Operating Cassandra NoSQL database across the globe

Jiří Horký

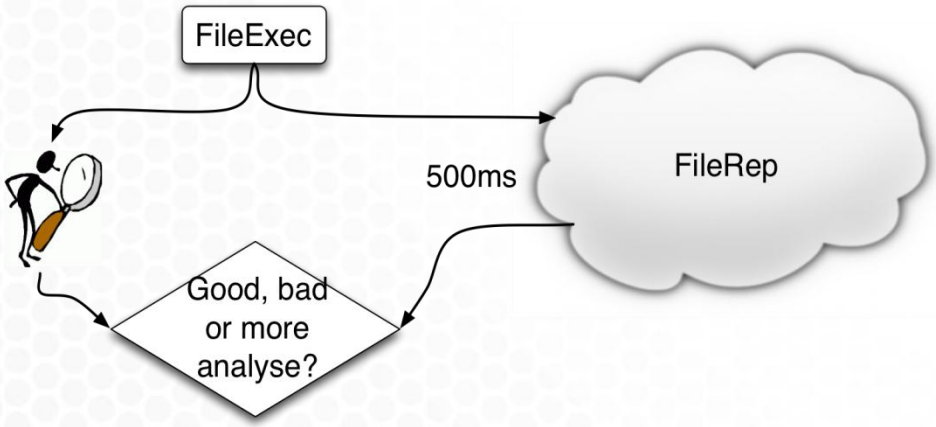
horky@avast.com

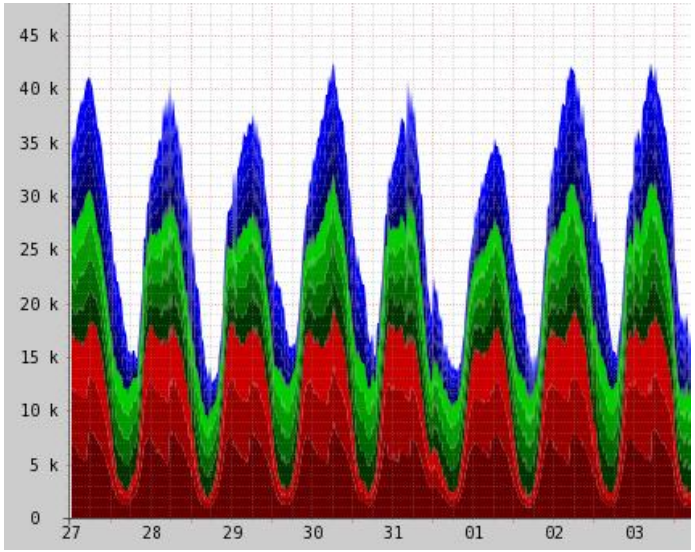
Cassandra @ Avast Cloud

- File Reputation service
  - 3 replicas, 4.5TB each, 1 billion of keys, 60 billion of columns
- Thumbnails of users' backups
  - 1 replica – 5TB of data across 5 nodes
- SecureLine (VPN) TCP sessions information
- Generally everywhere...

File Reputation of PXE

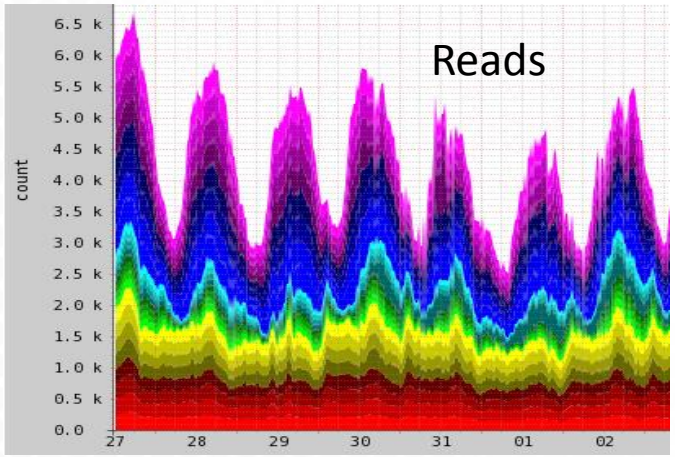
- Requirements
  - Distinguish between widely used EXE files (“Topstars”) and “loaners”
  - Key for the file is its **SHA256**
  - **~45k req/s** in peak
  - Respond in **500ms** in total (network lat. included) → **distributed DB**
  - Cache the topstars information in memory (~98%), query for rest in Cassandra

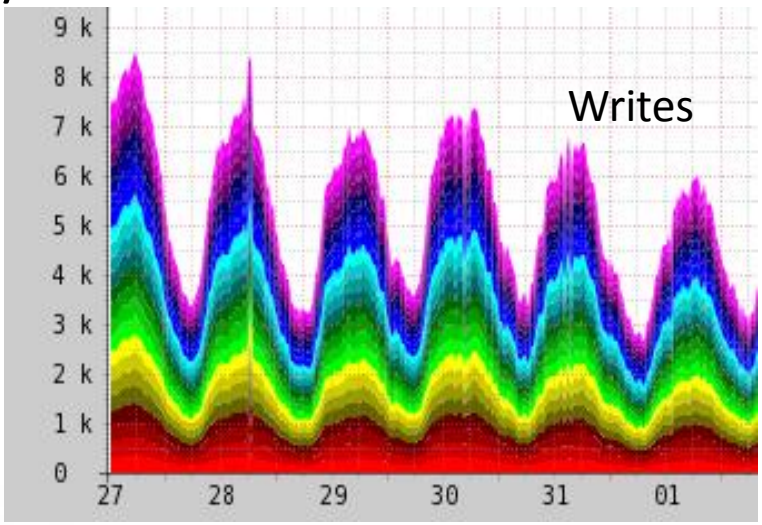




File Reputation - Cassandra

- Globally
  - 1 replica per DC (MIA, SEA, PRG)
- Each DC
  - **8 nodes, 4.5 TiB** data
  - **1 Billion** of keys (files)
  - peak: **6.5k** reads/s, **8k** writes/s

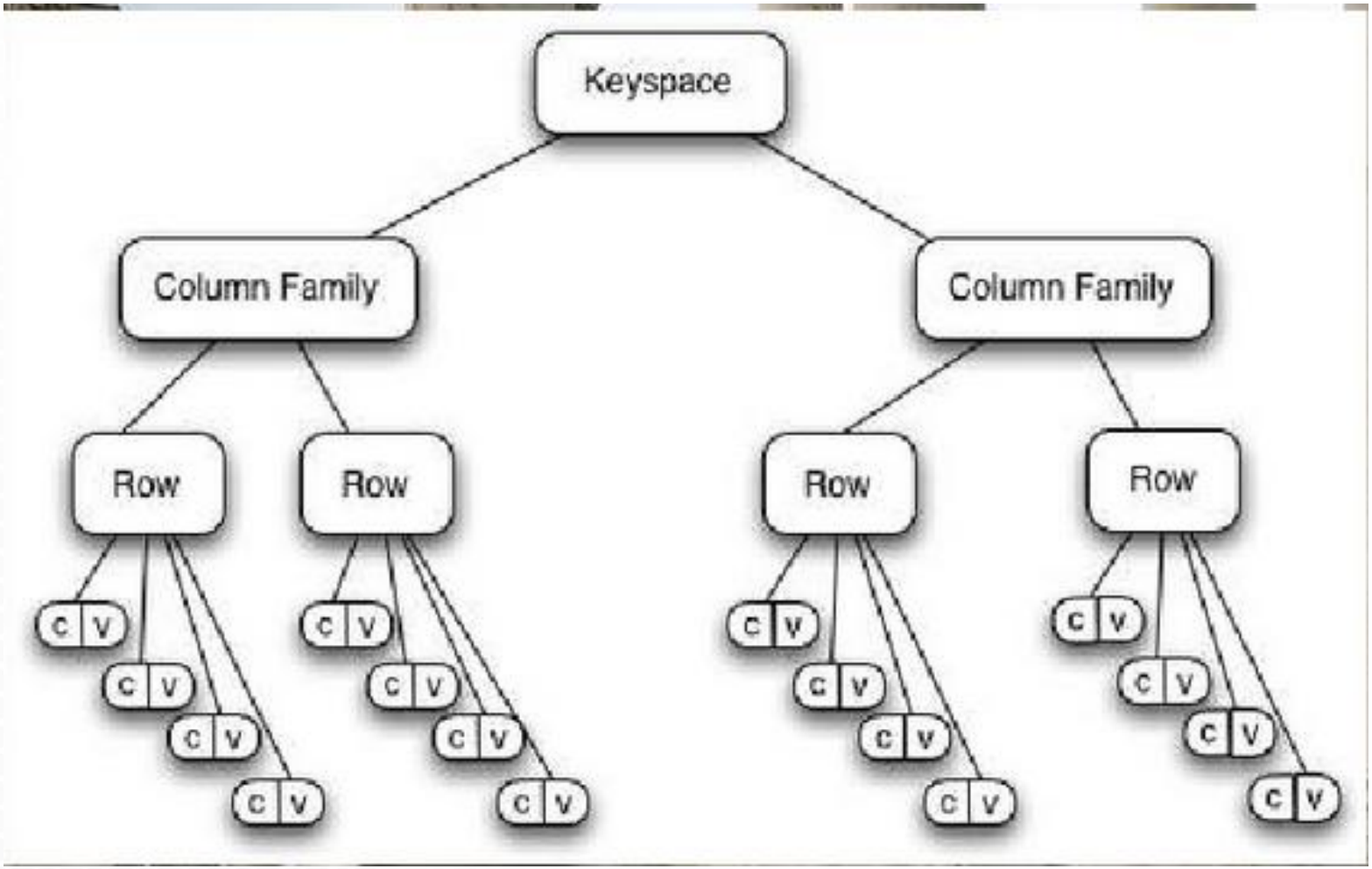




Data model

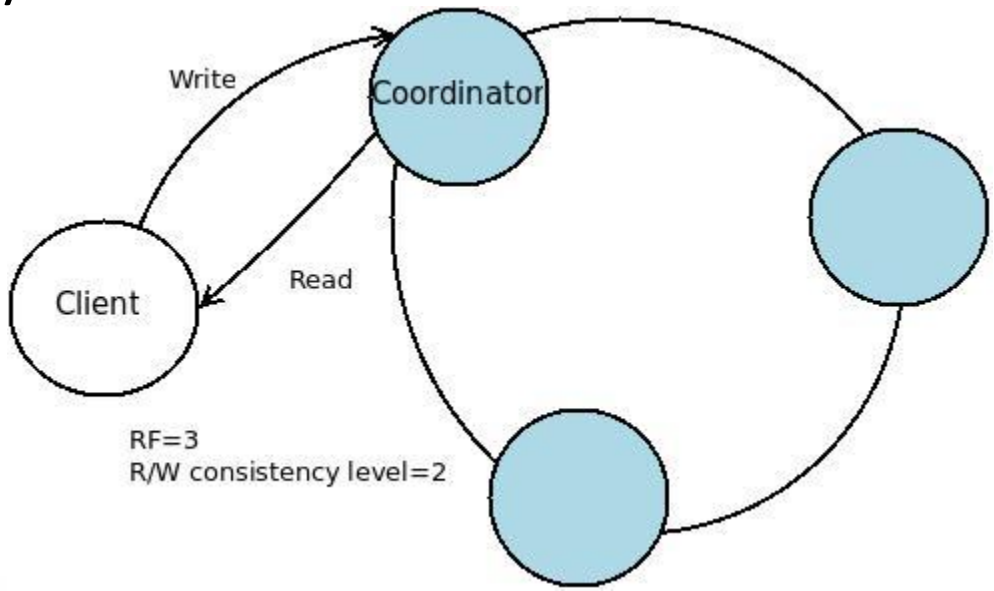
- Data are organized in **Keyspaces**
  - similar to “database” in SQL world
  - each Keyspace can have multiple **Column Families** (“tables”)
- Column Family consists of **rows**
  - each **row** has its **key** and many **Columns**
- Column consists of **column name** and **value**
  - **column name** can be different for each Column/Row
  - the information could be stored in column names
  - number of columns per row is not restricted – could be anything from zero to several million

Data model



Key Concepts

- No single point of failure
  - every node is equal to the others→ scalability
- Tunable tradeoff between consistency and latency
  - read/write consistency levels
  - replication factor



- Eventually consistent
  - “eventually” can mean 1s to 1 month (t.b. continued)
  - no locking

Key Concepts

- Data distribution based on consistent hashing of the **keys**
  - hash_funtion(**key**) = **token**
  - MD5, Murmur3 hashing -> **even** distribution of tokens to nodes even with uneven distribution of **keys**
- Nodes are assigned token ranges that they are responsible for
  - the range distribution is known to all nodes
  - every node knows which and **how many** nodes are responsible for the given key
- The distribution and consistency when reading/writing can be DC-aware
  - e.g. LOCAL_QUORUM consistency level

Conflict resolution

- If two clients write different values for the same key to different nodes, who wins?
- Conflicts are resolved by **timestamps**
  - each column:value pair has a timestamp

```
"key": "some_key",
  "columns": [
        ["name1","value1",1390041549969000],
        ["name2:","value2",1390041549968000]
   ]
```

- But...who should provide the timestamp?
  - Servers - may lead to write/delete paradox due to client connection pooling (CASS-6178)
  - Clients – this gives possibility to insert "unchangeable" content
- In both cases, keep your clock synchronized!

Conflict resolution

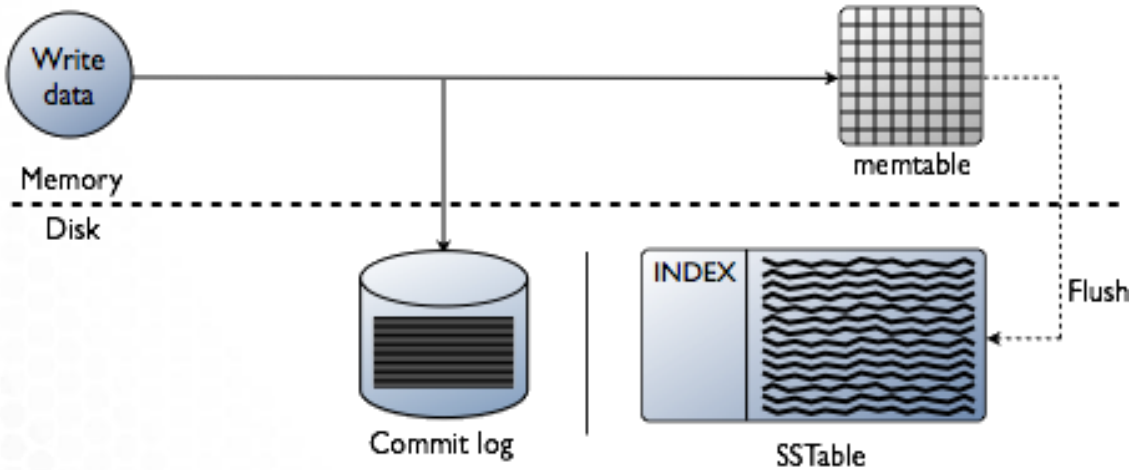
- We noticed that some columns are still present even after issuing an explicit delete command
  - initially suspecting a bug in Cassandra
  - discovered that the particular columns had their timestamps in **nano**seconds instead of default **micro**seconds
  - so even when the delete command got propagated, the original value was “newer” by several centuries!
  - actually a programmer’s fault by specifying wrong TS
  - real pain to get rid out of such entries

SSTables

- Data are stored in **immutable, sorted** files – sorted string tables – SSTables
  - rows are sorted by token, columns by comparator function (defined in schema of ColumnFamily)
    - allows effective **merging** and searching
- Since the files are never changed
  - updated values are placed in new files
  - snapshots are virtually free (hardlinks)
  - deletes are actually new values with special marker (tombstone) and TTL (gc_grace_period)
    - data could reappear if a node is down longer and missed the deletion!
  - sstables must be **compacted** (merged) to get rid of stale data

Write path

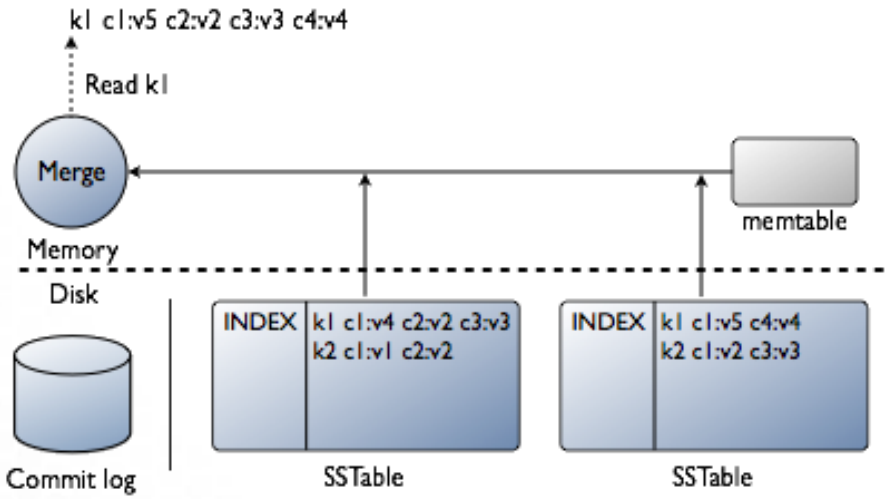
- Writes are **fast**
  - probably faster then in your old good SQL db



  - flush is asynchronous and typically triggered when the memtable reaches a size limit (default 4x SSTable size)
  - only sequential disk access needed

Read path

- Reads are ... less fast
  - a row can be spread in multiple sstables at once
  - need to merge with unflushed memtable



Read path

- Several optimizations to make things faster
  - bloomfilters of row keys per sstables cached in memory
    - space efficient probabilistic data structure
    - false positives possible, false negatives **not**
  - index files – contains offsets of row keys in data files
  - partition index – sampled row keys with offsets to index files
  - key cache, row cache
  - columns bloomfilter, columns indexes....
- Still, the lower the number of SStables a row is stored in, the better
  - depends mainly how **compactions** are done

Compactions

- Continues background bookkeeping process
- **Crucial** to overall performance
  - reduces number of SSTables a row is spread in
  - reduces disk space usage of stale data
  - constitutes a big part of IO operations being done
  - at least, the IO load is strictly sequential (sorted tables)

- Two types of compactions
  - Size Tiered Strategy – SStables are merged together based on size
  - Leveled Compaction Strategy – SStables are organized in levels

Size Tiered Compactions

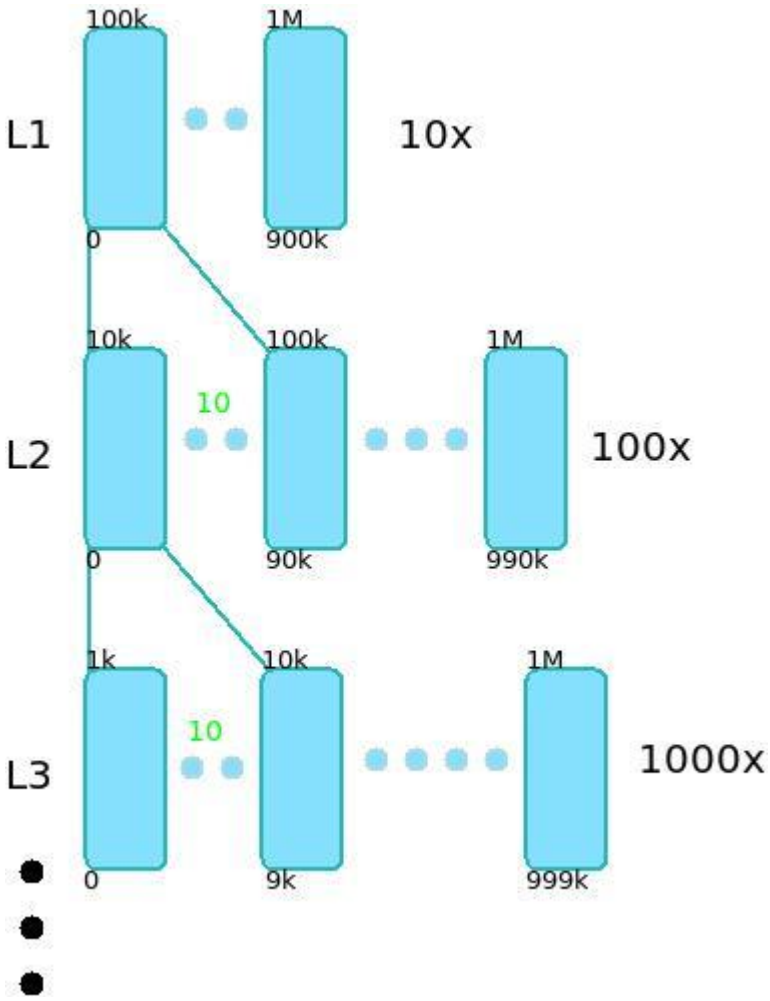
- If the memtable is full, flush sstables to disk
- If there are more than X (default 4) sstables with similar size, merge them together into one bigger sstable

- Twice as much disk space needed temporarily
  - **even more with more disks**
  - 4x 1TB disks, 300GB SStable on each of them
  - ~1.2TB disk space needed → compaction fails causing the affected node to get stuck – client timeouts for every request (C 1.1.0)

- No guarantee about how many sstables contain a single row
- Uneven distribution of compaction load over time

Leveled Compaction Strategy

- SStables organized in levels, tables in each level are **non-overlapping**
- Each level has 10x more SStables than the previous
  - with 64MB SStables, 6 levels is a real life maximum (70TB)
- Maximum number of SStables to read = number of levels
- Flushed tables are inserted to L0 and are immediately compacted with **overlaping** L1 SStables
- Newly created SStables forms new L1
- If there are more then 10 tables, pick an SStable and promote it down with **overlapping** SSTables in L2
- ....

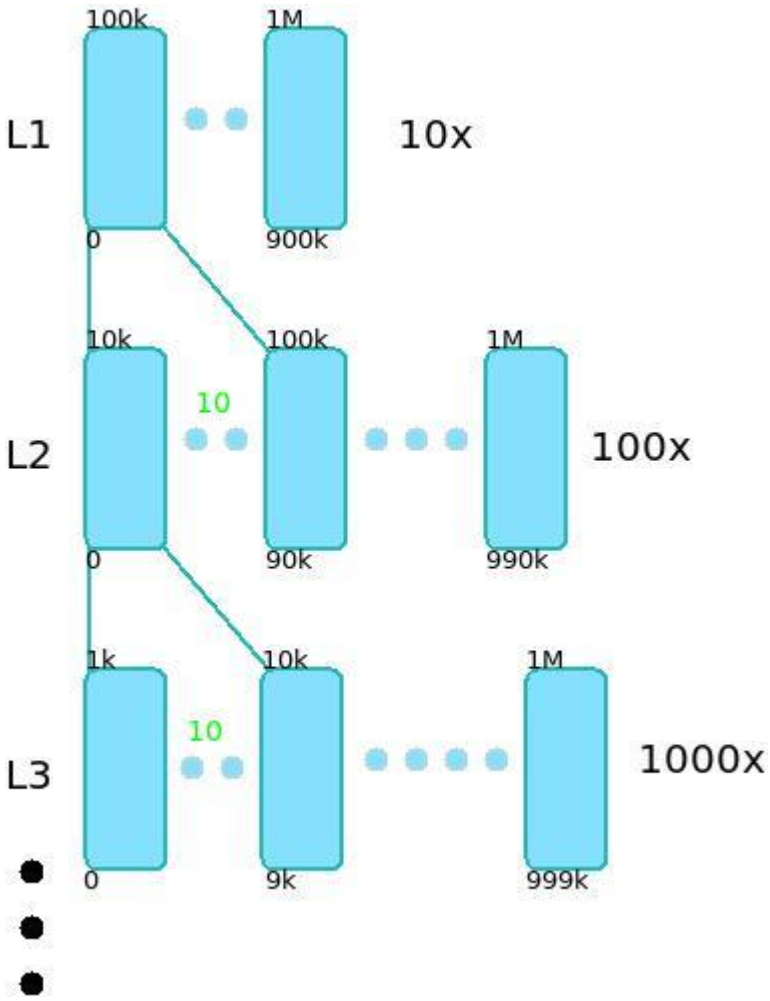


Leveled Compaction Strategy

- Because of hashing, the L0 SStables contains practically the whole token range for the node
- So it is necessary to compact all SStables in L1
- For each SStable in L1, there are ~10 SStables in L2 with overlapping token range
- The same applies all the way down
- Example: 4x64=**256MB** SStables flushed to L0 from memtable requires reading and writing of **~5.8GB** with 4 levels.

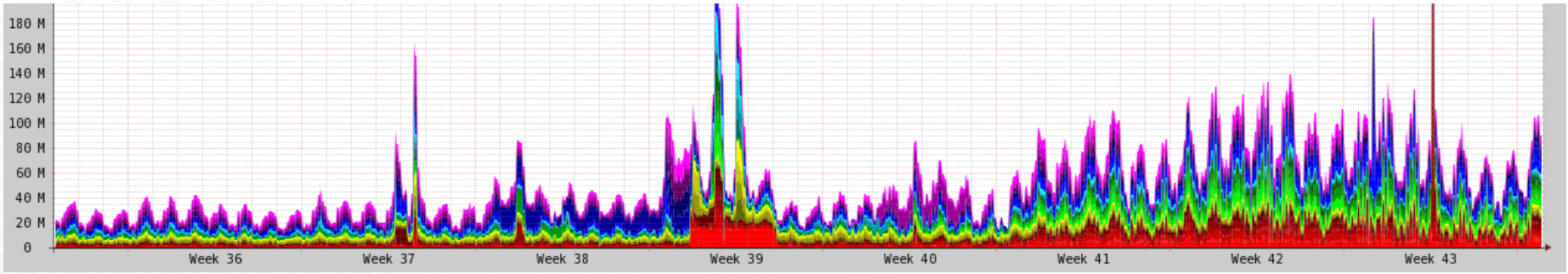
The overhead actually pays off in practice with read heavy workloads – it can be proven that 90% of rows are in one SStable only.

+ only ~10% disk space overhead



Leveled Compaction Strategy

- Because of read performance guarantees, we use LCS for FileRep
- Noticed gradual performance degradation after switching to Cassandra 2.0.0
  - ~3x more IO after two weeks
- Problem tracked down to a bug in LCS compaction CASS-6284
- The patch sent to community and accepted to v2.0.4

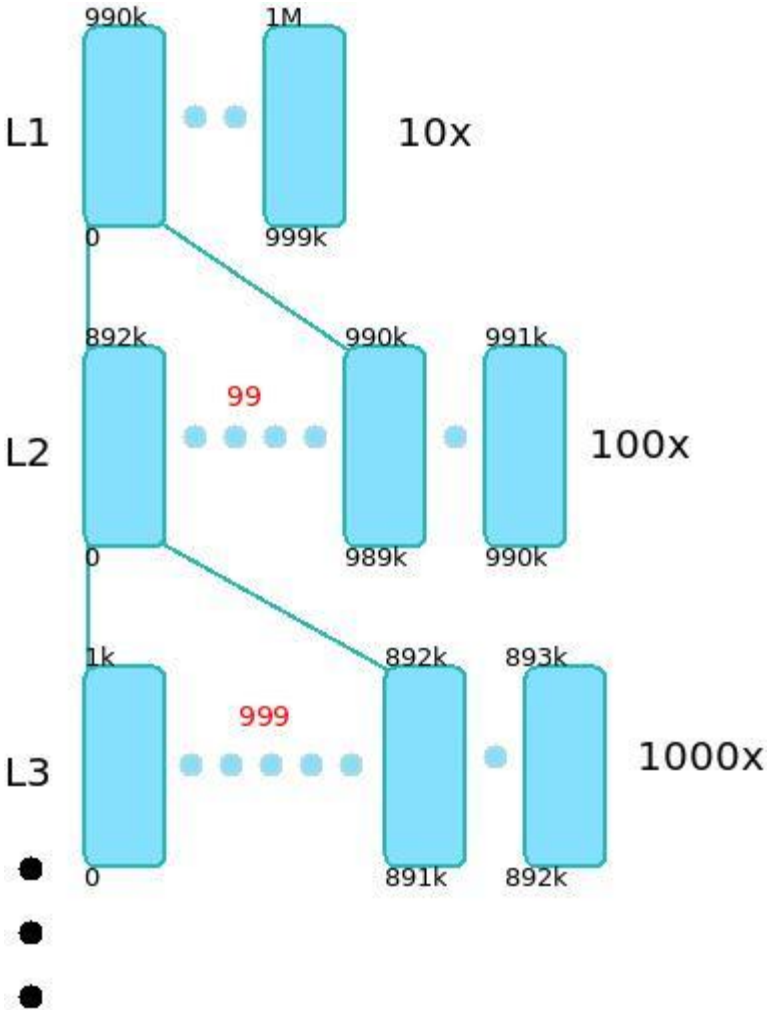


Leveled Compaction Strategy

- When promoting an SStable down if the current layer is overfull, the **candidate choice is important**
- If the first (in order) SStable is always chosen, the token distribution across SStables **degenerates**
- The problem gets worst and worst over time – but can stay unnoticed for some weeks...

We decided to revert back to previous Cassandra version.

But that needed lots of compaction work as well....



SSD needed

- TRIM command not functional on the particular RAID controller
  - would kill the SSD in 100days
- TRIM worked on an internal controller where DVD-ROM was attached to...

...desperate situations deserve desperate solutions

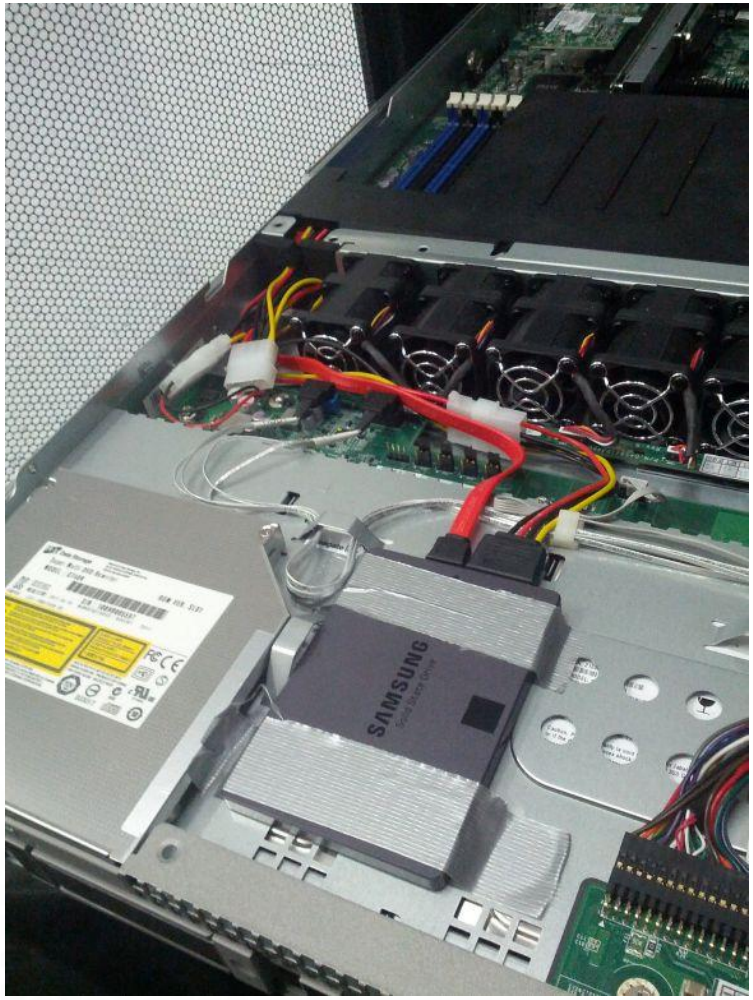
SSD needed

- TRIM command not functional on the particular RAID controller
  - would kill the SSD in 100days
- TRIM worked on an internal controller where DVD-ROM was attached to...

...desperate situations deserve desperate solutions

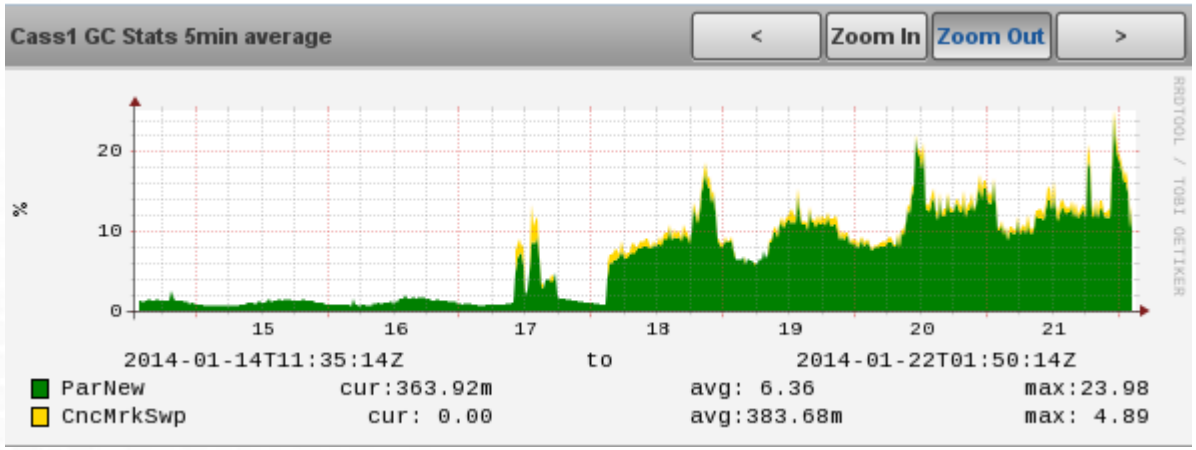
SSD needed

=



Hinted handoffs

- When a node responsible for is down, the coordinator node caches the data for it for a certain time (e.g. 1 day)
  - if the node is down for longer, manual **repair** is needed
- The hints are stored and transferred by **rows**
  - but what happens with **really (1GB) wide** rows?



- Serializing the big rows causes high memory pressure...
  - causing the affected node to flip
  - causing even more hints to be stored on other nodes..

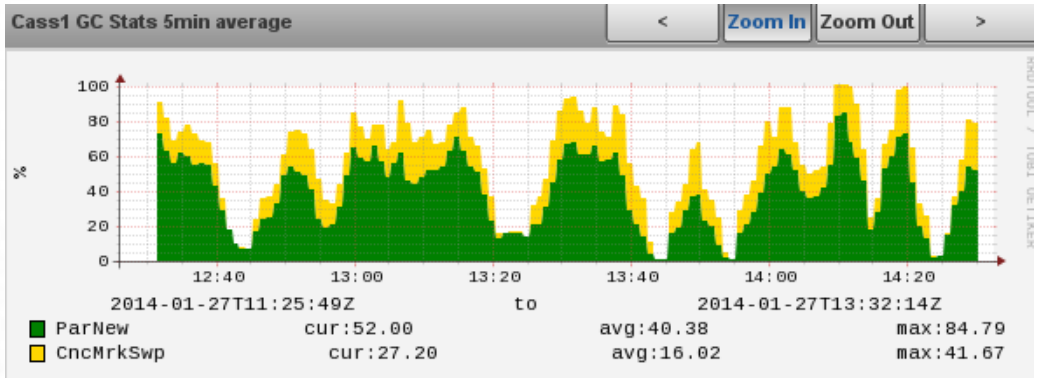
Repairs

- On each node, compute the keys a node is responsible for (merkle tree)
- Transfer it to all other nodes and compare
- Transfer the missing data files

- **Repairs are expensive**
  - But inevitable (node down than longer then gc_grace_period)
  - According to manual, should be run periodically (e.g. every 10days)
  - On the Filerep instance, it takes **~2months** with fine tuning to not influence the production

Beware of the row cache

- Even with key cache, a disk IO is needed
  - plenty of RAM...let's utilize the row cache!
  - store the entire row in memory
- The row cache can be off-heap
  - (de)serialization still needed



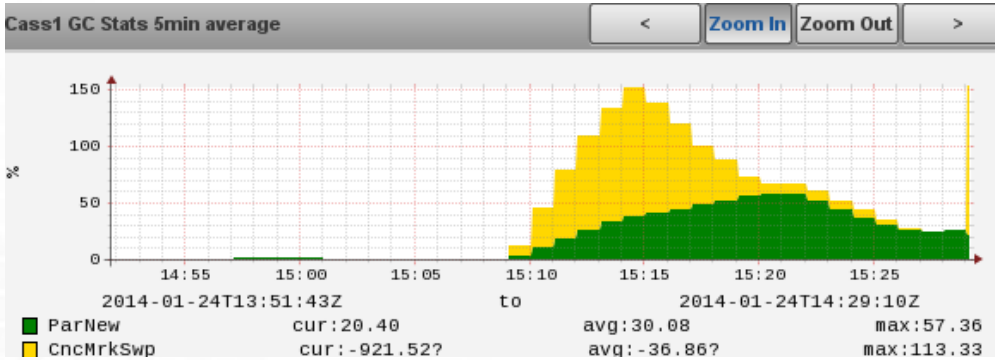
- Row cache is only useful in very narrow situations
  - In others, it actually hurts!
  - There is even a proposal to completely remove it: [CASS-5348]

Scanning through 3TB

- Needed to find files with certain properties in FileRep DB
  - full scan needed
- First approach – single threaded application, started on one of the 8 responsible cassandra nodes
  - would finish in 500 days

Scanning through 3TB

- Needed to find files with certain properties in FileRep DB
  - full scan needed
- First approach – single threaded application, started on one of the 8 responsible cassandra nodes
  - would finish in 500 days
- Better – use bigger batches, 10 threads and run against cluster with SSDs
  - 50 days, coordinator overloaded



Scanning through 3TB

- Needed to find files with certain properties in FileRep DB
  - full scan needed
- First approach – single threaded application, started on one of the 8 responsible cassandra nodes
  - would finish in 500 days
- Better – use bigger batches, 10 threads and run against cluster with SSDs
  - 50 days, coordinator overloaded
- Best – run the scan on each node of the cluster, scanning only the keys the node is responsible for
  - no inter node communication needed
  - 50 hours

Questions?

Thanks for attention!

Jiří Horký (horky@avast.com)
Jenda Kolena (kolena@avast.com)
Pavel Kučera (kucerap@avast.com)
Marcela Římalová (rimalova@avast.com)