Next Generation Cassandra Compaction,

Going beyond LCS

Joey Lynch

Speaker

Joey Lynch

Senior Software Engineer
Cloud Data Engineering at Netflix

Distributed system addict and data wrangler

Outline

Main Compaction challenges

General Purpose Compaction Strategies:

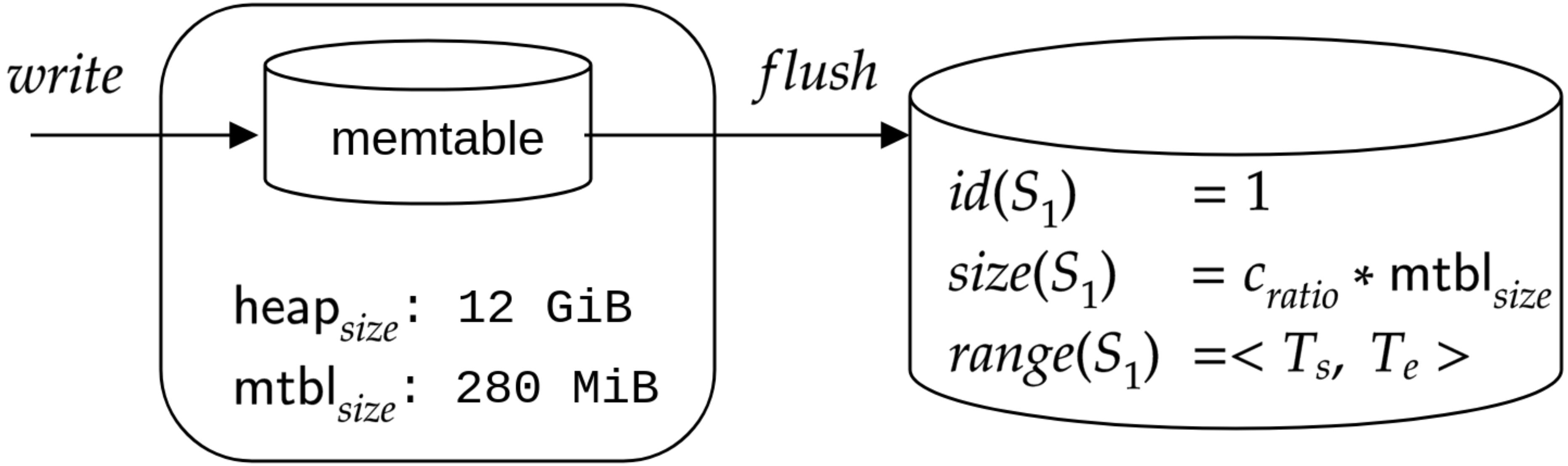
- Size Tiered
- Leveled

Next Generation Compaction Strategy

Compaction

Putting the merge in log structured merge (LSM)

Path to Compaction: Flush



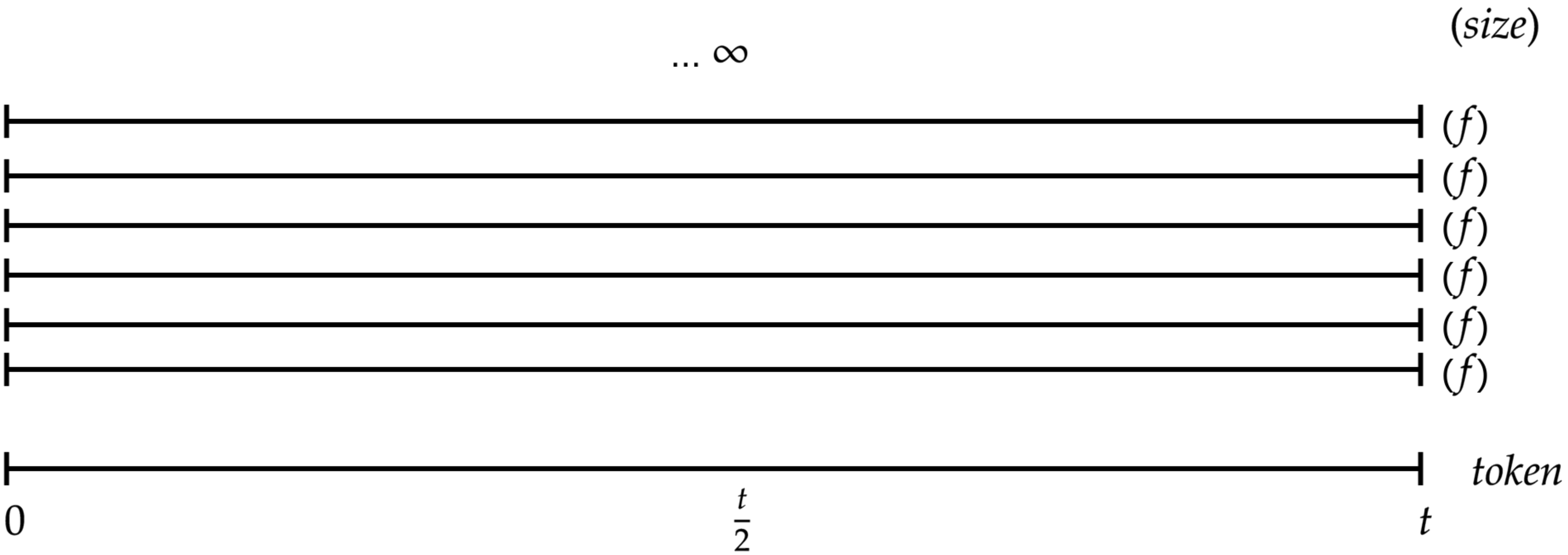
Writes are coalesced into a memtable, then flushed

What is an “SSTable”

```
$ ls -latr | grep 79649
      2468 Jul  1 16:38 mc-79649-big-Summary.db
   1353995 Jul  1 16:38 mc-79649-big-Index.db
      9784 Jul  1 16:38 mc-79649-big-Filter.db
 335548485 Jul  1 16:38 mc-79649-big-Data.db
        10 Jul  1 16:38 mc-79649-big-Digest.crc32
    189987 Jul  1 16:38 mc-79649-big-CompressionInfo.db
        92 Jul  1 16:38 mc-79649-big-TOC.txt
     11959 Jul  1 16:38 mc-79649-big-Statistics.db
```

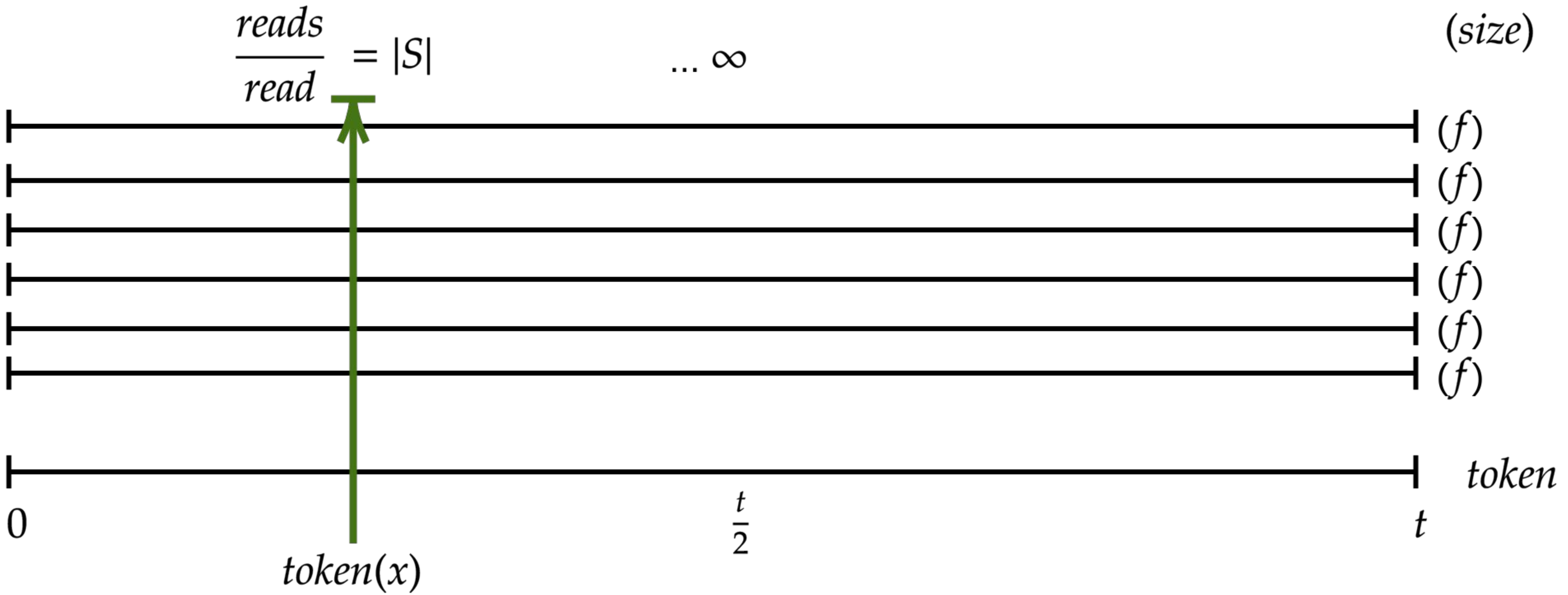
One “SSTable” actually multiple “components”

Problem: Space Amplification



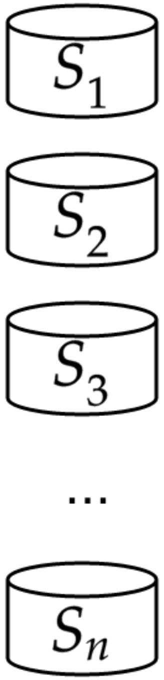
We have unbounded number of SSTables

Problem: Read Amplification



You wanted to *read* your data?

Compaction Inputs



$size(S_i)$ = Either compressed or uncompressed size

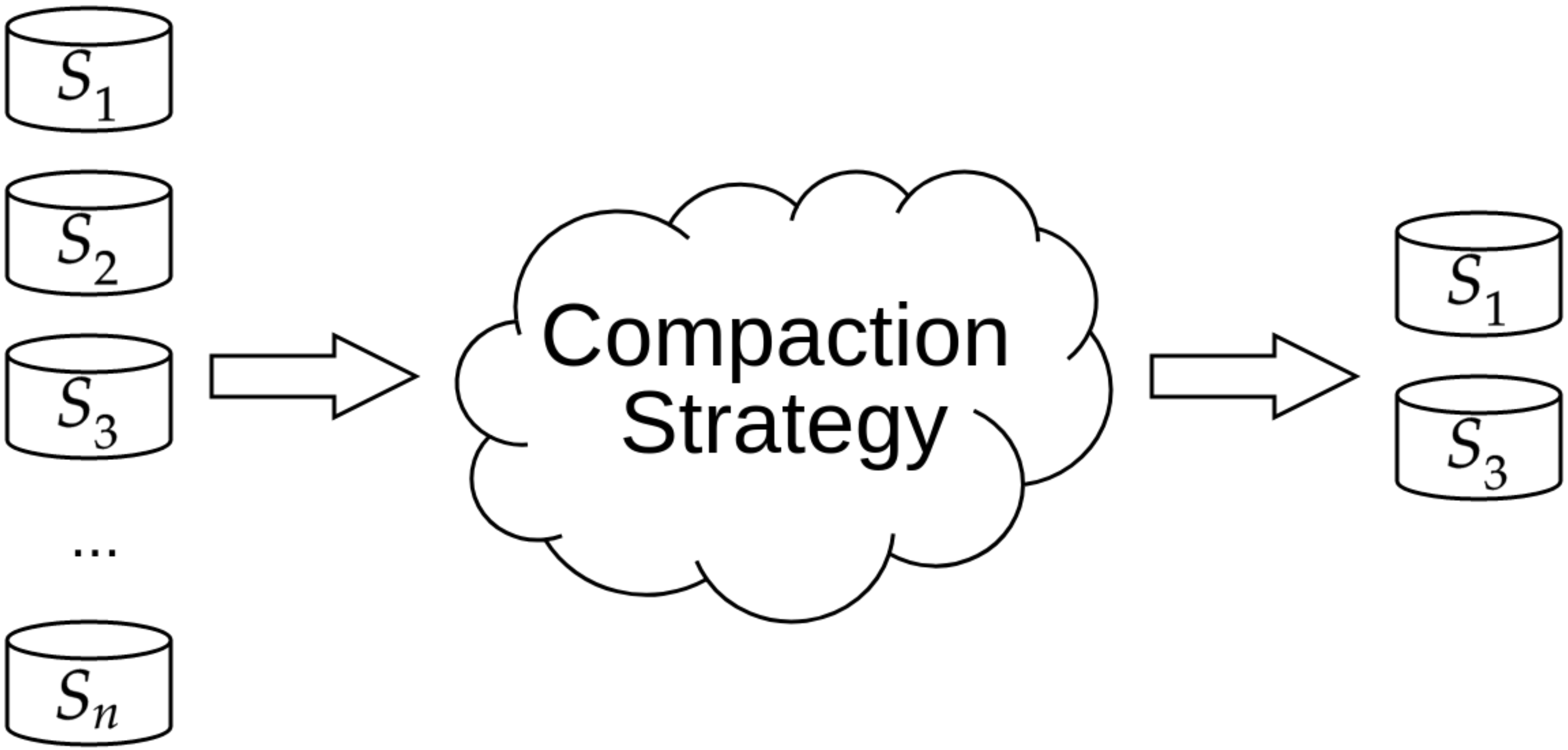
$range(S_i)$ = What range of tokens does the SSTable span?

$level(S_i)$ = Non-overlapping guarantee

Have some SSTables

Have some metrics

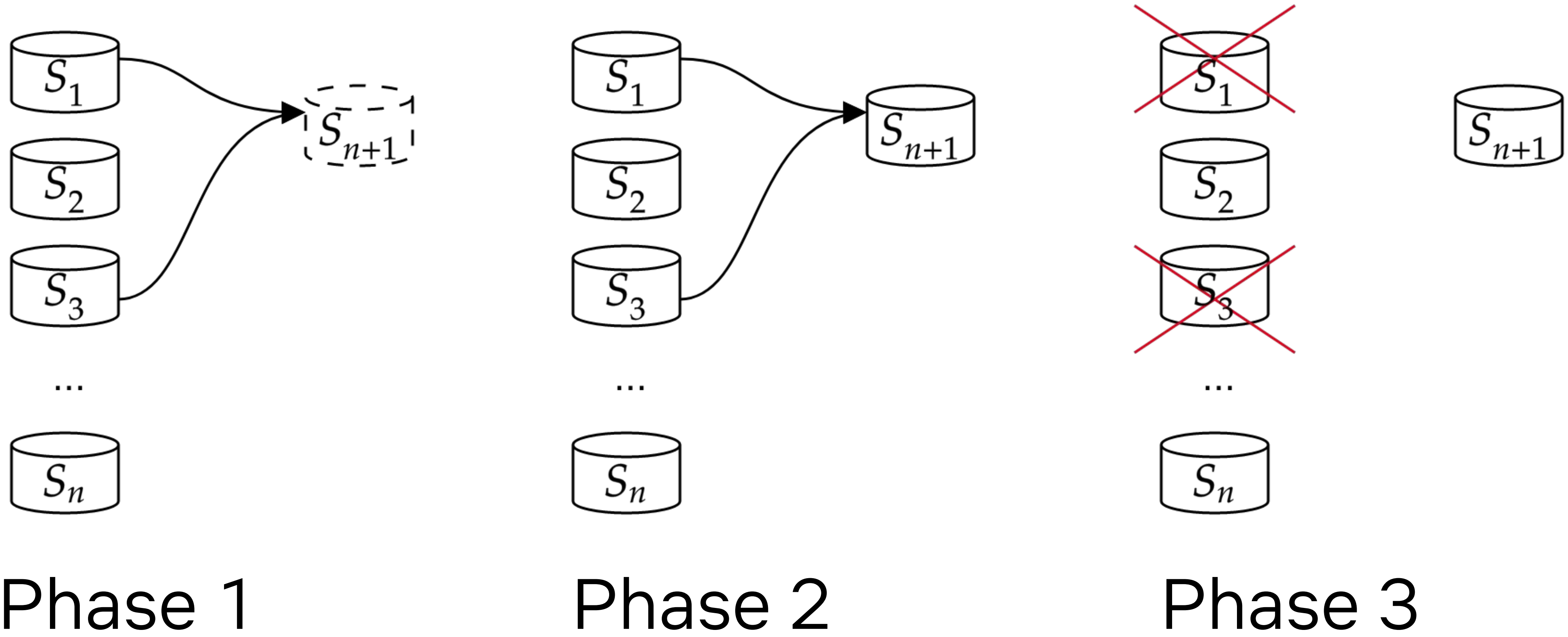
Compaction Strategy Picks Candidates



$$\text{let } C = candidates(S)$$
$$s.t.\ C \subseteq S$$

$$S = \{S_1, S_2, \dots S_n\}$$

Compaction Outputs



Compaction Is Expensive

Phase 1

Reads through a bunch of data, materializing it into heap

This is very expensive

Phase 2

Re-builds view

This can be expensive

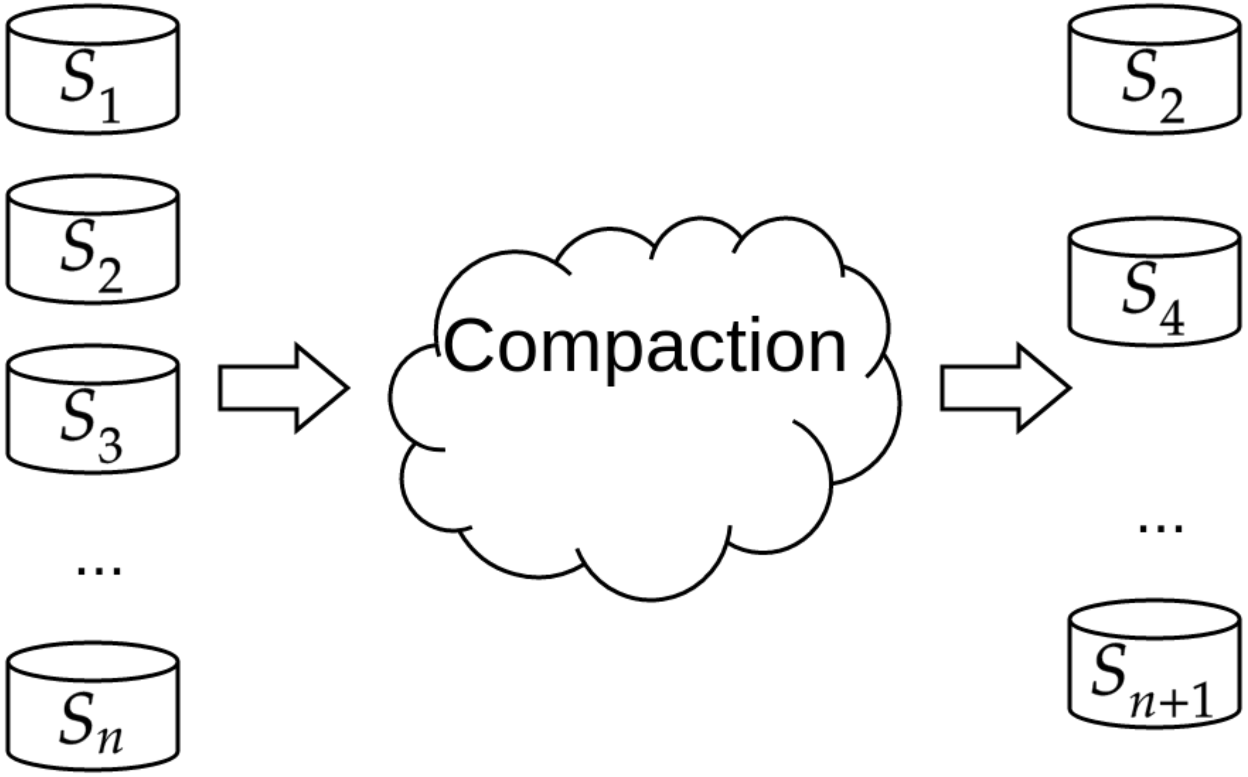
Phase 3

Re-builds view, Drops old data, OS page cache drops

This can be expensive

Problems?	**Why?**
Space Amplification	Can’t take infinite disk space ...
Read Amplification	Reads should be fast
Write Amplification	Compaction is expensive

Compaction Strategy Goal



$$\begin{aligned}
&\textit{choose candidates } C \\
&\textit{let } S' = S - C + S_n + 1 \\
&s.t \\
&amp_r(S') < amp_r(S) \quad \wedge \\
&amp_s(S') < amp_s(S) \quad \wedge \\
&min(|C|)
\end{aligned}$$

$$S = \{S_1, S_2, \ldots S_n\} \qquad S' = \{S_2, S_4, \ldots S_{n+1}\}$$

“How much disk space did I reclaim?”

“Is the data really gone?”

“My reads are slow!!”

Problems?	**Why?**
Full compaction	Have to be able to make guarantees
Manual compaction	Targeted incidents (hot partitions)
Handle tons of small SSTables	Repair is fun this way

Lessons Learned

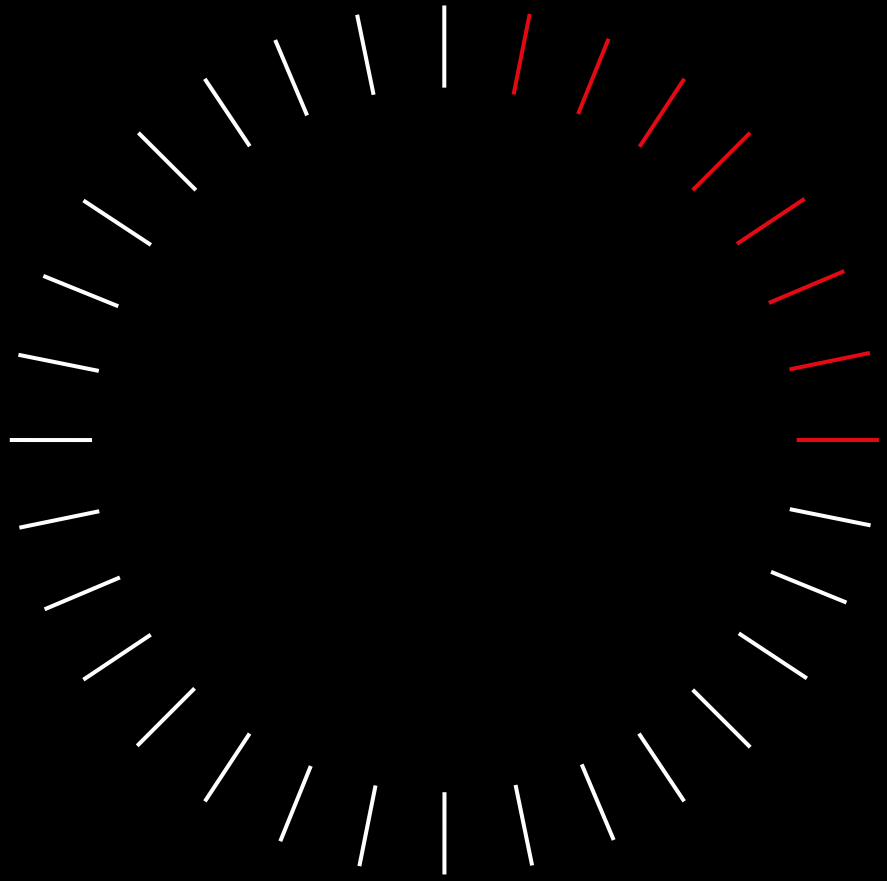
Compaction is an **optimization** problem

Trying to reduce space and read amplification

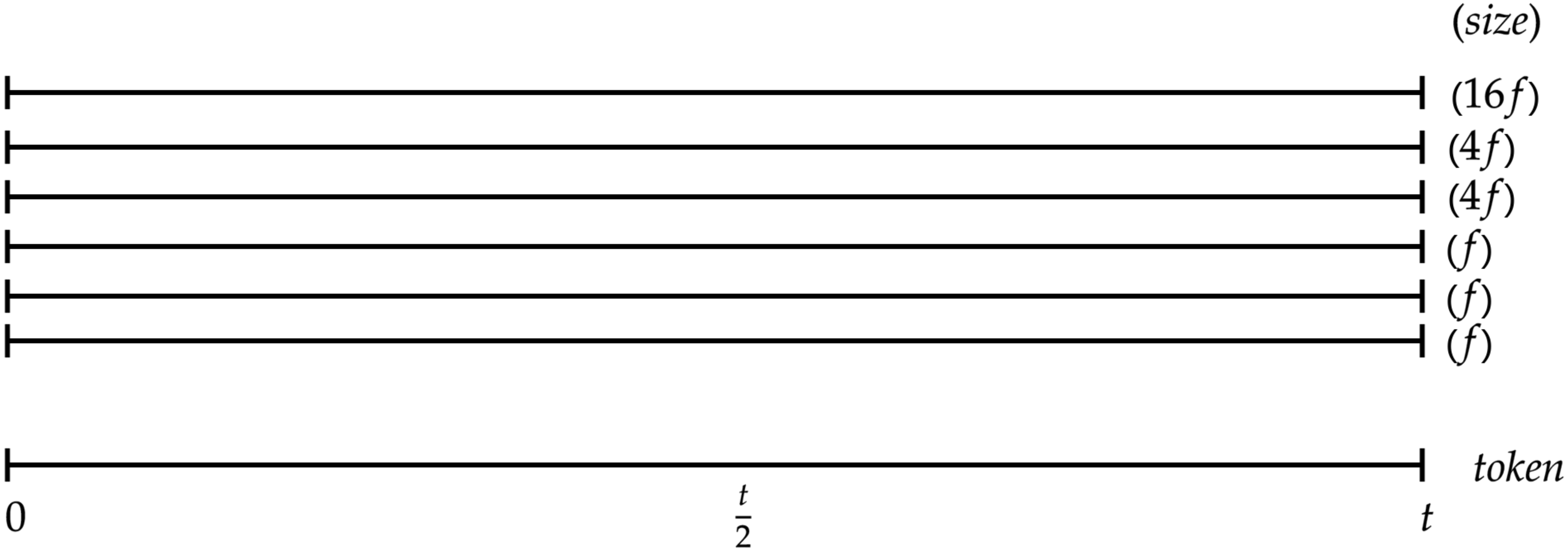
… while doing as little write amplification as we can

Size Tiered Compaction

Let’s do the most obvious thing

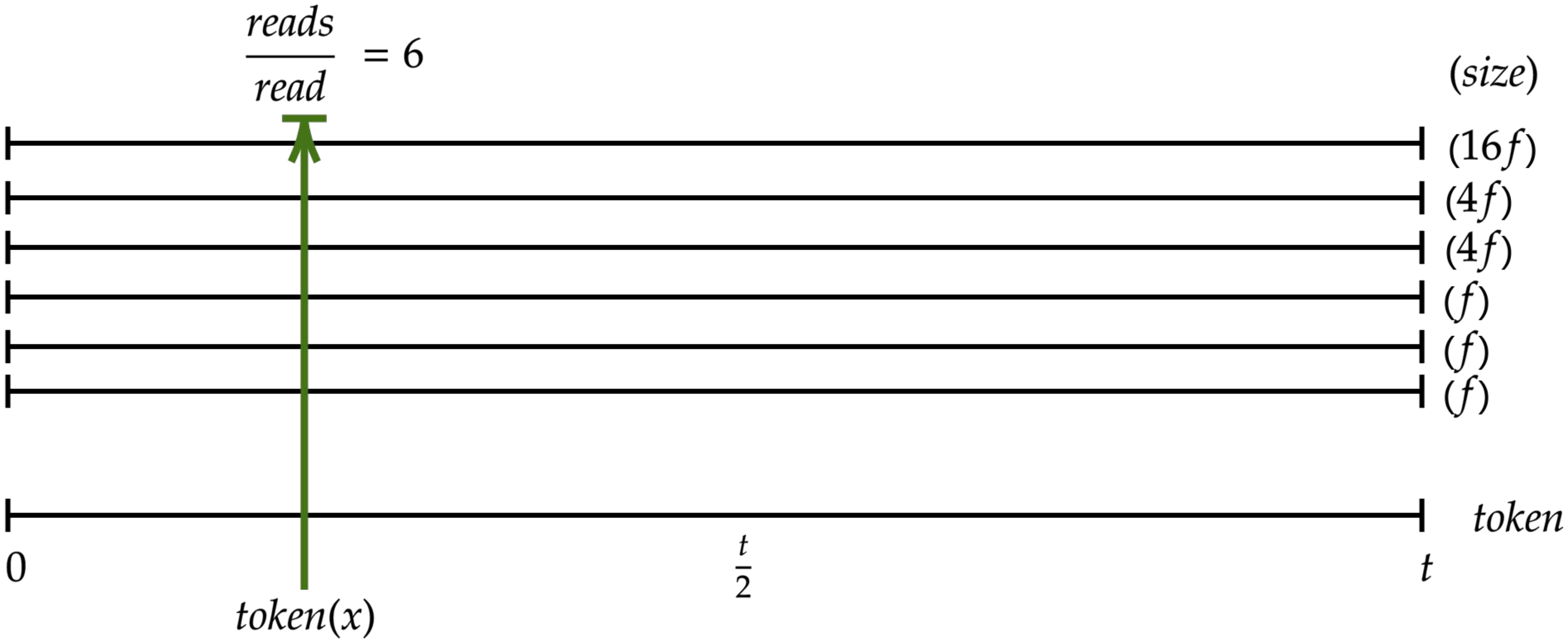


Group SSTables by Size



Main Tunables	Meaning?
`min_threshold`	This dictates the level of tiering
`bucket_low`	What should be bucketed together
`bucket_high`	

Group SSTables by Size



Advantages

Good for write only workloads

Full compaction works great

Super simple

Why?

Exponential re-write curve

Small number of files

Fewer bugs

Problems?	**Why do we care?**
50% space overhead	Hard to provision for
Giant SSTables	Hard to backup / transfer
Begat early re-open 🔥🔥	This had a **ton** of bugs
Read amplification	Variable read performance is bad

Problems?

Needless write amplification

Have to do periodic full compactions to give any guarantees

Why do we care?

Wasting CPU is bad

Re-writing the whole dataset all at once every few weeks seems like a suboptimal choice

Lessons Learned

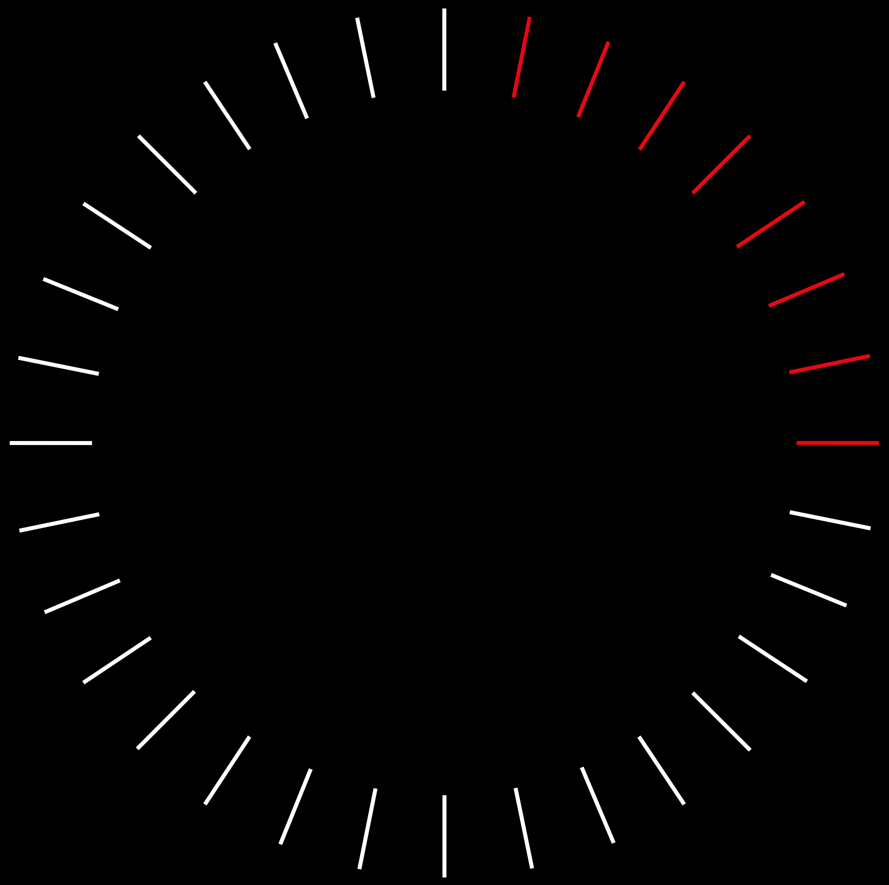
Large single files are problematic

Need to wait until we have enough work from flushes to do.

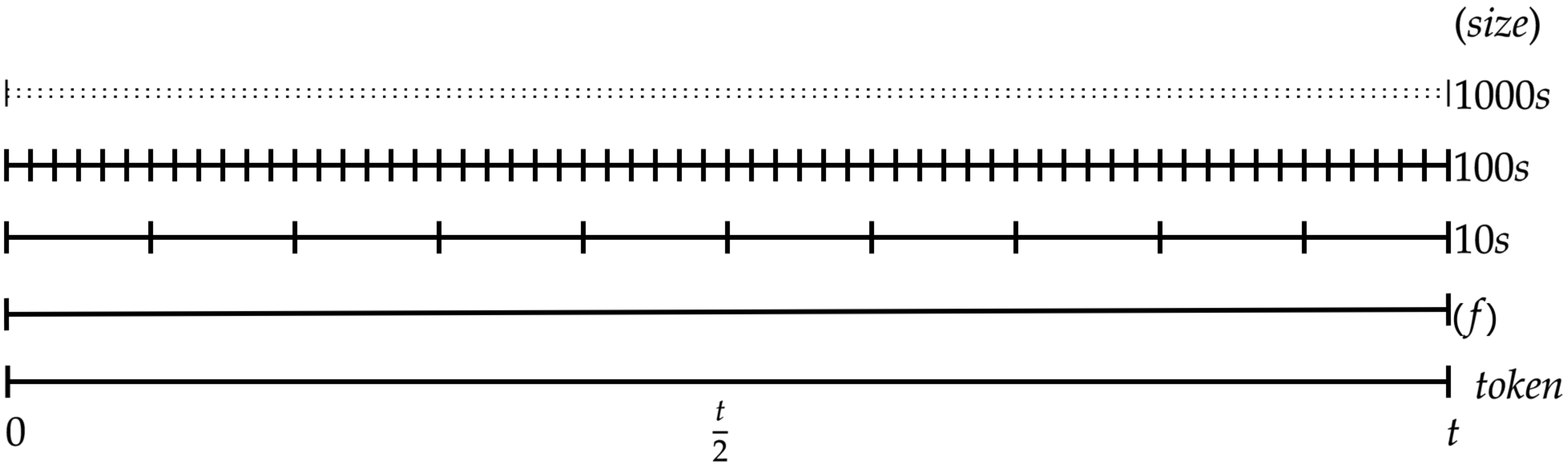
Time bounds on full compaction is nice

Leveled Compaction

What if we only did useful work?



Sorted runs to the rescue!



Advantages

Wicked fast reads

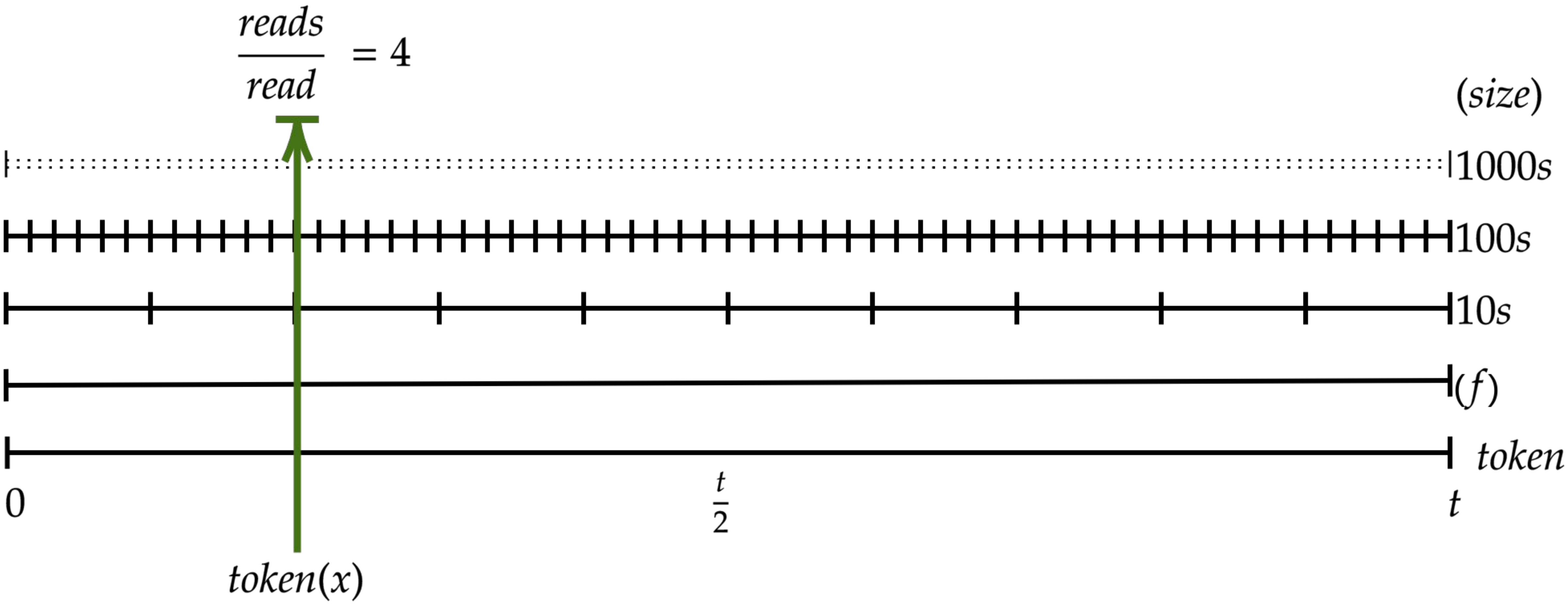
Relatively fixed disk usage

Why?

Bounded read amplification

Bounded space amplification

Sorted runs to the rescue!



Main Tunables	**Meaning?**
`sstable_size_in_mb`	The small, fixed size that LCS should output Each sorted run or "level" will contain 10x the previous sorted run

Advantages

Small files easy to work with

Good for TTL data

Good for updates
Good for reads

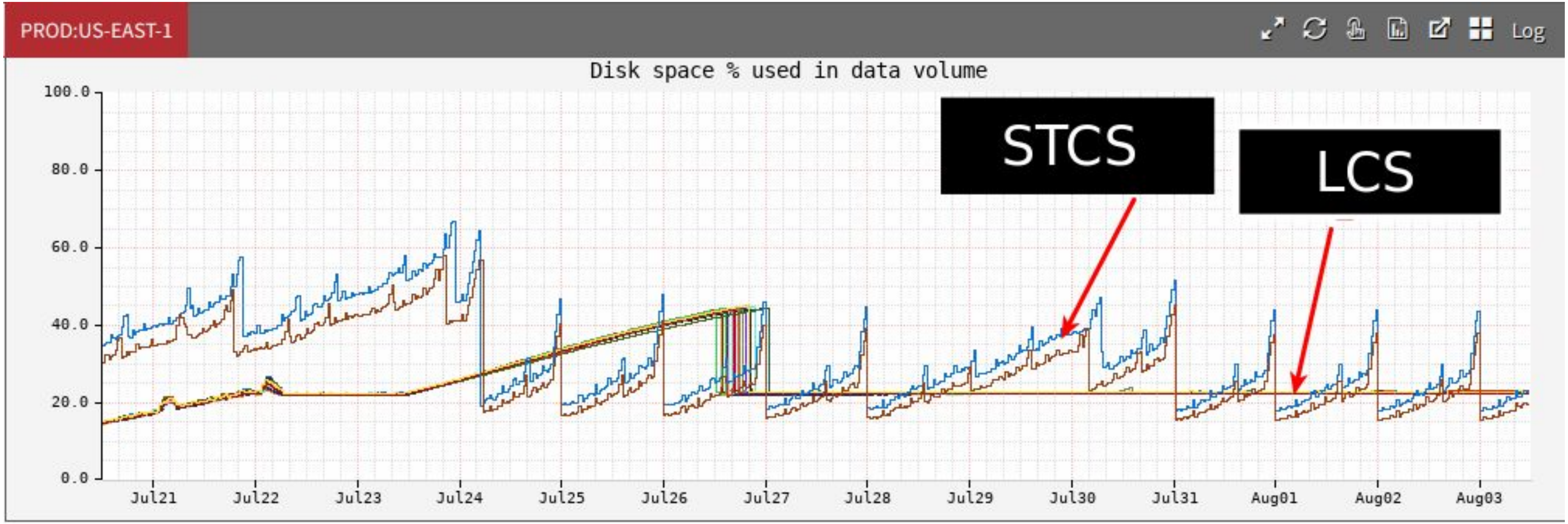
Why?

File parallelism is generally easy

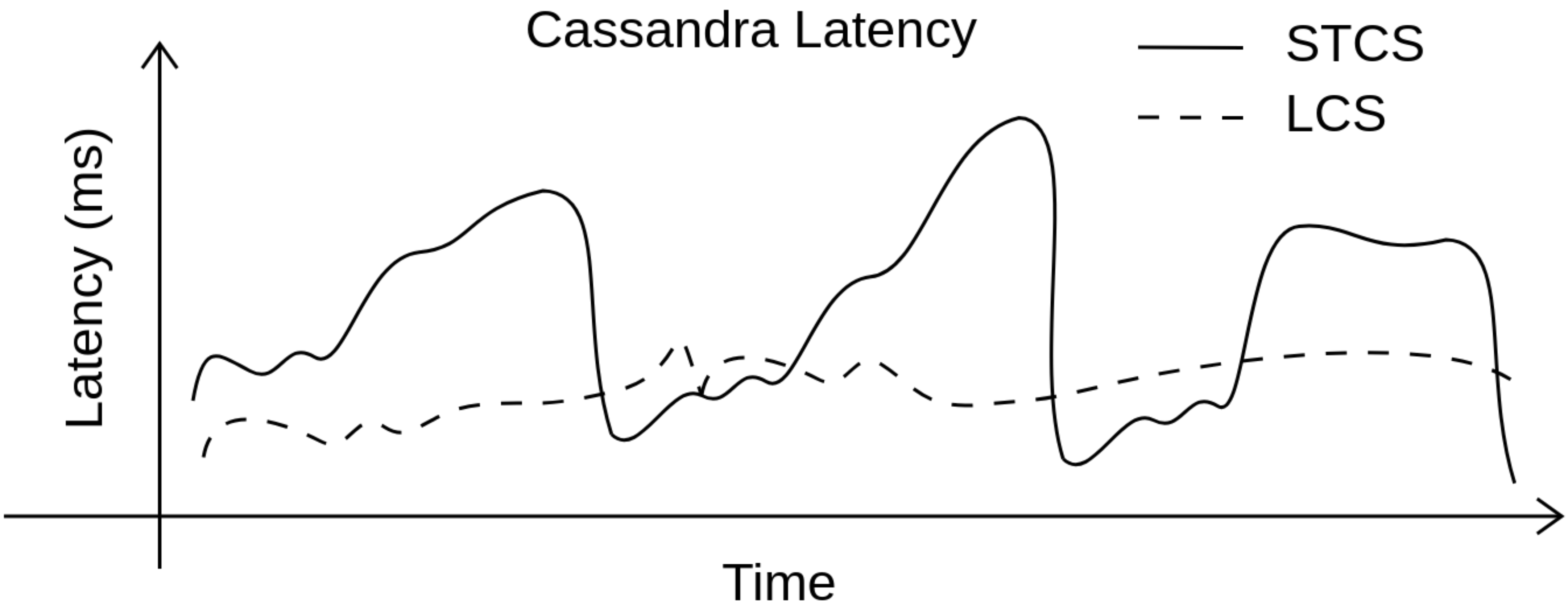
Self compactions viable

Levels are really nice ...

It Really Works



It Really Works



Problems?	**Why do we care?**
Data must go through L1	Can “fall behind” incoming writes
Does a lot of compaction	For spinning disks this can be a problem …
Really complicated	A large number of bugs

Problems?

Default 160 MiB can create a *lot* of sstables

SSTable size is underconstrained

Why do we care?

Upstream processors may not be able to handle 20k files

Have to raise SSTable size when all you want is larger L1

Problems?

Hard to get data from the upper levels “down”

Full compaction is currently very slow

Why do we care?

Getting updates (deletes) to find each other

Pretty huge operability issue

Four day compactions ...

Cassandra / CASSANDRA-14605

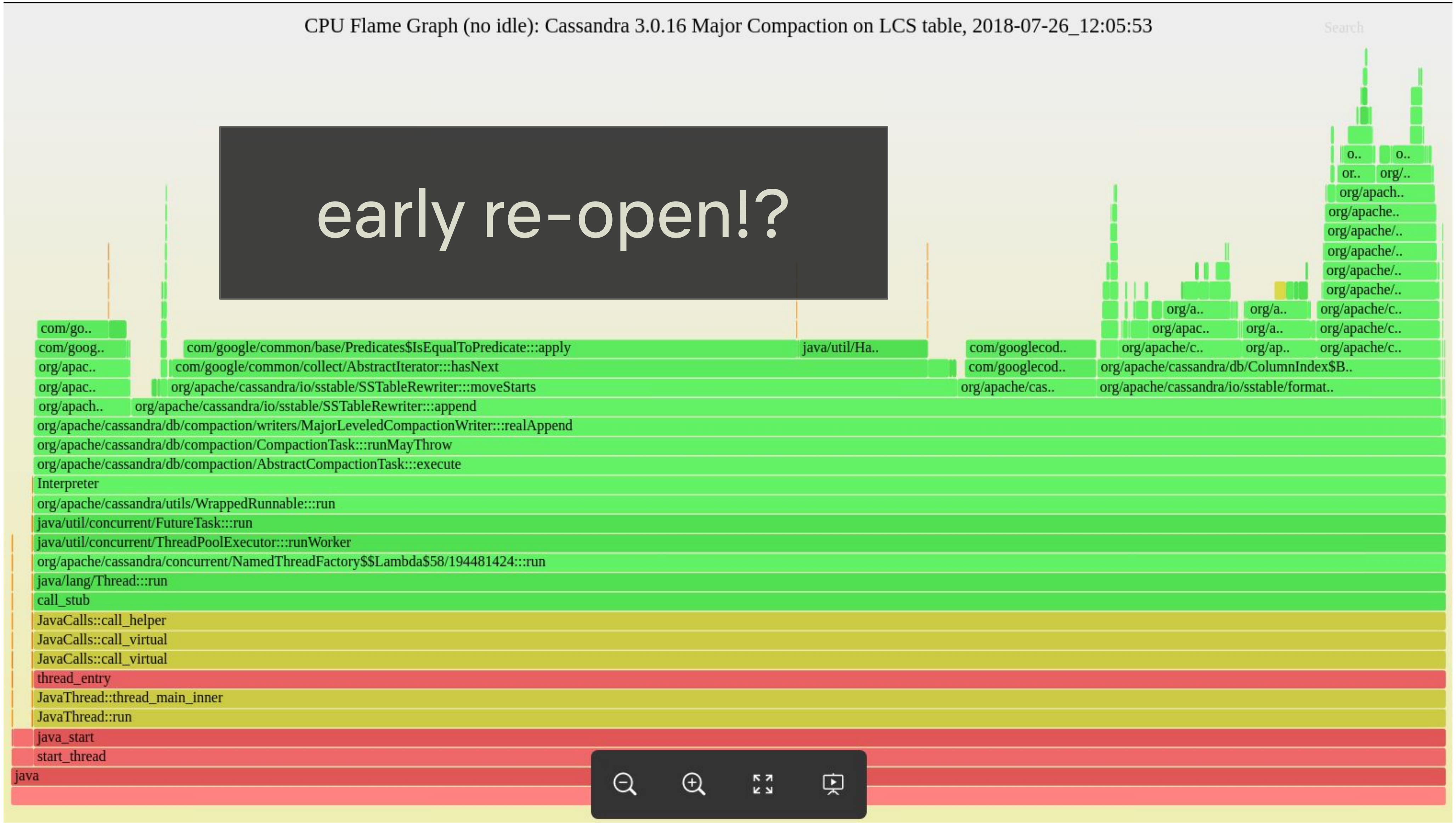
Major compaction of LCS tables very slow

Details

Description

We've recently started deploying 3.0.16 more heavily in production and today I noticed that full compaction of LCS tables takes a much longer time than it should. In particular it appears to be faster to convert a large dataset to STCS, run full compaction, and then convert it to LCS (with re-leveling) than it is to just run full compaction on LCS (with re-leveling).

Four day compactions ...



There is a solution!

Marcus Eriksson added a comment - 27/Jul/18 05:17

could you try setting `sstable_preemptive_open_interval_in_mb` to -1 and see how it performs?

Joseph Lynch added a comment - 30/Jul/18 20:25

Marcus Eriksson I've set the parameter and taken another flamegraph, all of the `moveStarts` usage is gone (nice!) and it appears to be proceeding about twice as fast (instead of taking 4 days it will take 2 days). I've attached the new flamegraph

Lessons Learned

Sorted runs are powerful

- Eliminates a lot of the need for early re-open

Bookkeeping complexity should be avoided if possible

Rigid structure (L0 -> L1) is not so good

Need more levels!

We need something better.

Target Overlap Compaction Strategy

Only do useful work, fewer rules

Goals

Sorted runs

Configurable read vs write amplification with one tunable

Capable of “full” compaction without significant extra disk space

Terminology

$size(S_i)$ = Either compressed or uncompressed size

$range(S_i)$ = What range of tokens does the SSTable span?

$level(S_i)$ = Non-overlapping guarantee

Sorted runs of SSTables

$$SS_i \leftarrow level(S_i)$$

$$density(SS_i) = \frac{\sum size(S_i)}{\sum range(S_i)}$$

High Level Idea

Aim for target number of reads per read (overlap)

Compaction only operates across sorted runs

Select candidates using `IntervalTree`, bucket by density. Once dense enough start de-overlapping

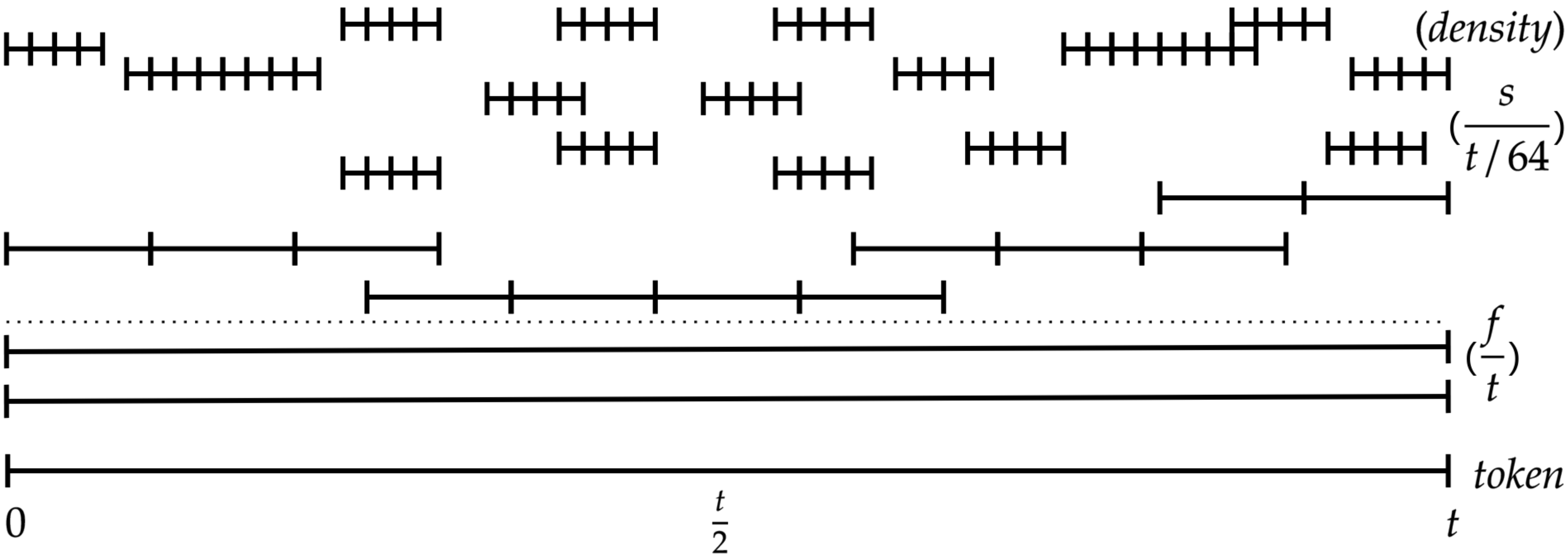
Target Overlap Configuration

```
CREATE TABLE ... (
  ...
) WITH compaction = {
  'class' : 'TargetOverlapCompactionStrategy',
  # What is your target “read/read”?
  'target_overlap': 4
};
```

Target Overlap Configuration

```
compaction = {
   # Standard STCS things
   'min_threshold': 2,
   'max_threshold': 32,
   'min_sstable_size_in_mb': 50,
   # Target read per read
   'target_overlap': 4,
   # How big should my segments be
   'target_sstable_size_in_mb': 256,
   # How many reads per read can I get?
   'max_overlap': target_overlap * min_threshold,
   # How many sstables can I handle max per node (e.g. due to backup)
   'max_sstable_count': 2000
   # How old should SSTables be allowed to get? 0 for unlimited
   'target_deadline': '10d',
};
```

Let the Levels* Run Free



* Now known as "Sorted Runs"

Algorithm: SS_0 (~= Level Zero)

Flushes are low quality sorted runs

Pure density tiering (~= size tiering)

Once we have `target_size * max_count` data, "promote" to a sorted run SS_{n+1}

Algorithm: SS_n

$$let\ O_t = target\ overlap$$
$$let\ O_m = max\ overlap$$
$$let\ D = deadline$$
$$let\ F = tier\ factor$$

$$
\begin{aligned}
& C \leftarrow \{\} \\
& \forall\ interval\ I \in range():\\
& \quad S \leftarrow overlapping(I) \\
& \quad SS \leftarrow group(S,\ level(S)) \\
& \quad if\ |SS| > O_m : \\
& \quad\quad C.insert(overlapping(SS)) \\
& \quad elif\ age(s \in SS) > D : \\
& \quad\quad C.insert(overlapping(s)) \\
& \quad elif\ |SS| > O_t : \\
& \quad\quad B_d \leftarrow bucket(SS,\ density) \\
& \quad\quad \forall\ b \in B_d\ s.t.\ |d| > F \\
& \quad\quad\quad C.insert(*b,\ score(d))
\end{aligned}
$$

Algorithm: SS_n

Iterate intervals in range

Group by sorted run

$$
\begin{aligned}
&C \leftarrow \{\} \\
&\boxed{\begin{aligned}&\forall\ interval\ I \in range():\\ &\quad S \leftarrow overlapping(I)\\ &\quad SS \leftarrow group(S,\ level(S))\end{aligned}} \\
&\quad if\ |SS| > O_m: \\
&\qquad C.insert(overlapping(SS)) \\
&\quad elif\ age(s \in SS) > D: \\
&\qquad C.insert(overlapping(s)) \\
&\quad elif\ |SS| > O_t: \\
&\qquad B_d \leftarrow bucket(SS,\ density) \\
&\qquad \forall\ b \in B_d\ s.t.\ |d| > F \\
&\qquad\quad C.insert(*b,\ score(d))
\end{aligned}
$$

Algorithm: SS_n

If we have enough overlapping ranges, force a compaction

$C \leftarrow \{\}$
$\forall\ interval\ I\ \in\ range():$
$\quad S\ \leftarrow overlapping(I)$
$\quad SS\ \leftarrow group(S,\ level(S))$
$\quad if\ |SS|\ >\ O_m:$
$\quad\quad C.insert(overlapping(SS))$
$\quad elif\ age(s\ \in SS)\ > D\ :$
$\quad\quad C.insert(overlapping(s))$
$\quad elif\ |SS|\ >\ O_t:$
$\quad\quad B_d\ \leftarrow bucket(SS,\ density)$
$\quad\quad \forall\ b\ \in B_d\ s.t.\ |d|\ > F$
$\quad\quad\quad C.insert(*b,\ score(d))$

Algorithm: SS_n

If any SSTables are too old, compact all overlapping with that SSTable

Probably with jitter…

$$
\begin{aligned}
& C \leftarrow \{\} \\
& \forall\ interval\ I \in range(): \\
& \quad S \leftarrow overlapping(I) \\
& \quad SS \leftarrow group(S,\ level(S)) \\
& \quad if\ |SS| > O_m: \\
& \quad\quad C.insert(overlapping(SS)) \\
& \quad \boxed{\begin{aligned} & elif\ age(s \in SS) > D: \\ & \quad C.insert(overlapping(s)) \end{aligned}} \\
& \quad elif\ |SS| > O_t: \\
& \quad\quad B_d \leftarrow bucket(SS,\ density) \\
& \quad\quad \forall\ b \in B_d\ s.t.\ |d| > F \\
& \quad\quad\quad C.insert(*b,\ score(d))
\end{aligned}
$$

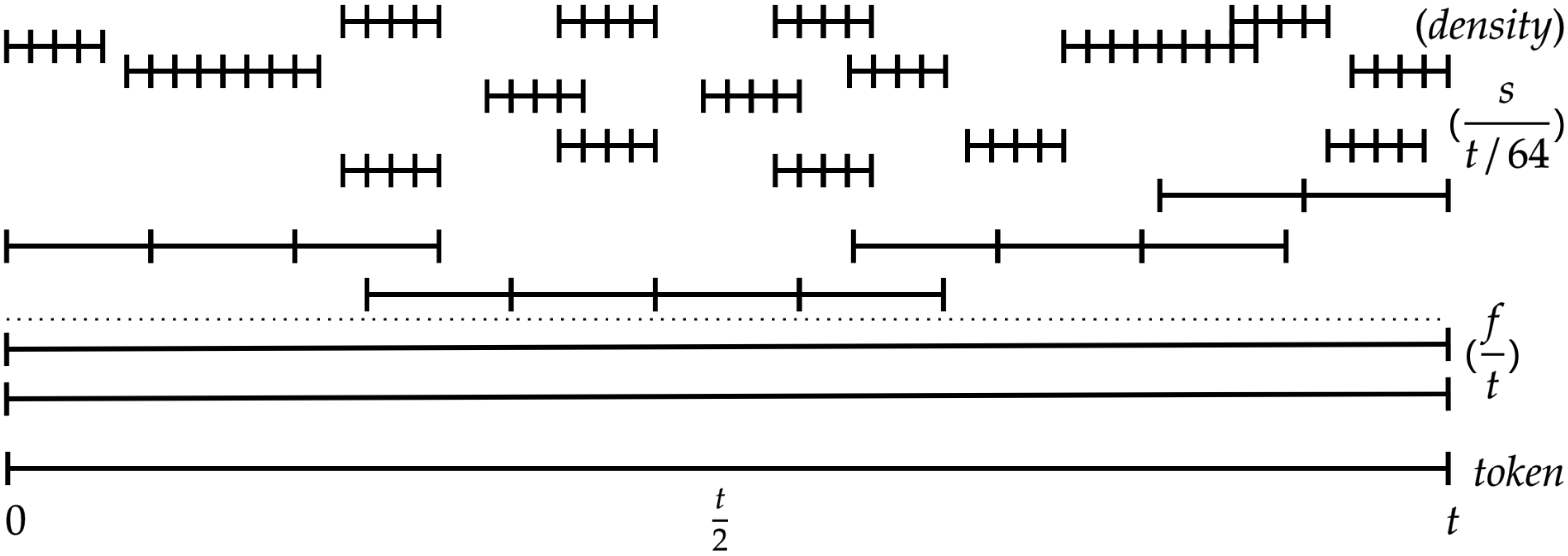
Algorithm: SS_n

Density tiering within the sorted runs.

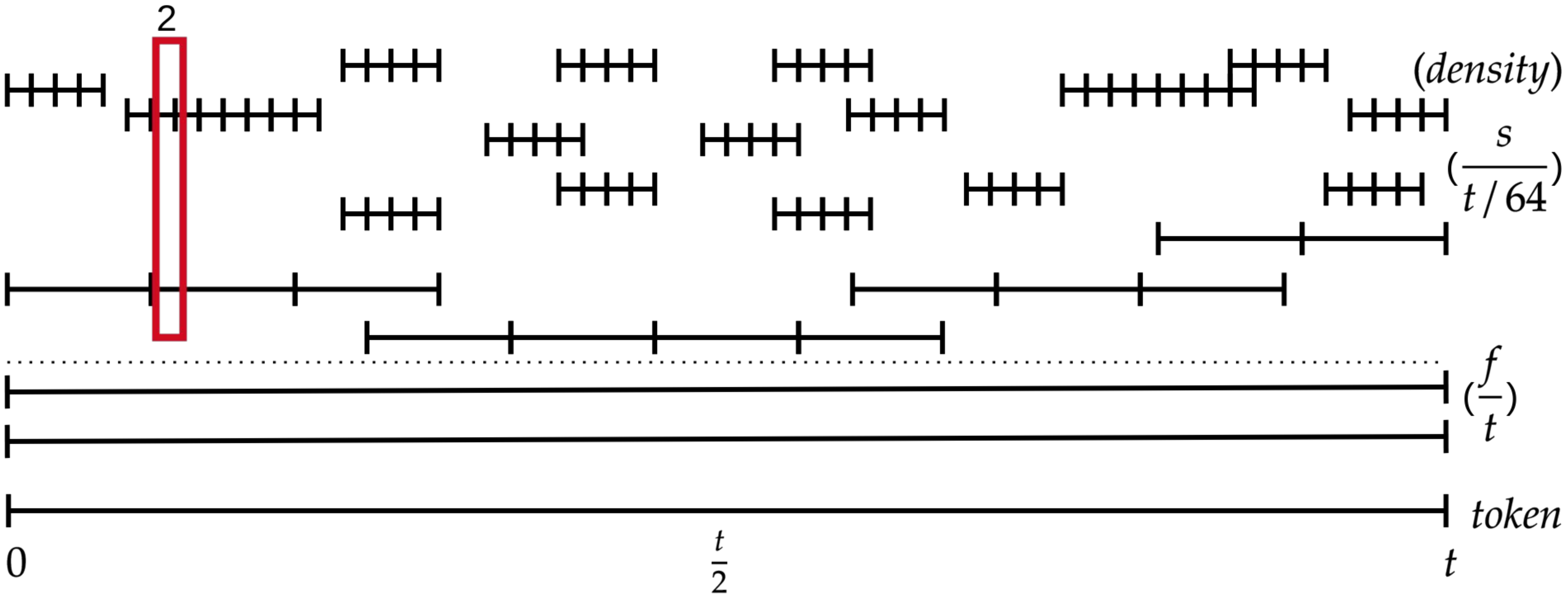
Prefer larger buckets first, break ties with density

$C \leftarrow \{\}$
$\forall\ interval\ I \in range():$
$\quad S \leftarrow overlapping(I)$
$\quad SS \leftarrow group(S,\ level(S))$
$\quad if\ |SS| > O_m:$
$\quad\quad C.insert(overlapping(SS))$
$\quad elif\ age(s \in SS) > D:$
$\quad\quad C.insert(overlapping(s))$
$\quad elif\ |SS| > O_t:$
$\quad\quad B_d \leftarrow bucket(SS,\ density)$
$\quad\quad \forall\ b \in B_d\ s.t.\ |d| > F$
$\quad\quad\quad C.insert(*b,\ score(d))$

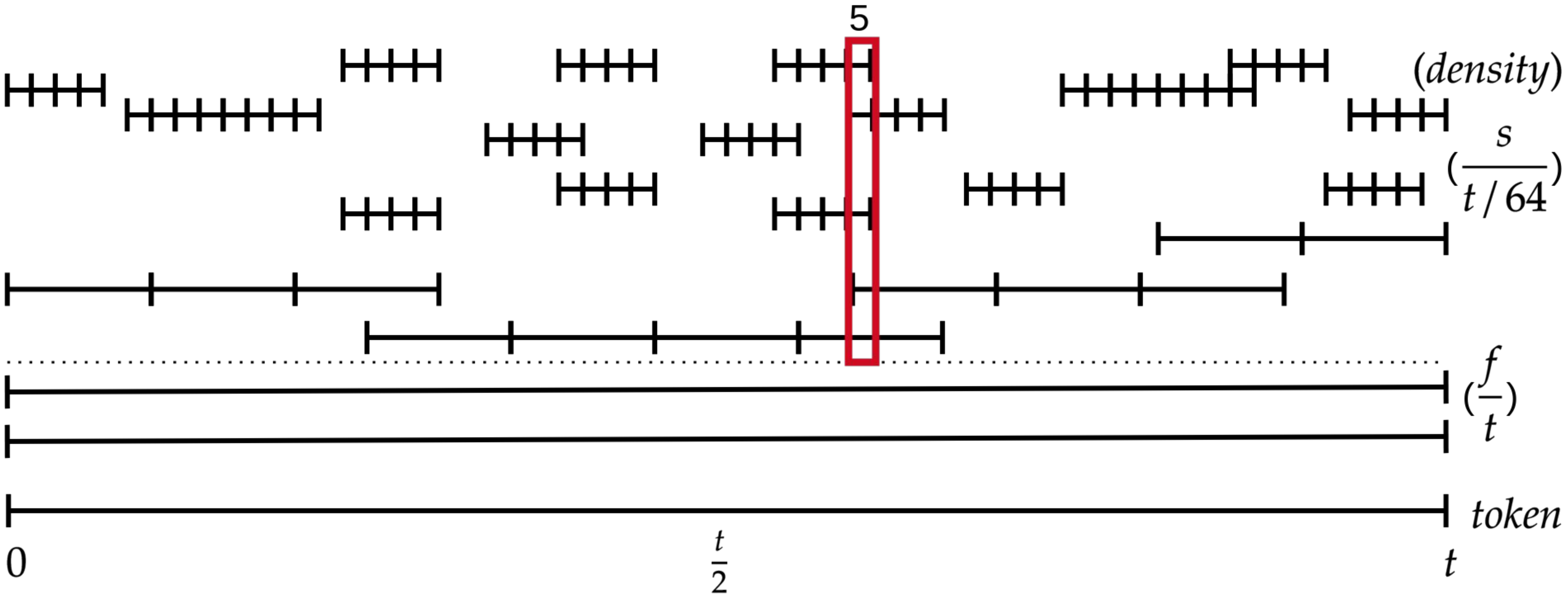
TOCS In Action



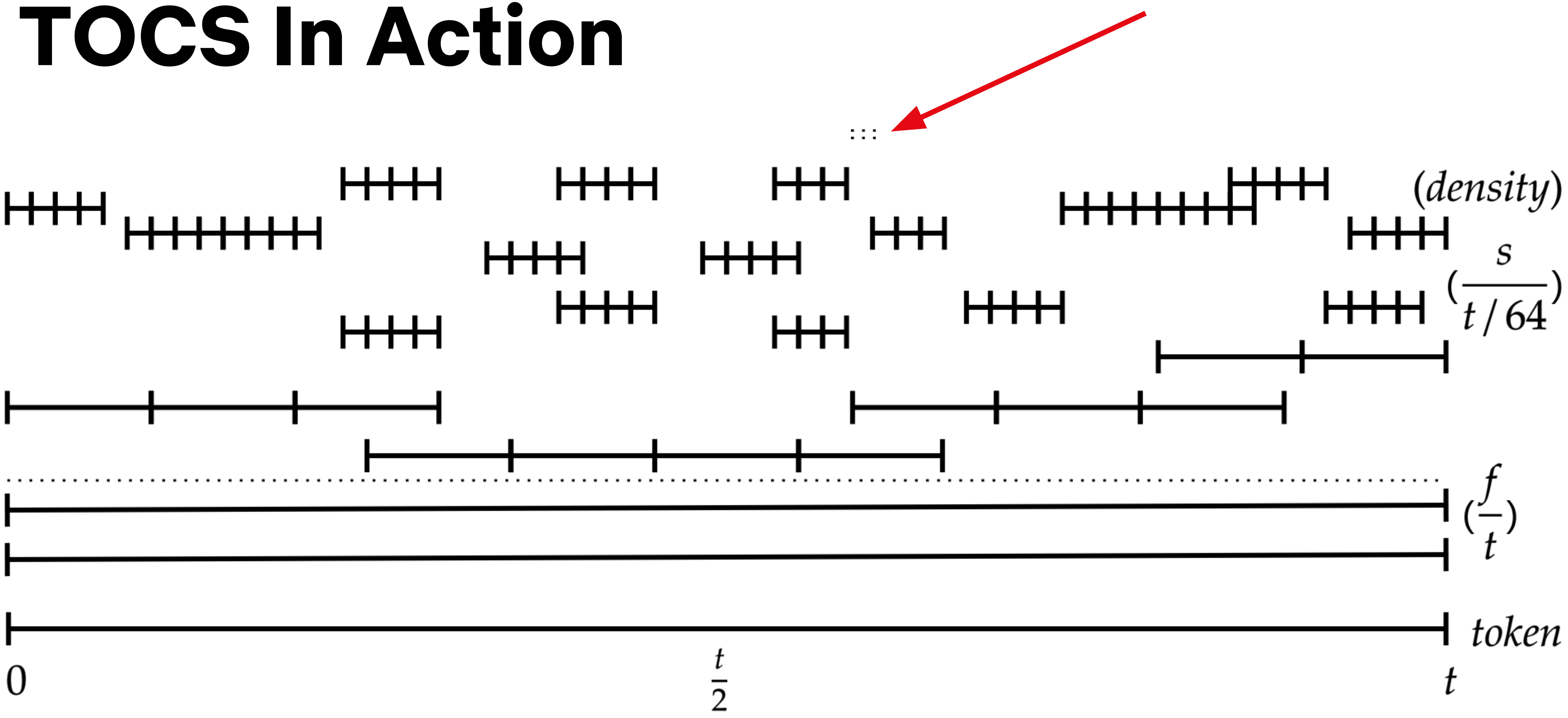
TOCS In Action



TOCS In Action



TOCS In Action



Property: Easy to Use

Single main tunable, `target_overlap`, allows easy to understand read vs write amplification tradeoff

Secondary tunable, `target_deadline`, allows easy to understand tombstone / update properties

Property: Low Overhead

Largest compaction will be ~1/32 of data size

Full compactions implemented through declarative desire rather than imperative command.

Property: Small predictable files

Since the strategy only produces sorted runs of tables about 256 MiB large, it is easy on:

1. Backup
2. Streaming (full sstable streaming)
3. Page cache (no more early re-open)

Still TODO

It’s not clear density tiering in the top levels is worth the complexity. Just do the whole vertical every time?

Should we split sorted runs into 32 ranges on promotion?

Take Away

Target Overlap (name?) is a potential way to unify our learnings from LCS and STCS.

Thank You.

Joey Lynch
josephl@netflix.com