Cellar Roadmap

This document tracks proposed features, improvements, and code cleanup opportunities for the Cellar disk cache library.

Feature Proposals

[Priority: High] Cache Type with Integrated State Management

Description: Add a high-level `Cache` type that wraps `CacheIndex` with `IO.Ref` for stateful operations, providing a simpler API for common use cases.

Rationale: Currently, users must manually manage the `CacheIndex` state and thread it through all operations. A stateful `Cache` type would reduce boilerplate and make the library easier to use correctly.

Proposed API:

```
structure Cache (K : Type) [BEq K] [Hashable K] where
  indexRef : IO.Ref (CacheIndex K)

namespace Cache
  def create (config : CacheConfig) : IO (Cache K)
  def get (cache : Cache K) (key : K) : IO (Option ByteArray)
  def put (cache : Cache K) (key : K) (data : ByteArray) : IO Unit
  def delete (cache : Cache K) (key : K) : IO Unit
  def clear (cache : Cache K) : IO Unit
end Cache
```

Affected Files: New file `Cellar/Cache.lean`

Estimated Effort: Medium

Dependencies: None

[Priority: High] Key-to-Path Mapping Typeclass

Description: Add a `KeyPath` typeclass that allows users to define how cache keys map to file paths, centralizing path logic.

Rationale: Currently, users must manually compute file paths for cache entries (as seen in afferent's `TileDiskCache`). A typeclass would standardize this pattern and reduce duplication.

Proposed API:

```
class KeyPath (K : Type) where
  toPath : CacheConfig -> K -> String
  fromPath : String -> Option K  -- For cache reconstruction

instance : KeyPath String where
  toPath config key := s!"{config.cacheDir}/{key}"
  fromPath path := some (System.FilePath.fileName path)
```

Affected Files: New file `Cellar/KeyPath.lean`, updates to `Cellar/Config.lean`

Estimated Effort: Small

Dependencies: None

[Priority: High] Cache Persistence and Reconstruction

Description: Add functionality to persist the cache index to disk and reconstruct it on startup by scanning the cache directory.

Rationale: Currently, the in-memory index is lost when the application exits. Users must rebuild the index by scanning the filesystem, but no helper functions exist for this.

Proposed API:

```
-- Scan directory and rebuild index
def CacheIndex.fromDirectory [KeyPath K] (config : CacheConfig) : IO (CacheIndex K)

-- Persist index to disk (optional, for faster startup)
def CacheIndex.save (index : CacheIndex K) (path : String) : IO Unit
def CacheIndex.load (path : String) : IO (Except String (CacheIndex K))
```

Affected Files: New file `Cellar/Persist.lean`, updates to `Cellar/IO.lean`

Estimated Effort: Medium

Dependencies: Key-to-Path Mapping Typeclass (for `fromDirectory`)

[Priority: Medium] TTL (Time-to-Live) Support

Description: Add optional TTL-based expiration in addition to LRU eviction.

Rationale: Some cache use cases require entries to expire after a fixed time period regardless of access patterns. This is common for HTTP caches and API response caching.

Proposed Changes: - Add optional `ttlMs : Option Nat` field to `CacheConfig` - Add `expiresAt : Option Nat` field to `CacheEntry` - Implement `selectExpiredEntries` function - Modify `selectEvictions` to consider TTL alongside LRU

Affected Files: `Cellar/Config.lean`, `Cellar/LRU.lean`

Estimated Effort: Medium

Dependencies: None

[Priority: Medium] Cache Statistics and Metrics

Description: Track cache hit/miss counts and other usage statistics.

Rationale: Useful for debugging, tuning cache sizes, and understanding cache effectiveness.

Proposed API:

```
structure CacheStats where
  hits : Nat
  misses : Nat
  evictions : Nat
  bytesWritten : Nat
  bytesRead : Nat
  deriving Repr, Inhabited

def CacheStats.hitRate (stats : CacheStats) : Float :=
  if stats.hits + stats.misses == 0 then 0.0
  else stats.hits.toFloat / (stats.hits + stats.misses).toFloat
```

Affected Files: New file `Cellar/Stats.lean`, updates to `Cellar/Config.lean`

Estimated Effort: Small

Dependencies: Cache Type with Integrated State Management (to track operations)

[Priority: Medium] Async/Concurrent Cache Operations

Description: Add thread-safe cache operations using Lean's `Mutex` or `IO.Ref.atomically`.

Rationale: Disk caches are commonly accessed from multiple tasks/threads (e.g., background prefetching, concurrent tile downloads). Thread-safety is essential for production use.

Proposed Changes: - Wrap `CacheIndex` access with mutex for thread safety - Consider using `IO.Ref.atomically` for atomic updates - Add async-friendly batch operations

Affected Files: `Cellar/Cache.lean` (new file), potentially new `Cellar/Concurrent.lean`

Estimated Effort: Medium

Dependencies: Cache Type with Integrated State Management

[Priority: Medium] Item Count Limit

Description: Add option to limit cache by number of entries in addition to byte size.

Rationale: Some use cases care more about the number of cached items than their total size.

Proposed Changes: - Add `maxEntries : Option Nat` to `CacheConfig` - Update `selectEvictions` to check both size and count limits

Affected Files: `Cellar/Config.lean`, `Cellar/LRU.lean`

Estimated Effort: Small

Dependencies: None

[Priority: Low] Compression Support

Description: Add optional transparent compression/decompression of cached data.

Rationale: Can significantly reduce disk usage for compressible data, though adds complexity.

Proposed Changes: - Add `compression : CompressionType` to `CacheConfig` (None, Gzip, Zstd) - Implement compression wrappers in IO module - Store compression metadata in cache entries

Affected Files: `Cellar/Config.lean`, `Cellar/IO.lean`, potentially new `Cellar/Compress.lean`

Estimated Effort: Large (requires FFI for compression libraries)

Dependencies: None

[Priority: Low] Cache Namespaces/Partitions

Description: Support multiple independent cache namespaces within a single cache directory.

Rationale: Allows grouping related entries and managing them independently (e.g., clear all tiles without affecting other cached data).

Proposed API:

```
def CacheConfig.withNamespace (config : CacheConfig) (ns : String) : CacheConfig

-- Clear all entries in a namespace
def Cache.clearNamespace (cache : Cache K) (ns : String) : IO Unit
```

Affected Files: `Cellar/Config.lean`, Cache module

Estimated Effort: Medium

Dependencies: Cache Type with Integrated State Management

Code Improvements

[Priority: High] Replace Except with Proper Error Type

Current State: Functions in `Cellar/IO.lean` return `Except String Unit` or `Except String ByteArray`, losing structured error information.

Proposed Change: Define a proper error ADT for cache operations:

```
inductive CacheError
  | fileNotFound (path : String)
  | ioError (message : String)
  | permissionDenied (path : String)
  | diskFull
  | corruptedData (path : String)
  deriving Repr, Inhabited

abbrev CacheIO := ExceptT CacheError IO
```

Benefits: Better error handling, pattern matching on error types, clearer API contracts.

Affected Files: `Cellar/IO.lean`, all modules using IO functions

Estimated Effort: Medium

[Priority: High] Improve LRU Algorithm Efficiency

Current State: `selectEvictions` in `Cellar/LRU.lean` converts HashMap to list, sorts the entire list, then iterates.

```
let sorted := index.entries.toList.map Prod.snd
  |>.toArray.qsort (fun a b => a.lastAccessTime < b.lastAccessTime)
  |>.toList
```

Proposed Change: Use a priority queue or maintain a separate sorted structure for LRU ordering. Consider: - `BinaryHeap` for O(log n) insert/extract-min - Doubly-linked list with HashMap for O(1) operations (classic LRU cache pattern)

Benefits: Better asymptotic performance for large caches.

Affected Files: `Cellar/LRU.lean`, possibly new `Cellar/PriorityQueue.lean`

Estimated Effort: Medium

Dependencies: May require additional data structure from batteries or custom implementation

---

[Priority: Medium] Add Monadic Interface for Cache Operations

Current State: Cache operations return raw values, requiring manual threading of state.

Proposed Change: Add a `CacheM` monad that encapsulates cache state and IO:

```
abbrev CacheM (K : Type) [BEq K] [Hashable K] := StateT (CacheIndex K) IO

def runCacheM [BEq K] [Hashable K] (index : CacheIndex K)
    (action : CacheM K  ) : IO (  × CacheIndex K) :=
  action.run index
```

Benefits: Cleaner composition of cache operations, easier error handling.

Affected Files: New file `Cellar/Monad.lean`

Estimated Effort: Small

[Priority: Medium] Improve Atomic Write Robustness

Current State: `writeFile` uses temp file + rename, but doesn't handle: - Cleanup of temp files on crash - Cross-filesystem moves (rename fails) - fsync for durability

Proposed Changes:

```
-- Add fsync before rename for durability
IO.FS.Handle.sync handle

-- Handle cross-filesystem case
if sameFilesystem tmpPath path then
  IO.FS.rename tmpPath path
else
  -- Fall back to copy + delete
  IO.FS.writeBinFile path data
  deleteFile tmpPath
```

Benefits: More robust cache operations, especially for production use.

Affected Files: `Cellar/IO.lean`

Estimated Effort: Small

[Priority: Medium] Add Typeclass Constraints Documentation

Current State: Functions require `[BEq K] [Hashable K]` but don't explain why.

Proposed Change: Add doc comments explaining the typeclass requirements and their purpose.

Affected Files: `Cellar/Config.lean`, `Cellar/LRU.lean`

Estimated Effort: Small

[Priority: Low] Consider Using System.FilePath More Consistently

Current State: Mix of `String` and `System.FilePath` types for paths.

Proposed Change: Use `System.FilePath` consistently throughout the codebase for type safety.

Affected Files: All files

Estimated Effort: Small

Code Cleanup

[Priority: High] Add Test Suite

Issue: The project has no tests (noted in CLAUDE.md: "Projects without a test target: canopy, cellar, crucible").

Location: Project-wide

Action Required: 1. Add crucible as a dependency in `lakefile.lean` 2. Create `Cellar/Tests/` directory with test files: - `ConfigTests.lean` - Test CacheConfig, CacheEntry, CacheIndex - `LRUTests.lean` - Test eviction logic - `IOTests.lean` - Test file operations (with temp directories) 3. Add test target to lakefile

Estimated Effort: Medium

[Priority: High] Clean Up Temp Files on Write Failure

Issue: In `Cellar/IO.lean`, `writeFile` can leave orphaned temp files if the rename fails after the temp file is written.

Location: `Cellar/IO.lean`, line 30-34

Action Required: Add error handling to clean up temp file on rename failure:

```
try
  IO.FS.rename tmpPath path
  pure (.ok ())
catch e =>
  -- Clean up temp file before returning error
  try IO.FS.removeFile tmpPath catch _ => pure ()
  pure (.error (toString e))
```

Estimated Effort: Small

[Priority: Medium] Add Module-Level Documentation

Issue: Source files have minimal documentation beyond brief header comments.

Location: All `.lean` files

Action Required: - Add comprehensive module docstrings - Document each public function with examples - Add usage examples in doc comments

Estimated Effort: Small

[Priority: Medium] Validate CacheConfig on Creation

Issue: `CacheConfig` can be created with invalid values (e.g., empty `cacheDir`, zero `maxSizeBytes`).

Location: `Cellar/Config.lean`, line 10-15

Action Required: Add validation function or smart constructor:

```
def CacheConfig.create (cacheDir : String) (maxSizeBytes : Nat) : Except String CacheConfig :=
  if cacheDir.isEmpty then .error "cacheDir cannot be empty"
  else if maxSizeBytes == 0 then .error "maxSizeBytes must be positive"
  else .ok { cacheDir, maxSizeBytes }
```

Estimated Effort: Small

[Priority: Medium] Consider BEq Instance for CacheEntry

Issue: The current `BEq` instance for `CacheEntry` only compares `filePath`, ignoring other fields. This may lead to unexpected behavior.

Location: `Cellar/Config.lean`, lines 29-30

Action Required: Either: 1. Remove the instance and let users compare entries explicitly 2. Document the behavior clearly 3. Change to compare all relevant fields

```
-- Current (potentially confusing):
instance [BEq K] : BEq (CacheEntry K) where
  beq a b := a.filePath == b.filePath

-- Alternative (more explicit):
instance [BEq K] : BEq (CacheEntry K) where
  beq a b := a.key == b.key && a.filePath == b.filePath
```

Estimated Effort: Small

[Priority: Low] Add Repr Instance for CacheIndex

Issue: `CacheIndex` has `Inhabited` but not `Repr`, making debugging harder.

Location: `Cellar/Config.lean`, line 33-40

Action Required: Add `Repr` instance:

```
instance [Repr K] [BEq K] [Hashable K] : Repr (CacheIndex K) where
  reprPrec idx _ :=
    s!"CacheIndex(entries: {idx.entries.size}, totalSize: {idx.totalSizeBytes}B)"
```

Estimated Effort: Small

[Priority: Low] Use Nat Subtraction Safely

Issue: In `Cellar/LRU.lean`, `removeEntries` uses natural number subtraction which could silently underflow if `removedSize > totalSizeBytes`.

Location: `Cellar/LRU.lean`, line 46-53

Action Required: Add guard or use saturating subtraction:

```
let newSize := if removedSize > index.totalSizeBytes then 0
               else index.totalSizeBytes - removedSize
```

Estimated Effort: Small

[Priority: Low] Consistent Error Handling Pattern

Issue: `deleteFile` always returns `(.ok ())` even on error, while other functions return errors. This inconsistency could be confusing.

Location: `Cellar/IO.lean`, lines 55-62

Action Required: Either: 1. Document this design decision clearly 2. Add a separate `deleteFileIfExists` function 3. Return the actual error but provide a helper that ignores errors

Estimated Effort: Small

Architecture Considerations

Separation of Concerns

The current design mixes pure index operations with IO. Consider a cleaner separation:

```
Cellar/
  Types.lean      -- Pure types (Config, Entry, Index)
  Index.lean      -- Pure index operations
  IO.lean         -- IO-only file operations
  Cache.lean      -- Stateful cache combining Index + IO
  Persist.lean    -- Index persistence
```

API Consistency

Consider aligning the API style with other workspace projects: - Use `do` notation consistently in IO operations - Consider a builder pattern for `CacheConfig` - Add `Cellar.` namespace prefix consistently

Dependency Management

The project currently has no dependencies beyond Std. Consider: - Adding crucible for tests - Adding batteries if more data structures are needed (priority queue, ordered map)

Summary

Category	High	Medium	Low	Total
Features	3	4	2	9
Improvements	2	3	1	6
Cleanup	2	3	4	9
Total	**7**	**10**	**7**	**24**

Recommended Starting Points: 1. Add test suite (High priority cleanup) 2. Implement Cache type with state management (High priority feature) 3. Add proper error types (High priority improvement) 4. Implement cache persistence (High priority feature)