Minard Loader Specification

Status: Planned **Target Language**: Rust or Go **Purpose**: Fast, reliable data ingestion for the Minard database

Overview

The loader is responsible for extracting information from PureScript (and eventually Haskell) codebases and populating the Minard DuckDB database. It must be fast enough to run on every compile without causing friction.

Current State

The existing loader (`database/loader/ce-loader.js`, ~800 lines Node.js) works but has issues:

Issue	Impact
~30s load time for 1000 modules	Discourages frequent updates
Single-threaded JSON parsing	CPU bottleneck
Full reload only	No incremental support
Brittle error handling	Fails silently on malformed input

Requirements

Functional Requirements

FR1: Project Discovery

- Accept path to PureScript project root
- Locate `spago.yaml`, `spago.lock`, `output/` directory
- Support both workspace (monorepo) and single-package projects

FR2: Data Extraction

Source	Data Extracted	Tables Populated
`spago.lock`	Package names, versions, dependencies	`package_versions`, `package_dependencies`
`output/*/docs.json`	Module names, exports, declarations, type signatures	`modules`, `declarations`, `child_declarations`
`output/*/corefn.json`	Function calls, import references	`function_calls`, `module_imports`
`src/**/*.purs`	Lines of code, source spans	`modules.loc`, `declarations.source_code`

Source	Data Extracted	Tables Populated
`git log`	Commits, authors, timestamps	`commits`, `module_commits`
`git diff` (optional)	Changed files since last snapshot	Incremental loading

FR3: Graph Construction

- Build module import graph (adjacency list)
- Build function call graph
- Compute transitive closures for reachability analysis
- Identify orphaned modules (not reachable from entry points)

FR4: Metrics Computation

- Module LOC (lines of code)
- Declaration count per module
- Coupling metrics (external calls in/out)
- Git churn (commits, authors, recency)

FR5: Database Operations

- Create snapshot record
- Insert all extracted data within transaction
- Support `--incremental` mode (only changed modules)
- Support `--replace` mode (delete previous snapshot first)

FR6: Multi-Project Support

- Track multiple projects in same database
- Each project has independent snapshots
- Cross-project queries remain possible

Non-Functional Requirements

NFR1: Performance

- Target: <5 seconds for 1000 modules (full load)
- Target: <1 second for incremental load (10 changed modules)
- Parallel file I/O and JSON parsing

NFR2: Reliability

- Graceful handling of malformed JSON
- Clear error messages with file paths
- Transaction rollback on failure
- No partial/corrupt database states

NFR3: Portability

- Single static binary (no runtime dependencies)
- macOS (ARM + x64), Linux (x64)

- Windows nice-to-have

NFR4: Observability

- Progress output (files processed, time elapsed)
- `--verbose` mode with detailed logging
- `--quiet` mode for CI integration
- JSON output mode for programmatic use

Command-Line Interface

```
minard-loader [OPTIONS] <COMMAND>

Commands:
  load        Load a project into the database
  snapshot    Create a new snapshot of an existing project
  incremental Update database with only changed files
  orphans     Report orphaned modules (no database write)
  stats       Show database statistics

Options:
  -d, --database <PATH>   Path to DuckDB file [default: ./minard.duckdb]
  -v, --verbose           Verbose output
  -q, --quiet             Suppress non-error output
  --json                  Output as JSON (for tooling)

load:
  minard-loader load <PROJECT_PATH> [OPTIONS]

  Options:
    -n, --name <NAME>       Project name [default: directory name]
    -l, --label <LABEL>     Snapshot label [default: timestamp]
    --no-git                Skip git history extraction
    --no-calls              Skip function call extraction (faster)

incremental:
  minard-loader incremental <PROJECT_PATH> [OPTIONS]

  Options:
    --since <COMMIT>        Only files changed since this commit
    --since-last            Only files changed since last snapshot

orphans:
  minard-loader orphans <PROJECT_PATH> [OPTIONS]

  Options:
    --entry <MODULE>        Entry point module [default: Main]
    --format <FMT>          Output format: text, json, csv
```

Data Structures

Input Parsing

```
// From docs.json
struct DocsJson {
    name: String,           // "Data.Array"
    comments: Option<String>,
    declarations: Vec<Declaration>,
    reExports: Vec<ReExport>,
}

struct Declaration {
    title: String,          // "map"
    comments: Option<String>,
    info: DeclarationInfo,
    sourceSpan: Option<SourceSpan>,
    children: Vec<ChildDeclaration>,
}

enum DeclarationInfo {
    ValueDeclaration { type_: Type },
    DataDeclaration { dataDeclType: String, typeArguments: Vec<...>,
constructors: Vec<...> },
    TypeSynonymDeclaration { ... },
    TypeClassDeclaration { ... },
    // ...
}

// From corefn.json
struct CorefnJson {
    moduleName: Vec<String>,  // ["Data", "Array"]
    imports: Vec<Import>,
    exports: Vec<String>,
    decls: Vec<Decl>,
}

struct Decl {
    // Contains Expr trees with function applications
    // Parse to extract call sites
}

// From spago.lock
struct SpagoLock {
    packages: HashMap<String, PackageInfo>,
}

struct PackageInfo {
    version: String,
    dependencies: Vec<String>,
}
```

Internal Representation

```rust
struct Project {
    name: String,
    root_path: PathBuf,
    packages: Vec<Package>,
    modules: Vec<Module>,
}

struct Module {
    id: ModuleId,
    name: String,           // "Data.Array"
    package: PackageId,
    path: Option<PathBuf>,  // src/Data/Array.purs
    loc: u32,
    imports: Vec<ModuleId>,
    declarations: Vec<Declaration>,
}

struct CallGraph {
    // caller (module.decl) -> [(callee_module, callee_decl)]
    edges: HashMap<(ModuleId, DeclId), Vec<(ModuleId, String)>>,
}
```

Algorithm Outline

```
1. DISCOVER
   - Find spago.yaml, determine project type (workspace vs single)
   - Find spago.lock, parse package versions
   - Find output/ directory, enumerate module directories
   - Find src/ directory, enumerate source files

2. PARSE (parallel)
   - For each output/<Module>/docs.json: parse declarations
   - For each output/<Module>/corefn.json: parse imports, calls
   - For each src/**/*.purs: count LOC, extract source spans
   - Parse git log for commit history

3. BUILD GRAPHS
   - Build module import graph
   - Build function call graph
   - Compute package dependencies from module imports

4. COMPUTE METRICS
   - Module LOC, declaration counts
   - Coupling: external calls in/out per declaration
   - Git: commit count, author count, days since modified

5. GENERATE SQL
   - Create snapshot record
   - Batch INSERT statements (1000 rows per statement)
   - Use prepared statements for type safety
```

```
6. EXECUTE
   - Begin transaction
   - Execute all INSERTs
   - Commit (or rollback on error)

7. REPORT
   - Print summary: modules loaded, declarations, time elapsed
   - Return exit code 0 on success
```

Incremental Loading

For `--incremental` mode:

1. Query database for last snapshot's file timestamps
2. Compare with current file mtimes
3. Parse only changed files
4. Delete old records for changed modules
5. Insert new records
6. Update snapshot metadata

Key insight: Module-level granularity is sufficient. If any file in a module changes, reload the entire module.

Future Extensions

HTTPurple Route Extraction

Parse server code to extract API routes:

```
-- Pattern to detect
route :: RouteDuplex' Route
route = root $ sum
  { "GetUser": path "api/users" (int segment)
  , "CreateUser": path "api/users" noArgs
  , "Health": path "health" noArgs
  }
```

Extract:

- Route name → URL pattern mapping
- HTTP method (from handler inspection)
- Request/response types (from type signatures)

Populate new tables:

- `api_routes (id, module_id, name, method, url_pattern)`
- `api_route_types (route_id, request_type, response_type)`

WebSocket Endpoint Extraction

Similar pattern matching for WebSocket handlers.

Cross-Reference with Frontend

Link frontend API calls to backend routes:

- Frontend: `fetch "/api/users"` or `Affjax.get "/api/users"`
- Backend: `"GetUser": path "api/users" ...`
- Result: `api_calls (frontend_module_id, backend_route_id)`

This enables:

- Dead API detection (defined but never called)
- Missing API detection (called but not defined)
- Full-stack dependency graphs

Haskell Support

The loader architecture should support Haskell projects:

- Parse `.cabal` or `package.yaml` instead of `spago.yaml`
- Parse Haddock output instead of `docs.json`
- Parse `.hi` files or GHC output for call graphs

Registry Integration

For PureScript:

- Fetch package metadata from registry API
- Populate `package_versions.description`, `license`, `repository`
- Track package release history

For Haskell:

- Fetch from Hackage/Stackage APIs

Testing Strategy

Unit Tests:

- JSON parsing (docs.json, corefn.json, spago.lock)
- Graph algorithms (transitive closure, orphan detection)
- SQL generation

Integration Tests:

- Load a known test project
- Verify expected tables populated
- Verify query results match expected

Performance Tests:

- Benchmark on large project (1000+ modules)

- Compare against Node.js baseline
- Track regression

Test Projects:

- `test-fixtures/minimal/` - 3 modules, basic structure
- `test-fixtures/workspace/` - Monorepo with multiple packages
- `test-fixtures/large/` - Generated 1000 modules (for perf)

Implementation Notes

Why Rust or Go?

Criterion	Rust	Go
Performance	Excellent	Very good
JSON parsing	serde (excellent)	encoding/json (good)
SQLite/DuckDB	rusqlite, duckdb-rs	go-duckdb
Binary size	~5MB	~10MB
Build time	Slow	Fast
Learning curve	Steep	Gentle

Recommendation: Go for faster iteration, Rust if performance is critical. Either is fine.

DuckDB Integration

Both languages have DuckDB bindings:

- Rust: `duckdb-rs` crate
- Go: `go-duckdb` package

Use prepared statements and batch inserts for performance.

Parallelism

- File I/O: Parallel directory traversal
- JSON parsing: Worker pool (num_cpus threads)
- SQL execution: Single-threaded (DuckDB limitation)

Error Handling

- Parse errors: Log warning, skip file, continue
- Missing files: Log warning, continue
- Database errors: Rollback, exit with error code
- All errors include file path and line number where applicable

Milestones

1. **M1: Basic Loading** - Parse docs.json, spago.lock, populate core tables
2. **M2: Call Graph** - Parse corefn.json, extract function calls
3. **M3: Git Integration** - Extract commit history, compute metrics
4. **M4: Incremental** - Detect changes, partial reload
5. **M5: Performance** - Parallel parsing, batch inserts, <5s target
6. **M6: API Routes** - HTTPurple route extraction (future)

Open Questions

1. Should we support loading from tarball (for CI without git)?
2. Should we embed DuckDB or require external installation?
3. How to handle private/unpublished packages in spago.lock?
4. Should orphan detection be in loader or separate tool?

This spec is intended to be complete enough for implementation by someone unfamiliar with the codebase. Questions welcome.