Row Cache for Apache HBase (Get / MultiGet)

1) Summary

HBase's existing caching is primarily **block-based** (BlockCache/BucketCache). This is great for scans and predictable access, but it can be inefficient for small point reads (Get/MultiGet) that touch only a tiny fraction of the 64KB HFile blocks that end up cached anyway. The proposal is to add a **row-level cache** to HBase, similar in spirit to Google Bigtable's row cache, to improve point-read throughput/latency and reduce wasted cache space and CPU overhead.

This document proposes an **HBase-native row cache** for Get/MultiGet, with:

- Correctness/coherency rules aligned with HBase consistency expectations.
- A memory-efficient representation (ideally sparse / "only what was requested").
- Pluggable storage: on-heap/off-heap/SSD.
- Production-grade ops: metrics, sizing, eviction, safety rails, and rollout controls.

2) Background & Motivation

2.1 Why block cache isn't enough (especially in cloud / disaggregated storage)

- **Point reads cache whole blocks** even when returning a small row → poor cache efficiency.
- **CPU overhead** remains high: point reads traverse MemStore + multiple StoreFiles and build the result, even if the same row is repeatedly requested.
- In disaggregated storage, cache is "the performance": losing locality or wasting cache harms tail latency.
- **Compaction sensitivity**: BlockCache entries are tied to HFile blocks. When compaction rewrites HFiles, corresponding cached blocks are invalidated, leading to cache churn and temporary performance degradation (this can be mitigated for some deployments by allocating caches which can hold the whole datasets and enabling caching on write).

In contrast, a row cache operates at logical row granularity, independent of physical HFile layout. As long as row-level mutations are correctly handled, cached row results remain valid across compactions, making row cache inherently resilient to compaction-induced churn.

2.2 Bigtable's row cache (reference behavior)

Google describes a **row cache** that "builds on the block cache" to cache data at **row granularity**; cached rows use a **sparse representation** that retains only the data accessed by the query. If a subsequent query needs missing data, it falls back to block

reads and fills the row cache with missing parts. Row and block caches share a single memory pool and eviction balances both. (cloud.google.com)

2.3 Current HBase row cache work in the community

Apache HBase JIRA **HBASE-29585** proposes a row-level cache to optimize Get operations because Gets can cache entire 64KB blocks even when retrieving a single row, reducing cache hit ratio efficiency and wasting BlockCache; row granularity aims to reduce BlockCache consumption and CPU. (issues.apache.org)

Related sub-tasks include “Add row cache framework” (HBASE-29668) which links to an upstream PR. (issues.apache.org)

2.4 Existing reference implementation

The repo **hbase-row-cache** implements Row Cache as an **HBase RegionObserver coprocessor**. Key implications:

- **No HBase core patches required**: it can be deployed/removed operationally as a coprocessor without modifying HBase server code. (github.com)

- The cache key is effectively **table + rowkey + column family**, and the implementation caches the CF data for Get/MultiGet, with invalidation on mutation for that rowkey/family. (github.com)

This approach is attractive for rapid adoption and experimentation because it avoids invasive changes. The trade-offs are typical for coprocessors: operational governance (loading CPs), integration constraints (limited access to certain internal hooks), and the need to carefully align correctness/ordering with HBase’s mutation path.

3) Goals & Non-goals

Goals

1. **Speed up Get/MultiGet** (and optionally very small, bounded scans later).
2. **Improve cache efficiency** for small point reads (avoid caching whole 64KB blocks when not needed).
3. **Reduce CPU cost** of repeated point reads by bypassing repeated MemStore/StoreFile processing.
4. **Predictable behavior** under mutations (clear coherency rules).
5. **Operationally safe**: metrics, sizing, guardrails, and rollout toggles.

Non-goals (initially)

- Not a general scan/range cache (unless a later phase adds it).
- Not a replacement for BlockCache/BucketCache; it should **complement** them.
- Not a new distributed cache system (keep it within RS process boundaries).

4) High-level Design

4.1 Where it sits in the read path

The row cache was implemented without modifying HBase core code by leveraging the coprocessor framework.

Coprocessor-based interception (current implementation)

- Implemented as a RegionObserver coprocessor.
- Intercepts Get/MultiGet operations at the RegionServer level.
- Performs cache lookup before normal read path execution.
- On miss, allows the standard HBase read pipeline to proceed and then populates the cache with the result.
- On mutation (Put/Delete/etc.), invalidates (now) or updates (future) the corresponding row cache entry.

Key property:

- No changes to HBase server internals are required.
- No patching of the HBase codebase.
- Deployment and rollback are operational (configuration-based), not code-based.

This approach minimizes upstream intrusion and reduces risk, while still allowing validation of semantics, performance characteristics, and operational behavior under real workloads.

4.2 Data model for cache entries

A cache entry represents the result of a **Get** over:

- Row key
- Column family
- Optional qualifier set / filter / time range (for sparse row representation)

Two representation options:

Option 1: "Whole CF" entry (current version)

- Cache: table + row + family → serialized KeyValues for the full CF.
- Simple and fast for repeated full-CF reads.
- But can over-cache if apps request only a few qualifiers.

Option 2: Sparse / partial row representation (Bigtable-like)

- Cache only the qualifiers/versions actually requested.
- If a later request needs missing parts → read-through + merge + update.
- Better efficiency for workloads that read a subset of wide rows. (cloud.google.com)

Recommendation:

- Implement **sparse representation** as the target design.

- Allow a configuration switch to “cache whole CF” for simpler workloads.

4.3 Memory & storage placement

- Keep row cache in a **separate pool (current implementation)** or share a common pool with block cache. Bigtable shares a single pool and uses an eviction policy balancing both. (cloud.google.com)
- In HBase, practical options:
    - On-heap (simple, but GC risk). There is no support for On-heap caches in the current implementation (can be added if there will be a request).
    - Off-heap (preferred - current implementation)
    - SSD-backed (optional - current implementation can be configured to use SSD as well)

Current implementation provides almost “zero GC” and large cache sizes by relying on Carrot Cache with optional disk/hybrid modes. (github.com)

5) Coherency Rules (Correctness)

A row cache must be safe under mutations (Put/Delete/Increment/Append).

Proposed coherency model (v1 - current version)

- **Write-through is not required**.
- On any mutation that affects a row:
    - Invalidate that row’s cache entry (at least for the affected CF).
    - This matches the current approach (rowkey:family invalidated on any mutation). (github.com)

Upgrade path (v2+)

- Support in-place updates / sparse updates:
    - Update only impacted qualifiers in the cached sparse structure.
    - The repo explicitly lists “update (not delete) cached row on mutation” and “sparse rows support” as TODOs. (github.com)

Filters, versions, and time ranges

- For sparse row support cache key must include any query-shaping dimensions that change results (this is not necessary for v1 - full row cache):
    - TimeRange
    - MaxVersions

- Column selection / Filter hash

Consistency model

- HBase provides strong consistency per row via single RS ownership; row cache must not return stale results after the RS applies a mutation.
- Therefore, cache invalidation/update must occur in the same RS process and be ordered w.r.t. the mutation application.

6) Eviction & Admission

Key to success is preventing the row cache from becoming a “heap of tiny objects”. Community review explicitly worries about per-entry map overhead for many small rows. (github.com). HBase Row Cache uses Carrot Cache as an underlying caching library (Apache 2.0 license). Carrot Cache provides extremely low metadata overhead (as low as 6-8 bytes) and per object expiration time, all data including metadata is kept off Java heap space, therefore it is suitable for caching of large numbers of small objects.

Admission control

Various admission policies may be applied:

- Don’t cache rows larger than a threshold.
- Cache only when repeated access is detected (TinyLFU-style admission)

Carrot Cache provides an API to integrate a custom Admission Controller into the core cache logic; it also provides a generic, queue - based admission controller implementation which can be enabled in the configuration (similar to TinyLFU).

Eviction

- Segmented LRU is the current eviction algorithm, another available algorithm is LRU; size-aware eviction is required (future work).
- Bigtable notes row caching considers row size and uses an eviction algorithm balancing block + row cache in a shared pool. (cloud.google.com)

7) API / Configuration

Minimal surface area:

- Enable row cache per-table and per-CF.

The current implementation uses the new HBase attribute `ROWCACHE` on table and CF, with CF overriding table. (github.com) to allow configuring Row cache per table and per table - column family.

Suggested HBase configs (illustrative):

- `hbase.rowcache.enabled` (global)
- `hbase.rowcache.maxEntryBytes`
- `hbase.rowcache.admission = always|tinylfu|queue`

Per-table/CF: `ROWCACHE => true/false` (table descriptor / CF descriptor)

The repo provides command-line tools for row cache administration - Administration.

8) Metrics & Observability

Must-have metrics:

- RowCache hit/miss, hit ratio
- Entries, bytes, evictions
- Admission rejects (size / policy)
- Invalidation count (by mutation type)
- Latency breakdown: cache hit path vs miss path

9) Comparison: Coprocessor approach vs HBASE-29585 direction

This section highlights a key architectural difference between the two approaches.

9.1 This proposal (coprocessor, no core patches)

- Implemented as a **RegionObserver coprocessor**.
- **Does not modify HBase server code**; deployable as configuration/runtime artifact.
- Can iterate quickly and validate behavior on real workloads without waiting for upstream merge cycles. (github.com)
- **Low metadata overhead**: avoids large on-heap metadata maps typical of block-based cache indexing.
- **Minimal Java heap usage** (off-heap–oriented design), reducing GC pressure and allowing larger practical cache sizes. Java on heap overhead <<1 byte per cached object, Java off heap overhead for meta is ~8-12 bytes per cached object.
- **Advanced data compression** (optional). Carrot Cache (the underlying cache implementation) supports Zstandard (Zstd) with dictionary compression. The dictionary is trained dynamically on incoming data and periodically retrained as the data distribution evolves. This approach is particularly efficient for small- to medium-sized objects that share common structural patterns.
- **High scalability**: designed to scale cache capacity without significant heap growth, making multi-terabyte cache sizes feasible on modern hardware.

9.2 HBASE-29585 / upstream proposal direction

HBASE-29585 proposes adding a row cache capability for Get operations to improve efficiency vs block caching. It is being developed as an **in-tree** feature that requires changes across multiple code paths (e.g., Get path interception and integration). (issues.apache.org)

Key concerns to call out explicitly:

- **Intrusiveness**: implementing row cache "inside" HBase typically requires patching multiple server components and is therefore a higher risk/higher review burden.

- **BucketCache-based implementation** inherits the **metadata/heap overhead** characteristics of the existing caching stack (e.g., large in-memory maps/indexes). This can limit practical cache size and reduce memory efficiency.

9.3 Recommendation

- Use the coprocessor implementation as a **reference implementation and proving ground** for semantics, benchmarks, and operational behavior.
- For upstream, require clear answers on:
    - How metadata is stored (avoid large on-heap maps where possible)
    - How duplication with BlockCache is prevented/managed
    - Where to hook into the read path to minimize invasive changes while preserving correctness

10) Open Questions (to resolve before upstreaming)

1. **Sparse representation** design (encoding, merge, versioning, filter awareness).
2. **Security/ACL interactions** (no special handling yet). (github.com)
3. **Region movement/reassignment safety:** the repo notes a stale-data scenario when a region moves away and returns. (github.com)
4. **Bulk upload handling.**
5. **Support for Hadoop 3.x.**

11) Proposed Phased Plan

Phase 0: Bench + validate

- Reproduce point-read inefficiency on cloud object-store deployments.
- Quantify: CPU, cache hit ratio, tail latency.

Phase 1: Full row cache, invalidate-on-write

- RowCache for Get/MultiGet
- Per-table/CF toggles
- Off-heap preferred

- Full invalidation on any mutation to (row, CF)

Phase 2: Full row cache, update-in-place

- Update cached entry on mutation (instead of a full invalidate)

Phase 3: Sparse + update-in-place

- Cache only requested qualifiers/versions
- Update cached entry on mutation (instead of full invalidate)

Phase 4: Optional scan/range cache (separate proposal)

- To cache results of small scan operations. Only if justified; keep coherency tractable.

References

1. Bigtable performance blog including **row cache** description and sparse representation. (cloud.google.com)
2. Row Cache implementation repo and README details. (github.com)
3. HBASE-29585 JIRA description & links to PR/doc. (issues.apache.org)
4. HBASE-29668 / HBASE-29669 subtask pages. (issues.apache.org)