-loading data and the bulk loads create larger StoreFiles than the StoreFiles which are holding data older than the bulk-loaded data. This can "trick" HBase into choosing to perform a major compaction each time a compaction is needed, and cause a lot of extra overhead. With the ExploringCompactionPolicy, major compactions happen much less frequently because minor compactions are more efficient.

In general, ExploringCompactionPolicy is the right choice for most situations, and thus is the default compaction policy. You can also use ExploringCompactionPolicy along with Section 9.7.6.7.6, "Experimental: Stripe Compactions".

The logic of this policy can be examined in `hbase-`

server/src/main/java/org/apache/hadoop/hbase/regionserver/compactions/ExploringCompactionPolicy.java. The following is a walk-through of the logic of the ExploringCompactionPolicy.

1. Make a list of all existing StoreFiles in the Store. The rest of the algorithm filters this list to come up with the subset of HFiles which will be chosen for compaction.
2. If this was a user-requested compaction, attempt to perform the requested compaction type, regardless of what would normally be chosen. Note that even if the user requests a major compaction, it may not be possible to perform a major compaction. This may be because not all StoreFiles in the Column Family are available to compact or because there are too many Stores in the Column Family.
3. Some StoreFiles are automatically excluded from consideration. These include:
    - StoreFiles that are larger than hbase.hstore.compaction.max.size
    - StoreFiles that were created by a bulk-load operation which explicitly excluded compaction. You may decide to exclude StoreFiles resulting from bulk loads, from compaction. To do this, specify the hbase.mapreduce.hfileoutputformat.compaction.exclude parameter during the bulk load operation.
4. Iterate through the list from step 1, and make a list of all potential sets of StoreFiles to compact together. A potential set is a grouping of hbase.hstore.compaction.min contiguous StoreFiles in the list. For each set, perform some sanity-checking and figure out whether this is the best compaction that could be done:
    - If the number of StoreFiles in this set (not the size of the StoreFiles) is fewer than hbase.hstore.compaction.min or more than hbase.hstore.compaction.max, take it out of consideration.
    - Compare the size of this set of StoreFiles with the size of the smallest possible compaction that has been found in the list so far. If the size of this set of StoreFiles represents the smallest compaction that could be done, store it to be used as a fall-back if the algorithm is "stuck" and no StoreFiles would otherwise be chosen. See Section 9.7.6.7.1.1, “Being Stuck”.
    - Do size-based sanity checks against each StoreFile in this set of StoreFiles.
        - If the size of this StoreFile is larger than hbase.hstore.compaction.max.size, take it out of consideration.
        - If the size is greater than or equal to hbase.hstore.compaction.min.size, sanity-check it against the file-based ratio to see whether it is too large to be considered. The sanity-checking is successful if:
            - There is only one StoreFile in this set, or
            - For each StoreFile, its size multiplied by hbase.store.compaction.ratio (or hbase.hstore.compaction.ratio.offpeak if off-peak hours are configured and it is during off-peak hours) is less than the sum of the sizes of the other HFiles in the set.
5. If this set of StoreFiles is still in consideration, compare it to the previously-selected best compaction. If it is better, replace the previously-selected best compaction with this one.
6. When the entire list of potential compactions has been processed, perform the best compaction that was found. If no StoreFiles were selected for compaction, but there are multiple StoreFiles, assume the algorithm is stuck (see Section 9.7.6.7.1.1, “Being Stuck”) and if so, perform the smallest compaction that was found in step 3.

9.7.6.7.1.3. RatioBasedCompactionPolicy Algorithm

The RatioBasedCompactionPolicy was the only compaction policy prior to HBase 0.96, though ExploringCompactionPolicy has now been backported to HBase 0.94 and 0.95. To use the RatioBasedCompactionPolicy rather than the ExploringCompactionPolicy, set hbase.hstore.defaultengine.compactionpolicy.class to RatioBasedCompactionPolicy in the hbase-site.xml file. To switch back to the ExploringCompactionPolicy, remove the setting from the hbase-site.xml.

The following section walks you through the algorithm used to select StoreFiles for compaction in the RatioBasedCompactionPolicy.

1. The first phase is to create a list of all candidates for compaction. A list is created of all StoreFiles not already in the compaction queue, and all StoreFiles newer than the newest file that is currently being compacted. This list of StoreFiles is ordered by the sequence ID. The sequence ID is generated when a Put is appended to the write-ahead log (WAL), and is stored in the metadata of the HFile.
2. Check to see if the algorithm is stuck (see Section 9.7.6.7.1.1, “Being Stuck”, and if so, a major compaction is forced. This is a key area where Section 9.7.6.7.1.2, “The ExploringCompactionPolicy Algorithm” is often a better choice than the RatioBasedCompactionPolicy.
3. If the compaction was user-requested, try to perform the type of compaction that was requested. Note that a major compaction may not be possible if all HFiles are not available for compaction or if too may StoreFiles exist (more than hbase.hstore.compaction.max).

4. Some StoreFiles are automatically excluded from consideration. These include:

    - StoreFiles that are larger than `hbase.hstore.compaction.max.size`
    - StoreFiles that were created by a bulk-load operation which explicitly excluded compaction. You may decide to exclude StoreFiles resulting from bulk loads, from compaction. To do this, specify the `hbase.mapreduce.hfileoutputformat.compaction.exclude` parameter during the bulk load operation.

5. The maximum number of StoreFiles allowed in a major compaction is controlled by the `hbase.hstore.compaction.max` parameter. If the list contains more than this number of StoreFiles, a minor compaction is performed even if a major compaction would otherwise have been done. However, a user-requested major compaction still occurs even if there are more than `hbase.hstore.compaction.max` StoreFiles to compact.

6. If the list contains fewer than `hbase.hstore.compaction.min` StoreFiles to compact, a minor compaction is aborted. Note that a major compaction can be performed on a single HFile. Its function is to remove deletes and expired versions, and reset locality on the StoreFile.

7. The value of the `hbase.store.compaction.ratio` parameter is multiplied by the sum of StoreFiles smaller than a given file, to determine whether that StoreFile is selected for compaction during a minor compaction. For instance, if hbase.store.compaction.ratio is 1.2, FileX is 5 mb, FileY is 2 mb, and FileZ is 3 mb:

    ```
    5 <= 1.2 x (2 + 3)           or           5 <= 6
    ```

    In this scenario, FileX is eligible for minor compaction. If FileX were 7 mb, it would not be eligible for minor compaction. This ratio favors smaller StoreFile. You can configure a different ratio for use in off-peak hours, using the parameter `hbase.hstore.compaction.ratio.offpeak`, if you also configure `hbase.offpeak.start.hour` and `hbase.offpeak.end.hour`.

8. If the last major compaction was too long ago and there is more than one StoreFile to be compacted, a major compaction is run, even if it would otherwise have been minor. By default, the maximum time between major compactions is 7 days, plus or minus a 4.8 hour period, and determined randomly within those parameters. Prior to HBase 0.96, the major compaction period was 24 hours. See `hbase.hregion.majorcompaction` in the table below to tune or disable scheduled major compactions.

9.7.6.7.1.4. Parameters Used by Compaction Algorithm

This table contains the main configuration parameters for compaction. This list is not exhaustive. To tune these parameters from the defaults, edit the `hbase-default.xml` file. For a full list of all configuration parameters available, see Section 2.3, “Configuration Files”

Parameter	Description	Default
hbase.hstore.compaction.min	The minimum number of StoreFiles which must be eligible for compaction before compaction can run. The goal of tuning `hbase.hstore.compaction.min` is to avoid ending up with too many tiny StoreFiles to compact. Setting this value to `2` would cause a minor compaction each time you have two StoreFiles in a Store, and this is probably not appropriate. If you set this value too high, all the other values will need to be adjusted accordingly. For most cases, the default value is appropriate. In previous versions of HBase, the parameter `hbase.hstore.compaction.min` was called `hbase.hstore.compactionThreshold`.	3
hbase.hstore.compaction.max	The maximum number of StoreFiles which will be selected for a single minor compaction, regardless of the number of eligible StoreFiles. Effectively, the value of `hbase.hstore.compaction.max` controls the length of time it takes a single compaction to complete. Setting it larger means that more StoreFiles are included in a compaction. For most cases, the default value is appropriate.	10
	A StoreFile smaller than this size will always be eligible for minor compaction. StoreFiles this size or larger are evaluated by `hbase.store.compaction.ratio` to determine if they are eligible.	

hbase.hstore.compaction.min.size	Because this limit represents the "automatic include" limit for all StoreFiles smaller than this value, this value may need to be reduced in write-heavy environments where many files in the 1-2 MB range are being flushed, because every StoreFile will be targeted for compaction and the resulting StoreFiles may still be under the minimum size and require further compaction. If this parameter is lowered, the ratio check is triggered more quickly. This addressed some issues seen in earlier versions of HBase but changing this parameter is no longer necessary in most situations.	128 MB
hbase.hstore.compaction.max.size	An StoreFile larger than this size will be excluded from compaction. The effect of raising `hbase.hstore.compaction.max.size` is fewer, larger StoreFiles that do not get compacted often. If you feel that compaction is happening too often without much benefit, you can try raising this value.	Long.MAX_VALUE
hbase.hstore.compaction.ratio	For minor compaction, this ratio is used to determine whether a given StoreFile which is larger than `hbase.hstore.compaction.min.size` is eligible for compaction. Its effect is to limit compaction of large StoreFile. The value of `hbase.hstore.compaction.ratio` is expressed as a floating-point decimal. A large ratio, such as `10`, will produce a single giant StoreFile. Conversely, a value of `.25`, will produce behavior similar to the BigTable compaction algorithm, producing four StoreFiles. A moderate value of between 1.0 and 1.4 is recommended. When tuning this value, you are balancing write costs with read costs. Raising the value (to something like 1.4) will have more write costs, because you will compact larger StoreFiles. However, during reads, HBase will need to seek through fewer StpreFo;es to accomplish the read. Consider this approach if you cannot take advantage of Section 14.6.4, "Bloom Filters". Alternatively, you can lower this value to something like 1.0 to reduce the background cost of writes, and use Section 14.6.4, "Bloom Filters" to limit the number of StoreFiles touched during reads. For most cases, the default value is appropriate.	1.2F
hbase.hstore.compaction.ratio.offpeak	The compaction ratio used during off-peak compactions, if off-peak hours are also configured (see below). Expressed as a floating-point decimal. This allows for more aggressive (or less aggressive, if you set it lower than `hbase.hstore.compaction.ratio`) compaction during a set time period. Ignored if off-peak is disabled (default). This works the same as `hbase.hstore.compaction.ratio`.	5.0F
hbase.offpeak.start.hour	The start of off-peak hours, expressed as an integer between 0 and 23, inclusive. Set to `-1` to disable off-peak.	-1 (disabled)
hbase.offpeak.end.hour	The end of off-peak hours, expressed as an integer between 0 and 23, inclusive. Set to `-1` to disable off-peak.	-1 (disabled)

hbase.regionserver.thread.compaction.throttle	There are two different thread pools for compactions, one for large compactions and the other for small compactions. This helps to keep compaction of lean tables (such as `hbase:meta`) fast. If a compaction is larger than this threshold, it goes into the large compaction pool. In most cases, the default value is appropriate.	2 x hbase.hstore.compaction.max x hbase.hregion.memstore.flush.size (which defaults to 128)
hbase.hregion.majorcompaction	Time between major compactions, expressed in milliseconds. Set to 0 to disable time-based automatic major compactions. User-requested and size-based major compactions will still run. This value is multiplied by `hbase.hregion.majorcompaction.jitter` to cause compaction to start at a somewhat-random time during a given window of time.	7 days (604800000 milliseconds)
hbase.hregion.majorcompaction.jitter	A multiplier applied to `hbase.hregion.majorcompaction` to cause compaction to occur a given amount of time either side of `hbase.hregion.majorcompaction`. The smaller the number, the closer the compactions will happen to the `hbase.hregion.majorcompaction` interval. Expressed as a floating-point decimal.	.50F

9.7.6.7.2. Compaction File Selection

Legacy Information

This section has been preserved for historical reasons and refers to the way compaction worked prior to HBase 0.96.x. You can still use this behavior if you enable Section 9.7.6.7.1.3, "RatioBasedCompactionPolicy Algorithm" For information on the way that compactions work in HBase 0.96.x and later, see Section 9.7.6.7, "Compaction".

To understand the core algorithm for StoreFile selection, there is some ASCII-art in the Store source code that will serve as useful reference. It has been copied below:

```
/* normal skew:
 *
 *         older ----> newer
 *
 *     _
 *    | |   _
 *    | |  | |   _
 *  --|-|- |-|- |-|---_-------_-------  minCompactSize
 *    | |  | |  | |  | | _ | |
 *    | |  | |  | |  | ||| | |
 */
```

Important knobs:

- `hbase.store.compaction.ratio` Ratio used in compaction file selection algorithm (default 1.2f).
- `hbase.hstore.compaction.min` (.90 hbase.hstore.compactionThreshold) (files) Minimum number of StoreFiles per Store to be selected for a compaction to occur (default 2).
- `hbase.hstore.compaction.max` (files) Maximum number of StoreFiles to compact per minor compaction (default 10).
- `hbase.hstore.compaction.min.size` (bytes) Any StoreFile smaller than this setting with automatically be a candidate for compaction. Defaults to `hbase.hregion.memstore.flush.size` (128 mb).
- `hbase.hstore.compaction.max.size` (.92) (bytes) Any StoreFile larger than this setting with automatically be excluded from compaction (default Long.MAX_VALUE).

The minor compaction StoreFile selection logic is size based, and selects a file for compaction when the file <= sum(smaller_files) * `hbase.hstore.compaction.ratio`.

9.7.6.7.3. Minor Compaction File Selection - Example #1 (Basic Example)

This example mirrors an example from the unit test `TestCompactSelection`.

- `hbase.store.compaction.ratio` = 1.0f
- `hbase.hstore.compaction.min` = 3 (files)
- `hbase.hstore.compaction.max` = 5 (files)