IAVL Handover

Repo: https://github.com/cosmos/iavl
API docs: https://pkg.go.dev/github.com/cosmos/iavl
Dev docs: https://github.com/cosmos/iavl/blob/master/docs/overview.md

For a detailed description of IAVL internals, see dev docs above (some parts may possibly be outdated).

Overview

- IAVL is a persistent key/value store with Merkelization and versioning.
- Used by Cosmos SDK, but not Tendermint.
- Key/value pairs are organized in a variant of an AVL binary search tree.
    - Guarantees O(log n) access time.
    - Branches differ in height by at most 1, enforced by node rotation.
    - Values are only stored in leaf nodes.
    - Inner nodes are used for Merkelization and to guide key lookups.
- Node data is stored in a node key/value database.
    - Uses a tm-db DB implementation for storage (e.g. LevelDB, MemDB, etc).
    - Custom node serialization based on varints and length-prefixed byte slices.
    - Key formats for nodes, roots (versions), and orphans.
- Supports versioning by staging key/value modifications until the next version is saved or rolled back.
    - Copy-on-write scheme indexed by node hash (see nodeKeyFormat).
    - Each version points to a root node (see rootKeyFormat).
    - Deleted or changed nodes that are no longer visible in the tree head but used by past versions are called orphaned (see orphanKeyFormat). These keep track of the range they are used by, and will be removed when these versions are deleted.
    - MutableTree has the latest (staged) version of the tree, which can be modified and will be persisted on SaveVersion().
    - ImmutableTree contains a read-only tree of a specific version.
- Can provide presence proofs, range proofs, and absence proofs.
    - Legacy Tendermint proof format.
    - New ICS23 proof format.
- Also has a recent gRPC server interface, for use by non-Go projects.

Project Management

IAVL has been maintained “on the side”, with a very simple low-effort approach.

- Single development branch in `master`.

    - Exception: during 0.15 development, branch 0.14.x was used as an ad hoc maintenance branch for 0.14 patch releases. This is mostly obsolete now that 0.15 has been released.
- Submit PRs from feature branches or forks.
    - Must be approved by at least one code owner.
    - CI tests must pass (uses GitHub Actions to run e.g. go test and linters).
    - Update `CHANGELOG.md` with change info, if necessary.
- Releases are cut from `master`.
    - Use semantic versioning (patch releases should be backwards compatible)
    - Update `CHANGELOG.md` with release date and other info.
    - Create a new release via the GitHub release page
        - Tag ‘vX.Y.Z’, e.g. ‘v0.15.4’
        - Title same as tag.
        - Description: permalink to changelog entry anchor.
        - Mark as pre-release for alpha/beta/rc releases (with appropriate tag suffix).
- Docker images for the gRPC server are built and published by GitHub Actions for each Git tag and each push to master. These currently use the interchainio/iavl Docker image name.

Code Structure

- Main API and AVL tree implementation
    - mutable_tree.go: MutableTree for tree modifications
    - immutable_tree.go: ImmutableTree for (historical) tree queries
- Storage
    - node.go: tree node data structure and (de)serialization
    - nodedb.go: node database for storage
    - encoding.go: (de)serialization utility functions
    - key_format.go: database key encoding (node, root, orphan)
- Proofs
    - proof_ics23.go: ICS23-format proof code
    - proof*.go: Legacy Tendermint-format proof code
- Import/export (e.g. state sync)
    - export.go: node exporter
    - import.go: node importer
- gRPC server
    - proto/: Protobuf schemas and generated code
    - server/: gRPC server

Testing

Tests are implemented as normal Go tests, and can be run with e.g. ‘go test’. Test coverage is very spotty, and the few tests we do have don’t cover enough failure modes or edge cases.

In order to improve test coverage with fairly little effort, a randomized test suite was written in tree_random_test.go which runs a set of random operations (including e.g. version deletion and rollbacks) and mirrors them in a known-good map, comparing the results. This is currently the most reliable test suite for detecting bugs, but should be accompanied by comprehensive unit tests.

Tests are run in CI via GitHub Actions, see .github/workflows/ci.yml for details.

Challenges

IAVL appears to be correct, i.e. no known major bugs in the implementation, but:

- Performance is poor, e.g.:
    - Key lookups use an AVL tree layered on top of a database with its own lookup data structure (e.g. B-tree or LSM-tree). This causes lots of random access lookups in the underlying database as the AVL tree is searched. The database should do lookups for us.
    - Version deletion deletes each version separately, which means the orphans will be rewritten over and over for each deletion instead of simply removing them when deleting a version range.
- Not concurrency-safe, and must be protected by an external mutex.
    - However, ImmutableTree is safe for concurrent read-only access as long as the caller can guarantee that the version will not be deleted.
    - Should offload concurrency control and versioning to underlying database (e.g. MVCC), which can provide fine-grained concurrency control per key/version.
    - Should have better isolation. E.g. node db writes are staged in a database batch, and are not visible to reads, which makes e.g. orphaning logic hard to reason about and easy to get wrong. Should use ACID transactions instead.
- Versioning (orphaning) code is complex and difficult to reason about: attempts to change/extend it have caused severe data corruption bugs (see v0.13 pruning).
- Panics are often used for error handling, causing library users to crash. It should return errors instead.
- Tests are lacking (see above).

Overall, IAVL takes on too much responsibility itself, and does a poor job at it. It should offload most of the heavy lifting to a mature and battle-tested database.