Cassandra Storage Format Proposal for C* 3.1

Version 2

Benedict Elliott Smith

January 2015

Abstract

This document is a draft proposal for a generalised replacement of the sstable format used in Cassandra. The goal is to deliver a framework building on top of a number of abstracted components, with which any efficient sstable representation can be encapsulated. It is designed to directly align with modern CQL data modelling, however it should also support thrift models at least as efficiently as the current state of art.

1 Components

First we will outline the abstract definitions of the components we will use in this framework, the expectations we have of them, and the manner in which they will be used.

PartitionIndex

One per sstable. Performs a translation from TOKEN($PartitionKey$) to a set of addresses in a $DataFile$ that may contain data for the key. This translation is permitted to yield false positives, but should expect to produce either zero or one result on average, with no false negatives. Multiple keys may be stored at the same address in the $DataFile$, which can be exploited by an implementation to produce a more compact mapping. These keys will generally be adjacent, but some may be out-of-order due to page packing (see Extension Two). Persistence is managed by the implementation itself.

function LOOKUP($PartitionIndex, Token$)
return All positions possibly containing $PartitionKey$
function RANGE ($PartitionIndex, StartToken, EndToken$)
return $Page \mid \forall(Token \mapsto Page) \in PartitionIndex, StartToken <= Token < EndToken$

RowIndex

One per partition. Located by a $PartitionIndex$ lookup, it stores a translation from *Clustering Columns* (here on referred to as $RowKey$) to both a positional key into the $ValueStore$ and a $RowEpoch$. It also stores range tombstones. It will be necessary to support a variable number of clustering columns so that range deletes will be supported when the functionality is delivered. This may also be used to encode collection keys.

The contract with the framework is to manage a sequence of bytes that contain its data, indexed from zero. The framework manages the actual persistence of this byte sequence, and the location of the zero position. In actual implementation it may be helpful to abstract this sequence of bytes to a sequence of pages to help efficient indexing of large partitions, but we leave this complexity out of this document.

function READ($RowIndex, RowKey$)
return $\langle RowEpoch$, Positional key in $ValueStore$ for $RowKey\rangle \mid DeletedTimestamp$

ValueStore

One per partition. A sequence of bytes that combines the position retrieved from the $RowIndex$ with the columns to be queried and the $RowEpoch$ to return cell values. We assume here that, prior to insertion of a new row, a property Pos can be queried, returning the key to store in $RowIndex$ for future retrieval of the row we are about to insert. Like $RowIndex$, it is treated as a zero-indexed sequence of bytes, the persistence of which is dealt with by the framework.

function READ($ValueStore, ColIds, Pos, RowEpoch$)
return Cells for the $ColIds$ persisted by $ValueStore$ at positional key Pos

2 Implementation Details

The components defined in the prior section should compose reasonably naturally, as far as intuition is concerned. Here we attempt to define it more concretely, with some both descriptive and mathematically explicit pseudocode describing how they can be stitched together to form a generalised storage framework.

2.1 Notation Key

$X \Leftarrow Y$	Write/Add/Append Y To X
$X \leftarrow\!\!\text{-}\!\!- Y$	Read X From Y
$X \leftarrow Y$	Set X to Y
$\langle A, B \rangle$	A tuple containing the values A and B
$\{Item \mid Constraint\}$	The set of all $Item$ produced by $Constraint$
S_N	$\equiv \{S_n \mid n \in N\}$
$Stmt_x \mid \forall x \in X$	Execute statement S for each x in X
$\lvert X \rvert$	$\equiv \mathbf{card}(X)$, i.e. the number of elements in X

2.2 Core Algorithm

Definitions

- *PartitionEpoch*, *RowEpoch*

  Against each tier of our structure we will encode the minimum timestamp found in the next tier, so that each may be encoded more efficiently. The expectation is that this will permit many cells to persist their timestamp using only a few bytes, a single bit, or no data at all

- *DataFile*

  A persistent byte storage medium, supporting append and buffered sequential and random reads

- *ColId*

  A translation from column to a unique integer id in the range $[0..N)$

Algorithm 1 Core Algorithm - Descriptive

```
1: procedure WRITE(Partitions)
2:     Map each column to a unique integer id in the range [0..N)
3:     for all Partitions do
4:         Extract the PartitionEpoch
5:         for all Rows in the Partition do
6:             Extract the RowEpoch, subtracting it from each cell
7:             Add the ValueStore position/key and RowEpoch to RowIndex
8:             Add the cells to the ValueStore
9:         Add everything to our Buffer
10:        if Buffer is full then
11:            Add the partition keys to our PartitionIndex, mapping to our page index
12:            Write the tokens in our buffer (fixed-width)
13:            Write each partition's offset within the page (fixed-width)
14:            Write each partition's data

15: function READ(PartitionKey, Cols, RowKeys)
16:     Map Cols to ColIds using our metadata
17:     for all Pages associated with TOKEN(PartitionKey) in PartitionIndex do
18:         Binary search for TOKEN(PartitionKey)
19:         for all Matching tokens do
20:             Find and seek to the associated data offset
21:             if The key matches PartitionKey then
22:                 Read the PartitionEpoch
23:                 Initialize RowIndex and ValueStore to their starting positions
24:                 for all Matching rows in RowIndex do
25:                     if Is DeletedTimestamp then
26:                         Add the range tombstone to our response
27:                     else
28:                         Read the associated RowEpoch and Pos
29:                         Provide these and ColIds to ValueStore
30:                         Add the result to our response
                    return Constructed response
        return nil
```

Algorithm 2 Core Algorithm - Precise

```
1: procedure WRITE(Partitions)
2:     PartitionIndex, DataFile ← NEW
3:     ColId ← {c ↦ i | ∀c, d ∈ Columns, d < c ⟹ d ↦ j ∧ j < i}
4:     Metadata ⇐ ColId

5:     for p ← Partitions do
6:         RowIndex, ValueStore ← NEW
7:         PartitionEpoch ← min{v.timestamp | v ∈ r.Values, r ∈ p.Rows}

8:         for r ← p.Rows do
9:             RowEpoch ← min{v.timestamp | v ∈ r.Values}
10:            EpochDelta ← RowEpoch − PartitionEpoch
11:            RowIndex ⇐ ⟨r.RowKey ↦ ⟨EpochDelta, ValueStore.Pos⟩⟩
12:            Values ← {ColId(c) ↦ r.Values(c) | c ∈ Columns}
13:            v.Time ← v.Time − RowEpoch | ∀v ∈ ran(Values)
14:            ValueStore ⇐ Values

15:        Buffer ⇐ ⟨p.PartitionKey, PartitionEpoch, RowIndex, ValueStore, DataFile.Pos⟩
16:        Size ← DataFile.pos − min_{∀b∈Buffer} b.Pos
17:        if Size ≥ PageSize then
18:            FLUSH(Flush, PartitionIndex, DataFile)
19:            Buffer ← NEW
20:    FLUSH(Buffer, PartitionIndex, DataFile)

21: procedure FLUSH(Buffer, PartitionIndex, DataFile)
22:    PartitionIndex ⇐ ⟨TOKEN(PartitionKey)↦ DataFile.Pos⟩ | ∀PartitionKey ∈ Buffer
23:    DataFile ⇐ |Buffer|
24:    DataFile ⇐ TOKEN(PartitionKey) | ∀⟨PartitionKey, _, _, _⟩ ∈ Buffer⟩
25:    Offset ← 0
26:    for ⟨PartitionKey, _, _, _⟩ ← Buffer do
27:        DataFile ⇐ Offset
28:        Offset ← Offset+ SIZEOF(⟨PartitionKey, RowIndex, ValueStore,EPOCH,{ADDRESS}⟩)
29:    for ⟨PartitionKey, PartitionEpoch, RowIndex, ValueStore⟩ ← Buffer do
30:        DataFile ⇐ ⟨PartitionEpoch, PartitionKey⟩
31:        DataFile ⇐ DataFile.Pos+ SIZEOF(RowIndex) + SIZEOF({ADDRESS})
32:        DataFile ⇐ RowIndex
33:        DataFile ⇐ ValueStore

34: function READ(PartitionKey, Cols, RowKeys)
35:    ColGroup, ColId ⇠-- Metadata
36:    Token ← TOKEN(PartitionKey)
37:    CandidatePages ←LOOKUP(PartitionIndex, Token)
38:    for Page ← CandidatePages do
39:        Count ⇠-- DataFile[Page]
40:        TokenBase ← Page+ SIZEOF(Count)
41:        OffsetsBase ← TokenBase + Count×SIZEOF(TOKEN)
42:        for i ← FIND(DataFile[TokenBase . . . OffsetBase), Token) do
43:            OffsetPosition ← OffsetsBase + i× SIZEOF(OFFSET)
44:            Offset ⇠-- DataFile[OffsetPosition]
45:            PartitionEpoch, CandidateKey, Pos ⇠-- DataFile[Offset]
46:            if PartitionKey = CandidateKey then
47:                Offset ← Offset+ SIZEOF(⟨PartitionEpoch, CandidateKey, Pos⟩)
48:                RowIndex ← DataFile[Offset]
49:                return BUILD(RowKeys, PartitionEpoch, ReadGroup, ColId, RowIndex, Pos, DataFile)
    return nil

50: function BUILD(RowKeys, PartitionEpoch, ColId, Pos, DataFile)
51:    Partition ← NEW
52:    ValueStore ⇐ DataFile[Pos]
53:    for RowKey ← RowKeys do
54:        RowInfo ← READ(RowIndex, RowKey)
55:        if RowInfo ∈ Timestamp then
56:            Partition ⇐ RowInfo
57:        else
58:            Pos, EpochDelta ← RowInfo
59:            RowEpoch ← PartitionEpoch + EpochDelta
60:            Partition ⇐ READ(ValueStore, ColId(Columns), Pos, RowEpoch)
    return Partition
```

2.3 Extension One - Column Groupings

When reading data from an sstable, fetching data for any single column necessitates also reading any columns that were written near-in-time, or have been compacted together. For tables with many columns this can result in suboptimal query performance when reading only a subset of the columns.

By permitting fields to be grouped for purposes of persistence, we can pack more tightly together data that may be accessed in isolation within a large partition, permitting range queries over these columns to incur fewer IOPs. Conversely, rarely accessed fields can be separated into their own group to avoid their polluting all other common queries.

Within the framework we also support multiple schemes for value encoding and retrieval, which may fit different columns stored in the table differently, or may only support a subset of the columns on a table. For example columnar or hybrid value storage may be ideal for densely populated non-collection timeseries fields. This extension permits us to use these value stores for such fields and row-oriented for the remainder; or any other optimal configuration.

At the same time, this approach permits us to separate values from the partition index in situations we expect many intra-partition range queries, so that the partition indexes can expect better cache occupancy. It also supports duplicating columns to meet multiple different access patterns with minimal IO, at the expense of write costs.

Definitions

- (Column) $Group_N$

  N groupings of columns, each containing a subset of the total columns defined on the table, that will be persisted together in a corresponding $DataFile_N$. By default there will be one group containing all columns, but rarely accessed columns or columnar layouts may be separated into their own group.

- $ColId_N$

  A translation from column to a unique integer id within each $Group_n$

Components

- $ValueStore_N$

  One per $Group_N$, per partition. Otherwise as defined originally.

- $DataFile_N$

  One per $Group_N$, per partition, storing the associated $ValueStore_N$, with $DataFile_0$ storing our $RowIndex$ and partition-wide data.

Algorithm 3 Column Groupings

procedure WRITE($Partitions, Group_N$)
  $PartitionIndex, Buffer \leftarrow$ NEW
  $DataFile_n \leftarrow$ NEW $\mid \forall n \in N$

  $ColGroup \leftarrow \{c \mapsto n \mid \forall c \in Group_n, \forall n \in N\}$
  $ColId_n \leftarrow \{c \mapsto i \mid \forall c, d \in G, d < c \implies d \mapsto j \wedge j < i\} \mid \forall n \in N, G \equiv Group_n$
  $Metadata \Leftarrow ColGroup$
  $Metadata \Leftarrow ColId$

  for $p \leftarrow Partitions$ **do**
    $RowIndex \leftarrow$ NEW
    $ValueStore_n \leftarrow$ NEW $\mid \forall n \in N$
    $PartitionEpoch \leftarrow \min\{v.timestamp \mid v \in r.Values, r \in p.Rows\}$

    for $r \leftarrow p.Rows$ **do**
      $RowEpoch \leftarrow \min\{v.timestamp \mid v \in r.Values\}$
      $EpochDelta \leftarrow RowEpoch - PartitionEpoch$
      $RowIndex \Leftarrow \langle r.RowKey \mapsto \langle EpochDelta, \{v.Pos \mid v \in ValueStore_N\}\rangle\rangle$
      $Values_n \leftarrow \{ColId(c) \mapsto r.Values(c) \mid c \in Group_n\} \mid \forall n \in N$
      $v.Time \leftarrow v.Time - RowEpoch \mid \forall v \in \mathbf{ran}(Values_n), \forall n \in N$
      $ValueStore_n \Leftarrow Values_N$

    $Pos_n \leftarrow DataFile_n.Pos | \forall n \in N$
    $Buffer \Leftarrow \langle p.PartitionKey, PartitionEpoch, RowIndex, ValueStore_N, Pos_N\rangle$
    $Size \leftarrow \sum_{n=0}^{N} (DataFile_n.pos - \min_{\forall b \in Buffer} b.Pos_n)$
    if $Size \geq N \times PageSize$ **then**
      $Flush \leftarrow$ SELECT($Buffer$)
      FLUSH($Flush, PartitionIndex, DataFile_N$)
      $Buffer \leftarrow$ NEW
  FLUSH($Buffer, PartitionIndex, DataFile_N$)

procedure FLUSH($Buffer, PartitionIndex, DataFile_N$)
  $PartitionIndex \Leftarrow \langle$TOKEN($PartitionKey$)$\mapsto DataFile_0.Pos\rangle \mid \forall PartitionKey \in Buffer$
  $DataFile_0 \Leftarrow |Buffer|$
  $DataFile_0 \Leftarrow$ TOKEN($PartitionKey$) $\mid \forall\langle PartitionKey, \_, \_, \_\rangle \in Buffer\rangle$
  $Offset \leftarrow 0$
  for $\langle PartitionKey, \_, \_, \_\rangle \leftarrow Buffer$ **do**
    $DataFile_0 \Leftarrow Offset$
    $Offset \leftarrow Offset +$ SIZEOF($\langle PartitionKey, RowIndex, ValueStore_0,$ EPOCH, $\{$ADDRESS$^N\}\rangle$)
  for $\langle PartitionKey, PartitionEpoch, RowIndex, ValueStore_N\rangle \leftarrow Buffer$ **do**
    $DataFile_0 \Leftarrow \langle PartitionEpoch, PartitionKey\rangle$
    $Pos_n \leftarrow DataFile_n.Pos \mid \forall n \in N$
    $Pos_0 \leftarrow Pos_0 +$ SIZEOF($RowIndex$) + SIZEOF($\{$ADDRESS$^N\}$)
    $DataFile_0 \Leftarrow Pos_N$
    $DataFile_0 \Leftarrow RowIndex$
    $DataFile_n \Leftarrow ValueStore_n \mid \forall n \in N$
    Pad $DataFile_n$ to a page boundary $\mid \forall n \in N$

Algorithm 3 Column Groupings (contd.)

```
1: function Read(PartitionKey, Cols, RowKeys)
2:     ColGroup, ColId_N <-- Metadata
3:     ReadGroup_N ← ReadGroups(Cols, GolGroup_N)
4:     Token ← Token(PartitionKey)
5:     CandidatePages ←Lookup(PartitionIndex, Token)
6:     for Page ← CandidatePages do
7:         Count <-- DataFile_0[Page]
8:         TokenBase ← Page+ SizeOf(Count)
9:         OffsetsBase ← TokenBase + Count×SizeOf(Token)
10:        for i ← Find(DataFile_0[TokenBase ... OffsetBase), Token) do
11:            OffsetPosition ← OffsetsBase + i× SizeOf(Offset)
12:            Offset <-- DataFile_0[OffsetPosition]
13:            PartitionEpoch, CandidateKey, Pos_N <-- DataFile_0[Offset]
14:            if PartitionKey = CandidateKey then
15:                Offset ← Offset+ SizeOf(⟨PartitionEpoch, CandidateKey, Pos_N⟩)
16:                RowIndex ← DataFile_0[Offset]
17:                return Build(RowKeys, PartitionEpoch, ReadGroup_N, ColId_N, RowIndex, Pos_N, DataFile_N)
    return nil

18: function Build(RowKeys, PartitionEpoch, ReadGroup_N, ColId_N, Pos_N, DataFile_N)
19:     Partition ← New
20:     ValueStore_n ⇐ ⟨DataFile_n[Pos_n]⟩ | ∀n ∈ N
21:     for RowKey ← RowKeys do
22:         Row ← New
23:         Pos_N, EpochDelta ← Read(RowIndex, RowKey)
24:         RowEpoch ← PartitionEpoch + EpochDelta
25:         for n ← N : ReadGroup_n ≠ ∅ do
26:             Row ⇐ Read(ValueStore, ColId_n(ReadGroup_n), Pos_n, RowEpoch)
27:         Partition ⇐ Row
    return Partition
```

2.4 Extension Two - Partition Reordering

An sstable represents an ordered collection of partitions, however this doesn't require the data be in order on disk. By permitting records to appear out-of-order we can reduce the number of records that cross a page boundary, reducing the number of IOPs needed per query. This should result in increased throughput and reduced maximum latencies. For linear scan performance we need ordered traversal, however this only necessitates records occurring *almost* in order. We define a $BufferLimit$ which bounds the amount of buffer space any reader can be required to have to produce results in order. Flushing enforces this by buffering some multiple K of $BufferLimit$, and selecting from this buffer a well packed page that favours records earlier in the stream, while leaving a good distribution of records to help pack future pages.

Algorithm 4 Partition Reordering

procedure WRITE($Partitions, Group_N$)
    ...

    for $p \leftarrow Partitions$ **do**
        ...
        $Distance \leftarrow \sum_{n=0}^{N} (DataFile_n.pos - \min_{\forall b \in Buffer} b.Pos_n)$
        if $Distance \geq K \times BufferLimit$ **then**
            $Flush \leftarrow$ SELECT($Buffer$)
            FLUSH($Flush, PartitionIndex, DataFile_N$)
            $Buffer \leftarrow Buffer \setminus Flush$

function SELECT($Buffer$)
    $Selection \leftarrow ?$
Ensure: $Selection \subseteq Buffer$
Ensure: $s \in Selection \cdot s.Pos = \min_{\forall b \in Buffer} b.Pos_n$
Ensure: $(\exists k \cdot \sum_{s \in Selection} s.Size_n \approx k \times PageSize) \mid \forall n \in N$
Ensure: Future calls to SELECT can also meet criteria
    return $Selection$

procedure SCAN($StartToken, EndToken, Cols, RowKeys, Out$)
    $ColGroup, ColId_N \leftarrow\!\!- Metadata$
    $Read_N \leftarrow$ READGROUPS($Cols, GolGroup_N$)
    for $Page \leftarrow$RANGE($PartitionIndex, Start, End$) **do**
        $Count \leftarrow\!\!- DataFile_0[Page]$
        $TokenBase \leftarrow Page +$ SIZEOF($Count$)
        $OffsetsBase \leftarrow TokenBase + Count \times$SIZEOF(TOKEN)
        for $0 \leq i < Count$ **do**
            $TokenPosition \leftarrow BasePos + i \times$ SIZEOF(TOKEN)
            $Token \leftarrow DataFile_0[TokenPosition]$
            if $StartToken \leq Token < EndToken$ **then**
                $OffsetPosition \leftarrow OffsetsBase + i \times$ SIZEOF(OFFSET)
                $Offset \leftarrow\!\!- DataFile_0[OffsetPosition]$
                $PartitionEpoch, PartitionKey, Pos_N, RowIndex \leftarrow\!\!- DataFile_0[Offset]$
                $Buffer \Leftarrow$ BUILD($RowKeys, PartitionEpoch, ReadGroup_N, ColId_N, RowIndex, Pos_N, DataFile_N$)
                $BufferSize \leftarrow$ SIZEOF($Buffer$)
                if $BufferSize \geq BufferLimit$ **then**
                    $Out \Leftarrow \min(Buffer)$
                    $Buffer \leftarrow Buffer / \min(Buffer)$

2.5 Extension Two Alternative - Partition Redirection

An alternative to the approach just outlined, but achieving the same ends, is producing multiple sstables at flush time, of logarithmically smaller size. When flushing we write the contents of our buffer in order, but omit a subset that permit us to pack our pages better. Any omitted are siphoned into their own buffer for a smaller sstable, on which the logic is repeated (until returns are minimal). The advantage of this approach is it may combine well with techniques we will outline later for managing hash partition indexes, and does not require partition index implementations to cope with out-of-order records.

One slight further variant on this approach is to write all of these pages to the same sstable, persisting somewhere which subset of pages contain "out-of-order" records - really a parallel order. This could meet the constraints imposed by *Partition Reordering* by ensuring at most $\frac{BufferLimit}{PageSize}$parallel-order pages must be retained by any scanner. This also avoids $PartitionIndex$ support for out-of-order records, but requires a separate bloom filter and partition index for each parallel order.

2.6 Notes

Large Partitions

These pseudocode implementations have ignored certain complexities of dealing with very large partitions, and assumed we can assemble the entire partition in memory. This was to keep the complexity of our pseudocode to a minimum. To solve this problem, partitions with a $RowIndex$ spread across many pages can be filtered into a separate file, with $Group_0$ being shifted to $Group_{-1}$, so that no values are stored alongside the $RowIndex$. Thus we do not need to know its length, since there is no $ValueStore_0$ to locate. Since only one partition will be serialized we also do not need to know the position of the next $RowIndex$.

For most sstables, a majority of records will fall one side of this barrier, so there should not be a significant increase in sstable count. Since this will only be performed for large partitions spread over multiple pages, there will also be no IO penalty.

Alternatively, if providing a page-oriented byte layer abstraction, on encountering an oversized record we can immediately flush the contents of our write buffer, then interleave serialization of $RowIndex$ and $ValueStore_0$ pages for the large partition.

$RowKey$ Abstraction

For simplicity of presentation we have assumed a set of $RowKey$ will be provided for any query. In reality range slices will be common. Behaviour is not meaningfully different, and clarity is improved by making this assumption for presentation.

Dynamic Block Size

At a minimum each file should select a block size optimal for the storage medium and the data distribution being written to it. For SSDs this should mean a block size

just large enough to fit a single partition, or just its row index, if such a block size exists; otherwise, a 4Kb block size (also our minimum block size). For spinning disks, a larger block size can be used unconditionally; say 32-64Kb, as is currently the case. We should consider/explore using a dynamic block size within each file.

Decoupling of $ValueStore_N$ from $DataFile_N$

We have assumed each $ValueStore_n$ has its own $DataFile_n$, but there is no requirement that this holds true, nor that a selection of alike $ValueStore$ could not be grouped and indexed by the same $RowIndex$ entry. Pure column oriented value stores could all be grouped in this manner. These are simple additions that only complicate the pseudocode unnecessarily.

Collections

Collections have not been dealt with at all directly in this document. We will touch on them briefly here, and outline some of the ways they can be dealt with.

1. *RowIndex* Presence

   Perhaps the most obvious approach would be to continue the current paradigm of treating the collection key as a suffix of the clustering data. While $RowIndex$ implementations will need to support variable length prefixes of clustering data, supporting variable length suffixes may be an unwelcome burden, and this would also require some special treatment in row-oriented $ValueStore$ to reduce the amount of row metadata.

2. Direct Value Encoding

   Alternatively, we could persist the entire collection as a dynamic width field in the row. For small collections this is likely optimal in both performance and implementation complexity. The data could be stored using a $RowIndex$ implementation whose zero index is the dynamic column offset.

3. Indirect Value Encoding

   If the collection is large, we could instead store a fixed offset to a page in the $DataFile_0$, or some special spill-over $DataFile_S$ where the contents are stored. In implementation there would be little difference between this and *Direct Value Encoding*; both can utilise a $RowIndex$ implementation, and only vary the locality of storage.

3 Component Variants

Here we will outline the main variants for each component, and some of the details of their implementation.

3.1 Partition Index

The partition index most likely needs just two implementations, although it may be that subtle variations of each are tried over time. The implementations depend on the nature of the partitioning scheme, which fall into two categories: Ordered and Hashed.

Ordered

The obvious choice is a B$^+$-tree, although since our data is static some alternatives are viable. Since this partitioning scheme is uncommon, however, it may not be worth investing labour in a major new approach, and a simplification of the current scheme may be sensible, with some modifications to match our new layout.

Since each page has its own internal micro index, our macro index need only store the first token occurring in each page. A summary index can be constructed that does the same for the index itself, within which we binary search for the index record to consult. This summary can reside in memory, directing us to an index page within which we perform a binary search to find the data page. This translates directly to the sstable representing the bottom three levels of a B$^+$-tree, with the highest remaining resident in memory as our summary, and the middle being the highly cacheable index. For small datasets we could omit the summary level and store the entire index in memory, since the required space for the same page size will be significantly lower than the current approach. Only 8 bytes are necessary per page, instead of 16 bytes + $\textsc{Len}(PartitionKey)$ per partition. If we pack P partitions per page, this translates to a minimum fraction of $\frac{1}{2P}$, with $\frac{1}{4P}$for $UUID$ partition keys.

Hashed

The basic building block for hash partitioning will be the same as *Ordered*, however we can exploit the random data distribution to improve the computational complexity of lookup and the space required, both in memory and on disk.

The basic intuition to exploit is that, given a uniform token distribution and a known start/end for an sstable (CASSANDRA-6696) being helpful here), we can calculate the average boundary token that can be expected for each page. This expectation will typically be quite wrong, but the *error* will be distributed over a small range, and this can be exploited in three ways:

1. **Hash Indexed Summary**

   Instead of performing binary search on the summary, we can predict the correct summary page based on the expected distribution of data, at write-time ensuring this is never incorrect by more than a predetermined bound. A bounded number of linear probes from the predicted location within this page can then be performed to find the desired token boundary, or a binary search can be performed over a known maximal range either side of the predicted position.

2. **Hash Indexed Index**

   We can eliminate the summary entirely, instead padding the index file so that our expected boundaries for the index file are always correct, requiring exactly one seek into the index file. The amount of padding necessary will typically be small, but if it is large this approach can be abandoned in favour of another.

3. **Probabilistic Index Search**

   We can alternatively eliminate the summary by predicting a location to begin our index search in, along with an expected range to binary search around, precomputed at write-time. This range will contain the desired page mapping

with $P(X) \geq Limit$, for some configured $Limit$. If the search fails we expand the area by a similar logic. This yields a predictable performance characteristic with fixed memory constraints.

4. **Golomb Encoded Page Boundaries**
   We can potentially eliminate both the summary and the index. Instead of encoding our page boundaries as a list of tokens, we can instead encode as the distance between the real token boundary of any two pages and the one we predict. The error will be distributed around zero, following a geometric distribution, and so can be stored efficiently with Golomb codes with a constant multiplier. This permits tunable accuracy, yielding some $P(X) \geq Limit$ that we will select the correct page; if we do not find the correct page, we can be certain the adjacent page is correct, so that on average each query performs $2 - Limit$ seeks. The level of compression achieved is dependent on this $Limit$, and the distribution of the data. These two variables combine to require a scale of multiplier; the multiplier scale in turn determines the range of Golomb codes needed. The smaller $Limit$, or better distributed the data, the larger the viable multiplier and hence the smaller the Golomb codes. This technique needs some exploration, but some simple simulation suggests viability beyond the theoretical.

   This technique may compose well with *Extension Two - Partition Redirection*, as we could select records to persist based both on page density and page boundary.

Out of Order Records

If we opt to support out-of-order persistence, a generalized pre-index may be useful, in which we store the few tokens we have permitted to be re-ordered. Lookups hit this index first, and on finding no results fallback to the normal index.

3.2 Row Index

The Row Index has by far the most scope for variation, so we will only touch briefly on the various categories here, with a high level overview of their implications. We will mention retrieval and merge performance, by intuitive description of characteristics only.

Entry Per Cell

This isn't really possible in the new world, but is worth discussing for comparison since it describes the current state-of-art. Here we have the entire clustering prefix repeated for each cell. The cost of merging is as suboptimal as possible:

1. There are $\times|Columns|$ more items to merge than necessary
2. Shared clustering prefixes must be compared in full, so each comparison is costlier
3. All rows must be compared; there is no pruning of known disjoint descendant sets

Linear Collection

This is the closest to the current scenario, except that we only repeat the data once per row, not per cell. The idea would be to store each complete clustering prefix in sequence, and linearly scan for the relevant record. For very large partitions this would not support any true indexing, but it is very simple. For small partitions it would be acceptable, and is optimal for single rows. The cost of merging is improved by eliminating (1).

Column Trie, Linear Internal Collections

The next simplest approach is to split each clustering column into its own set of linear sequences of data, so that with multiple clustering columns we do not repeat data present in higher tiers. This permits data compression. Merging is also improved by significantly reducing the effects of (2) and (3). For partitions whose clustering column tiers are each smaller than a single page this is a fairly optimal approach as binary search can be performed on each tier as you descend.

Column Trie, BTree Internal Collections

When the trie levels are larger than a page, we need to introduce paging, and to do this a BTree makes perfect sense. This is still a very general collection, since any kind of comparison can be performed on each BTree item, so it can support all current and custom data types. This optimisation permits further improvement to (3) for merging, as well as optimal search costs for custom data types and those not supporting binary prefix comparison.

Binary Trie

If the clustering prefixes can be compared by any binary prefix, a binary trie can be used. This permits optimal behaviour for (2) and (3), making merges extremely cheap as the number of rows and partitions grow, which is likely to be of significant benefit, given how CPU constrained some users are on these costs. This also permits superior data compression. There are a number of possible binary trie variants to explore, but this warrants a separate document or JIRA ticket to discuss the options.

3.3 Value Store

The value store likely has only two major variants, and a hybrid between them: column- and row-oriented. There is no requirement that a given table subscribe to one or the other, however. Within a single sstable there can be a mix, with each field grouping selecting its own value persistence approach.

Row Oriented

This most closely resembles the current storage, except that we consider each row a discrete unit and encode certain information prefixing it to permit indexed access within the row, and to permit compression of the structural data.

- $HasCell$ Bitmap

  Each column that occurs at least once in the file will have an index in this bitmap, which encodes if the column appears in this row, to permit efficient indexing within the row. It may be that some columns appear in every row, and these may be encoded in the file metadata to remove them from the row bitmaps.

- $HasTimestamp$ Bitmap

  Each column with its bit set in $HasCell$ will have an index in $HasTimestamp$; this will indicate if there is any timestamp offset necessary from *RowEpoch*; a value of zero indicates *RowEpoch* is enough by itself to construct the cell timestamp.

- $HasValue$ Bitmap

  Each column with its bit set in $HasCell$ will have an index in $HasValue$; this will indicate if there is any actual data associated, or if the "value" is a tombstone.

- $HasExpiry$ Bitmap

  Each column with its bit set in $HasValue$ will have an index in $HasExpiry$; this will indicate if there is an expiry associated with the value.

- $HasInfo$ Bitmap

  A bitmap indicating if $HasCell$, $HasTimestamp$, $HasValue$ or $HasExpiry$ are necessary to encode.

- $TimestampLength$

  If any bits are set in $HasTimestamp$, this will encode how many bytes are needed to encode them. Each will be encoded with the same width. This and $HasInfo$ can be encoded in the same byte.

- $Timestamps$

  A fixed-width array of timestamps for each column marked in $HasTimestamp$.

- $Expiries$

  A fixed-width array of expiries for each column marked in $HasExpiry$.

- $ColumnWidth : ColId \mapsto \mathbb{N}$

  A table level property indicating which columns are fixed width, and their widths. A value of ∞ indicates the column is dynamic width.

- $ColumnCount$

  A table level property indicating the number of columns persisted against this value store.

- $Values_B$

  The cells appearing in $HasValue$ whose types permit fixed-width encoding of length B will appear next, so that they may indexed directly without any further information. All columns will appear grouped by size, but also in $ColId$ order; we will construct $ColId$ to enforce this at write-time.

- $Offsets$

  The index of any dynamic length fields present in $HasValue$ will follow, so that they may also be accessed directly. Encoded as $Values_{\infty}$.

- $DValues$

  Finally we encode the dynamic length fields themselves.

Notes

- It is necessary that our bitmaps support efficient rank operations, i.e. count the number of bits less than an index. For small bitmaps ($\leq$ 64 bits) this is trivial, but for larger bitmaps it requires a little extra data to implement efficiently. However it may be easiest to encode rows as linked-lists of $\leq$ 64 possible columns, since more should be rare.
- The pseudocode implementation assumes we read all of our bitmaps out of the row, but this is unnecessary and done only to aid clarity. Other optimisations with presentation issues are similarly left out.
- To help read implementation, we select $ColId$ values that correspond to the size of the column, with dynamically sized columns occurring last. This permits us to walk the columns that are being queried in the order the data is stored on disk, and to perform fewer calculations for indexing into it.

Algorithm 5 Row Oriented Value Retrieval

```
1: function READ(ValueStore, ColIds, Pos, RowEpoch)
2:     HasInfo, TimestampLength ←-- ValueStore
3:     HasCell, HasValue ←-- ValueStore
4:     HasTimestamp, HasExpiry, Timestamps, Expiries ← ∅
5:     if 'HasCell' ∈ HasInfo then
6:         HasCell ←-- ValueStore[... ColumnCount / 8)
7:     if 'HasTimestamp' ∈ HasInfo then
8:         HasTimestamp ←-- ValueStore[... |HasCell| / 8)
9:     if 'HasValue' ∈ HasInfo then
10:        HasValue ←-- ValueStore[... |HasCell| / 8)
11:    if 'HasExpiry' ∈ HasInfo then
12:        HasExpiry ←-- ValueStore[... |HasValue| / 8)
13:    if |HasTimestamp| > 0 then
14:        Timestamps ←-- ValueStore[... |HasTimestamp| × TimestampLength)
15:    if |HasExpiry| > 0 then
16:        Expiries ←-- ValueStore[... |HasExpiry| × 8)

17:    Row ← NEW
18:    Width, WidthStart, WidthEnd ← 0, 0, 0
19:    for ColId ← ColIds do
20:        if ColId ∈ HasValue then
21:            Expiry ← 0
22:            Timestamp ← RowEpoch
23:            if ColId ∈ HasTimestamp then
24:                Timestamp ← Timestamp + Timestamps[RANK(ColId, HasTimestamp)]
25:            if ColId ∉ HasValue then
26:                Row ⇐ ⟨ColId, Timestamp, 'Deleted', Expiry⟩
27:            else
28:                if ColId ∈ HasExpiry then
29:                    Expiry ← Expiries[RANK(ColId, HasExpiry)]
30:                if ColumnWidth(ColId) ≠ Width then
31:                    PrevWidth, Width ← Width, ColumnWidth(ColId)
32:                    for Fetch ← {w | w ∈ ran(Width), PrevWidth < w ≤ Width} do
33:                        LastColId ← max {ColId | ColumnWidth(ColId) = Fetch}
34:                        WithStart ← WidthEnd
35:                        WidthEnd ← 1+ RANK(HasValue, LastColId)
36:                        Count ← WidthEnd − WidthStart
37:                        Values_Fetch ←-- ValueStore[... Count × Fetch)
38:                if Width ≠ ∞ then
39:                    Value ←-- Values_Width[WidthStart+ RANK(HasValue, ColId)]
40:                else
41:                    Index ← WidthStart+ RANK(HasValue, ColId)
42:                    StartOffset ←-- Values_∞[Index]
43:                    EndOffset ←-- Values_∞[Index + 1]
44:                    Value ←-- ValueStore[StartOffset ... EndOffset)
45:                Row ⇐ ⟨ColId, Timestamp, Value, Expiry⟩
46:    return Row
```

Column Oriented

Full column-oriented storage supports only fixed-width data types, and is comparatively trivial to define: each row is assigned an index within the partition, and every single column-oriented store multiplies this with its base offset and the width of the data type to locate the position where data is stored. No muss, no fuss.

Hybrid (Fixed-Width Block Encoding)

The idea here is to exploit fixed-width and densely populated data distributions to further compress data storage requirements, and make indexing to a given row/column involve fewer computational steps. The basic idea is to store all of the values in a column-oriented fashion within a single data page only, so a row index would store the page number, and the row offset within the page. The goal is to support row-oriented workloads, just more efficiently. This model could support non-fixed-width types, but there is probably little benefit to be had when these could instead be persisted in a row-oriented value store by placing the fields in a different $Group_N$.

- $TimestampSize$
  Occupying 2 bits per field in the row, indicates the size of timestamp encoding, with 0 indicating all values are equal to the row epoch; 1: 2 bytes; 2: 4 bytes; 3: 8 bytes. This must support efficient rank operations, like bitmaps for row-oriented storage, but based on cumulative value as opposed to index.
- $IsDeleted$ Bitmap
  Encoded once per row, if $HasDeleted$ is set. A bitmap to be checked before returning the value encoded, that indicates if the cell is deleted. Treated as a pseudo column, of fixed width.
- $HasDeleted$ Boolean
  Indicates if $IsDeleted$ is present.
- $HasExpiry$ Bitmap
  Bitmap indicating which columns need to save expiry and deleted information.
- $ColumnWidth : ColId \mapsto \mathbb{N}$
  A table level property indicating which columns are fixed width, and their widths. A value of ∞ indicates the column is dynamic width. Must also support a cumulative value rank based on fixed-width size, but this can be precomputed.
- $ColumnCount$
  A table level property indicating the number of columns persisted against this value store.

Algorithm 6 Hybrid Value Retrieval

```
1: function READ(ValueStore, ColIds, Pos, RowEpoch)
2:     TimestampSize, HasExpiry, HasDeleted ←-- ValueStore
3:     Row ← NEW
4:     IsDeletedId ← ColumnCount
5:     if HasDeleted then
6:         ColumnWidths ← ColumnWidths ∪ {IsDeletedId ↦ IsDeletedId/8}
7:         TimestampSize ← TimestampSize ∪ {IsDeletedId ↦ 0}
8:         ColIds ← ColIds ∪ {IsDeletedId}
9:     else
10:        ColumnWidths ← ColumnWidths ∪ {IsDeletedId ↦ 0}
11:    RowLength ← ColumnWidths(IsDeletedId) + |HasExpiry| × 4 + ∑ TimestampSize
12:    IsDeleted ← ∅
13:    if HasDeleted then
14:        Index ← RowLength × (Pos + 1) − ColumnWidth(IsDeletedId)
15:        IsDeleted ←-- ValueStore[Index ... Index + ColumnWidth(IsDeletedId))
16:    for ColId ← ColIds do
17:        Index ←RANK(HasExpiry, ColId) ×4
18:        Index ← Index+RANK(HasTimestamp, ColId)
19:        Index ← Index+RANK(HasDeleted, ColId)
20:        Index ← Index+RANK(ColumnWidth, ColId)
21:        Expiry, Timestamp ← 0, RowEpoch
22:        if ColId ∈ HasExpiry then
23:            Expiry ← ValueStore[Index ... Index + 4)
24:            Index/getsIndex + 4
25:        Timestamp ← ValueStore[Index ... Index + TimestampSize(ColId)]
26:        Index ← Index + TimestampSize(ColId)
27:        if ColId ∈ IsDeleted then
28:            Value ←′ Deleted′
29:        else
30:            Value ← ValueStore[Index ... Index + ColumnWidth(ColId))
31:        Row ⇐ ⟨ColId, Timestamp, Value, Expiry⟩
32:    return Row
```