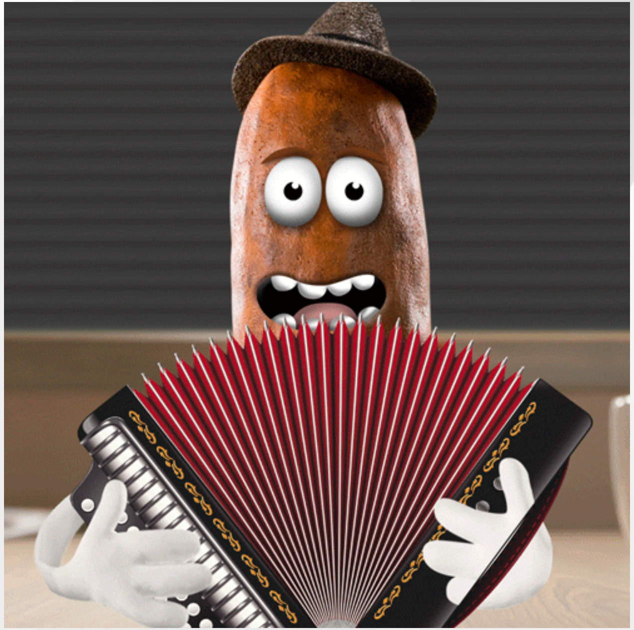
YAHOO!



Accordion: HBase Breathes with In-Memory Compaction

Eshcar Hillel, Anastasia Braginsky, **Edward Bortnikov** | HBaseCon, Jun 12, 2017

The Team

Edward Bortnikov

Eshcar Hillel
(committer)

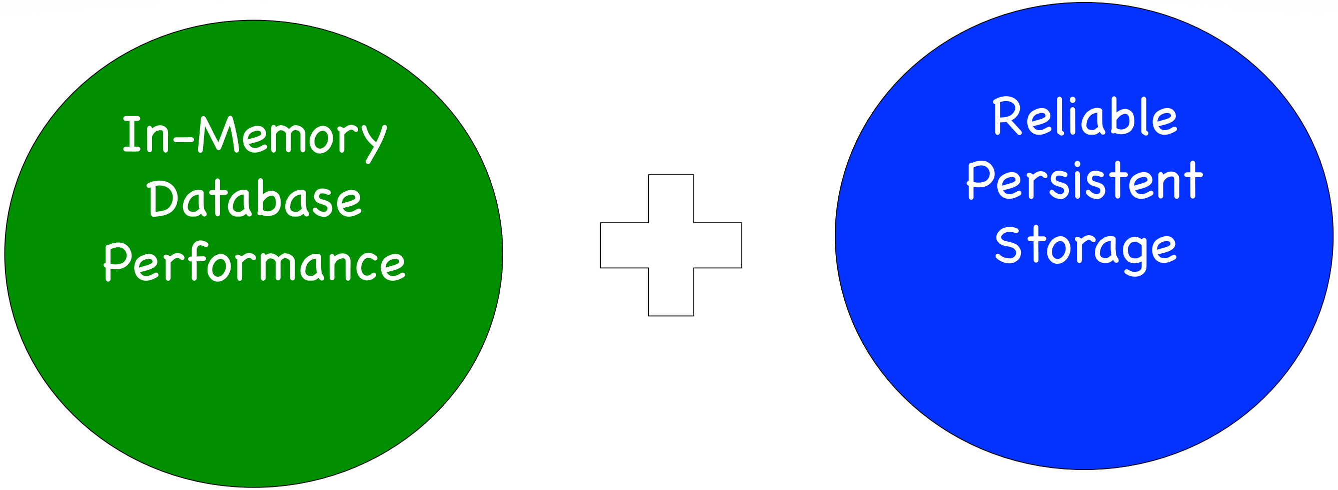
Anastasia Braginsky
(committer)

Michael Stack

Anoop Sam John

Ramkrishna Vasudevan

Quest: The User's Holy Grail



What is Accordion?

Novel Write-Path Algorithm

Better Performance of Write-Intensive Workloads

Write Throughput ↗, Read Latency ↘

Better Disk Use

Write amplification ↘

GA in HBase 2.0 (becomes default MemStore implementation)

In a Nutshell

Inspired by Log-Structured-Merge (LSM) Tree Design

- Transforms random I/O to sequential I/O (efficient!)
- Governs the HBase storage organization

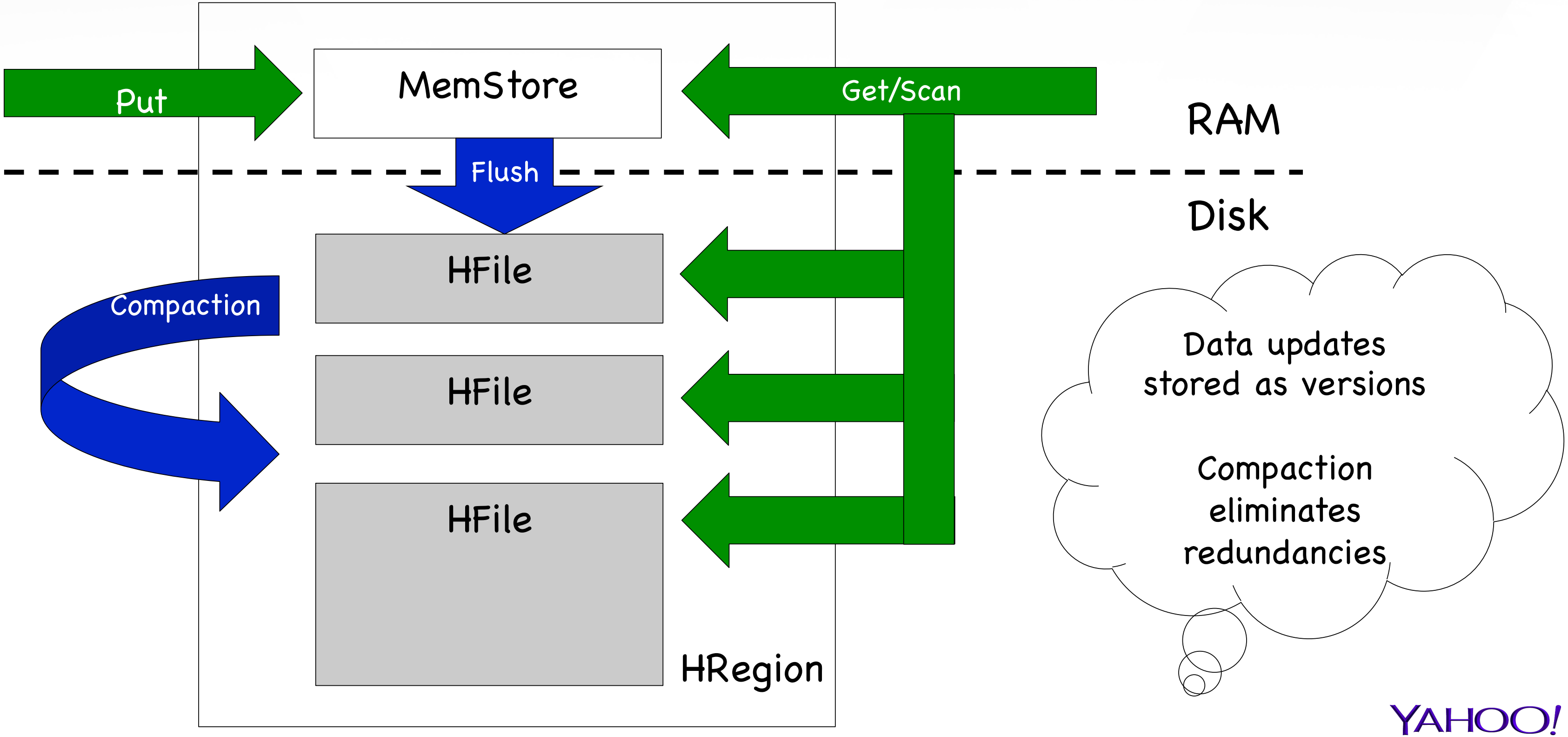
Accordion reapplies the LSM Tree design to RAM data

→ Efficient resource use – data lives in memory longer

→ Less disk I/O

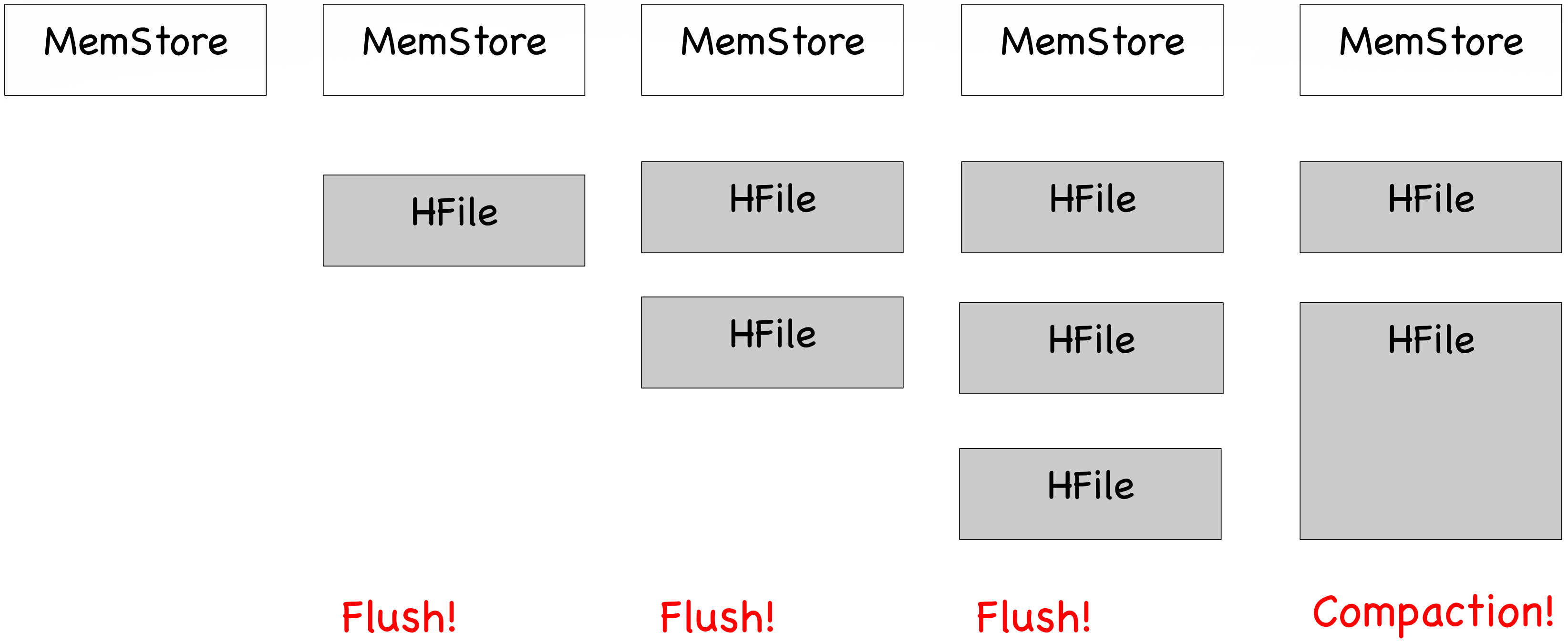
→ Ultimately, higher speed

How LSM Trees Work

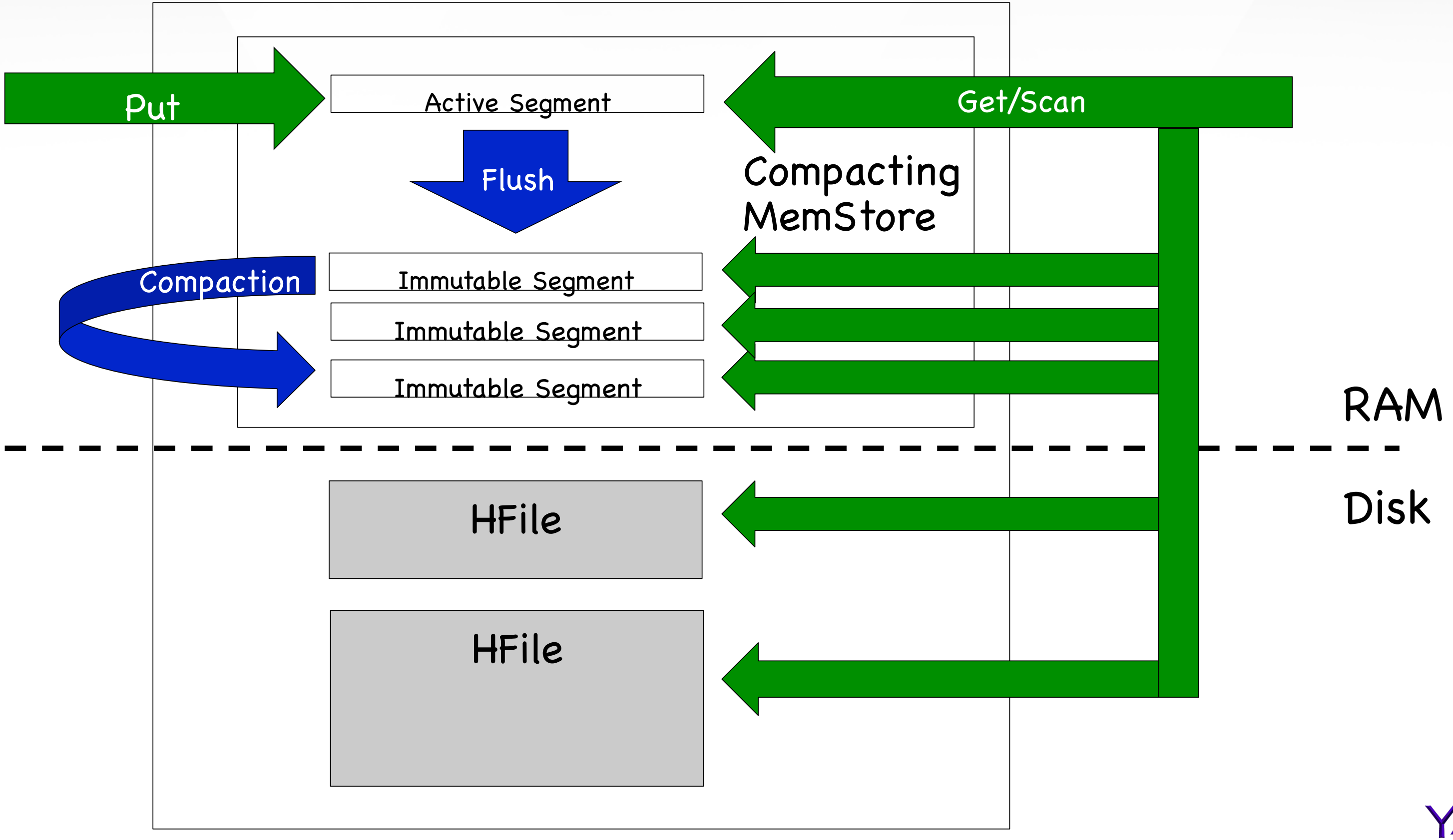


YAHOO!

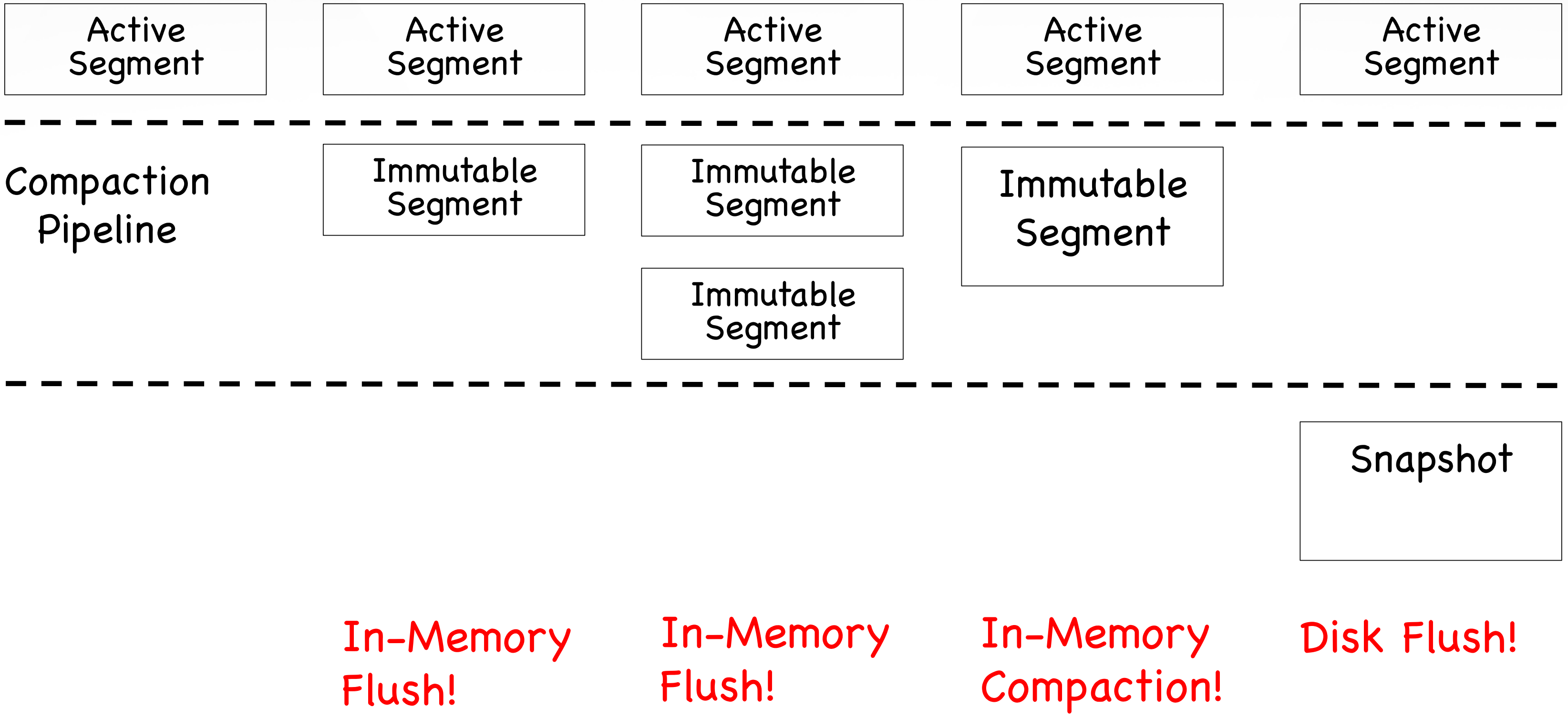
LSM Trees in Action



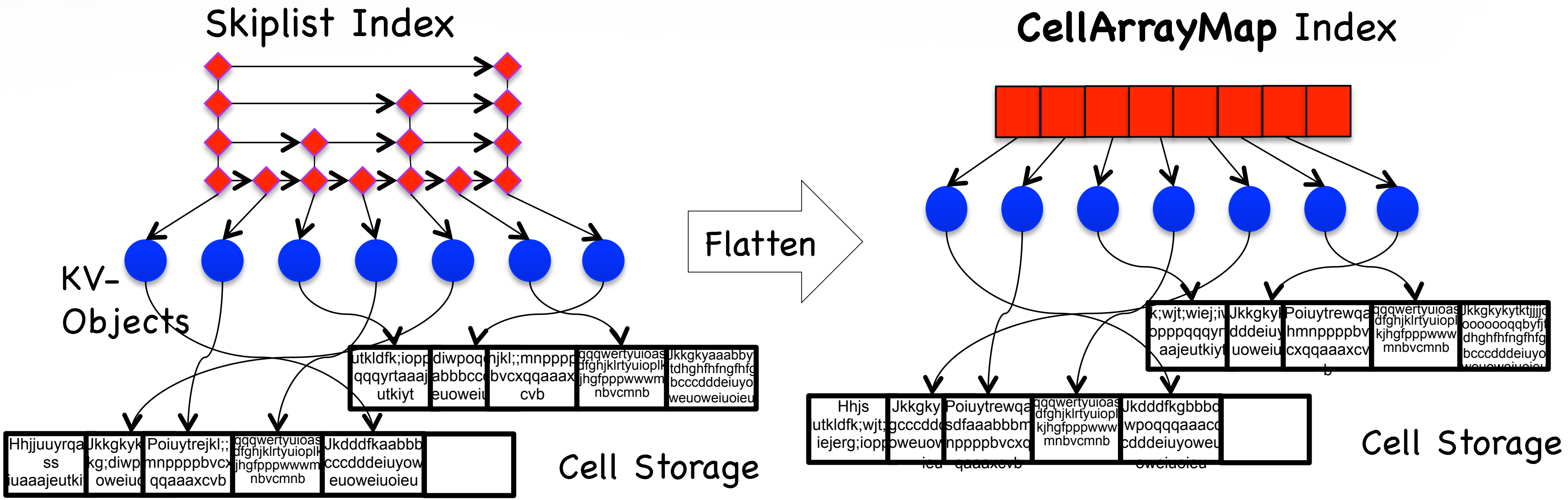
Accordion: In-Memory LSM Tree



Accordion in Action



Flat Immutable Segment Index



Lean footprint – the smaller the cells the better!

Redundancy Elimination

In-Memory Compaction **merges** the pipelined segments

- **Get** access latency under control (less segments to scan)

BASIC compaction

- Multiple indexes merged into one, cell data remains in place

EAGER compaction

- Redundant data versions eliminated (SQM scan)

BASIC vs EAGER

BASIC: universal optimization, avoids physical data copy

EAGER: high value for highly redundant workloads

- SQM scan is expensive
- Data relocation cost may be high (think MSLAB!)

Configuration

- BASIC is default, EAGER may be configured
- Future implementation may figure out the right mode automatically

Compaction Pipeline: Correctness & Performance

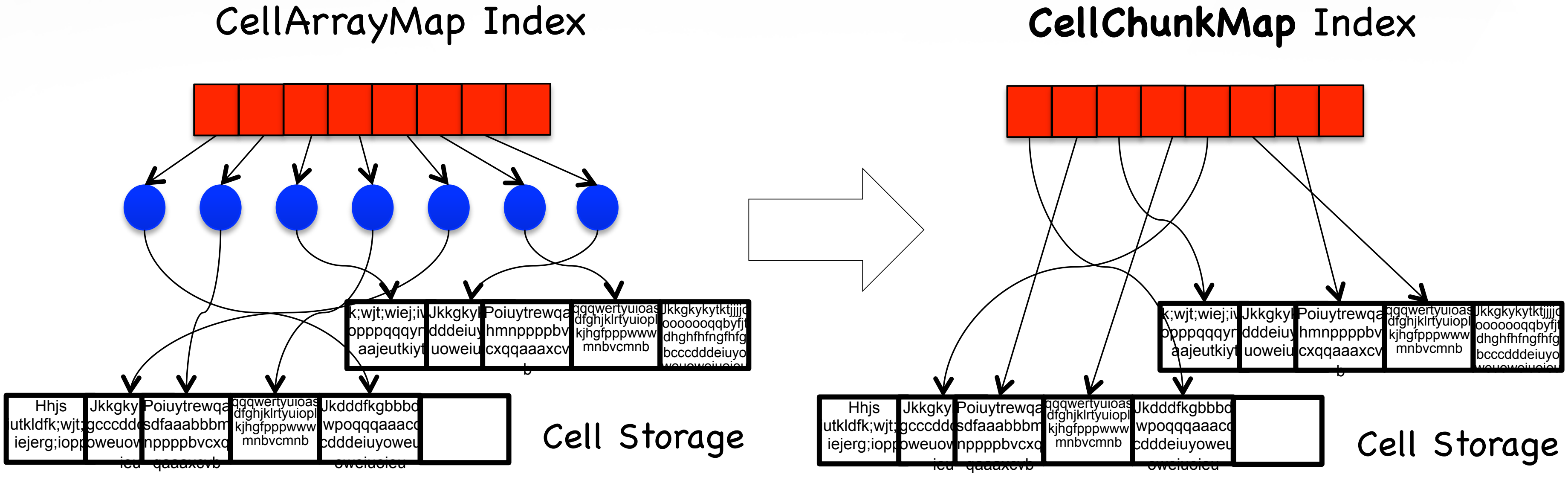
Shared Data Structure

- Read access: Get, Scan, in-memory compaction
- Write access: in-memory flush, in-memory compaction, disk flush

Design Choice: Non-Blocking Reads

- Read-Only pipeline clone – no synchronization upon read access
- Copy-on-Write upon modification
- Versioning prevents compaction concurrent to other updates

More Memory Efficiency - KV Object Elimination



Lean Footprint (no KV-Objects). Friendly to Off-Heap Implementation.

The Software Side: What's New?

CompactingMemStore: BASIC and EAGER configurations

DefaultMemStore: NONE configuration

Segment Class Hierarchy: Mutable, Immutable, Composite

NavigableMap Implementations: CellArrayMap, CellChunkMap

MemStoreCompactor: compaction algorithms implementation

CellChunkMap Support (Experimental)

- Cell objects embedded directly into CellChunkMap (CCM)
  - New cell type - reference data by unique **ChunkID**
- ChunkCreator: Chunk allocation + ChunkID management
  - Stores mapping of ChunkID's to Chunk references
  - **Strong** references to chunks managed by CCM's, **weak** to the rest
  - The CCM's themselves are allocated via the same mechanism
- Some exotic use cases
  - E.g., jumbo cells allocated in one-time chunks outside chunk pools

Evaluation Setup

System

- 2-node HBase on top of 3-node HDFS, 1Gbps interconnect
- Intel Xeon E5620 (12-core), 2.8TB SSD storage, 48GB RAM
- RS config: 16GB RAM (40% Cache/40% MemStore), on-heap, no MSLAB

Data

- 1 table (100 regions, 50 columns), 30GB-100GB

Workload Driver

- YCSB (1 node, 12 threads)
- Batched (async) writes (10KB buffer)

YAHOO!

Experiments

Metrics

Write throughput, read latency (distribution), disk footprint/amplification

Workloads (varied at client side)

Write-Only (100% Put) vs Mixed (50% Put/50% Get)

Uniform vs Zipfian Key Distributions

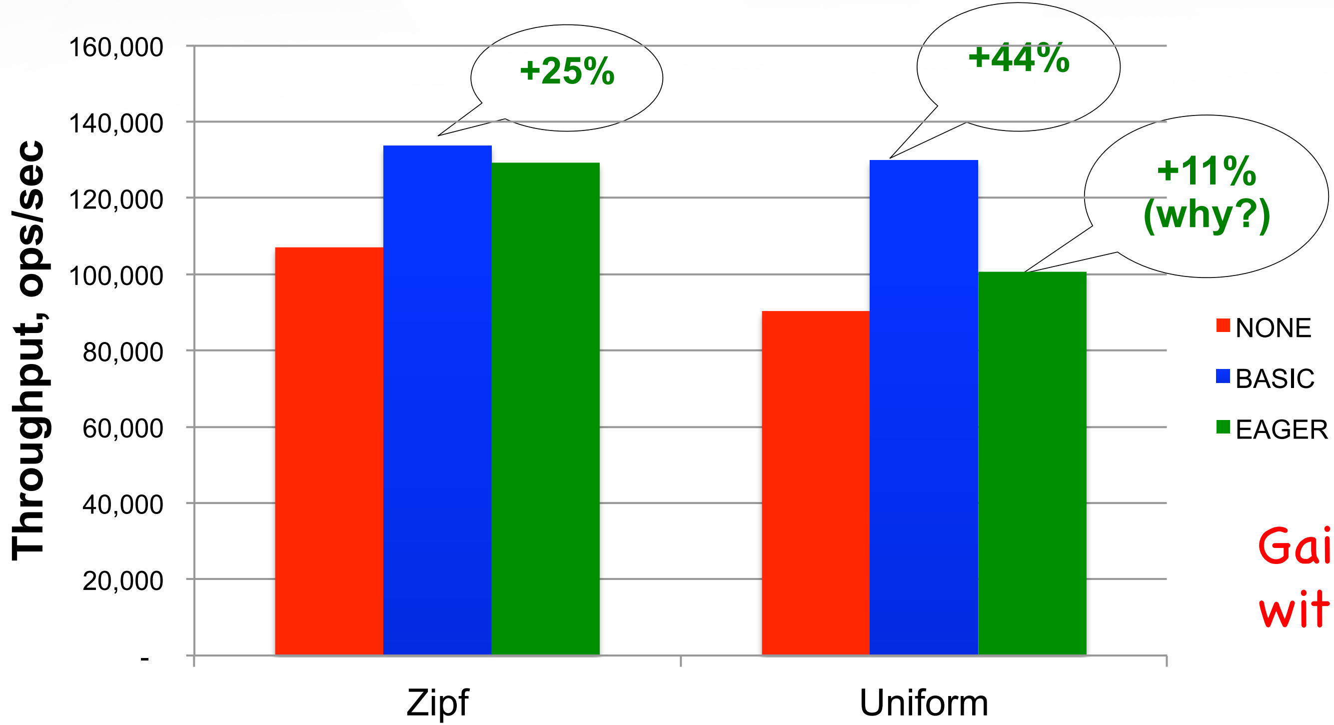
Small Values (100B) vs Big Values (1K)

Configurations (varied at server side)

Most experiments exercise Async WAL

YAHOO!

Write Throughput

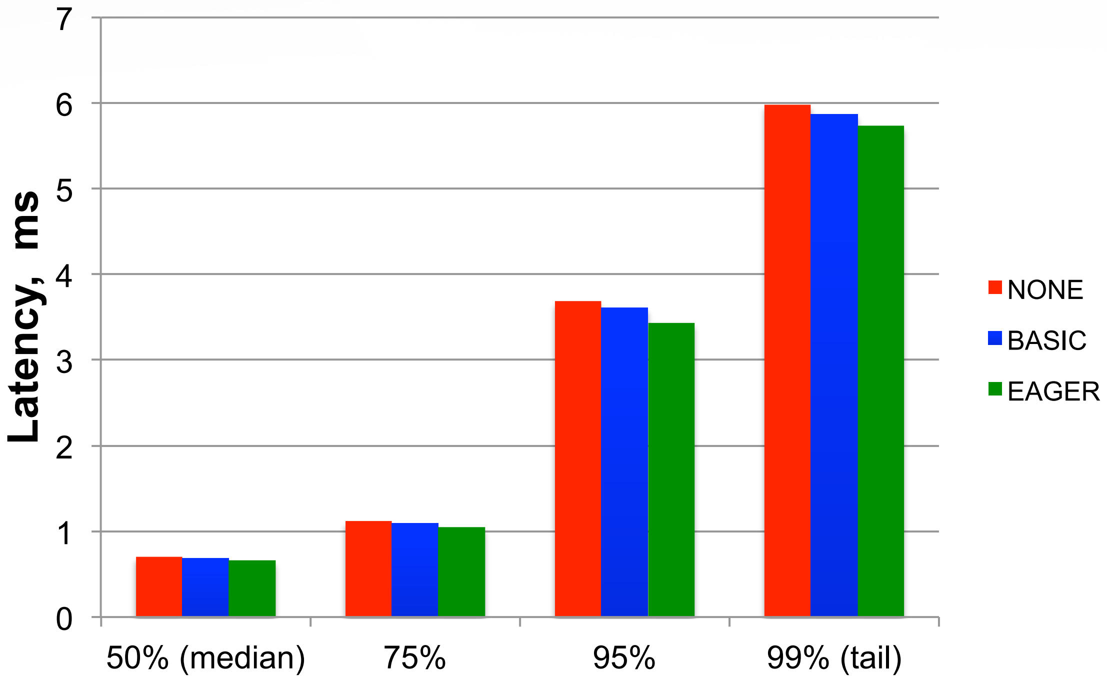


100GB Dataset
100% Writes
100B Values

Every write updates a single column

Gains less pronounced with big values (1KB)

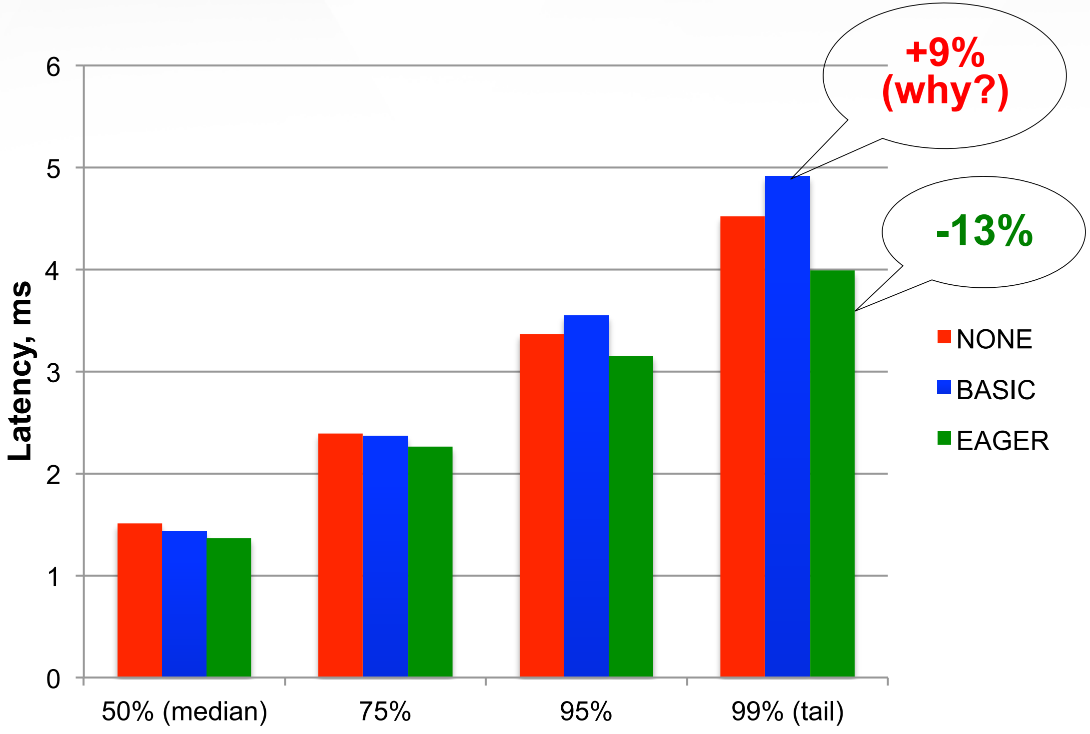
Single-Key Write Latency



100GB Dataset
Zipf distribution
100% Writes
100B Values

YAHOO!

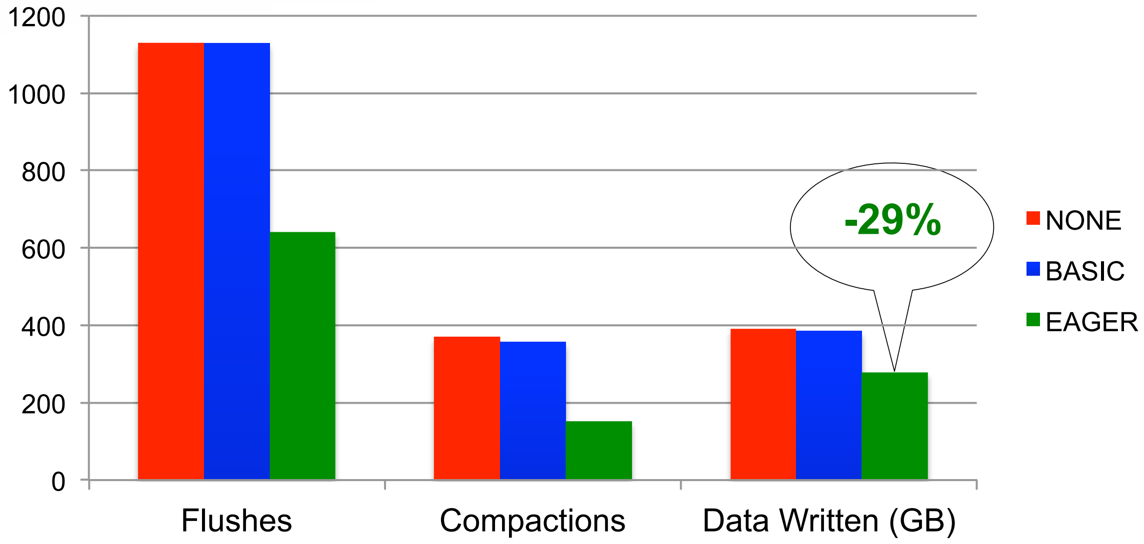
Single-Key Read Latency



30GB Dataset
Zipf Distribution
50% Writes/50% Reads
100B Values

YAHOO!

Disk Footprint/Write Amplification



100GB Dataset
Zipf Distribution
100% Writes
100B Values

YAHOO!

Status

In-Memory Compaction GA in **HBase 2.0**

- Master JIRA HBASE-14918 complete (~20 subtasks)
- Major refactoring/extension of the MemStore code
- Many details in Apache HBase blog posts

CellChunkMap Index, **Off-Heap** support **in progress**

- Master JIRA HBASE-16421

Summary

Accordion = a leaner and faster write path

Space-Efficient Index + Redundancy Elimination → **less I/O**
Less Frequent Flushes → **increased write throughput**
Less On-Disk Compaction → **reduced write amplification**
Data stays longer in RAM → **reduced tail read latency**

Edging Closer to In-Memory Database Performance

Thanks to Our Partners for Being Awesome