How ScyllaDB makes LSM-tree compaction state-of-art by leveraging RUM conjecture and controller theory

Raphael “Raph” Carvalho
ScyllaDB

Raphael “Raph” Carvalho

Computer programmer who loves kernel programming, and a musician wannabe who had some fun learning electronic keyboard. Has been working distributed database ScyllaDB since 2015. Prior to that, worked on creation of unikernel OSv and developed file system drivers for Syslinux.

SCYLLA.

Agenda

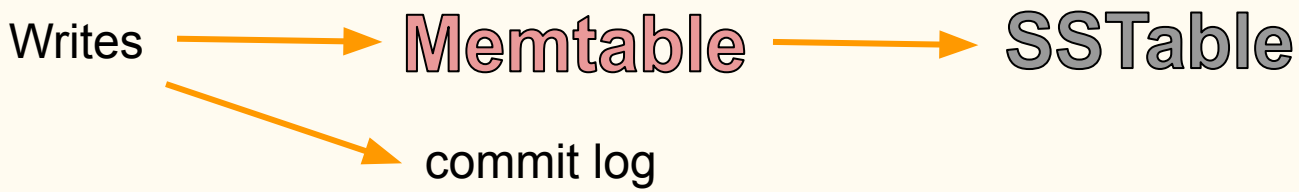
- What is LSM-tree compaction?
- Compaction strategies (policies) and their trade-offs
- The famous incremental compaction
- Hybrid policy in incremental compaction
- Self-tuning compaction by leveraging controller theory

What is LSM-tree compaction?

LSM storage engine's write path:

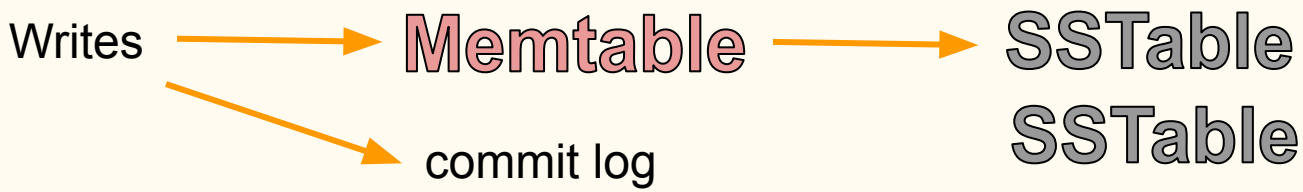
What is LSM-tree compaction?

LSM storage engine's write path:



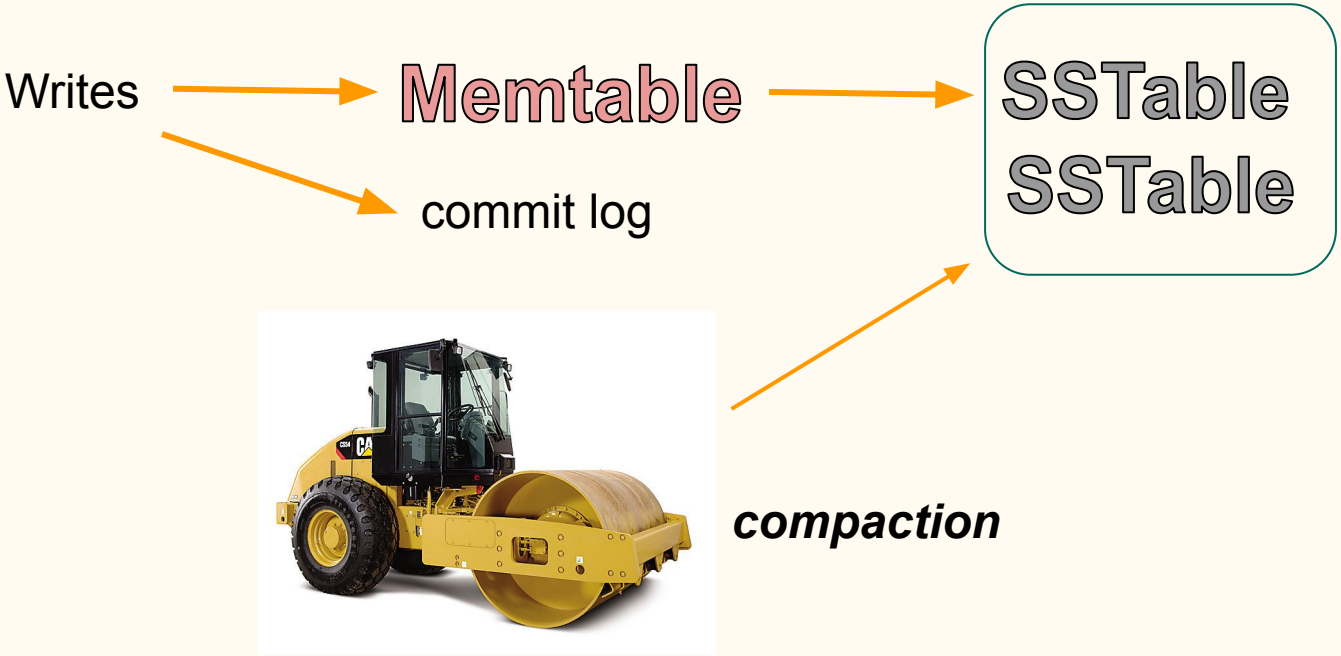
What is LSM-tree compaction?

LSM storage engine's write path:



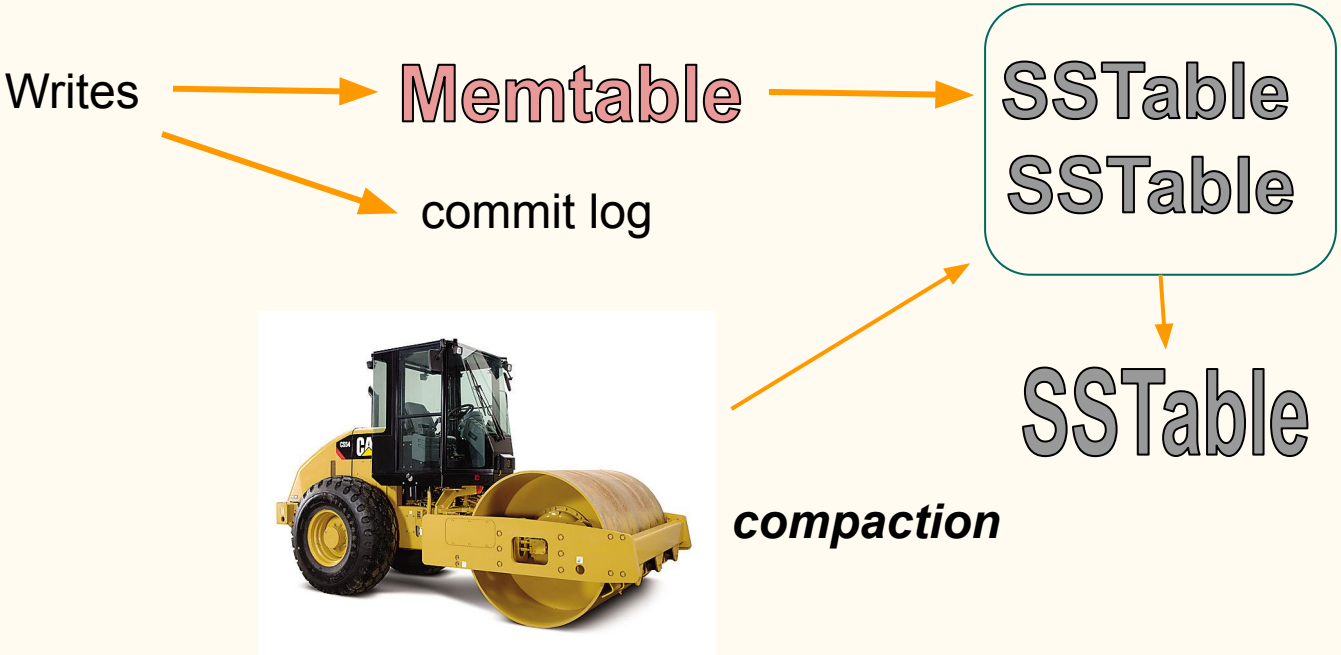
What is LSM-tree compaction?

LSM storage engine's write path:



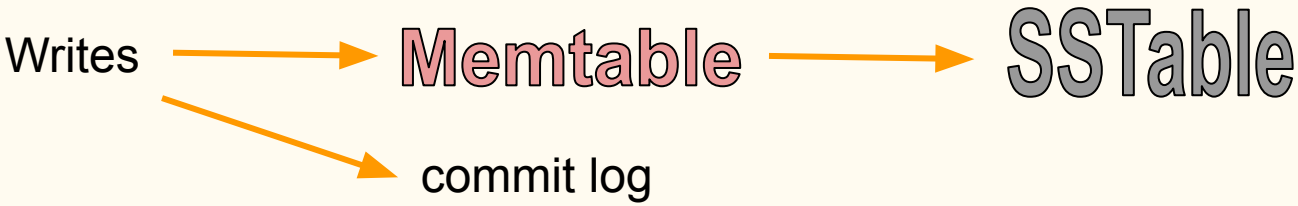
What is LSM-tree compaction?

LSM storage engine's write path:



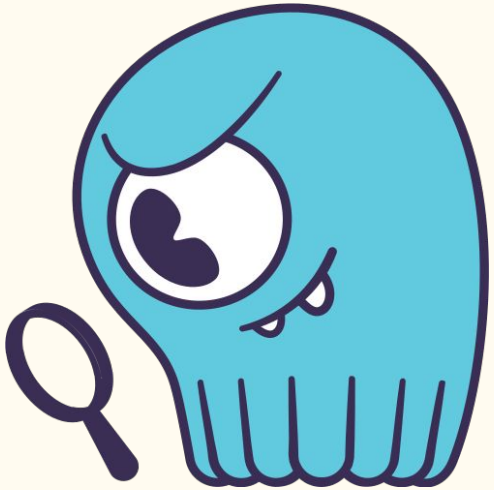
What is LSM-tree compaction?

LSM storage engine's write path:



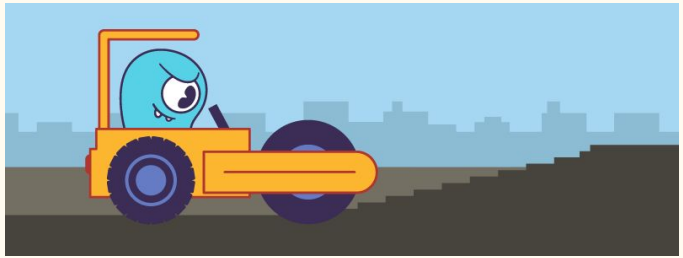
What is compaction? (cont.)

- This technique of keeping sorted files and merging them is well-known and often called **Log-Structured Merge** (LSM) **Tree**
- Published in 1996, earliest popular application that I know of is the Lucene search engine, 1999
  - High performance write.
  - Immediately readable.
  - Reasonable performance for read.



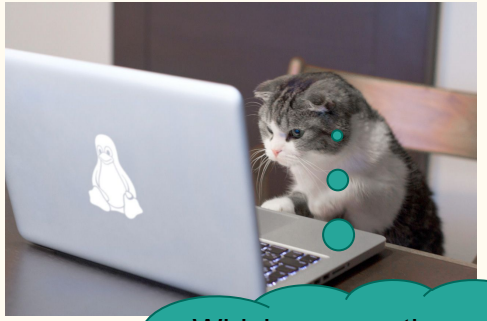
(Compaction efficiency requirements)

- SSTable merge is efficient
  - Merging sorted sstables efficient, and contiguous I/O for read and write
- Background compaction does not increase request tail-latency
  - Scylla breaks compaction work into small pieces
- Background compaction does not fluctuate request throughput
  - “Self-tuning”: compaction done not faster than needed



Compaction Strategy (a.k.a. File picking policy)

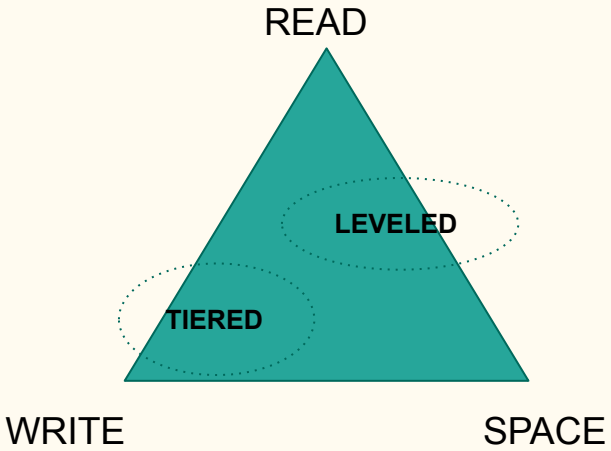
- Which files to compact, and when?
- This is called the **compaction strategy**
- The goal of the strategy is low amplification:
  - Avoid read requests needing many sstables.
    - **read amplification**
  - Avoid overwritten/deleted/expired data staying on disk.
  - Avoid excessive temporary disk space needs (scary!)
    - **space amplification**
  - Avoid compacting the same data again and again.
    - **write amplification**



Read, write and space amplification - Make a choice!

- This choice is well known in distributed databases like with CAP, etc.
- The RUM Conjecture states:
  - we cannot design an access method for a storage system that is optimal in all the following three aspects - Reads, Updates, and, Memory.
- Impossible to decrease read, write & space amplification, all at once
- A policy can e.g. optimize for write, while sacrificing read & space
- Whereas another can optimize for space and read, while sacrificing write

Read, write and space amplification - Make a choice!



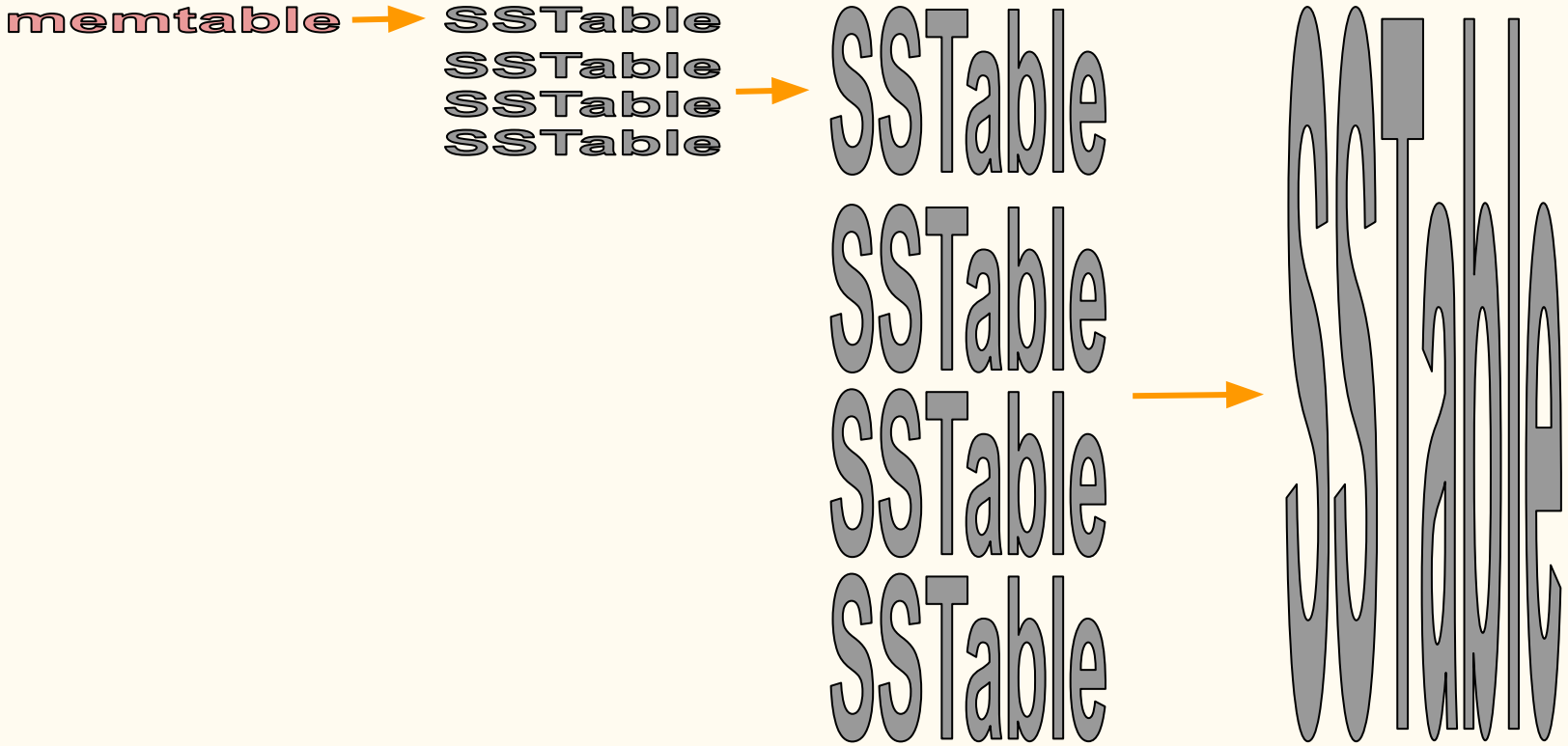
A brief history of compaction policies

- Starts with size tiered compaction policy
  - Efficient write performance
  - Inconsistent read performance
  - Substantial waste of disk space = bad space amplification (due to slow GC)
- To fix read / space issues in tiered compaction, leveled compaction is introduced
  - Fixes read & space issues
  - **BUT** it introduces a new problem - write amplification

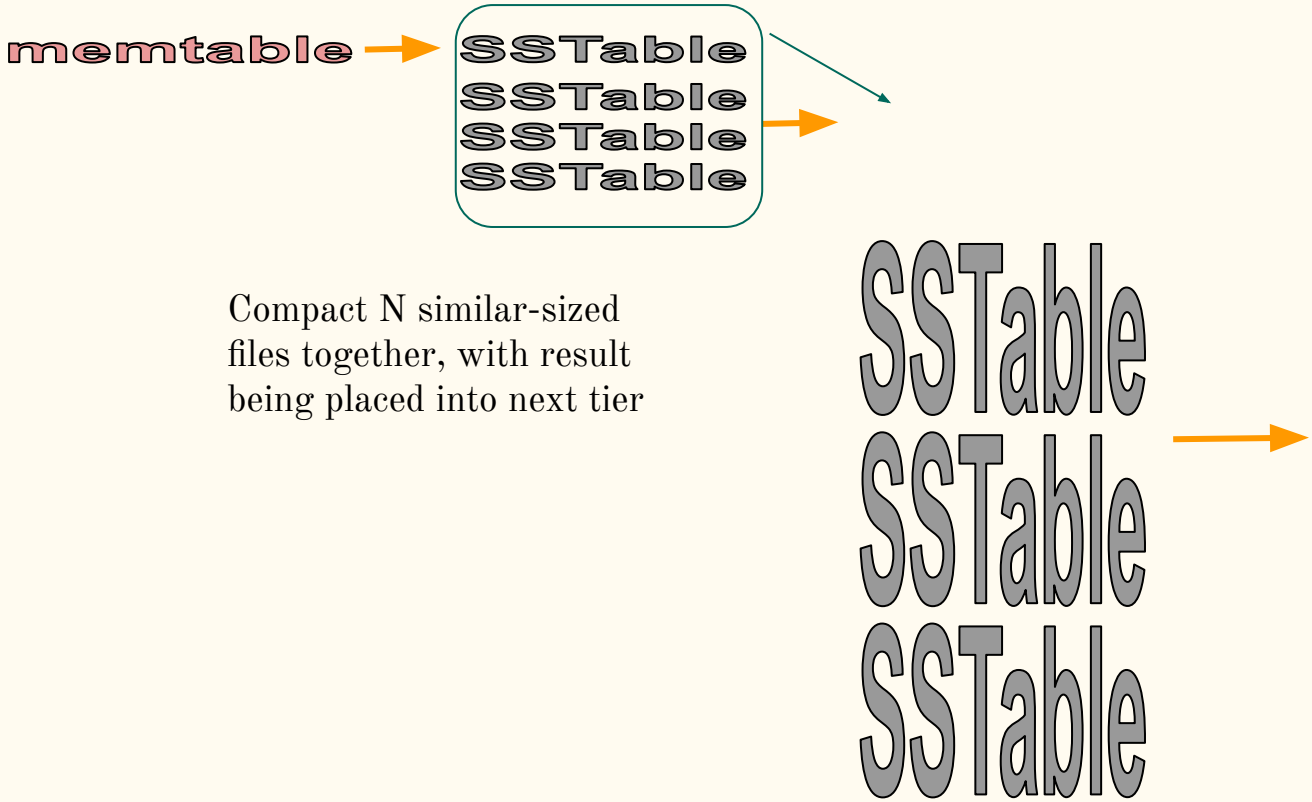
Strategy #1: Size-Tiered Compaction

- Cassandra's oldest, and still default, compaction strategy
- Dates back to Google's BigTable paper (2006)
  - Idea used even earlier (e.g., Lucene, 1999)

Size-Tiered compaction strategy

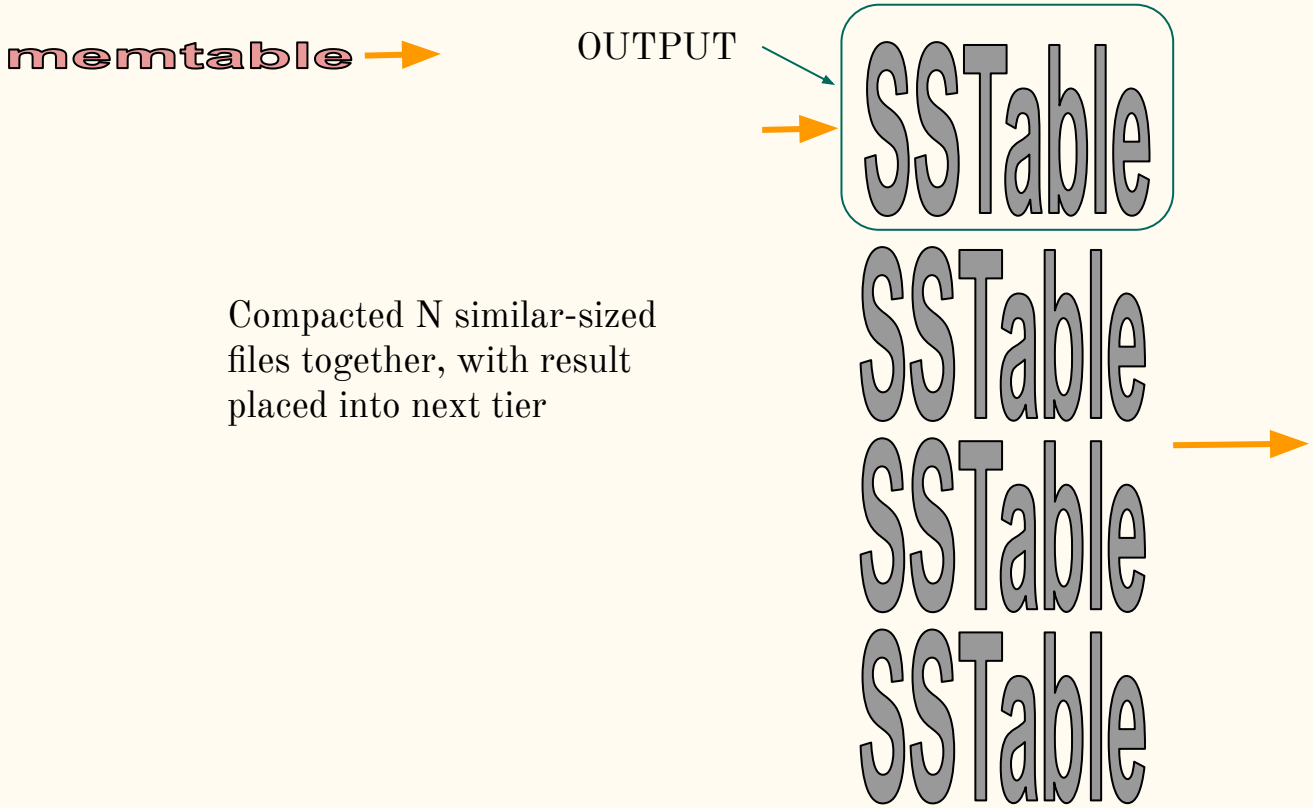


Size-Tiered compaction strategy



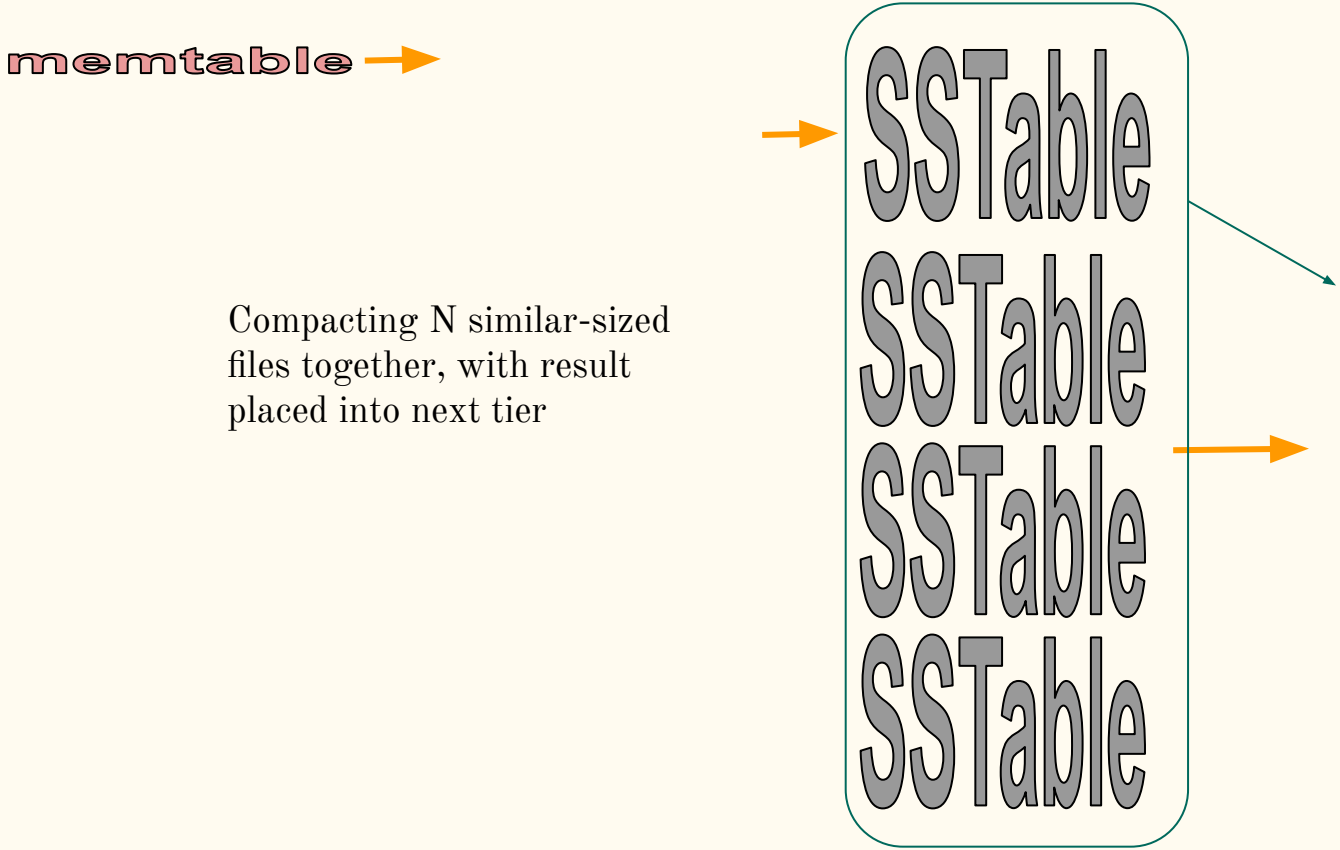
Compact N similar-sized files together, with result being placed into next tier

Size-Tiered compaction strategy



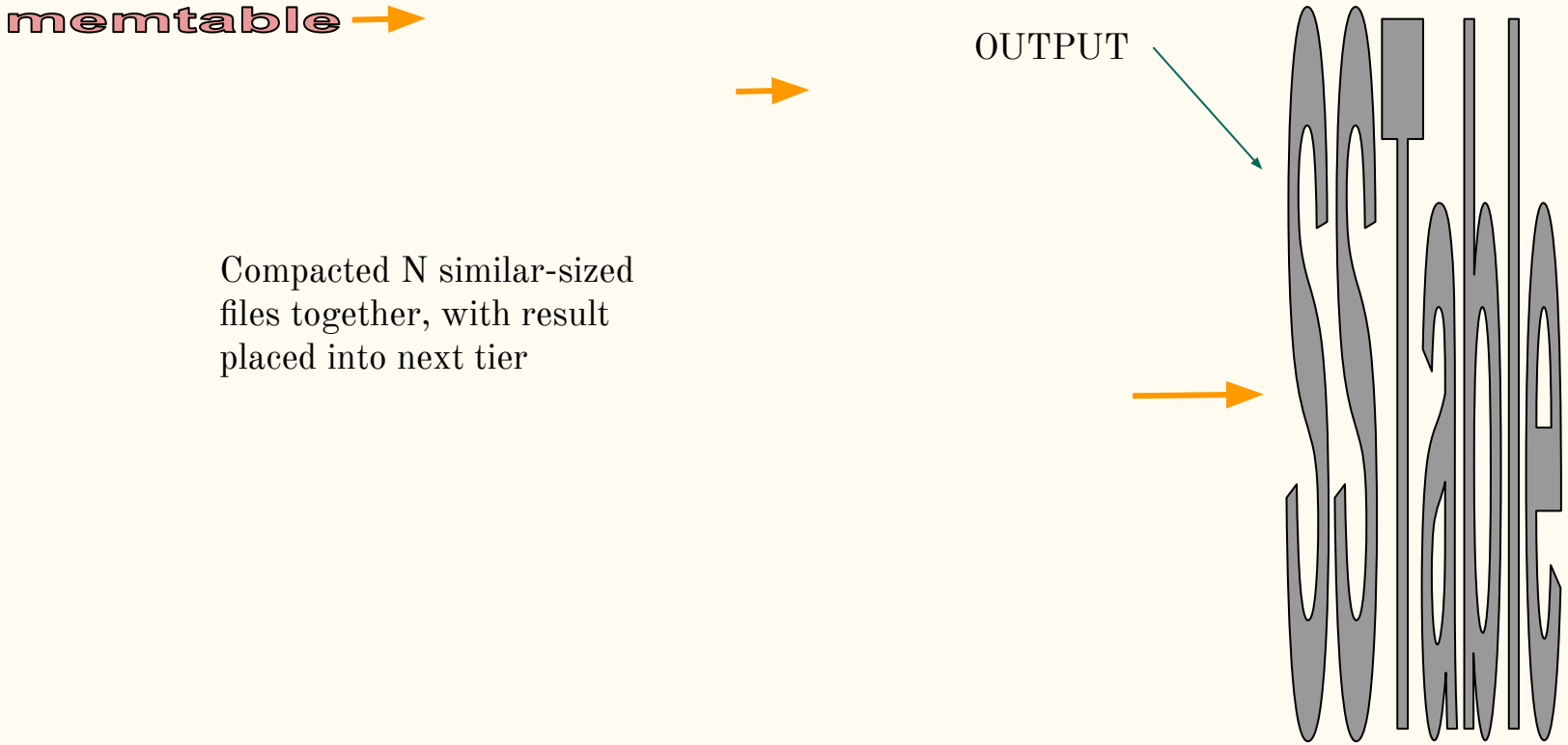
Compacted N similar-sized files together, with result placed into next tier

Size-Tiered compaction strategy



Compacting N similar-sized files together, with result placed into next tier

Size-Tiered compaction strategy



Compacted N similar-sized files together, with result placed into next tier

Size-Tiered compaction - amplification

- **write amplification**: O(logN)
  - Where “N” is (data size) / (flushed sstable size).
  - Most data is in highest tier - needed to pass through O(logN) tiers
  - This is asymptotically optimal

Size-Tiered compaction - amplification

What is **read amplification?** O(logN) sstables, but:

- If workload writes a partition once and never modifies it:
  - Eventually each partition's data will be compacted into one sstable
  - In-memory *bloom filter* will usually allow reading only **one** sstable
  - Optimal
- But if workload continues to update a partition:
  - All sstables will contain updates to the same partition
  - O(logN) reads per read request
  - Reasonable, but not great ☹

Size-Tiered compaction - amplification

- **Space amplification** ☹

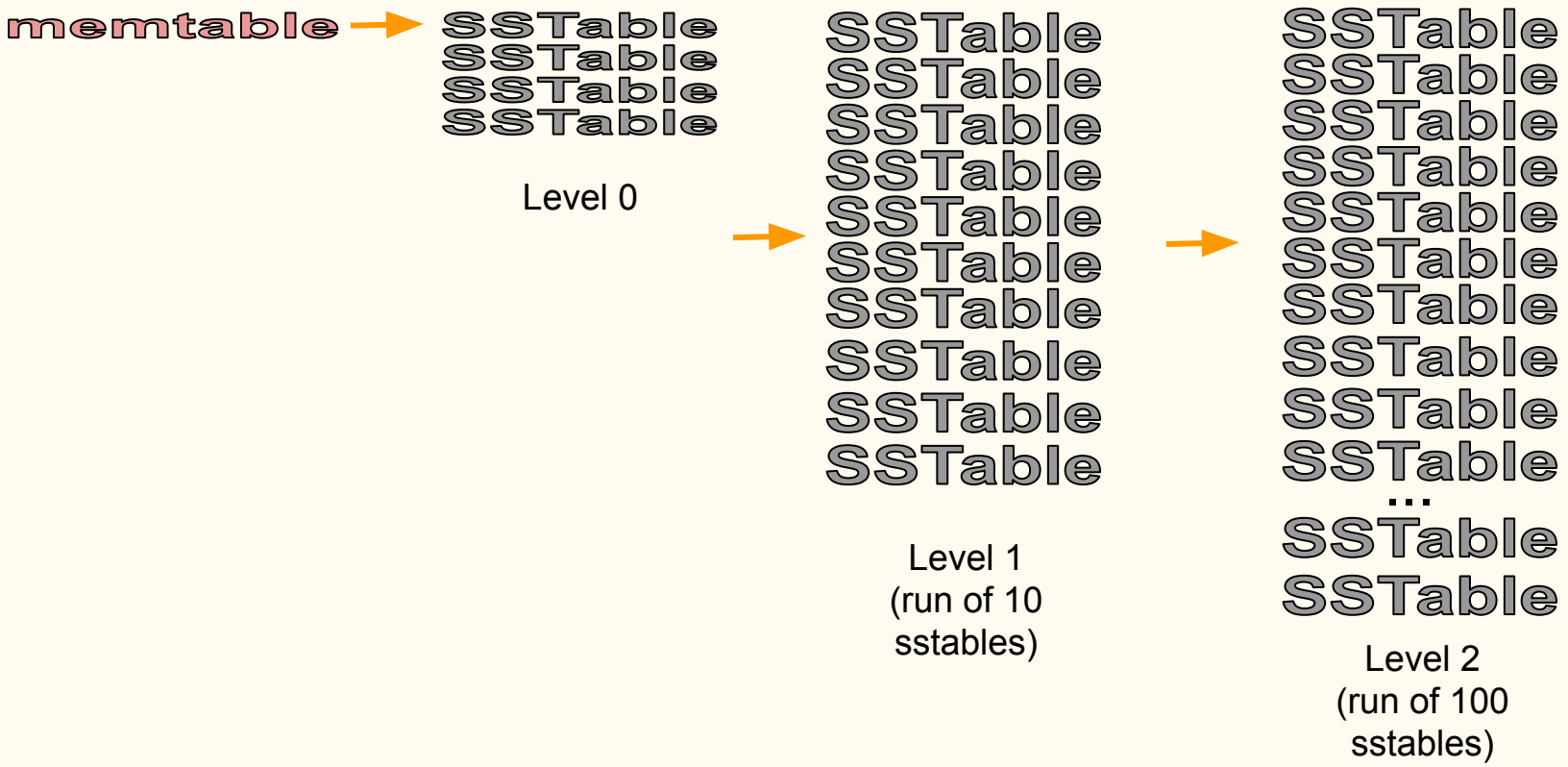
Size-Tiered compaction - amplification

- **Space amplification**:
  - Obsolete data in a huge sstable will remain for a very long time
  - Compaction needs a lot of temporary space:
    - Worst-case, needs to merge all existing sstables into one and may need **half** the disk to be empty for the merged result. (2x)
    - Less of a problem in Scylla than Cassandra because of sharding
  - When workload is overwrite-intensive, it is even worse:
    - We wait until 4 large sstables
    - All 4 overwrote the same data, so merged amount is same as in 1 sstable
    - 5-fold space amplification!
    - Or worse - if compaction is behind, there will be the same data in several tiers and have unequal sizes

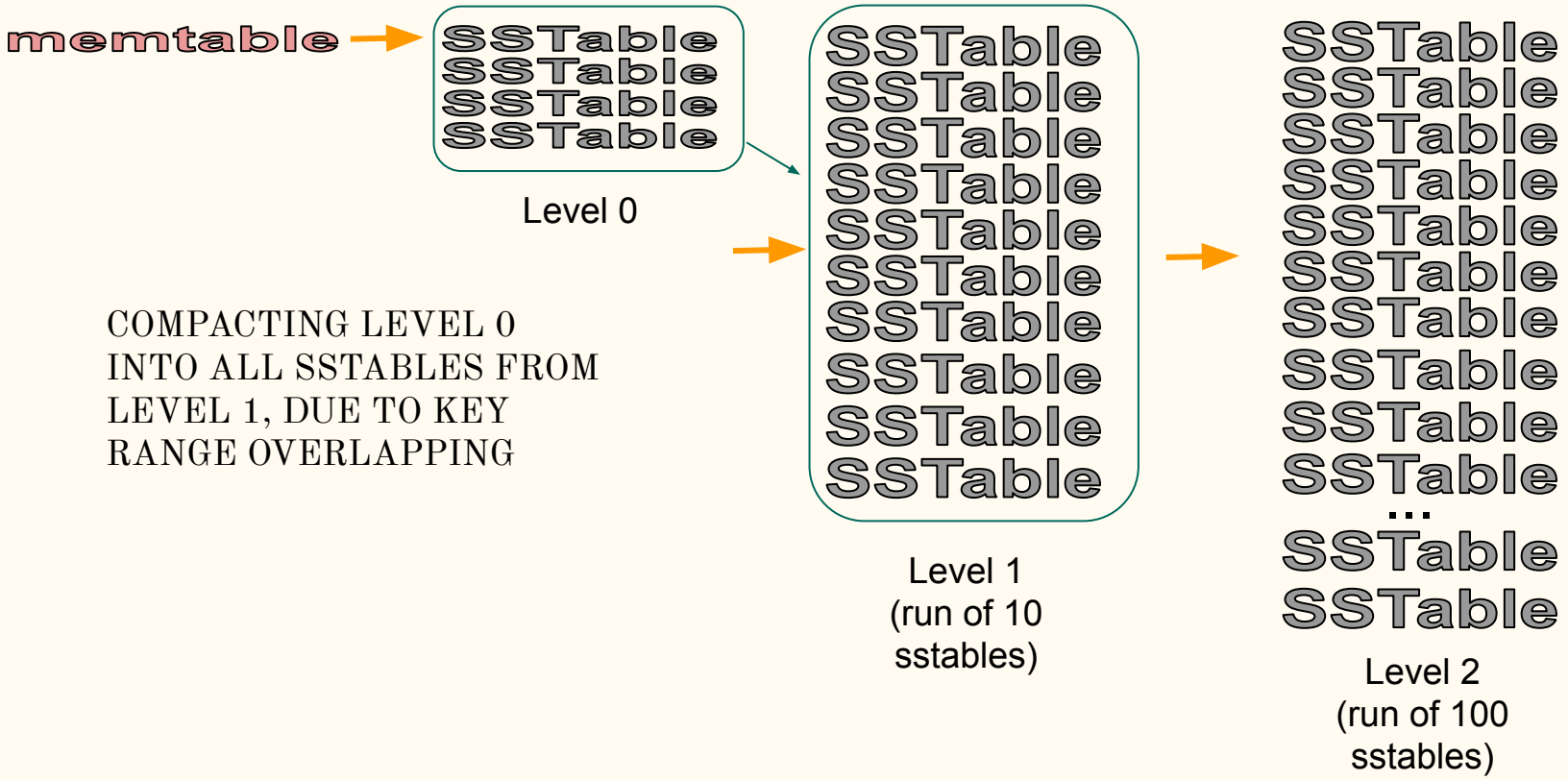
Strategy #2: Leveled Compaction

- Introduced in Cassandra 1.0, in 2011.
- Based on Google's LevelDB (itself based on Google's BigTable)
- No longer has size-tiered's huge sstables
- Instead have **runs**:
  - A **run** is a collection of small (160 MB by default) SSTables
  - Have non-overlapping key ranges
  - A huge SSTable must be rewritten as a whole, but in a run we can modify only parts of it (individual sstables) while keeping the disjoint key requirement
- In leveled compaction strategy:

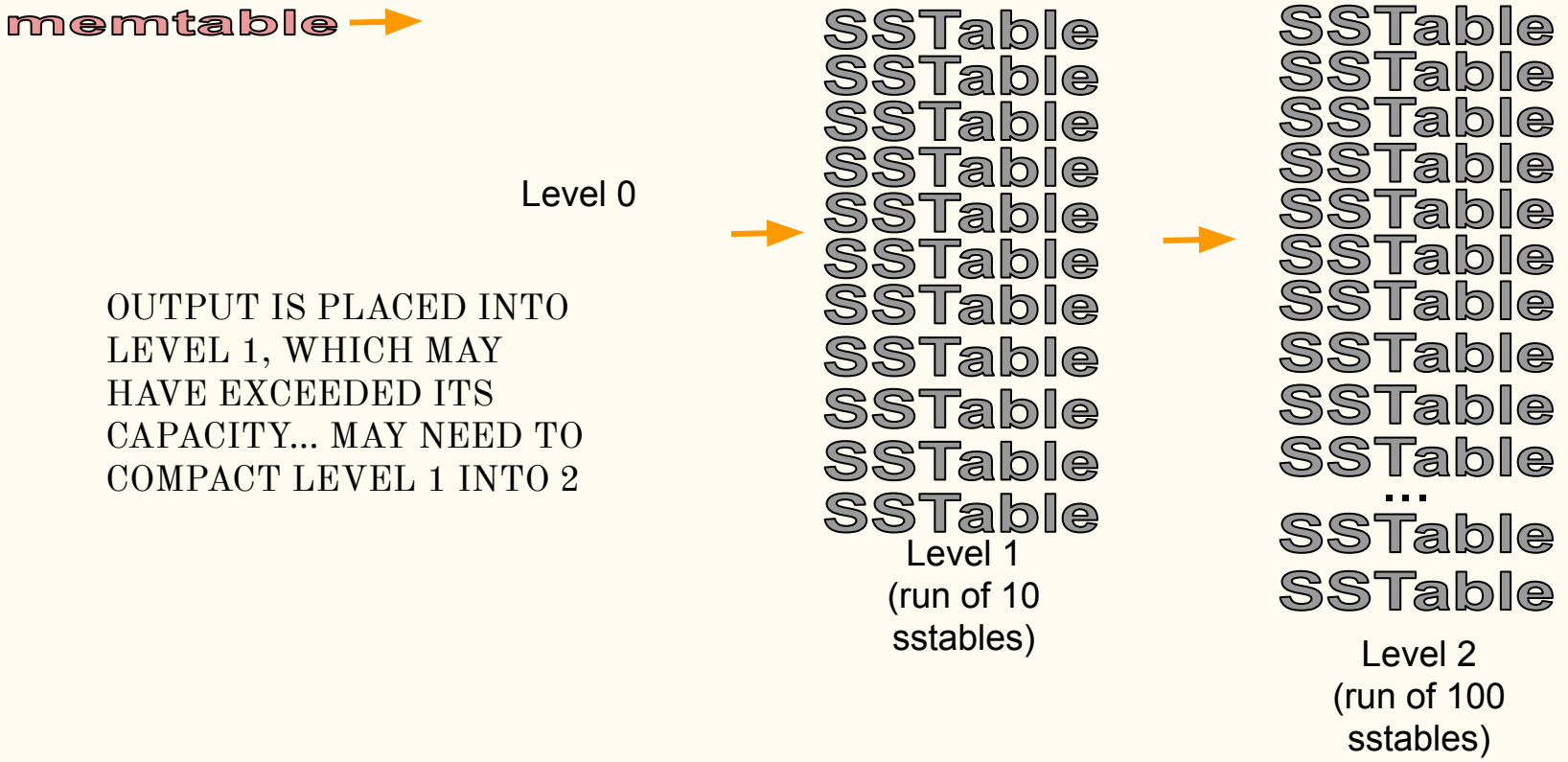
Leveled compaction strategy



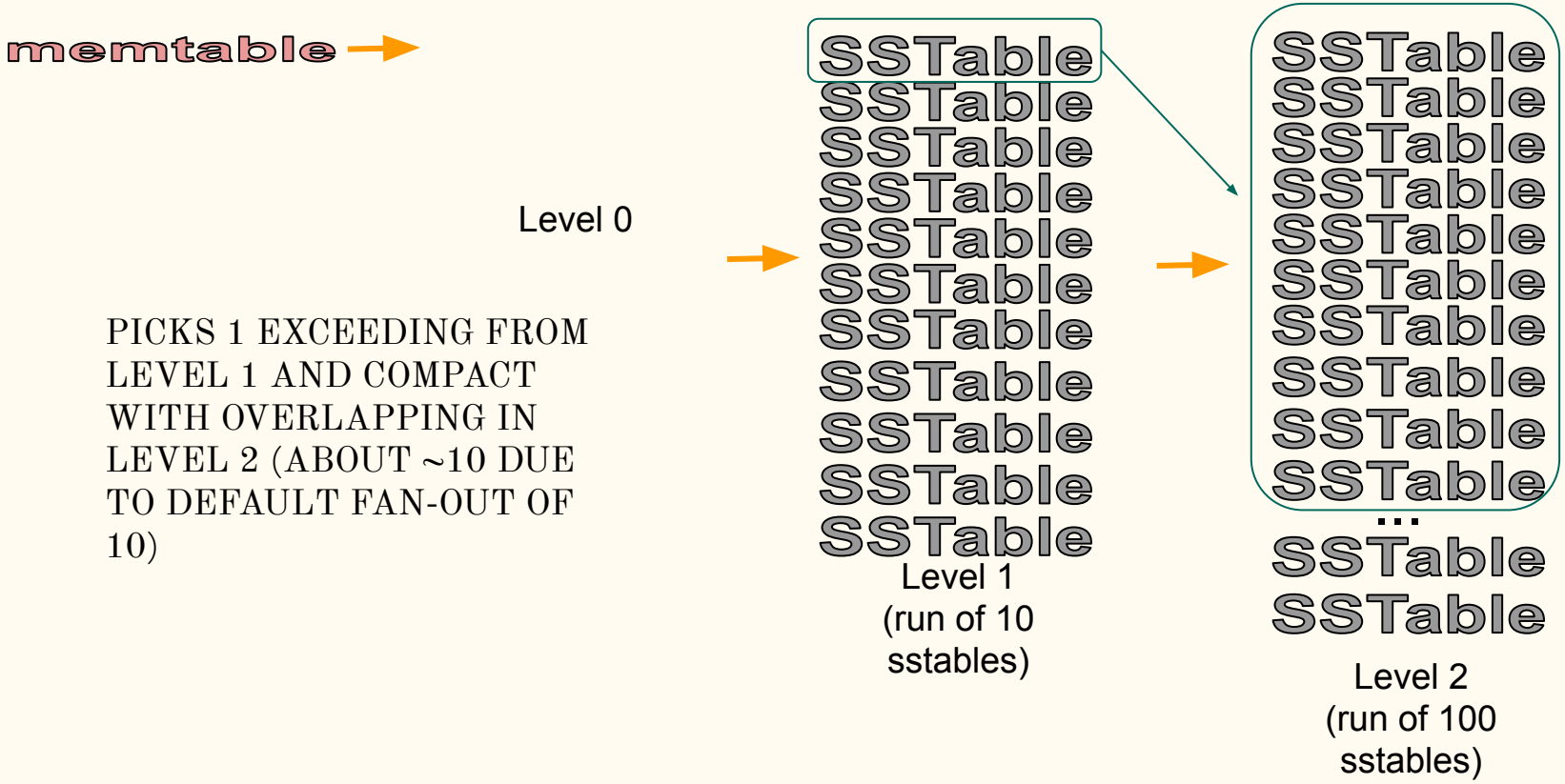
Leveled compaction strategy



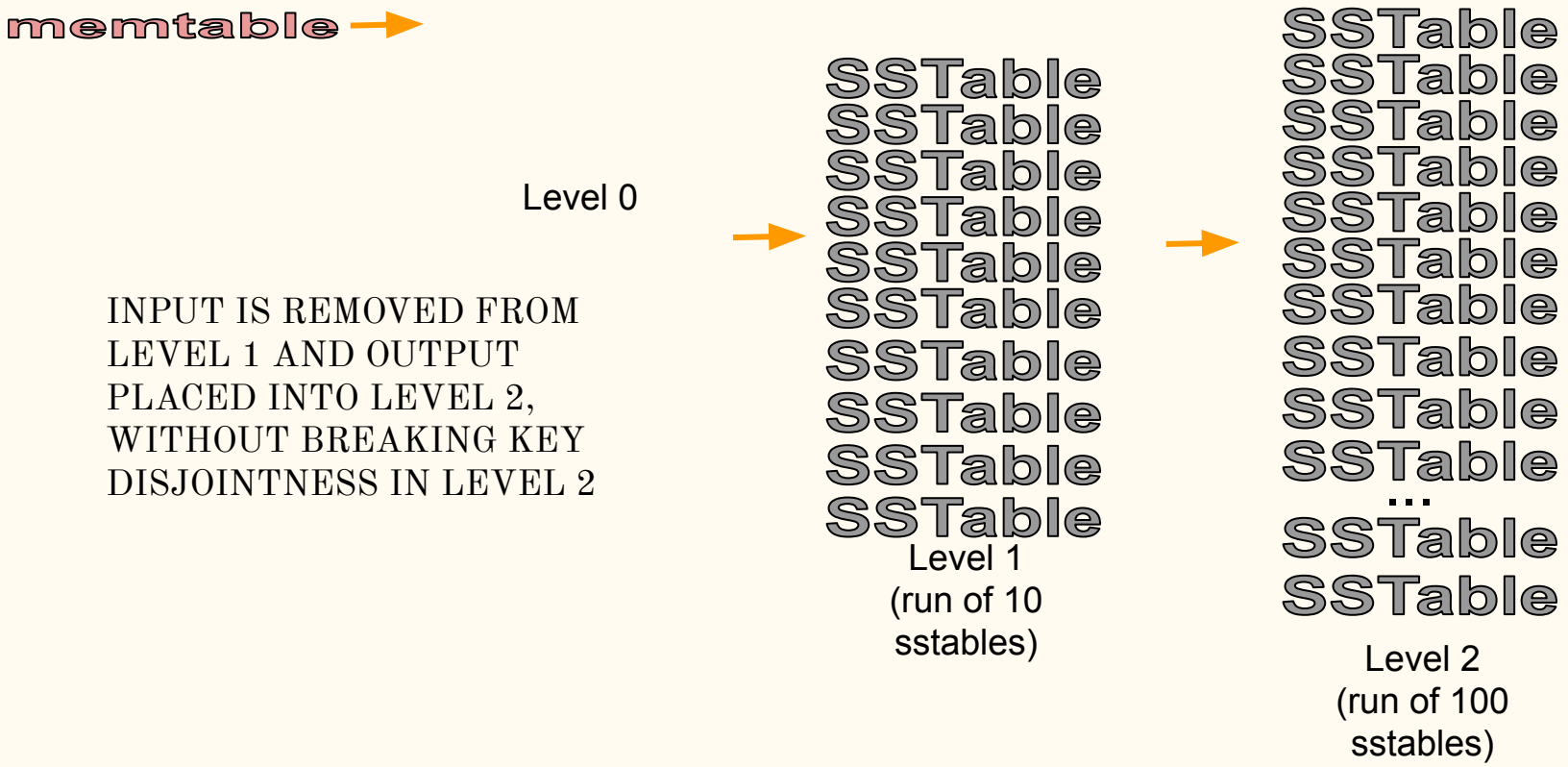
Leveled compaction strategy



Leveled compaction strategy



Leveled compaction strategy



Leveled compaction - amplification

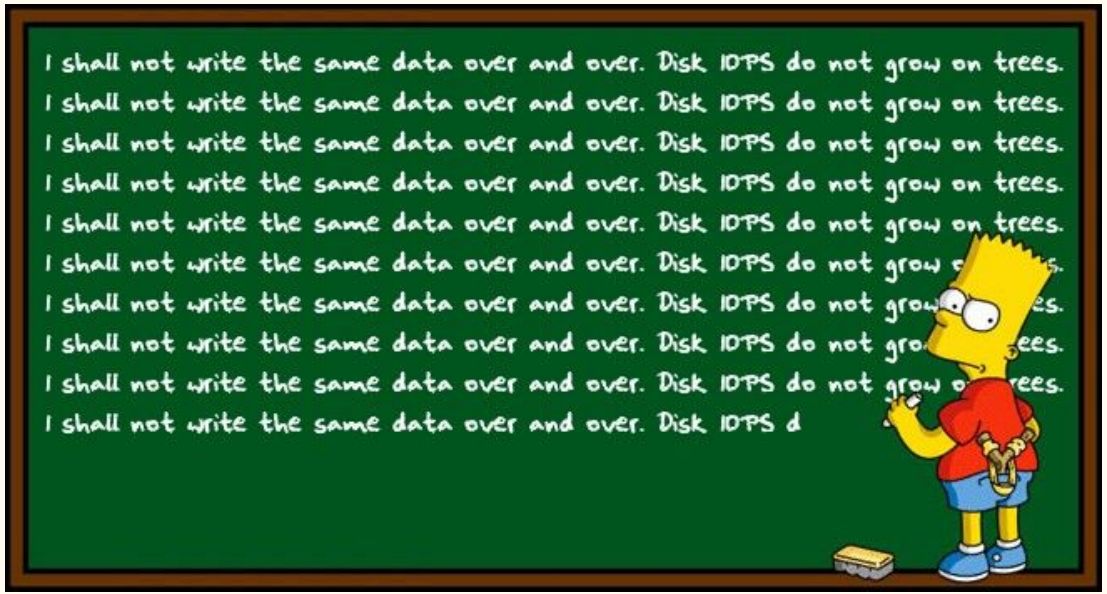
- **Space amplification:**
  - Because of sstable counts, 90% of the data is in the deepest level (if full!)
  - These sstables do not overlap, so it can’t have duplicate data!
  - So at most, 10% of the space is wasted
  - Also, each compaction needs a constant (~12*160MB) temporary space
  - Nearly optimal

Leveled compaction - amplification

- **Read amplification:**
  - We have O(N) tables!
  - But in each level sstables have disjoint ranges (cached in memory)
  - Worst-case, O(logN) sstables relevant to a partition - plus L0 size.
  - Under some assumptions (update complete rows, of similar sizes) space amplification implies: 90% of the reads will need just one sstable!
  - Nearly optimal

Leveled compaction - amplification

- **Write amplification**: ☹️



Leveled compaction - amplification

- **Write amplification**:
  - Again, most of the data is in the deepest level k
    - E.g., k=3 is enough for 160 GB of data (per shard!)
    - All data was written once in L0, then compacted into L1, … then to Lk
    - So each row written k+1 times
  - For each input (level i>1) sstable we compact, we compact it with ~12 overlapping sstables in level i+1. Writing ~13 output sstables. (lower for L0)
  - Worst-case, write amplification is around 13k
  - Also O(logN) but higher constant factor than size-tiered...
  - If **enough writing and LCS can't keep up**, its read and space advantages are lost
  - If also have cache-miss reads, they will get less disk bandwidth

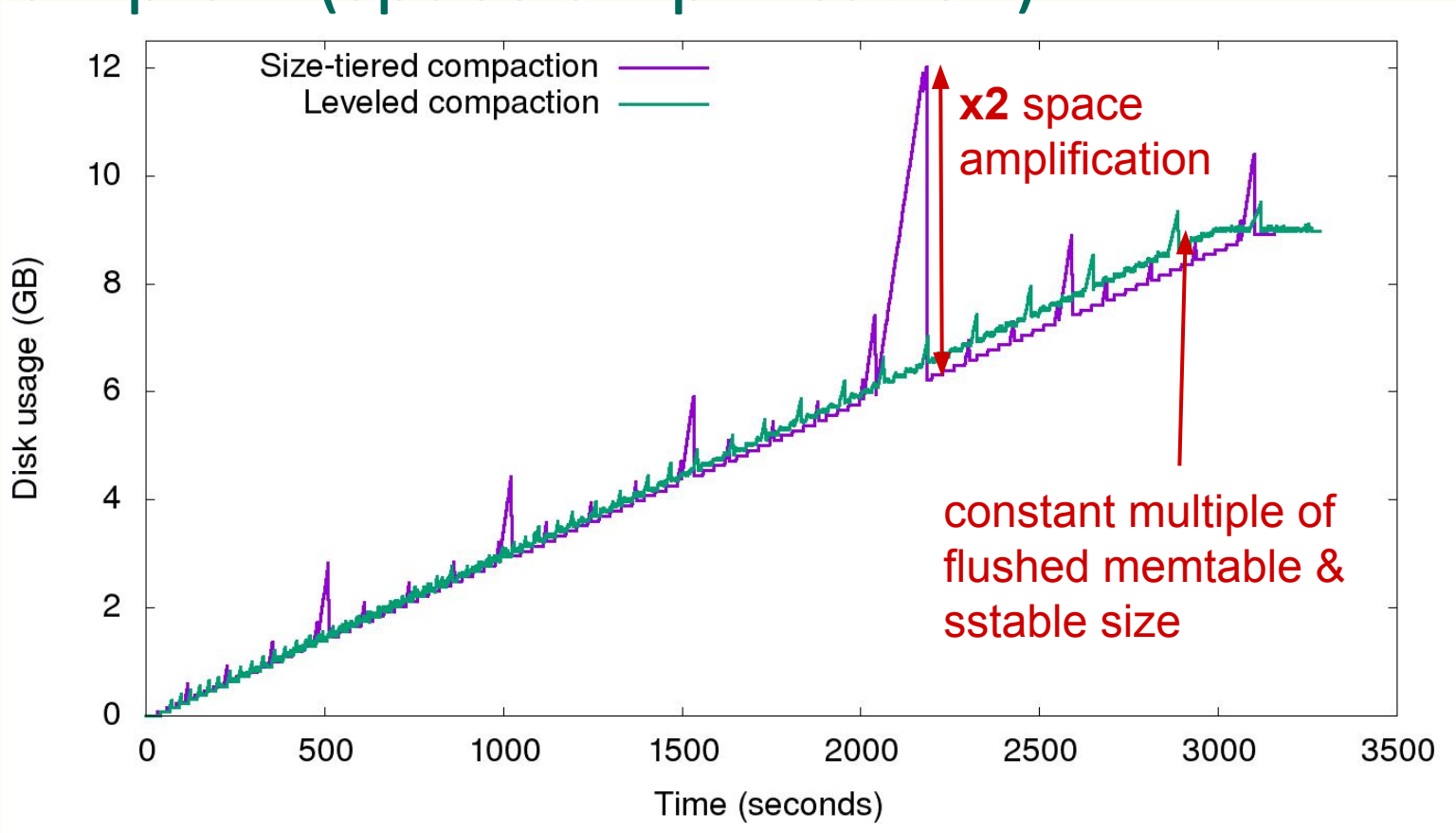
Example 1 - write-only workload

- Write-only workload
  - Cassandra-stress writing 30 million partitions (about 9 GB of data)
  - Constant write rate 10,000 writes/second
  - One shard

Example 1 - write-only workload

- Size-tiered compaction:
  at some points needs twice the disk space
  - In Scylla with many shards, “usually” maximum space use is not concurrent
- Level-tiered compaction:
  more than double the amount of disk I/O
  - Test used smaller-than default sstables (10 MB) to illustrate the problem
  - Same problem with default sstable size (160 MB) - with larger workloads

Example 1 (space amplification)



Example 1 (write amplification)

- Amount of actual data collected: 8.8 GB
- Size-tiered compaction: 50 GB writes (4 tiers + commit log)
- Leveled compaction: 111 GB writes

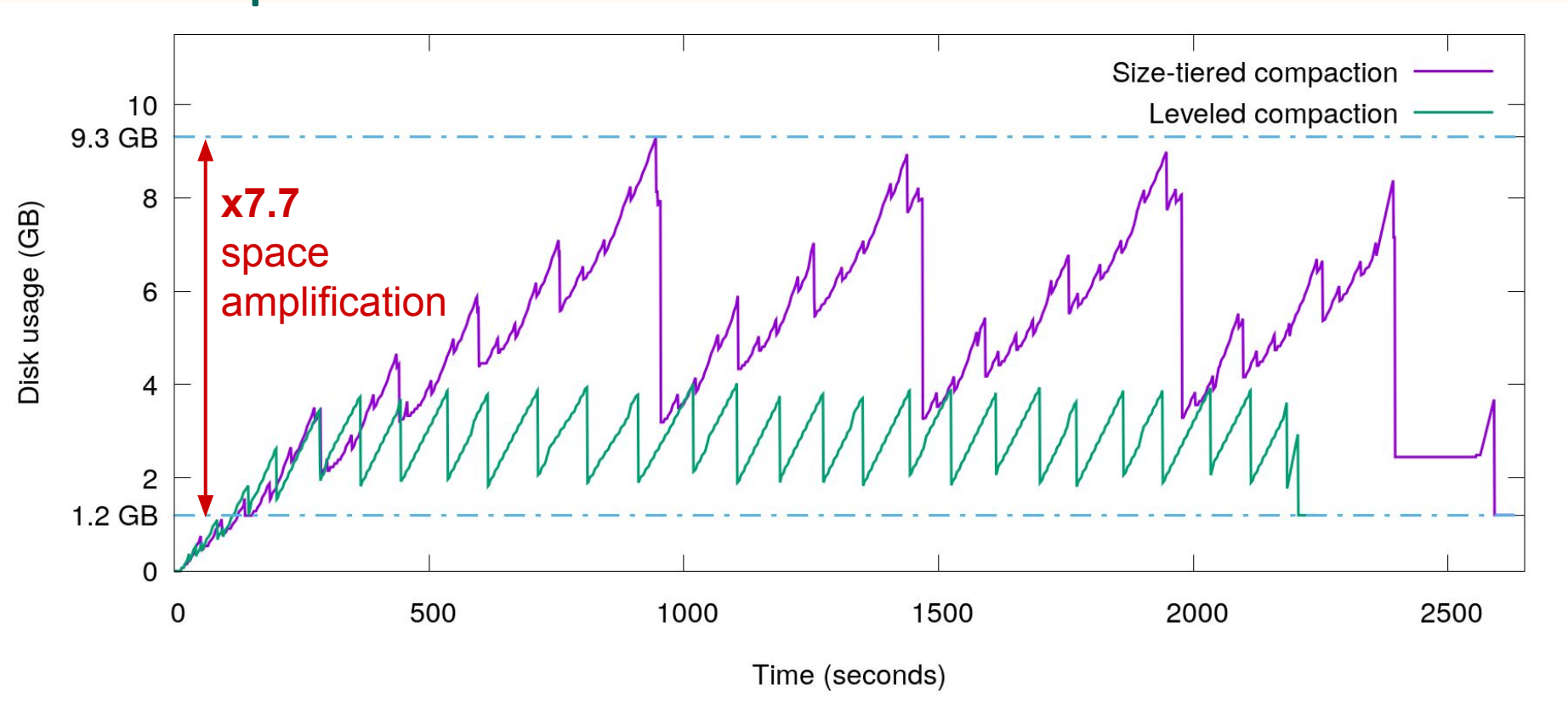
Example 1 - note

- Leveled compactions write amplification is not only a problem with 100% write...
- Can have just 10% writes and an amplified write workload so high that
  - Uncached reads slowed down because we need the disk to write
  - Compaction can’t keep up, uncompacted sstables pile up, even slower reads
- Leveled compaction is unsuitable for many workloads with a non-negligible amount of writes even if they seem “read mostly”

Example 2 - overwrite workload

- Write 15 times the same 4 million partitions
  - cassandra-stress write n=4000000 -pop seq=1..4000000 -schema "replication(strategy=org.apache.cassandra.locator.SimpleStrategy,factor=1)"
  - In this test cassandra-stress not rate limited
  - Again, small (10MB) LCS tables
- Necessary amount of sstable data: 1.2 GB
- STCS space amplification: x7.7 !
- LCS space amplification lower, constant multiple of sstable size
- Incremental will be around x2 (if it decides to compact fewer files)

Example 2

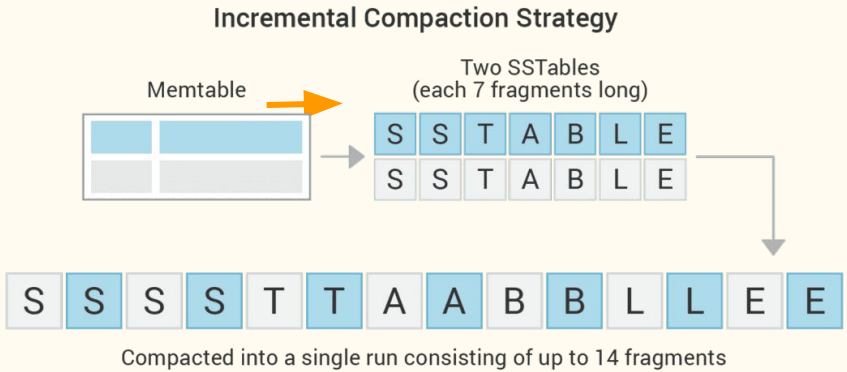

Size-tiered compaction
Leveled compaction
10
9.3 GB
8
6
4
2
1.2 GB
0
x7.7
space
amplification
Disk usage (GB)
0
500
1000
1500
2000
2500
Time (seconds)

Can we create a new compaction strategy with

- Low write amplification of size-tiered compaction
- Without its high temporary disk space usage during compaction?
  - Meaning no longer a need to leave 50% of the disk unused

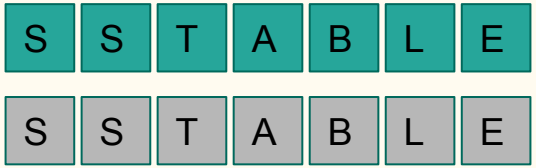
Strategy #3: Incremental Compaction

- Size-tiered compaction needs temporary space because we only remove a huge sstable after we fully compact it.
- Let's split each huge sstable into a **run** (a la LCS) of "fragments":
  - Treat the entire run (not individual sstables) as a file for STCS
  - Remove individual sstables as compacted. Low temporary space.



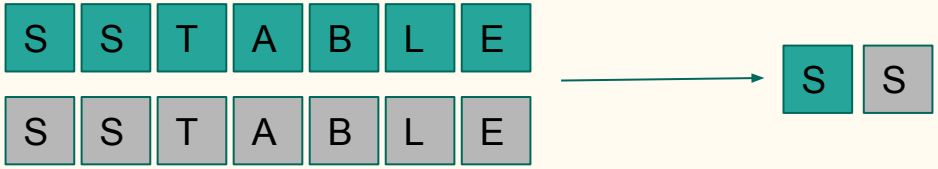
Strategy #3: Incremental Compaction

TWO SSTABLE RUNS (7 FRAGMENTS EACH)



Strategy #3: Incremental Compaction

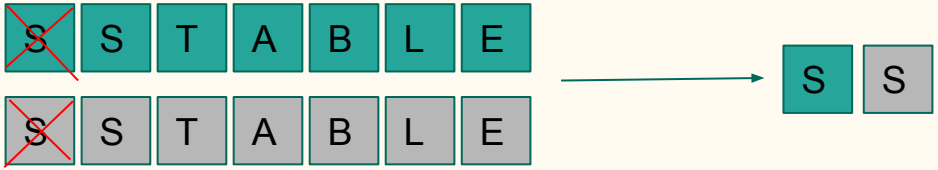
TWO SSTABLE RUNS (7 FRAGMENTS EACH)



WRITES 2 OUTPUT FRAGMENTS, EXHAUSTING THE FIRST FRAGMENT OF INPUT SSTABLE RUNS

Strategy #3: Incremental Compaction

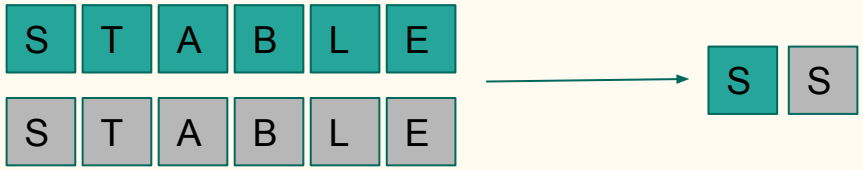
TWO SSTABLE RUNS (7 FRAGMENTS EACH)



RELEASES FIRST FRAGMENT OF INPUT SSTABLE RUNS AS THEY'RE EXHAUSTED

Strategy #3: Incremental Compaction

TWO SSTABLE RUNS (7 FRAGMENTS EACH)



RELEASED FIRST FRAGMENT OF INPUT SSTABLE RUNS AS THEY'RE EXHAUSTED

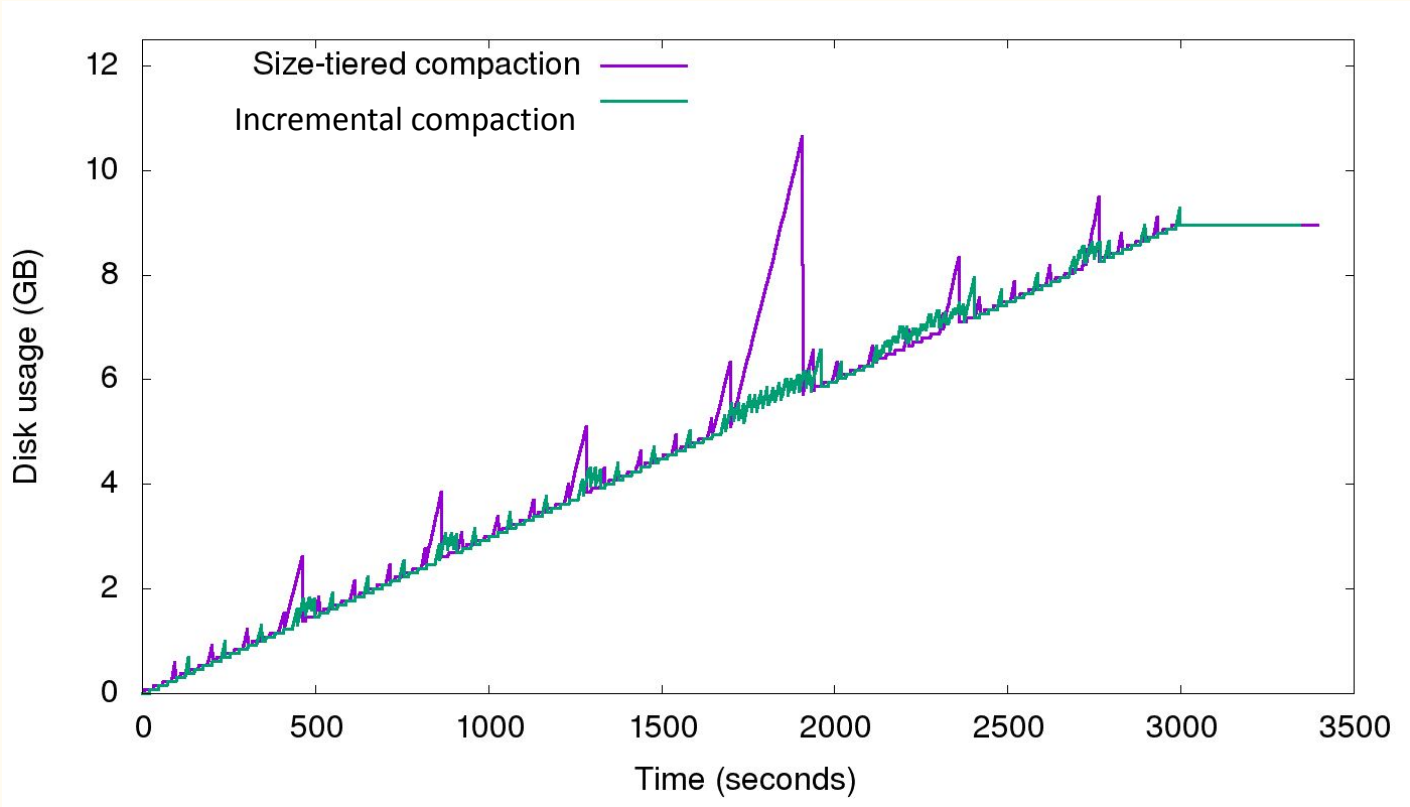
Strategy #3: Incremental Compaction

- Solve 4x worst-case in overwrite workloads with other techniques:
  - Compact fewer sstables if disk is getting full
    - Not a risk because small temporary disk needs
  - Compact fewer sstables if they have large overlaps

Incremental compaction - amplification

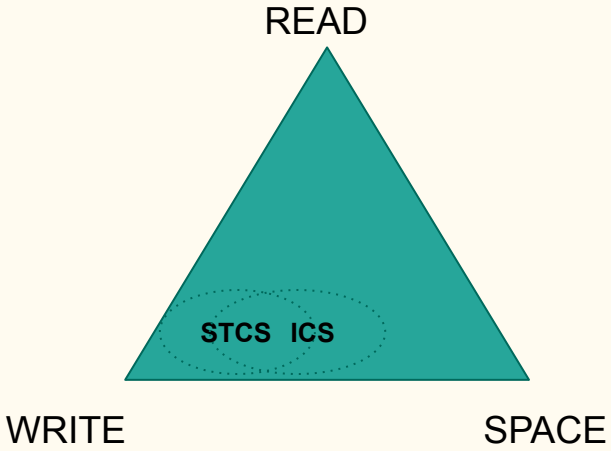
- **Space amplification**:
  - Small constant temporary space needs, even smaller than LCS (M*S per parallel compaction, e.g., M=4, S=160 MB)
  - Overwrite-mostly still a worst-case, but 2-fold instead of 5-fold
  - Optimal.
- **Write amplification**:
  - O(logN), small constant — same as Size-Tiered compaction
- **Read amplification**:
  - Like Size-Tiered, at worst O(logN) if updating the same partitions

Example 1 - Size Tiered vs Incremental



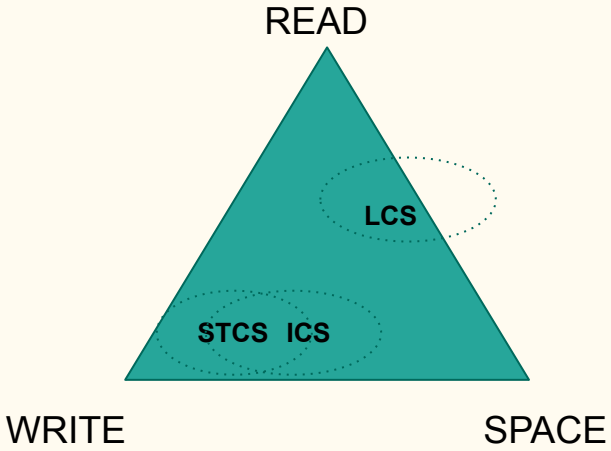
Is it enough?

- Space overhead problem was efficiently fixed in Incremental (ICS), however…
- Incremental (ICS) and size-tiered (STCS) strategies share the same space amplification (~2-4x) when facing overwrite intensive workloads, where:
  - They cover a similar region in the three-dimensional efficiency space (RUM trade-offs):



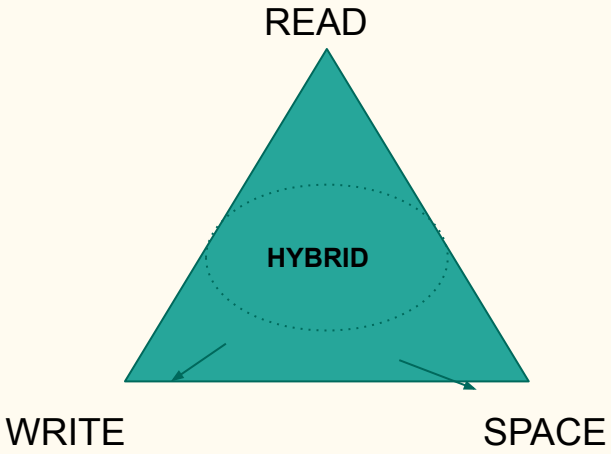
Turns out it’s not enough. But can we do better?

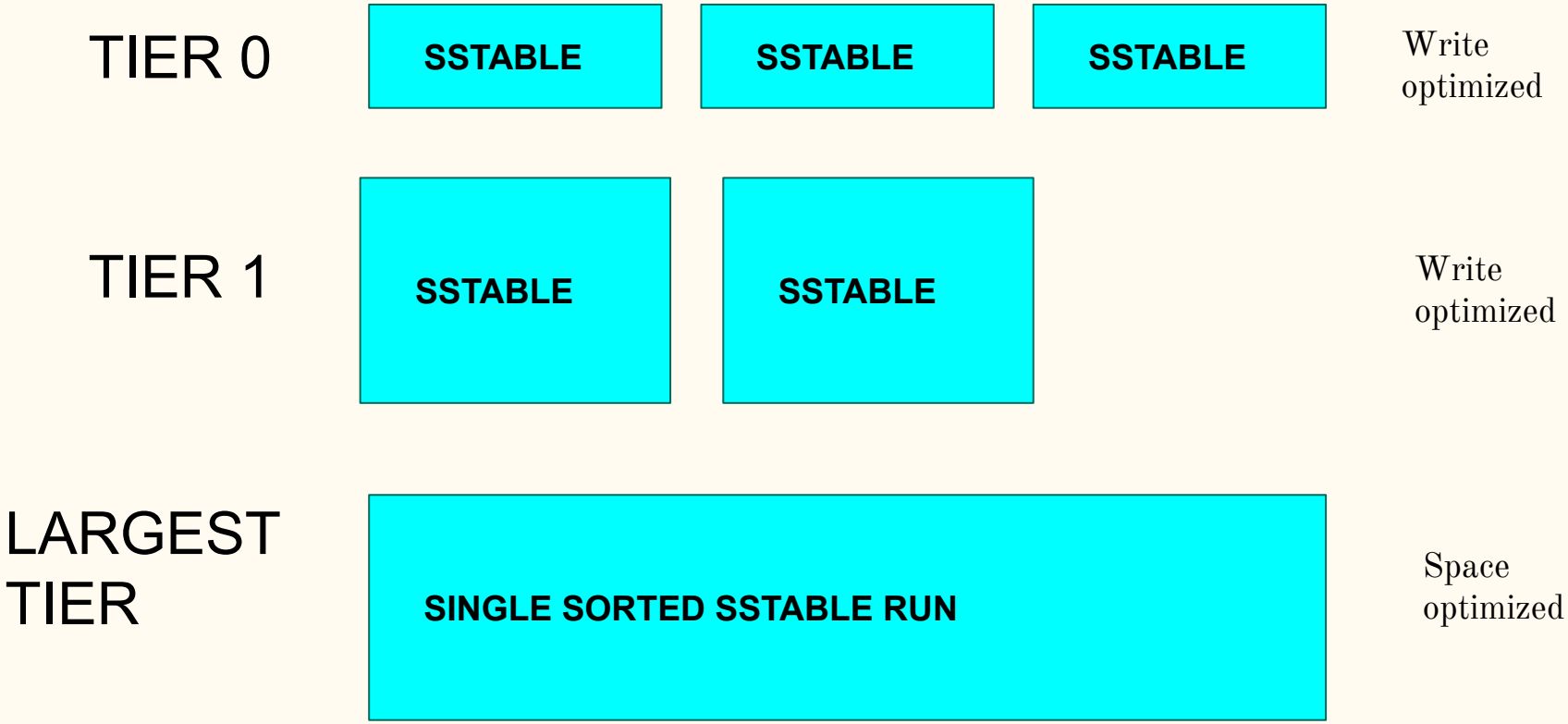
- Leveled strategy and Size-tiered (or ICS) cover different regions
  - Interesting regions cannot be reached with either strategies.
  - But interesting regions can be reached by combining data layout of both strategies
    - i.e. a hybrid (tiered+leveled) approach

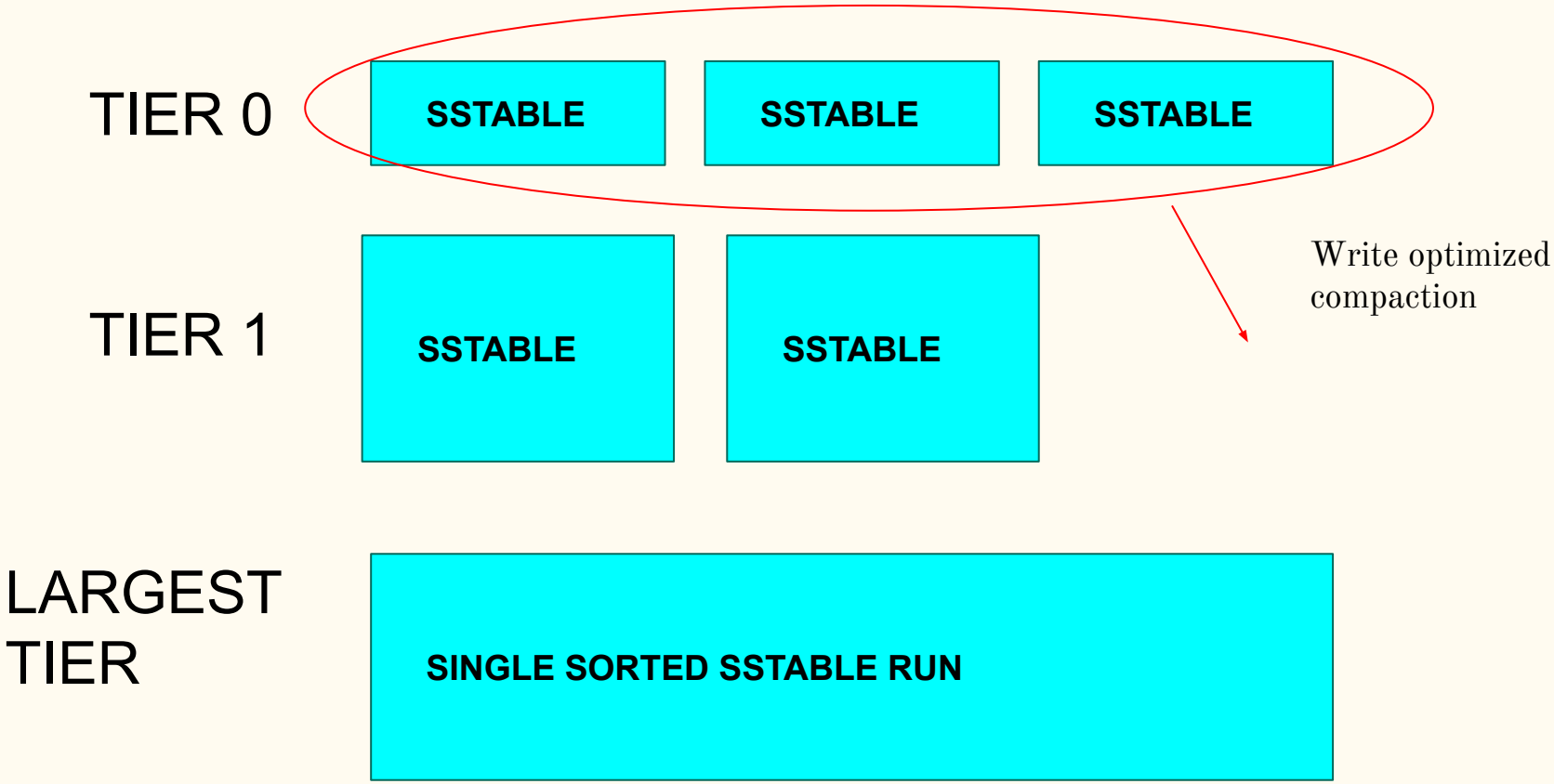


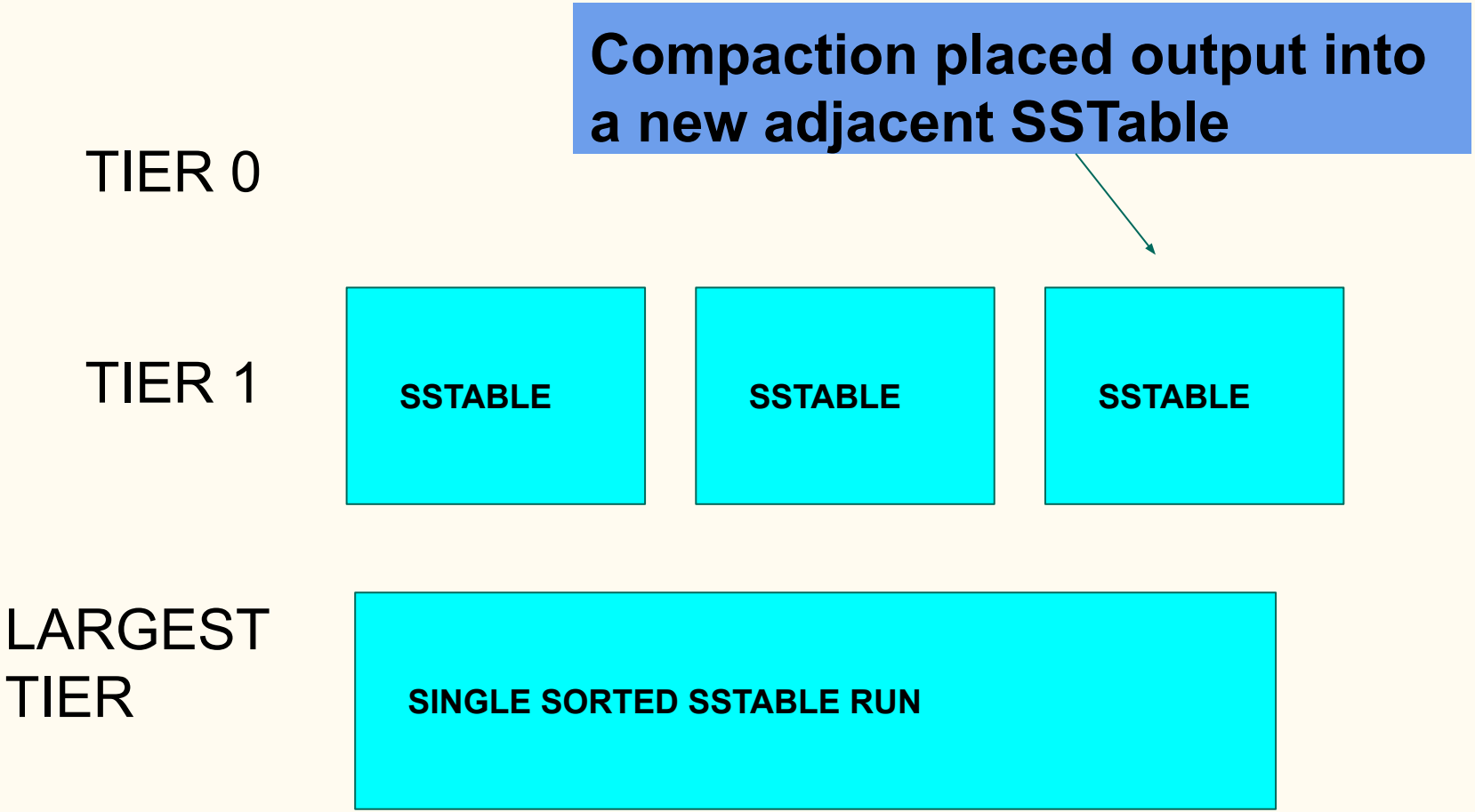
How does hybrid work under the hood?

- All the size tiers **but the largest one** can have more than 1 sorted run = write optimized
- **Largest tier** will be a single sorted run = space optimized
- The hybrid approach will allow compaction to dynamically adapt to the workload
  - Under heavy write load, compaction strategy will work to meet write latencies.
  - Otherwise, optimize space efficiency



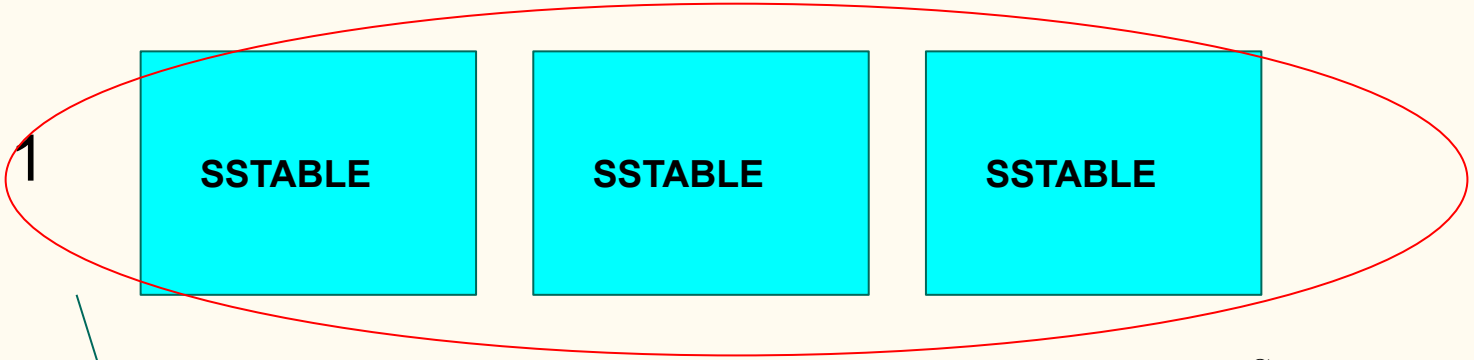

TIER 0
SSTABLE
SSTABLE
SSTABLE
Write optimized
TIER 1
SSTABLE
SSTABLE
Write optimized
LARGEST TIER
SINGLE SORTED SSTABLE RUN
Space optimized


TIER 0
SSTABLE
SSTABLE
SSTABLE
TIER 1
SSTABLE
SSTABLE
Write optimized
compaction
LARGEST
TIER
SINGLE SORTED SSTABLE RUN

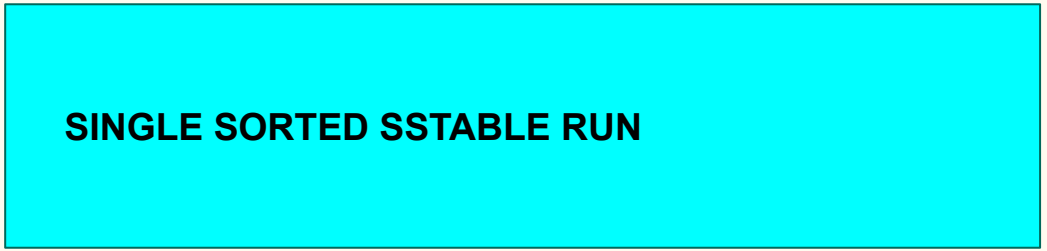

Compaction placed output into a new adjacent SSTable
TIER 0
TIER 1
SSTABLE
SSTABLE
SSTABLE
LARGEST TIER
SINGLE SORTED SSTABLE RUN

TIER 0

TIER 1



LARGEST TIER



Space optimized compaction

TIER 0

TIER 1

LARGEST TIER

TIER 1 and LARGEST TIER were compacted together (cross-tier compaction).

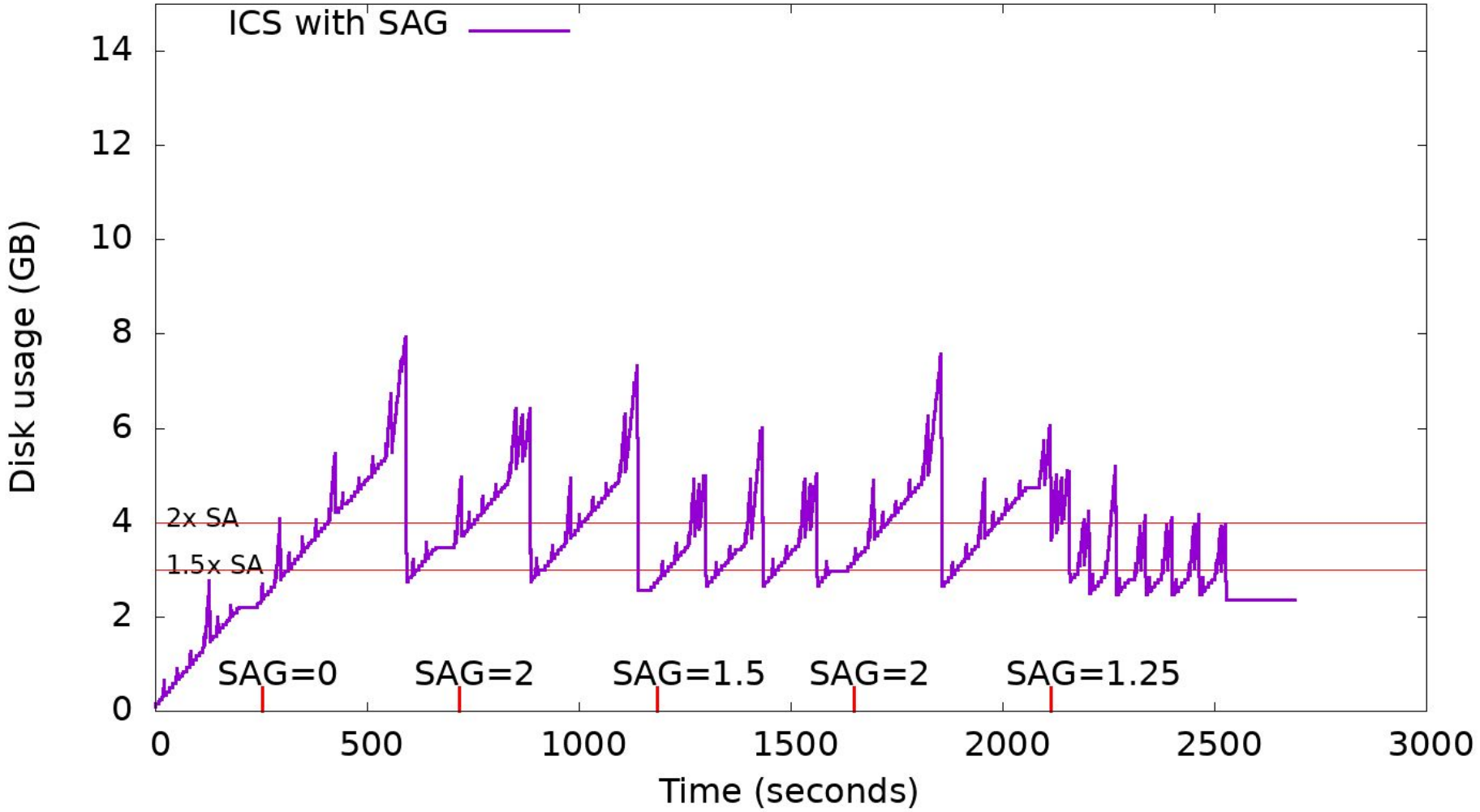
SINGLE SORTED SSTABLE RUN

Write amplification with hybrid policy

- When largest tier is not involved, the write amplification (WA) is exactly as in size-tiered. 1 WA per each tier.
- When doing cross-tier compaction, the WA depends on the size ratio of largest and second-largest tier.
  - The higher the size ratio, the higher will be the WA.
  - So it’s good to target a low size ratio to minimize the effect of cross-tier compaction on WA.
  - For example, wait for second-largest to be at least half the size of largest before triggering cross-tier compaction.

Hybrid policy through space amplification goal

- Space amplification goal (SAG) is a property to control the size ratio of the largest and the second largest-tier.
- It’s a value between 1 and 2 (defined in table’s schema). Value of 1.5 implies cross-tier when second-largest is half the size of largest.
- Effectively, helps controlling the space amplification. Not an upper bound on SA, but results show that compaction will be working towards reducing the actual SA to below the configured value.
- The lower the SAG value the lower the SA but the higher the WA. Up to user to find an optimal configuration value based on the needs.


ICS with SAG
Disk usage (GB)
Time (seconds)
0
2
4
6
8
10
12
14
0
500
1000
1500
2000
2500
3000
2x SA
1.5x SA
SAG=0
SAG=2
SAG=1.5
SAG=2
SAG=1.25

Policies' amplification summary

Workload	Size-Tiered	Leveled	Incremental	Time-Window
Write-only	2x peak space	2x writes	**Best**	-
Overwrite	Huge peak space	write amplification	**high peak space, but not like size-tiered**	-
Read-mostly, few updates	read amplification	**Best**	read amplification	-
Read-mostly, but a lot of updates	read and space amplification	write amplification may overwhelm	**read amplification**	-
Time series	write, read, and space ampl.	write and space amplification	write and read amplification	**Best**

…

Ok, this is all cool, but what can we do to not allow compaction to use more resources (CPU & IO) than needed? Can we self tune it?

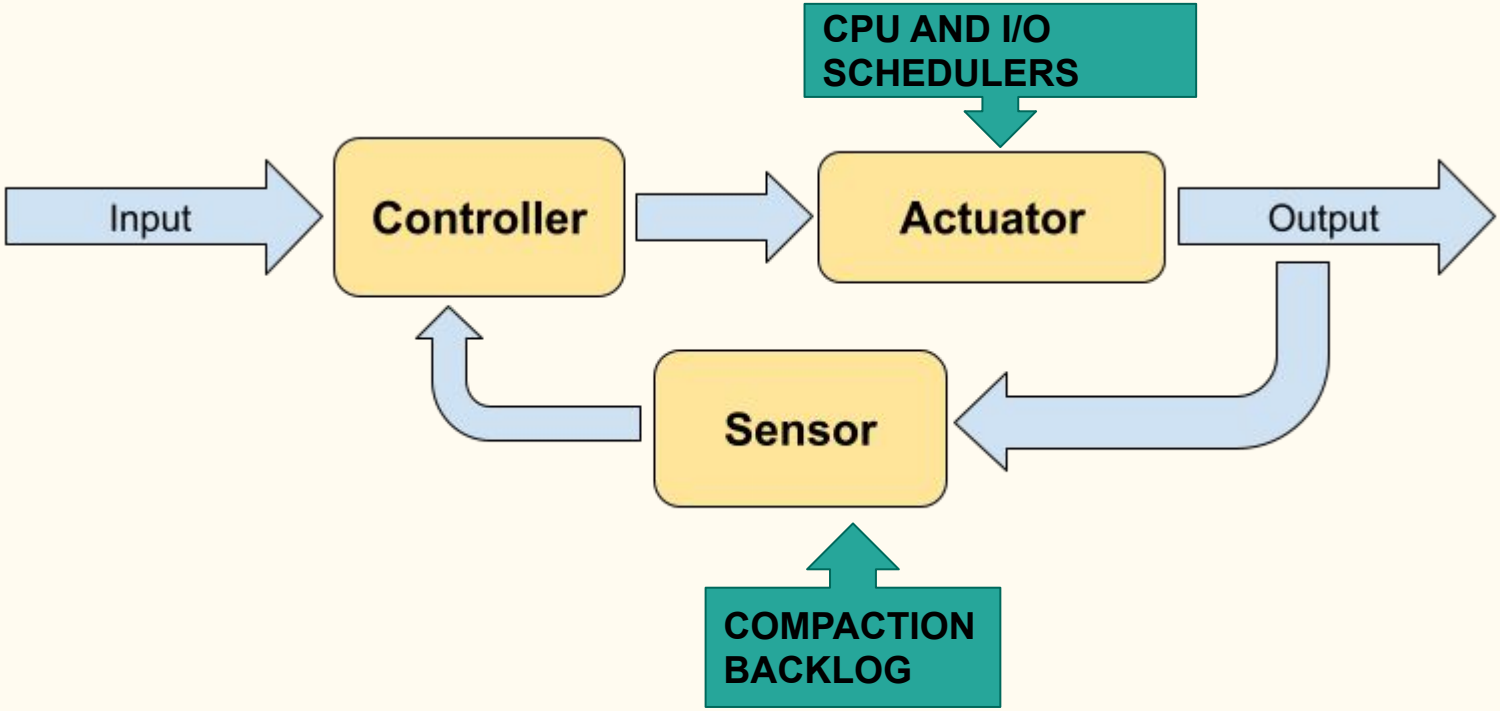
Problem statement

- If we spend fewer resources compacting existing files, we will likely be able to achieve faster write rates
- However, reads are going to suffer as they will now need to touch more files.
- The common option is to push the decision to the user in the form of tunables.
- The user is then responsible for a trial-and-error tuning cycle to try to find the right number for the matching workload.
- This approach is fragile. Manual tuning is not resilient to changes in the workload, many of which are unforeseen.
- Endless tuning cycle for DBAs

Leveraging PID controller theory for self tuning

- So compaction controller was born *(available since Scylla 2.2)*
  - Based on industrial PID controller
  - Essentially, a closed loop (feedback) control system
  - The world wants to be in a particular state
  - The current state is fed back to the control system
  - The control system acts to bring the system back to the goal
- Good example is air conditioner

The controller: Under the hood



Measuring compaction backlog

- Each compaction policy will have its own method for measuring the current backlog
- For size tiered policy, we're using the formula:

$$\mathbf{Bi = S - C * \log(T / S)}$$

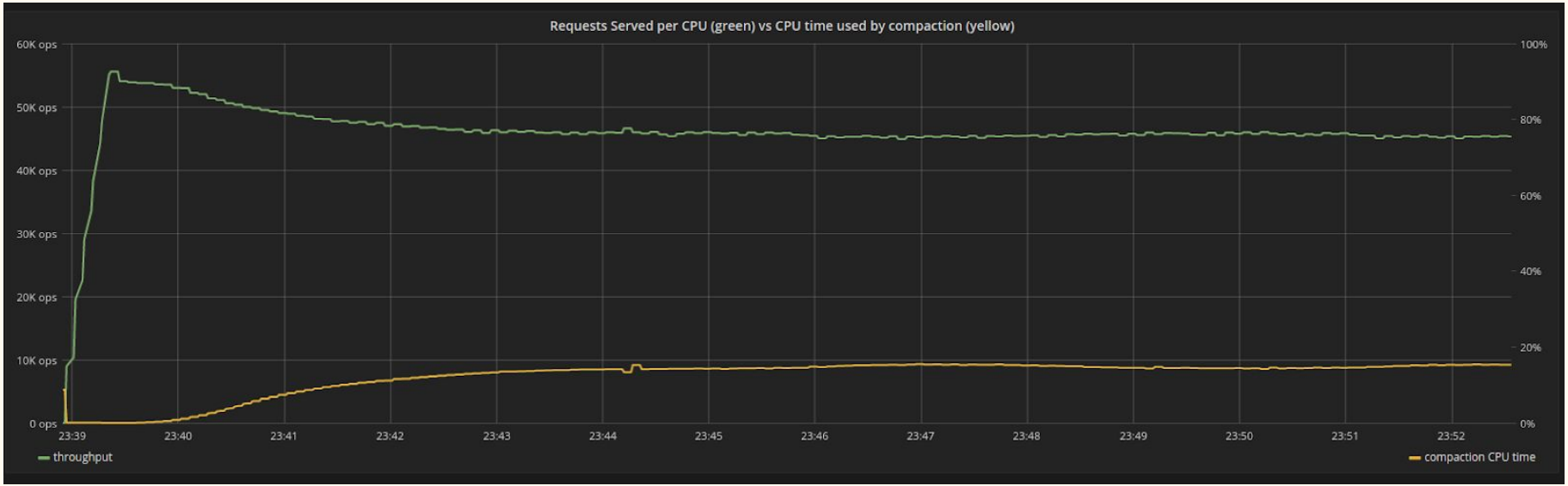
Where **S** is the size of the SSTable which needs to be compacted

C is the amount of bytes already compacted from the SSTable

T is the Table Size, **log (T / S)** is the write amplification factor.

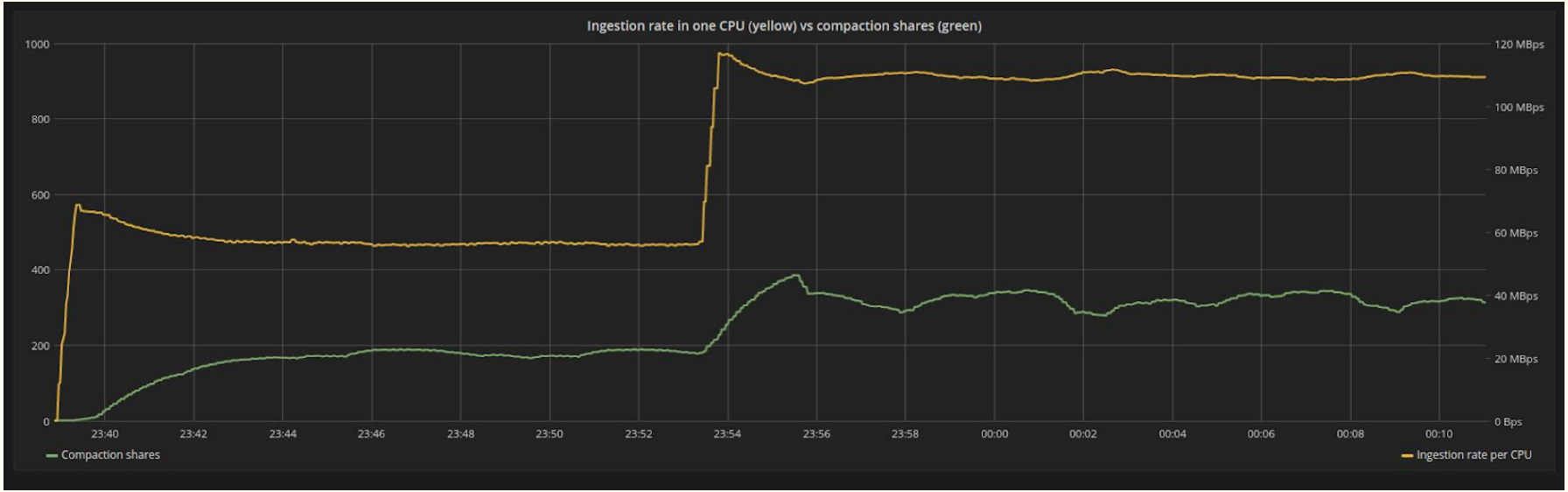
- The total backlog is normalized and then feeded into the controller
- Which in turn will decide to either decrease or devote more resources to compaction

Compaction controller in practice



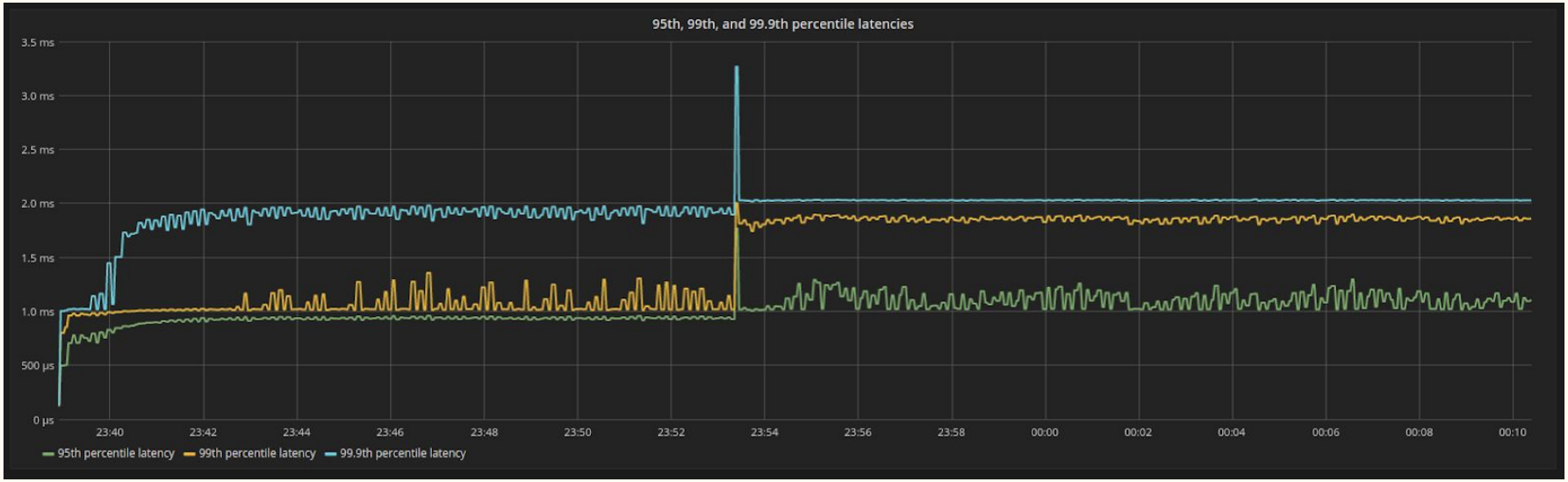
How the system responds to an ingestion-only workload; writing 1kB values into a fixed number of random keys so that the system eventually reaches steady state.

Compaction controller in practice #2



-

Compaction controller in practice #3



Controller goals

- Making sure that the incoming workload is not severely disrupted by compactions
- Nor that the compaction backlog is so big that reads are later penalized
- Allowing DBA to do more important tasks by not being stuck in a endless tuning cycle
- Win-win.

THANK YOU

Any questions?

Please stay in touch

raphaelsc@scylladb.com

twitter: @raphael_scarv