nornir-catalog

version 0.1.3

generated 2026-06-09

---

rendered by **nornir** · typst engine

Contents

1 nornir-catalog 3
1.1 nornir vs Nessie vs Polaris 3
1.1.1 Catalog read — `table_exists` (ops/sec, log scale) 4
1.1.2 Data write — streaming ingest (`data_pipe` , rows/sec) 4
1.1.3 Concurrent commit — the group-commit lever (`commit_burst` , commits/sec) 4
1.1.4 Analytical read — TPC-H SF1, 22 queries (`tpch_cmp` , total seconds) 5
1.2 Status 5
1.3 Read & write paths 5
1.4 Benchmarks 7
1.4.1 Catalog read RPC — `table_exists` (storage-free, runs on every backend) 7
1.4.2 Embedded read fast paths (nornir-only) 7
1.4.3 Data plane — single-writer / many-processor ingest 8
1.4.4 Commit-bursty — the catalog lever (group-commit) 8
1.4.5 TPC-H — analytical SQL (all 22 queries via DataFusion) 8
1.4.6 TPC-H large warehouse (opt-in, big iron) 9
1.4.7 TPC-H across catalogs × storage (nornir vs Nessie vs Polaris) 9
1.4.8 OSM GeoParquet ingest (data plane) 10
1.4.9 ZSTD decode — `zstd-sys-rs` vs the `zstd` crate 10
1.5 Quickstart 10
1.6 Multi-table atomic commits 11
1.7 Storage layout 11
1.8 Known shortcomings (0.1.2) 11
1.9 License 12

nornir-catalog

Pure-Rust, embedded Apache Iceberg catalog backed by redb.

crates.io v0.1.2

docs passing

license Apache-2.0

An `iceberg::Catalog` with cross-table transactions, implemented as an immutable, lock-free static search tree (Ragnar `STree64`) out front — continuously regenerated from a mutable redb backend, the ACID source of truth. The front is massively parallel (every core reads it lock-free); redb sits behind it. Keyed by immutable, content-addressed identifiers, the front never serves a stale value — only falls back to redb on a miss.

- **Orders of magnitude faster than a REST catalog.** It lives in-process — no network hop, no JVM, no double-JSON round-trip.
  - `table_exists` RPC — **2.2M ops/s embedded**, **492× Nessie** · **677× Polaris** (0.31 µs p50 vs 220 µs / 287 µs).
  - Warm metadata reads — **100–1000× lower latency** than a JVM REST catalog.
  - Even REST-fronted (axum) — **4–7× faster** than either JVM catalog.

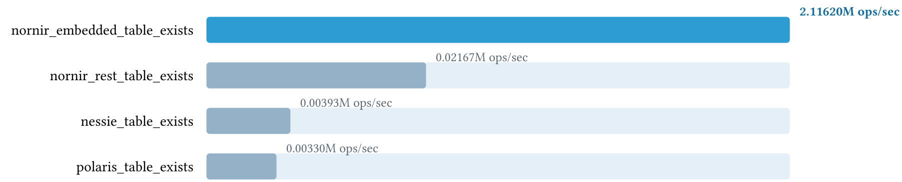
See Benchmarks.

- **ACID.** Every catalog mutation is one redb `WriteTransaction`. Crashes leave the catalog consistent.
- **Atomic multi-table commits.** See `RedbCatalog::atomic_release` — flip the pointers of N Iceberg tables in a single redb transaction.

nornir vs Nessie vs Polaris

The mashup: a bar chart **and** a head-to-head table per capability, **auto- generated by `nornir docs render` from the latest `nornir bench run`** (winner highlighted). Every nornir storage variant shows — local NVMe (`_nvme`), tmpfs RAM (`_ram`), and S3 (`_s3`) — alongside Nessie (S3) and Polaris (file); the storage is right in each bar/row label. `table_exists` is the storage-free catalog RPC. Full per-variant breakdowns are in Benchmarks.

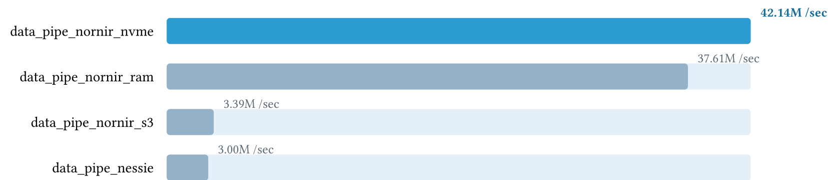
Catalog read — `table_exists` (ops/sec, log scale)



v0.4.0 · oden · 32 cores · 2026-06-08

workload	ops_sec	p50_us	p90_us	p99_us	mean_us	min_us
nornir_embedded_table_exists	**2.11620M**	**0.32**	**0.51**	**0.73**	**0.37**	**0.28**
nessie_table_exists	0.00393M	227.05	297.56	617.80	253.97	162.22
polaris_table_exists	0.00330M	288.28	325.42	470.99	302.56	236.02

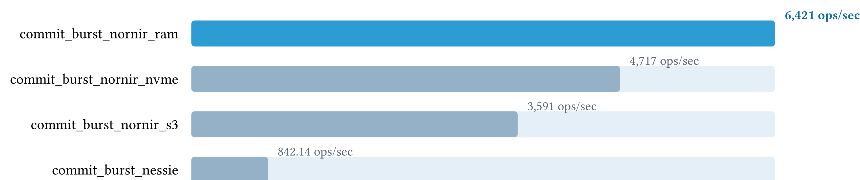
Data write — streaming ingest (`data_pipe` , rows/sec)



v0.4.0 · oden · 32 cores · 2026-06-08

workload	rows_per_sec
data_pipe_nornir_nvme	**42.14M**
data_pipe_nornir_ram	37.61M
data_pipe_nornir_s3	3.39M
data_pipe_nessie	3.00M

Concurrent commit — the group-commit lever (`commit_burst` , commits/sec)

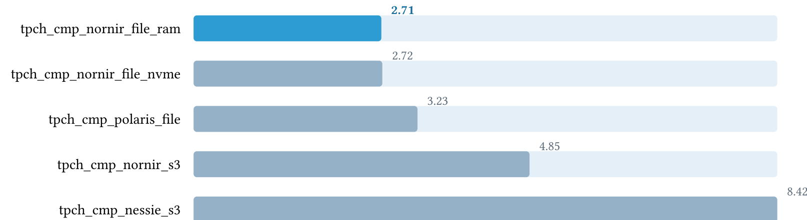


v0.4.0 · oden · 32 cores · 2026-06-08

workload	commits_per_sec	p50_us	p90_us	p99_us	mean_us	min_us
commit_burst_nornir_nvme	4,717	6,847	7,808	9,562	6,777	494.98
commit_burst_nornir_ram	**6,421**	**4,869**	**6,252**	**7,628**	**4,975**	**219.68**

commit_burst_nornir_s3	3,591	8,880	10,225	11,514	8,870	2,548
commit_burst_nessie	842.14	6,705	59,312	385,412	29,921	2,492

Analytical read — TPC-H SF1, 22 queries (`tpch_cmp` , total seconds)



v0.4.0 · oden · 32 cores · 2026-06-08

workload	total_s	query_s	build_s	slowest_query_s
tpch_cmp_nornir_file_nvme	2.72	**2.29**	0.44	**0.32**
tpch_cmp_nornir_file_ram	**2.71**	2.30	**0.41**	0.32
tpch_cmp_nornir_s3	4.85	4.17	0.68	0.49
tpch_cmp_nessie_s3	8.42	5.76	2.66	0.57
tpch_cmp_polaris_file	3.23	2.49	0.74	0.33

v0.4.0 · oden · 32 cores · 2026-06-08

Most dramatic query — `q03` : nornir is 1.8× faster (144.8 ms vs 253.7 ms).

query	tpch_cmp_nornir_s3	tpch_cmp_nessie_s3	speedup
`q03`	**144.8 ms**	253.7 ms	**1.8×**

Status

`0.1.2` . The Iceberg `Catalog` trait is fully implemented and tested against `iceberg = "0.9.1"` . The one gap is schema evolution: the `Transaction` actions that drive it aren't exposed in upstream iceberg-rust's public API yet; once they land (0.10) they work here unchanged. See Known shortcomings.

Read & write paths

Every call resolves left-to-right through these layers, falling through only on a miss. **Latest-state** reads ride the L1 pointer mirror; **time-travel** (snapshot-id) reads ride the Ragnar `STree64` index. redb is always the source of truth behind them — the in-memory layers are keyed by immutable, content-addressed identifiers, so they can only be *evicted*, never *stale*.

Read paths every call routes left → right, falling through to **redb** only on a miss

hit (cache serves) miss → redb

`table_exists(id)` → **L1 pointer mirror** `ArcSwap<foldhash map>` lock-free · every core → `bool`

`resolve_metadata(id)` → **L0 metadata cache** parsed · byte-bounded · single-flight → `Arc<TableMetadata>`

`load_table(id)` → **L1.5 handle cache** built Table · ~100 ns clone → `Table`

key → metadata_location content-addressed · never stale

miss miss → build

`load_table_at(id, sid)` time-travel — keyed by snapshot_id → **Ragnar STree64** static snapshot-id index

miss

FileIO read + parse `…/uuid.metadata.json`

redb ACID source of truth — durable, transactional `tables · commits · namespaces · namespace_props · meta`

layer	key ⟶ value	type	populated
L1 pointer mirror	`table_key → metadata_location`	`ArcSwap<imbl::HashMap<String, Arc<str>, foldhash>>`	full scan at open, then write-through
L1.5 handle cache	`metadata_location → Table`	`moka::Cache<String, Table>` (capacity-bounded)	on `load_table` miss
L0 metadata cache	`metadata_location → TableMetadata`	`moka::Cache<String, Arc<TableMetadata>>` (byte-bounded, single-flight)	on metadata miss
Ragnar static index	`snapshot_id → (table_key, metadata_location)`	`ArcSwapOption<STree64>`	warm-built at open, rebuilt by bg compactor (≥1024 commits)
redb source of truth	`tables` · `commits` · `namespaces` · `namespace_props` · `meta`	`Mutex<redb::Database>` (ACID)	every committed write

The same routing in full — including the time-travel, write, and redb-direct paths the diagram leaves out:

```
LATEST READS
  table_exists(id) ────────► L1 mirror ──hit──► true
                                  └─miss─► redb `tables` ───────► bool

  resolve_metadata(id) ───► L1 mirror ──► loc ──► L0 cache ──hit──► Arc<TableMetadata>
       (skips Table::build)       └─miss─► redb           └─miss─► FileIO read + parse JSON

  load_table(id) ──────────► L1 mirror ──► loc ──► L1.5 cache ──hit──► Table  (~100 ns
clone)
                                  └─miss─► redb            └─miss─► L0 cache ──►
```

```
Table::build() + insert

TIME-TRAVEL READS  (by snapshot_id: i64)
  load_table_at / resolve_metadata_at(id, sid)
        ──► Ragnar STree64 ──hit──► loc ──► L1.5 / L0 ──► Table / Arc<TableMetadata>
                 └─miss (sid above cutoff)─► redb `commits` live tail ──► loc ──► …

  resolve_many(id, [sid; N])
        ──► Ragnar batch probe (1 pipelined pass) ──► redb `commits` (1 txn, misses
only) ──► L0 per loc

WRITES  (create · register · update · drop · rename)
  …──► FileIO write …/<uuid>.metadata.json ──► redb WriteTransaction { tables CAS ·
commits log · meta++ }
        ──commit──► L1 mirror write-through (insert / remove) ──► maybe rebuild Ragnar
(background)

  update_table ──► group-commit: N concurrent commits coalesce into 1 redb txn / 1 fsync
← commit-burst lever

REDB-DIRECT  (no cache layer)
  list_namespaces · get_namespace · namespace_exists · list_tables ──► redb read txn
(range scan)
  create / update / drop_namespace ──────────────────────────────► redb
WriteTransaction
```

Benchmarks

All numbers below are **auto-filled by `nornir docs render`** from the latest `nornir bench run` (machine/cores/version in each header). Don't hand-edit inside the generated regions — re-run the bench instead. Reproduce with the harness in [`bench/`] (`containers/rustfs_up.sh` + `containers/nessie_up.sh`, then `nornir bench run nornir-catalog` from `workspace_nornir-catalog/`).

Catalog read RPC — `table_exists` (storage-free, runs on every backend)

`table_exists` is a pure catalog RPC (no object storage), the apples-to-apples comparable; Nessie/Polaris go through the same `iceberg::Catalog` REST client.

v0.4.0 · oden · 32 cores · 2026-06-08

workload	ops_sec	p50_us	p90_us	p99_us	mean_us	min_us
nessie_table_exists	0.00393M	227.05	297.56	617.80	253.97	162.22
nornir_embedded_table_exists	**2.11620M**	**0.32**	**0.51**	**0.73**	**0.37**	**0.28**
nornir_rest_table_exists	0.02167M	44.39	51.11	69.97	45.91	32.31
polaris_table_exists	0.00330M	288.28	325.42	470.99	302.56	236.02

Embedded read fast paths (nornir-only)

`resolve_metadata` skips `Table::build()` (the lock-free L1+L0 floor); `load_table` is the full `Catalog` trait path after the L1.5 handle cache.

v0.4.0 · oden · 32 cores · 2026-06-08

workload	ops_sec	p50_us	p90_us	p99_us	mean_us	min_us
nornir_embedded_load_table	127,316	1.74	29.27	39.24	7.71	0.72
nornir_embedded_resolve_metadata	**220,756**	**1.24**	**15.44**	**22.35**	**4.40**	**0.52**

Data plane — single-writer / many-processor ingest

All cores encode Parquet; one writer streams + commits (`data-pipe`). Backend- agnostic and storage-bound (shared client-side Parquet + S3), so rows/s converge across backends — this lifts all boats, it is **not** a catalog lever. Nessie runs the *same* shared RustFS S3 warehouse. Scan verifies the round-trip. The nornir local-FS row runs **two file destinations** — `_nvme` (PCIe-4.0 NVMe) and `_ram` (`/dev/shm` tmpfs) — to expose the pure storage floor.

v0.4.0 · oden · 32 cores · 2026-06-08

workload	rows_per_sec	files
data_pipe_nessie	3.00M	10
data_pipe_nornir_nvme	**42.14M**	10
data_pipe_nornir_ram	37.61M	10
data_pipe_nornir_s3	3.39M	10

Commit-bursty — the catalog lever (group-commit)

Many small concurrent metadata commits — where nornir coalesces commits into one redb txn/fsync. This is where the embedded catalog genuinely pulls ahead of a REST/JVM catalog (throughput **and** tail latency).

v0.4.0 · oden · 32 cores · 2026-06-08

workload	commits_per_sec	p50_us	p90_us	p99_us	mean_us	min_us
commit_burst_nessie	842.14	6,705	59,312	385,412	29,921	2,492
commit_burst_nornir_nvme	4,717	6,847	7,808	9,562	6,777	494.98
commit_burst_nornir_ram	**6,421**	**4,869**	**6,252**	**7,628**	**4,975**	**219.68**
commit_burst_nornir_s3	3,591	8,880	10,225	11,514	8,870	2,548

TPC-H — analytical SQL (all 22 queries via DataFusion)

The full 8-table TPC-H schema loaded into Iceberg tables, queried through DataFusion (`iceberg-datafusion`) — per-query latency for all 22 canonical queries over the catalog's scan/manifest path. Authentic `tpchgen` data; `TPCH_SF` sets the scale (small by default). Answer values aren't validated here (that needs SF=1) — this measures plan + scan + execute latency end-to-end.

v0.4.0 · oden · 32 cores · 2026-06-08

workload	max_us	mean_us	min_us	ops_sec	p50_us	p90_us	p999_us	p99_us	rows
tpch_q01	19,573	18,098	16,577	55.25	17,998	19,573	19,573	19,573	4
tpch_q02	42,073	40,558	39,324	24.66	39,886	42,073	42,073	42,073	4
tpch_q03	18,694	17,992	17,559	55.58	17,890	18,694	18,694	18,694	138
tpch_q04	13,872	13,167	12,427	75.95	12,974	13,872	13,872	13,872	5
tpch_q05	47,532	43,174	38,878	23.16	44,440	47,532	47,532	47,532	5
tpch_q06	3,689	3,445	3,190	290.29	3,442	3,689	3,689	3,689	1
tpch_q07	37,090	36,249	35,047	27.59	36,407	37,090	37,090	37,090	4
tpch_q08	54,632	51,634	49,143	19.37	50,568	54,632	54,632	54,632	2
tpch_q09	55,766	50,904	42,198	19.64	51,799	55,766	55,766	55,766	173
tpch_q10	32,739	30,300	29,548	33.00	29,740	32,739	32,739	32,739	399
tpch_q11	14,334	13,874	13,508	72.08	13,861	14,334	14,334	14,334	359
tpch_q12	18,139	17,611	16,977	56.78	17,492	18,139	18,139	18,139	2
tpch_q13	13,880	13,396	12,977	74.65	13,415	13,880	13,880	13,880	33
tpch_q14	6,980	6,506	6,048	153.69	6,591	6,980	6,980	6,980	1
tpch_q15	15,678	15,265	14,897	65.51	15,388	15,678	15,678	15,678	1

tpch_q16	22,129	21,697	21,209	46.09	21,771	22,129	22,129	22,129	296
tpch_q17	24,126	23,728	23,063	42.14	23,943	24,126	24,126	24,126	1
tpch_q18	34,074	32,598	30,110	30.68	32,847	34,074	34,074	34,074	2
tpch_q19	14,123	12,106	11,491	82.60	11,670	14,123	14,123	14,123	1
tpch_q20	25,943	25,568	25,293	39.11	25,512	25,943	25,943	25,943	1
tpch_q21	43,672	40,676	39,096	24.58	40,499	43,672	43,672	43,672	1
tpch_q22	12,512	12,124	11,667	82.48	12,226	12,512	12,512	12,512	7

TPC-H large warehouse (opt-in, big iron)

A full-scale run: ingest the 8-table warehouse at a large scale factor across all cores (partitioned parallel ingest), then run all 22 queries over it — reporting build vs query time, ingest rate, and the slowest query. Runs by default at a moderate scale; `TPCH_WAREHOUSE_SF` tunes it (default 10 ≈ 60 M lineitem rows / a few min; 100-200 for a 10-60 min big-iron workout), `TPCH_WAREHOUSE=0` disables it, `TPCH_WAREHOUSE_PARTS` sets ingest partitions (default = all cores).

v0.4.0 · oden · 32 cores · 2026-06-08

workload	build_s	ingest_rows	ingest_rows_per_s	parts	query_s	result_rows	scale_factor	slowest_query	slowest_query_s	total_s
tpch_warehouse	3.19	86.59M	27.12M	32	19.43	534,307	10	9	**2.81**	22.62

TPC-H across catalogs × storage (nornir vs Nessie vs Polaris)

The same SF warehouse built and all 22 queries run over each **(catalog × storage)** combination through DataFusion — the analytical-SQL path compared apples-to-apples (not just the `table_exists` RPC). The matrix covers every combo that's possible: nornir on **file-NVMe, file-RAM, and S3**; Nessie **S3-only** (it rejects a local file warehouse); Polaris on **file** (its S3 credential vending 301s against the RustFS endpoint — see `bench/src/containers.rs`). The REST targets need their container + the shared RustFS S3 warehouse (`bench-containers up all`), else they skip. `TPCH_COMPARE_SF` sets the scale (default 1.0 ≈ 8.66M rows).

> **Parallel scan patch:** stock `iceberg-datafusion` 0.9.1 scans a table as a single DataFusion partition (one serial Parquet-decode stream), so on a 32-core box only 1 core works and queries are 7-8× slower than they should be. The bench vendors a patched `iceberg-datafusion` (`bench/vendor/`, relative-path `[patch]`) that exposes **one partition per file**, so DataFusion decodes files across all cores. SF100 FILE query dropped 1660s ⟶ 197s (8.4×). These numbers reflect that patch.

At SF100 (866M rows) both S3-backed paths (`nornir_s3`, `nessie_s3`) fail with `hyper: connection closed before message completed` finishing the Parquet writer — an S3 write-robustness limit under sustained heavy PUT load (rust-s3 / RustFS), not a catalog issue; the FILE-backed paths complete cleanly.

v0.4.0 · oden · 32 cores · 2026-06-08

workload	build_s	q01_s	q02_s	q03_s	q04_s	q05_s	q06_s	q07_s	q08_s	q09_s	q10_s	q11_s	q12_s	q13_s	q14_s	q15_s	q16_s	q17_s	q18_s	q19_s	q20_s	q21_s	q22_s	query_s	rows	scale_factor	slowest_query	slowest_query_s	total_s
tpch_cmp_nessie_s3	2.66	0.14	0.26	0.25	0.18	0.41	0.10	0.40	0.49	0.57	0.24	0.14	0.18	0.18	0.16	0.21	0.10	0.38	0.38	0.18	0.20	0.48	0.12	5.76	8.66M	1	9	0.57	8.42
tpch_cmp_nornir_file_nvme	0.44	**0.06**	0.10	0.06	0.03	0.20	0.01	0.15	0.25	**0.32**	0.08	0.07	0.07	**0.05**	**0.02**	**0.04**	**0.04**	**0.18**	**0.22**	**0.04**	**0.06**	**0.21**	**0.03**	**2.29**	8.66M	1	9	**0.32**	2.72
tpch_cmp_nornir_file_ram	**0.41**	0.06	**0.09**	**0.06**	**0.03**	**0.19**	**0.01**	**0.13**	**0.23**	0.32	**0.08**	**0.07**	**0.06**	0.05	0.02	0.04	0.05	0.20	0.24	0.04	0.06	0.21	0.03	2.30	8.66M	1	9	0.32	**2.71**
tpch_cmp_nornir_s3	0.68	0.10	0.16	0.14	0.10	0.30	0.07	0.25	0.39	0.49	0.19	0.10	0.11	0.13	0.12	0.15	0.08	0.28	0.30	0.12	0.13	0.36	0.09	4.17	8.66M	1	9	0.49	4.85
tpch_cmp_polaris_file	0.74	0.06	0.11	0.07	0.04	0.21	0.03	0.15	0.26	0.33	0.09	0.08	0.07	0.06	0.03	0.05	0.06	0.19	0.23	0.05	0.07	0.21	0.04	2.49	8.66M	1	9	0.33	3.23

OSM GeoParquet ingest (data plane)

End-to-end ingest of **authentic OSM data**: a `nodes.parquet` produced by the `katana-osm` / `osm2geoparquet` converter (OSM `.osm` / `.bz2` / `.pbf` → GeoParquet) is read, its schema mapped to Iceberg, and ingested into a fresh `RedbCatalog` table via the single-writer / many-processor pipeline, then scanned back to verify. NVMe vs RAM destinations isolate storage cost. Point `OSM_GEOPARQUET` at the file (`OSM_MAX_ROWS` caps rows); the `osm_ingest_*` benchers record a failure if it's unset, like the container-backed targets.

v0.4.0 · oden · 32 cores · 2026-06-08

workload	commits	encode_workers	files	mb_per_sec	read_rows_per_sec	read_threads	rows	rows_per_sec	windows	write_rows_per_sec
osm_ingest_nornir_nvme	8	32	64	**912.24**	42.47M	32	50.33M	15.68M	1	**24.85M**
osm_ingest_nornir_ram	8	32	64	809.81	**67.52M**	32	50.33M	**16.63M**	1	22.06M

ZSTD decode — `zstd-sys-rs` vs the `zstd` crate

Aggregate (all-core) decompression throughput on a real OSM corpus (WKB geometry bytes from the converted Europe extract, parquet-page-sized frames), decoding the same frames two ways: the stock `zstd` crate (one-shot, allocates per frame) and `zstd-sys-rs` 's zero-copy path (a reused `ZSTD_DCtx` + reused output buffer). Both link the same static libzstd 1.5.7, so this validates the bindings and the zero-copy API on real data rather than chasing a codec difference. Needs `OSM_GEOPARQUET` (`ZSTD_CHUNK_KB` sets frame size).

v0.4.0 · oden · 32 cores · 2026-06-08

workload	decompressed_mb	frames	max_us	mb_per_sec	mean_us	min_us	ops_sec	p50_us	p90_us	p999_us	p99_us	threads
zstd_decode_crate	1,057	16,160	57,419	22,598	46,773	39,509	21.38	42,761	57,211	57,419	57,419	32
zstd_decode_sys_zerocopy	1,057	16,160	**56,227**	**24,807**	**42,608**	**39,093**	**23.47**	**42,244**	**43,125**	**56,227**	**56,227**	32

Quickstart

```
,no_run
use std::collections::HashMap;
use std::sync::Arc;

use iceberg::io::LocalFsStorageFactory;
use iceberg::spec::{NestedField, PrimitiveType, Schema, Type};
use iceberg::{Catalog, CatalogBuilder, NamespaceIdent, TableCreation};
use nornir_catalog::RedbCatalogBuilder;

# async fn run() -> anyhow::Result<()> {
let catalog = RedbCatalogBuilder::default()
    .db_path("/var/lib/nornir/catalog.redb")
    .warehouse_location("file:///var/lib/nornir/warehouse")
    .with_storage_factory(Arc::new(LocalFsStorageFactory))
    .load("nornir", HashMap::new())
    .await?;

let ns = NamespaceIdent::new("bench".to_string());
catalog.create_namespace(&ns, HashMap::new()).await?;

let schema = Schema::builder()
    .with_schema_id(0)
    .with_fields(vec![
        NestedField::required(1, "run_id",
Type::Primitive(PrimitiveType::String)).into(),
```

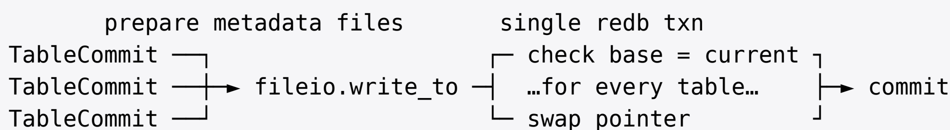
```
        NestedField::required(2, "ops_sec",
Type::Primitive(PrimitiveType::Double)).into(),
    ])
    .build()?;

let table = catalog
    .create_table(
        &ns,
        TableCreation::builder()
            .name("bench_runs".to_string())
            .schema(schema)
            .build(),
    )
    .await?;
# Ok(()) }
```

Multi-table atomic commits

Iceberg's per-table `Transaction::commit` only gives single-table atomicity. When a release logically spans several tables (e.g. publishing `bench_runs`, `dep_graph`, and `components` from a single CI run), readers can otherwise observe a half-published state.

redb gives every write transaction global atomicity across all of its tables. `RedbCatalog::atomic_release(commits)` exploits this: each `TableCommit` is staged (metadata blobs written to object storage), then a single redb write transaction performs an optimistic-concurrency check on every base pointer and either flips them all or none.



If iceberg-rust later exposes `TableCommit` construction publicly, the ergonomic story improves. Until then, `atomic_release` is most useful when you build commits via crates that have direct access to internal helpers (e.g. a nornir release driver).

Storage layout

Inside the redb file:

keyspace	key	value
`namespaces`	`<catalog>\x1f<ns_path>`	empty marker
`namespace_props`	`<catalog>\x1f<ns_path>\x1f<prop>`	property value
`tables`	`<catalog>\x1f<ns_path>\x1f<table>`	metadata location

Where `<ns_path>` is the dot-joined namespace ident (`"a.b.c"`). One redb file may host multiple logical catalogs by reusing different `<catalog>` prefixes; in practice you'll point each catalog at its own file.

Known shortcomings (0.1.2)

- **Schema evolution is not callable yet.** `iceberg-rust 0.9.1` seals the `TransactionAction` trait (`pub(crate)`), so downstream code can't construct `AddSchema` / `SetCurrentSchema` . The on-disk metadata already supports schema evolution — only the writer API is sealed — so it works here unchanged once upstream exposes it (0.10). These Transaction actions **do** work today: `update_table_properties` , `fast_append` , `replace_sort_order` , `update_location` , `update_statistics` , `upgrade_format_version` .

- **No orphan-file garbage collection.** A commit that fails *after* writing metadata blobs to object storage but *before* the redb pointer-swap commits leaves those blobs behind — harmless but unreferenced. Reap them with Iceberg's standard orphan-file procedures; there is no built-in sweeper.

License

Licensed under Apache License, Version 2.0, matching the upstream Apache Iceberg project.