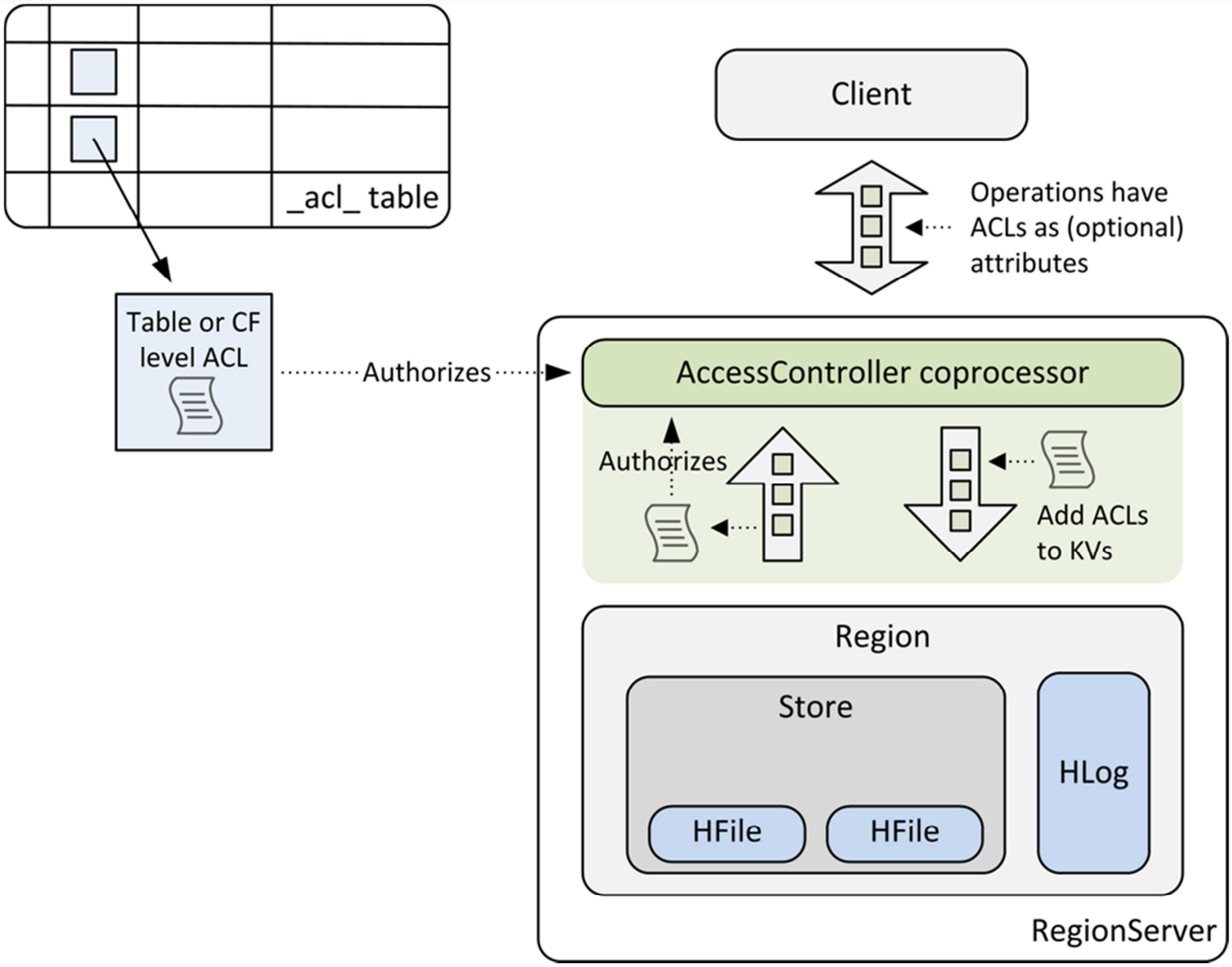
We extend the AccessController with mechanism to support authorization on a per cell basis. Our goal is to add per-cell ("per-KeyValue") security through as simple and straightforward extensions of the existing AccessController implementation as possible, with minimal changes to core code. We do not aim for Accumulo parity nor support for multi label security schemes, either of which could be follow on orthogonal work.

This is a high level diagram of AccessController authorization including our changes:



First we add a mechanism for storing per cell metadata. Two implementation options are provided as separate patches. The current proposal (*6222-aclcf.patch*) employs a "shadow" column family to store ACL data. This column family is automatically added to new and existing tables (currently without provision for resolving naming collision) and protected from all user or admin actions except for those with GLOBAL ADMIN privilege. As before we maintain the AccessController check for a user's access rights to the table and the CF. The ACL table and AccessControlProtocol endpoint are not changed. We then extend the AccessController to also evaluate the union of CF or table level permissions with any

permissions stored at KV scope (if any) covered by the pending operation. There are early outs and optimizations made possible by union-of-ACL semantics. Promotion of user privilege from KV to CF to table level scope is a potential performance optimization, if policy allows.

Requests created by HBase clients can include arbitrary attributes. We introduce an attribute for carrying ACLs and a helper function for serializing ACLs as protobuf for RPC. We can also consider adding methods for setting permissions directly to mutations types.

On the server, the AccessController applies the ACL of the request to the KVs submitted in the request.

ACLs are added only logically to KVs. The ACL metadata is stored in the shadow column family. However, note that on this issue we also provide an alternate implementation (as *6222-inline.patch*) that stores ACL metadata inline with the KVs, using a hack (as *kv-tags.patch*) to introduce tags to KVs in a backwards compatible way.

We treat ACLs on a KV as timestamped like the rest of the KV. This allows simple and straightforward evolution of security policy over time without requiring expensive updates. An ACL in a new mutation applies only to all KVs in that mutation. To actually change the ACL on an existing cell, the cell must be replaced by a new Put to its exact location.

We require mutations to have covering permission. This is defined as the union of the user's table perms, CF perms, and perms in any ACL metadata for the most recent visible version of a value, if the value exists. For operations that touch multiple values, all must allow the pending mutation in order for it to be applied. So for example for Deletes all visible prior versions that would be covered by the tombstone must authorize the operation.

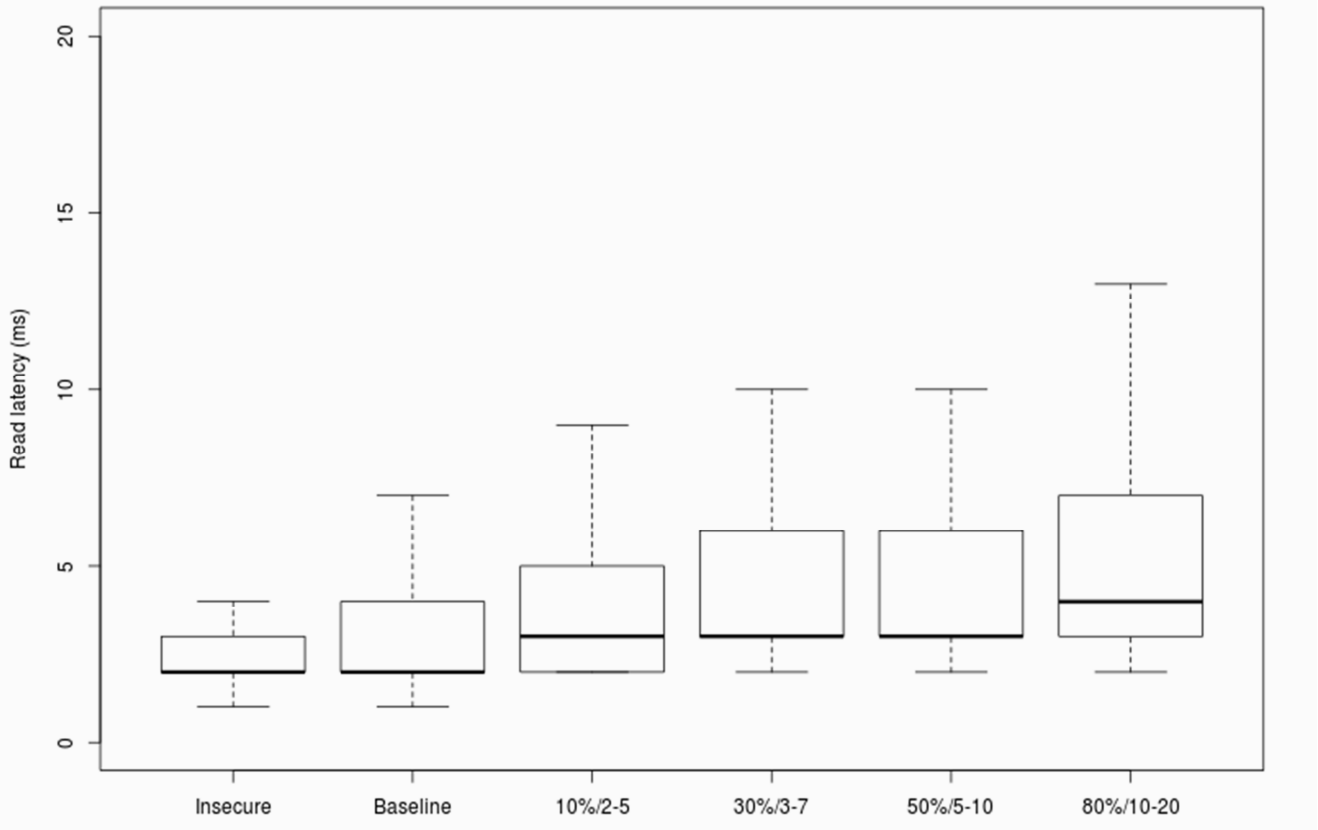
Both Gets and Scans use an InternalScanner to iterate over stores seeking the requested data. To enforce access control, the AccessController injects an AccessControlFilter that determines for each encountered value if it is visible to the user. We extend the AccessControlFilter to retrieve permissions metadata for values as they are streamed through. For mutations, in addition to existing table and CF permission checks we use an InternalScanner bounded on the parameters of the mutation (row, column, qualifier, timestamp (if provided)) to look for the most recent visible existing value within the given timerange; if one is found, permissions metadata is retrieved and checked. If there are table or CF permissions granting access for the user, we can early out and do not need to do the scan (no additional IO). For deletes, we use an InternalScanner bounded on some of the parameters of the Delete, but we must take care to look for all visible values that might be covered by the tombstone. If any permissions metadata for visible covered values do not grant the user permission, the Delete is rejected with an AccessDeniedException. Therefore, Delete#deleteFamily and Delete#deleteColumns can be expensive. For these we must iterate over every row in the region, but security always has some price, we pay it here instead of on the read side. Finally, for MultiActions, the actions are individually executed on the server, we do not need to do anything special here. Any AccessDeniedException thrown during processing of an action will be returned in the MultiResponse to indicate failure.

To determine the performance impact of these changes, we introduce a large unit test TestCellACLLoadTest, based on TestMiniClusterLoad, that extends the writer threads to add ACLs for five test cases:

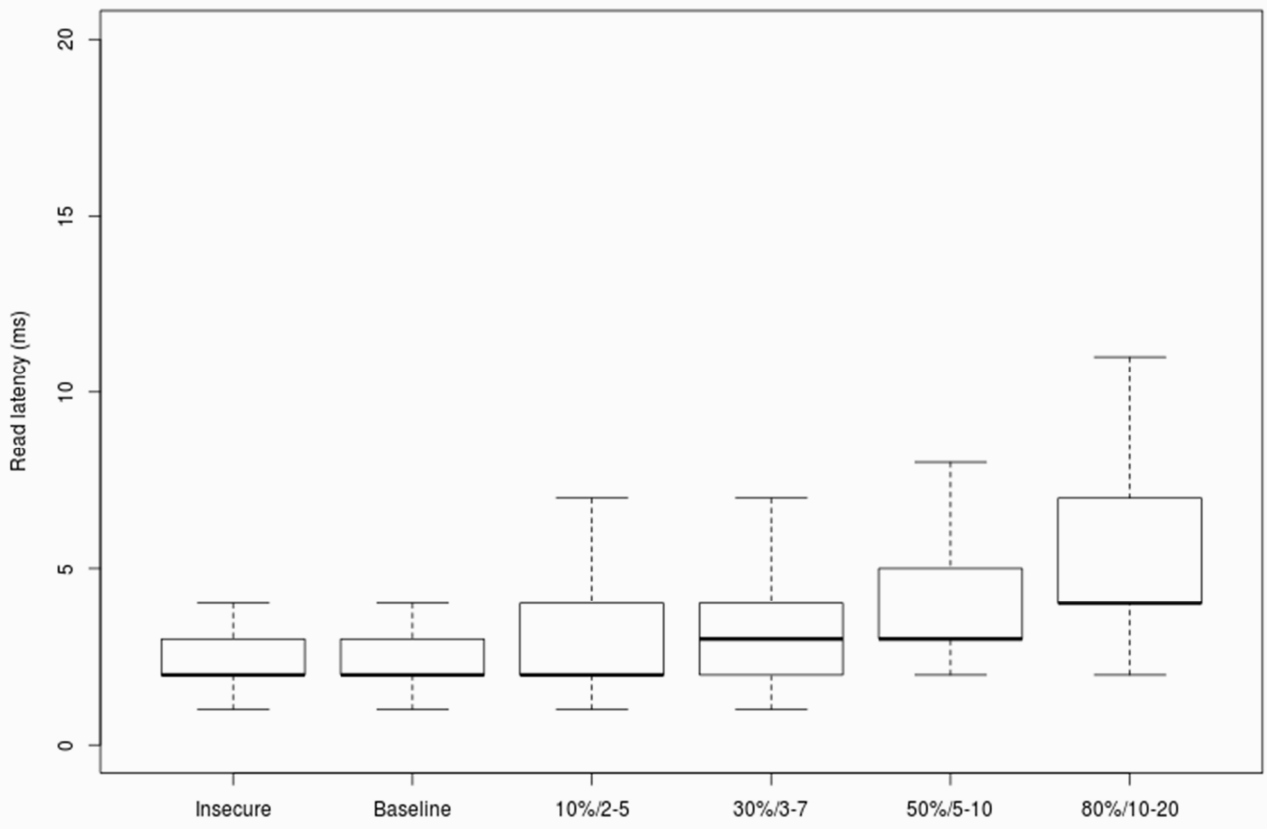
- Baseline
- 10% of cells have ACLs, 2-5 entries per ACL
- 30% of cells have ACLs, 3-7 entries per ACL
- 50% of cells have ACLs, 5-10 entries per ACL
- 80% of cells have ACLs, 10-20 entries per ACL

We also lower the RegionServer block cache size from 25% of heap to 5% of heap to emulate real world conditions under heap pressure (multiple tables, many regions, etc.).

Here are the read side latency results for the "shadow" CF storage option:



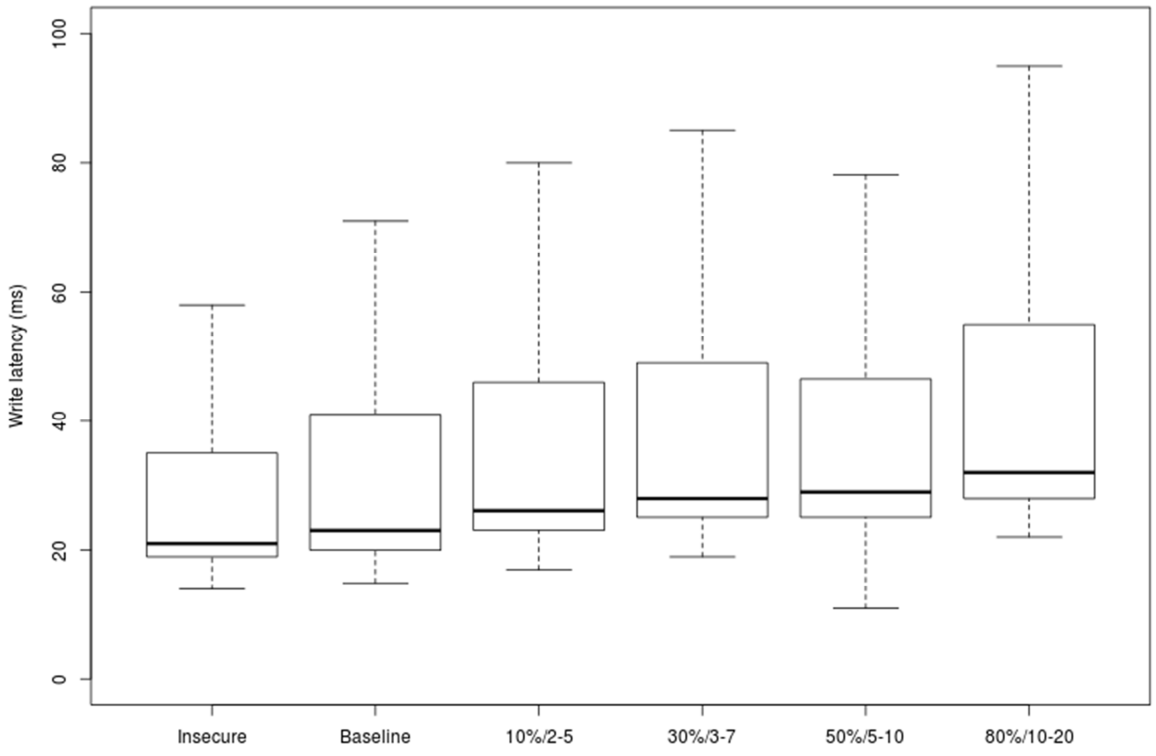
For comparison, the results for the inline KV tags storage option:



Write side measurements present an even clearer difference posed by the choice to store ACL data out of line with the KV itself, versus inline with the KV. See the next page.

Further work should focus on proper mechanism for inline storage of permissions metadata with the KV itself. This option benefits from the sequential read already in progress. In contrast, the “shadow" CF approach requires additional reads for blocks from the shadow column family. However, we cannot use this approach presently as HBase does not yet provide inline storage of KV metadata.

ACL CF



Inline Tags