Efficient and Consistent Time-to-Live (TTL) Implementation in MyRocks at Meta

Abhinav Sharma
Software Engineer

Meta

Why TTL?

- Compliance: Sensitive data must be deleted after retention period
- Resource management: Logs, time series, caches grow unbounded
- Queue-like patterns: MySQL tables used as queues need cleanup

Before TTL: Ad-hoc batch DELETE jobs

- Slow, resource-intensive, hard to maintain
- Generate massive binlog traffic replayed on every replica and backed up for PITR
- Consume SQL CPU that competes with production queries

Goal: Make rows disappear automatically with zero application work

Meta

The Cost of Explicit DELETE

Setup: 1M rows (~136 bytes/row), single primary key, no secondary indexes.
Passive TTL (rows expire by time) vs. batched DELETE (1000 rows per batch).

Metric	TTL	DELETE
SQL CPU	<15ms	20s
Binlog	0 MB	136 MB
Wall Time	300ms	25s

TTL expiration cost is effectively zero — no SQL, no binlog, no replication traffic.

DELETE writes 1M tombstones generating 136 MB of binlog that every replica must replay and retain for point-in-time recovery.

With 5 replicas: 680 MB of total replication traffic per expiration cycle.

Meta

Requirements

1. Row-level expiration
2. Immediate invisibility — expired rows must never be returned
3. Physical cleanup must reclaim disk space
4. Snapshot isolation preserved in presence of TTL
5. Primary-replica consistency must not be compromised
6. No noticeable performance degradation

Key insight: Most systems satisfy 3–4 of these. We need all six simultaneously.

Meta

Architecture: Two-Layer Expiration

Read Filtering (logical)

- Every read checks: row_timestamp + ttl_duration ≤ snapshot_time?
- If yes → skip the row
- Timestamp fixed per transaction
- Immediate invisibility
- Zero write overhead

Compaction Filter (physical)

- Piggybacks on LSM compaction
- Drops expired rows during merge
- Safe: only drops rows invisible to all active snapshots
- No tombstones, no binlog
- Single-pass cleanup

Read filtering = correctness. Compaction = space reclamation.

Meta

Row Format: 8-Byte TTL Timestamp

TTL definition (in CREATE TABLE):

```
CREATE TABLE user_sessions (
    id BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    session_data BLOB,
    created_at BIGINT UNSIGNED NOT NULL DEFAULT (UNIX_TIMESTAMP())
  ) ENGINE=ROCKSDB
    COMMENT='ttl_duration=86400; ttl_col=created_at';
```

Storage: 8-byte timestamp prepended to every key's value:

- Primary key value: [8B TTL ts][field data]
- Secondary index value: [8B TTL ts][unpack info]

Why in every index?

- Enables independent compaction per index (per column family)
- No cross-index lookups needed during cleanup
- Read filtering works on any access path (PK scan, SK scan, covering index)

Cost: 8 bytes × (1 PK + N SIs) per row. Negligible after compression.

Meta

Read Filtering: Immediate & Consistent

Per-transaction filtering timestamp:

- Assigned once at transaction start
- Derived from Hybrid Logical Clock (HLC)
- Immutable for the transaction's lifetime

Filter check (every row read):

expired = row_ts + ttl_duration ≤ snapshot_time

Guarantees:

- A row is either visible or invisible for the entire
- transaction
- No row changes TTL status mid-transaction
- ⇒Snapshot isolation preserved

Example:

Snapshot T =50:

- Row A (exp 30): hidden
- Row B (exp 60): visible
- Row C (exp 45): hidden
- Row D (exp 80): visible

Snapshot T =70:

- Row A (exp 30): hidden
- Row B (exp 60): hidden
- Row C (exp 45): hidden
- Row D (exp 80): visible

Same data, fewer live rows.

Compaction: Safe Physical Cleanup

Compaction filter rule: row_ts + ttl_duration ≤ oldest active snapshot_time

Only drops rows invisible to every active transaction.

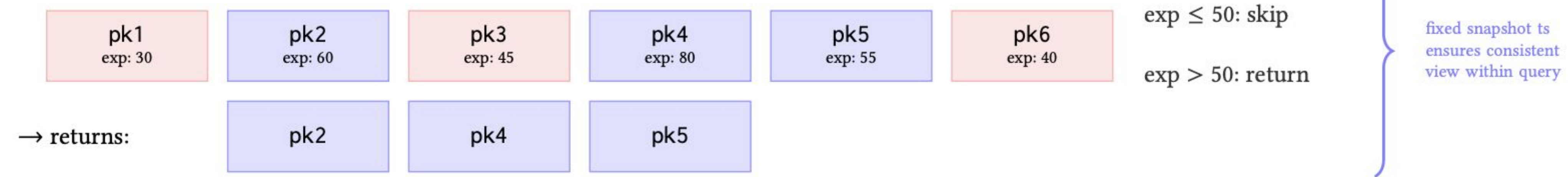
Advantages over DELETE-based cleanup:

- No tombstones → no tombstone cascading in level compaction
- No SQL CPU, no binlog, no replication traffic
- Single-pass: read once, write survivors
- DELETE requires two-pass in level compaction (2× write I/O)

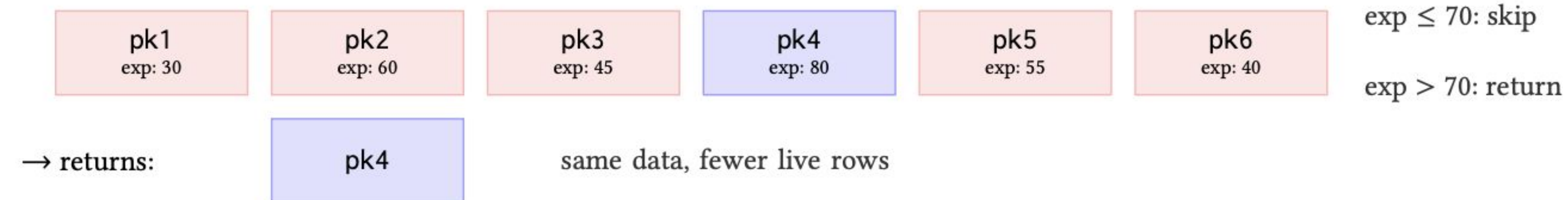
Challenge: RocksDB's scheduler doesn't know about TTL → we added TTL-aware compaction scheduling.

Read Filtering

Snapshot A (T=50):

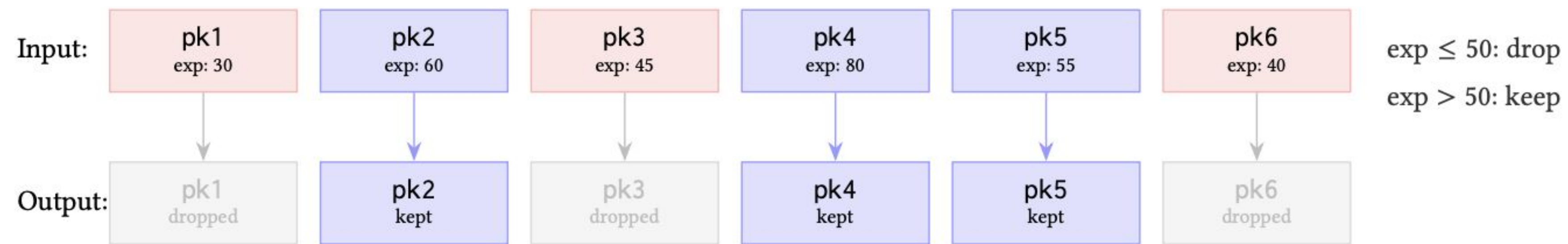


Snapshot B (T=70):

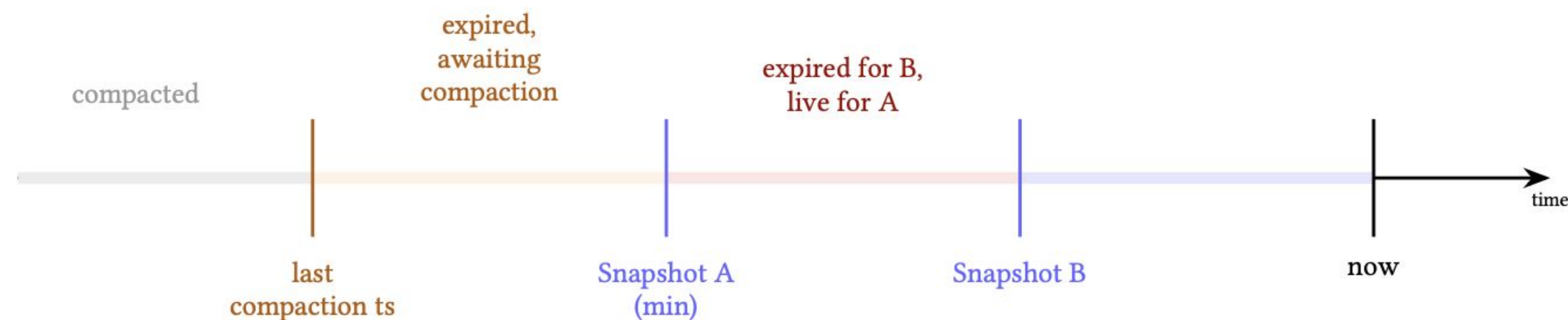


Compaction

Active snapshots: A (T=50), B (T=70) $\Rightarrow$ compaction safe to drop exp $\leq$ min(50, 70) = **50**



Expiration Timeline



Meta

Replication: The Hard Part

Primary	Replica
T=1 INSERT row A (expires at T=10)	← replicates → INSERT row A
T=8 UPDATE row A (extend TTL, now expires at T=20)	
	(UPDATE hasn't arrived yet)
T=11	Replica thinks row A expired, compaction runs → row A deleted ✗
T=12	UPDATE replicated "row not found" ERROR ✗

Meta

Replication: The Hard Part

Three consistency challenges:

1. Premature purge: Replica lags behind → compaction drops a row → pending replication entry tries to update it → "row not found"
2. Read drift: Replica uses its own clock for filtering → sees different set of live rows than primary → state that never existed on primary
3. Clock skew: Wall clock differences between nodes → same row appears expired on one node, live on another

Solution: Replicate timing metadata, not deletion events

- Primary writes its oldest snapshot time to the binlog. Replica uses that as an upper bound → replica never deletes a row before the primary does.
- Each transaction in the binlog carries the exact snapshot time used on the primary. Replica uses that same time for read filtering → sees the identical set of live vs expired rows.

Meta

Time Source: HLC, Not Wall Clock

Restoring a backup would immediately expire rows based on current wall clock time

Solution: Hybrid Logical Clock (HLC)

- Advances only on transaction commit or heartbeat
- Saved with backups → restored instance holds time at recovery point

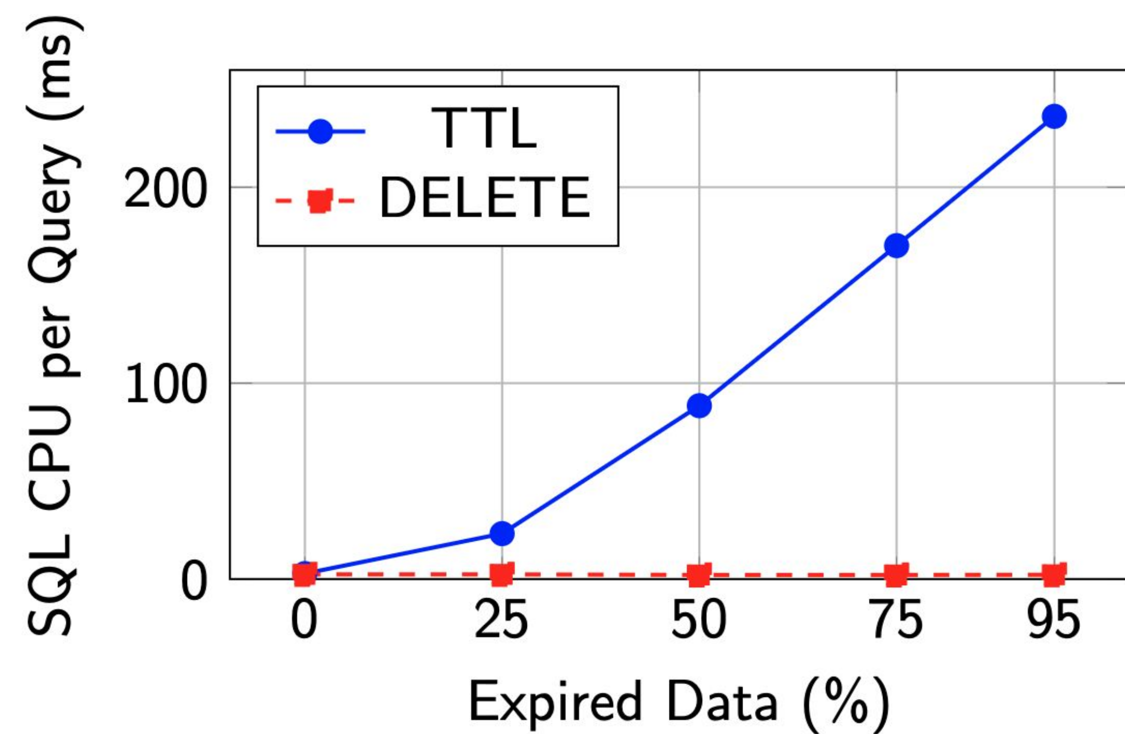
Meta

Evaluation: Storage Overhead

SI Count	Δ/row (bytes)	Expected
0	8.1	8
1	16.1	16
3	32.1	32
5	48.1	48
10	88.1	88

8 bytes per key, perfectly linear: $(N+1) \times 8$ bytes per row.
On-disk with compression: 0.1–2.7 bytes per row.
No extra key writes — only the value size increases.

Evaluation: Read Performance with Expired Data



TTL tradeoff:

- Read filtering adds~0.34 μs per expired row
- At 75% expired: 170 ms vs 2.8 ms baseline
- Drops to zero after compaction

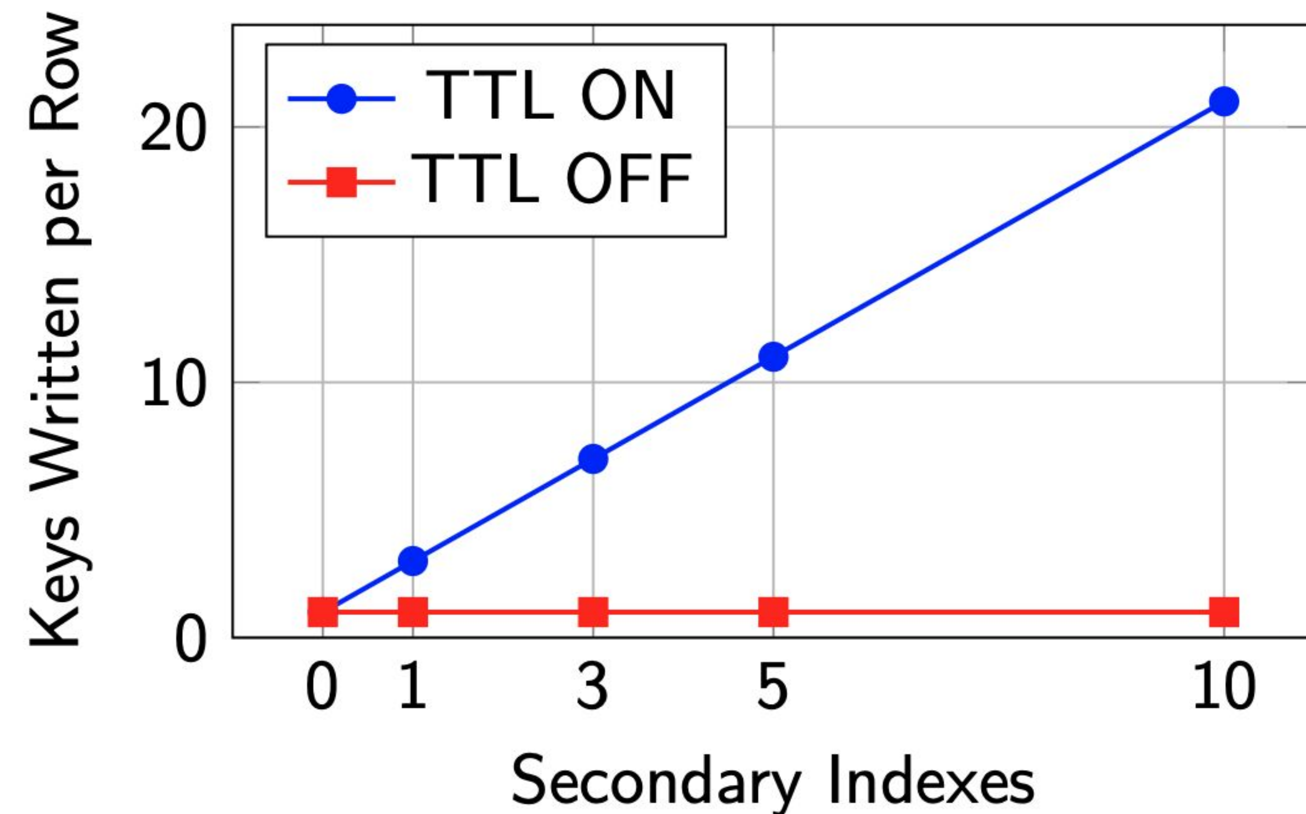
DELETE is constant (~2.2 ms):

- Tombstones resolved at iterator level
- But paid 20 s CPU + 136 MB binlog upfront

Takeaway: Timely compaction is essential for TTL tables.

Meta

Evaluation: TTL Timestamp Update Cost



When the TTL column is updated:

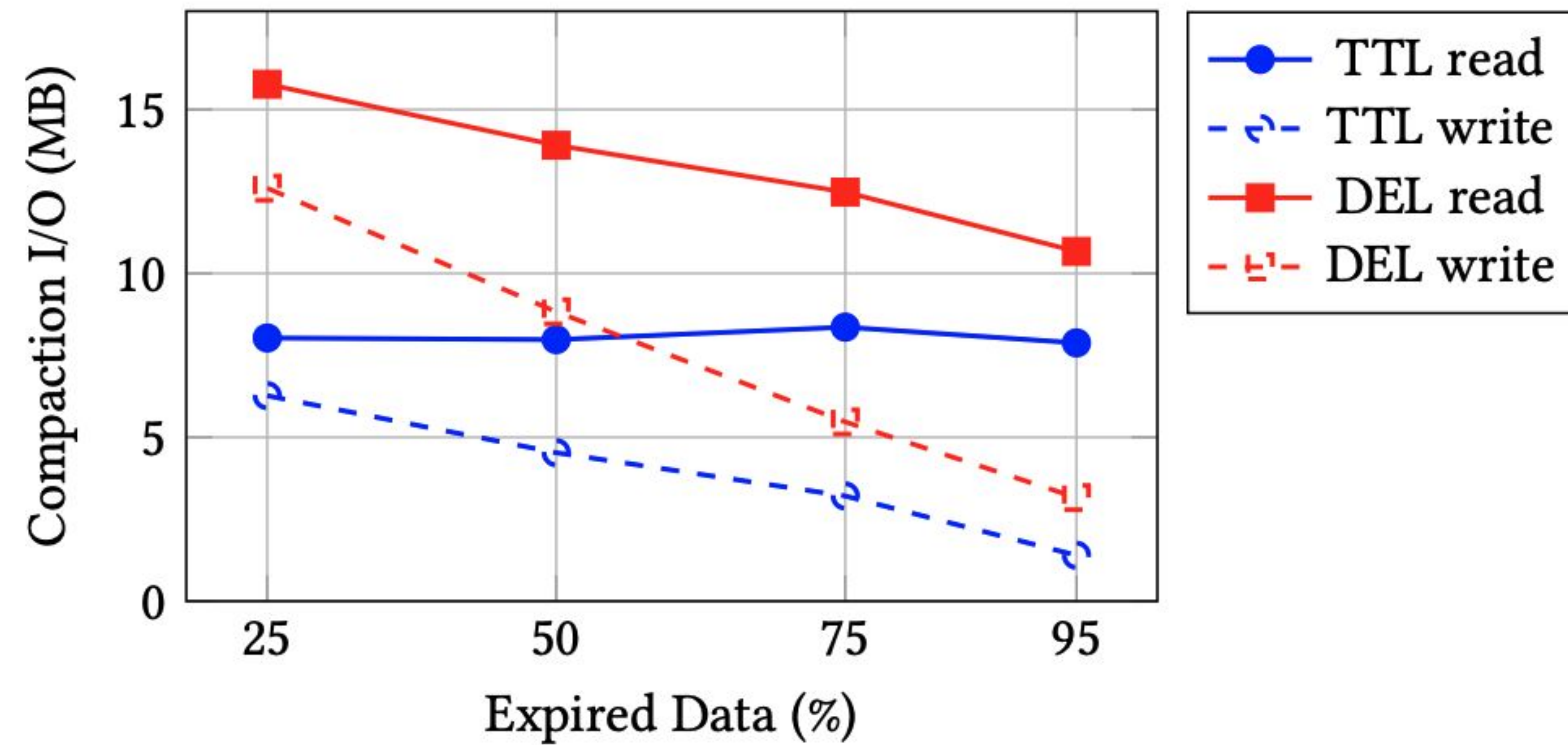
- Every SI value contains the TTL ts
- Must delete-old/insert-new for each SI
- 1 + 2N key writes per row
- At 10 SIs: 6.4×more CPU

In practice:

- Most tables use passive expiry (write once)
- This cost is zero for passive TTL
- Recommendation: minimize SIs on TTL
- tables

Meta

Evaluation: Compaction I/O



TTL: single-pass

- Read: constant (~8 MB)
- Write: decreases with more expired rows
- No tombstones →no cascading

DELETE: two-pass in level compaction

- 2×more write I/O at every ratio
- Tombstones require merge + rewrite

Correctness: TTL expired exactly 250K/500K/750K/950K rows at each ratio.

Meta

Production: What We Learned

Issues

- Table grew despite TTL working correctly: RocksDB scheduler doesn't know about expired rows (no tombstones!)
- Too many SIs and high rate of update to TTL column caused high write amplification

Fixes

- Make RocksDB scheduler aware of expired rows
- Remove unused indexes
- Avoid frequent TTL column updates

Automation Surprises

- Shard migration: destination compaction must be paused
- Checksum verification: must use explicit read filtering timestamp
- Binlog batching: cannot merge transactions with different TTL timestamps

Meta

Conclusion

TTL in MyRocks

- Read filtering + compaction filter = immediate invisibility + space reclamation
- Snapshot isolation preserved via per-transaction filtering timestamp
- Replica consistency via minimal timing metadata in binlog
- Zero overhead for passive expiry (the common case)

Future work

- Smarter compaction scheduling (prioritize by estimated expired ratio)
- Reduce SI write amplification without sacrificing correctness

Open source: github.com/facebook/mysql-5.6

Meta