SQLite as Drop-in Backend for InvenioRDM

SQLite as Drop-in Backend for InvenioRDM: Feasibility Notes

Executive Summary

Replacing PostgreSQL and OpenSearch with SQLite in InvenioRDM is technically feasible for Starter/small-institution use cases. The commonmeta-rs SQLite schema is a strong foundation: it already covers works, people, organizations, relations, and FTS5 search at production scale (billions of rows). The main effort is on the Python/InvenioRDM side — adapting invenio-db and invenio-search to target SQLite backends — and is realistically a 4–6 month project.

Current Architecture: InvenioRDM

Data stores

Layer	Technology	Role
Primary DB	PostgreSQL	Records, files metadata, users, communities, requests, access control
Search	OpenSearch / Elasticsearch	Full-text search, facets, aggregations
Task broker	Redis / RabbitMQ	Celery task queue

Layer	Technology	Role
Object store	S3 / local filesystem	Binary file storage

Key Python packages

- **invenio-db** — SQLAlchemy ORM, Alembic migrations; all database models live here
- **invenio-search** — thin wrapper around `opensearch-py`; indexing, query DSL
- **invenio-records** — base record model (JSON document stored in JSONB column)
- **invenio-rdm-records** — RDM-specific record types, deposit, access control
- **invenio-communities** — communities, membership, moderation
- **invenio-requests** — workflow engine (review, accept, decline)
- **invenio-files-rest** — file locations, buckets, objects (separate from S3 storage)

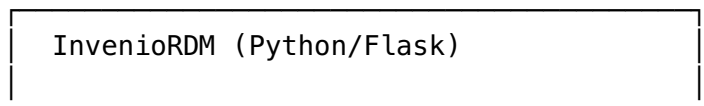
What PostgreSQL is actually used for

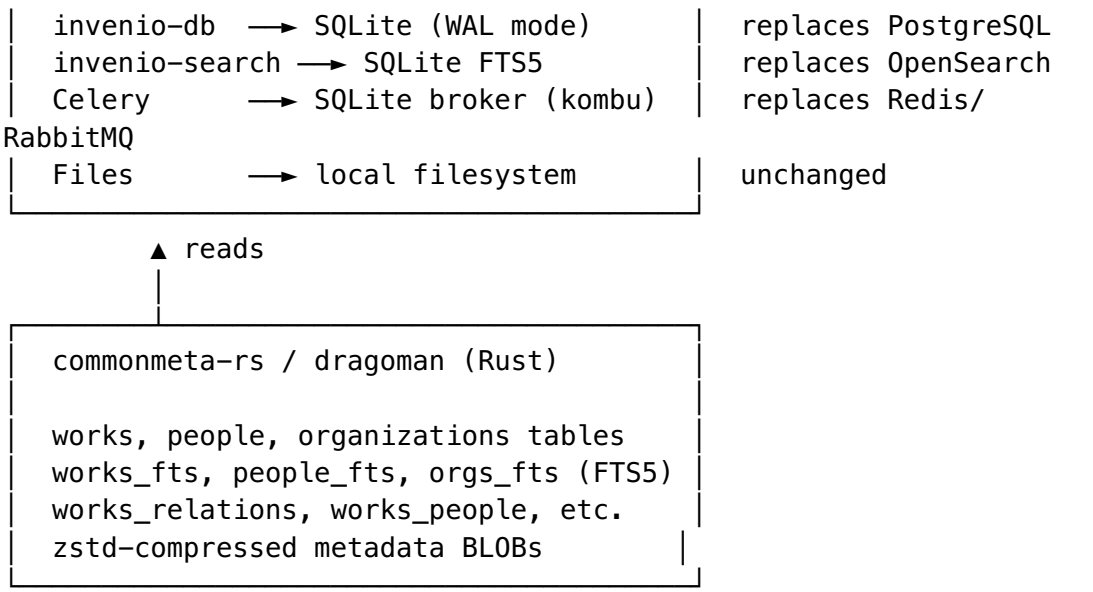
1. **Record versions** — `rdm_records`, `rdm_drafts`, parent records with JSONB `json` column
2. **File metadata** — `files_files`, `files_buckets`, `files_objecttags`
3. **User/auth** — `accounts_user`, `oauth2server_token`, `oauthclient_remotetoken`
4. **Communities** — `communities_community`, membership, featured
5. **Requests** — `request_metadata`, `request_events`
6. **Access control** — `access_systemroles`, `access_grants`
7. **Vocabularies** — subjects, resource types, languages, licenses (small tables)

What OpenSearch is used for

1. Record search (title, description, subjects, creators, dates)
2. Aggregations/facets (resource type, access status, subject, date range)
3. Community search
4. Community record search with permission filtering
5. Draft search (deposit UI)
6. Suggestions (autocomplete)

Proposed Architecture: SQLite-based





What the commonmeta-rs schema already provides

The SQLite schema in commonmeta-rs (`src/formats/commonmeta.rs`) is mature and covers:

- `works` — id (DOI/URL), type, title, subjects, language, date fields, provider, pmid, pmcid, openalex, arxiv, `metadata` BLOB (zstd-compressed commonmeta JSON)
- `people` — ORCID profiles with affiliations, external identifiers
- `organizations` — ROR data with location, types, relations
- Junction tables: `works_people`, `works_organizations`, `works_relations` (with relation type)
- FTS5 virtual tables: `works_fts`, `people_fts`, `organizations_fts` (content tables, unicode61 tokenizer)
- Schema migrations via `PRAGMA user_version` (v1–v9+)
- WAL mode + `mmap_size = 512 GB` for read performance
- Partial indexes (e.g. `works_pmid WHERE pmid != ''`)
- Settings table for versioned key-value metadata

This maps onto most of what InvenioRDM stores, with these gaps: - No user/auth tables (not in scope for the metadata layer) - No file metadata tables (separate concern) - No community/request tables (InvenioRDM-specific workflow)

SQLite 3.45 JSONB: What It Is and What It Is Not

What changed in 3.45 (January 2024)

SQLite 3.45 introduced a **binary JSONB storage format** — a compact, pre-parsed representation of JSON stored as a BLOB rather than UTF-8 text. The naming is coincidental with PostgreSQL's `JSONB`; the two are unrelated.

New surface area:

- `jsonb(x)` — converts text JSON → binary JSONB blob
- `jsonb_extract(blob, path)`, `jsonb_object()`, `jsonb_array()` — JSONB-native variants of existing functions
- All existing `json_*` functions now accept JSONB blobs as input (transparent upgrade)
- The `->` and `->>` operators (SQLite 3.38+) work on both text JSON and JSONB

What this gives you:

- ~10–15 % faster `json_extract()` on large documents (parse cost eliminated)
- Smaller on-disk representation for JSON-heavy workloads
- No schema changes required — store as `BLOB` instead of `TEXT`

What SQLite 3.45 JSONB does NOT provide (vs PostgreSQL JSONB)

Feature	PostgreSQL JSONB	SQLite 3.45 JSONB
GIN index on JSON keys/values	✓	✗
`@>` containment operator	✓	✗
Index on arbitrary JSON path	✓ (via expression index)	✗ directly; use generated column
`->` / `->>` operators	✓	✓ (since 3.38)
`jsonb_path_query` / JSONPath	✓	partial (3.38 JSONPath subset)
Partial index on JSON predicate	✓	✓ (on generated column only)
SQLAlchemy `postgresql.JSONB` type	works	does not map (falls back to TEXT)

The absence of GIN indexes is the most consequential gap. PostgreSQL can answer `WHERE json @> '{"metadata": {"resource_type": {"id": "dataset"}}}'` using a GIN index in O(log n). SQLite must scan every row and call `json_extract()` — O(n).

InvenioRDM-specific implications

`invenio-records` defines the record's `json` column as:

```
# invenio_records/models.py
json = db.Column(
    db.JSON().with_variant(postgresql.JSONB(none_as_null=True),
        "postgresql"),
    default=lambda: dict(),
    nullable=True,
)
```

On SQLite this falls back to `sa.JSON` which stores as TEXT. That is fine for storage but means any ORM filter like `.filter(Record.json["metadata"]["resource_type"]["id"] == "dataset")` becomes a full table scan.

Mitigation already in commonmeta-rs: pre-extract frequently-queried fields as indexed first-class columns (`type`, `language`, `date_published`). InvenioRDM would need the same for `resource_type_id`, `access_status`, `publisher`, `year`. Generated columns in SQLite 3.31+ make this transparent:

```
ALTER TABLE rdm_records_metadata ADD COLUMN resource_type TEXT
    GENERATED ALWAYS AS (json_extract(json,
        '$.metadata.resource_type.id')) STORED;
CREATE INDEX idx_rdm_resource_type ON
        rdm_records_metadata(resource_type);
```

Migration path: the SQLAlchemy event system can intercept column definitions at `create_engine` time and substitute generated columns for models that use JSONB field filters. This keeps the Python model code unchanged while the DDL adapts to the dialect.

SQLAlchemy type mapping

Replace in `invenio-db-sqlite` (fork):

```
from sqlalchemy.dialects import postgresql
import sqlalchemy as sa

# Before (PostgreSQL-only):
json = sa.Column(postgresql.JSONB)

# After (dialect-portable):
json = sa.Column(
```

```
    sa.JSON().with_variant(postgresql.JSONB(none_as_null=True),
        "postgresql")
)
```

`sa.JSON` maps to TEXT on SQLite (stores as UTF-8) and to `jsonb` on PostgreSQL. The Python interface is identical. The 3.45 binary format can be adopted later as an optimisation by storing with `jsonb()` at insert time; it is not required for correctness.

Write Concurrency in SQLite

The WAL model

SQLite WAL (Write-Ahead Log) mode — already used in commonmeta-rs — provides:

- **Readers never block writers; writers never block readers**
- **Only one writer at a time** — additional writers queue behind the first
- **Readers see a consistent snapshot** from before the write started
- **Automatic checkpoint** at 1000 WAL pages (configurable), briefly pauses writers

This is a fundamentally different model from PostgreSQL's MVCC, which allows concurrent writers on different rows. In SQLite, concurrent writes are serialized at the database level.

Write patterns in InvenioRDM

Operation	Frequency	Duration	Risk
Record deposit / publish	Low (user action)	< 50 ms	Low
Draft auto-save	Medium	< 10 ms	Medium
File metadata insert	Medium	< 10 ms	Low
Celery bulk reindex	Low	seconds	High (holds write lock)
Embargo lifting	Low (scheduled)	seconds	High
Access log writes	High	< 1 ms	Medium

The dangerous case is Celery tasks that hold a write transaction open for seconds. During that window, all Flask workers trying to write will block until `busy_timeout` expires.

Configuration

Apply these PRAGMAs on every new connection (SQLAlchemy `connect` event):

```
from sqlalchemy import event, text
from sqlalchemy.engine import Engine

@event.listens_for(Engine, "connect")
def set_sqlite_pragmas(dbapi_conn, _record):
    dbapi_conn.execute("PRAGMA journal_mode=WAL")
    dbapi_conn.execute("PRAGMA synchronous=NORMAL")   # safe with
        WAL; faster than FULL
    dbapi_conn.execute("PRAGMA busy_timeout=5000")    # retry
        writes for up to 5 s
    dbapi_conn.execute("PRAGMA wal_autocheckpoint=1000")
    dbapi_conn.execute("PRAGMA mmap_size=549755813888")  # 512 GB
        virtual mmap
    dbapi_conn.execute("PRAGMA cache_size=-32768")
        # 32 MB page cache per connection
    dbapi_conn.execute("PRAGMA temp_store=MEMORY")
```

`busy_timeout=5000` means SQLite retries the lock acquisition for up to 5 seconds before raising `OperationalError: database is locked`. For deposit workflows 5 s is generous; reduce to 2 s if you want faster failure under load.

`BEGIN IMMEDIATE` for write transactions

SQLite has three transaction modes:

- `DEFERRED` (default): acquires a shared lock on first read; upgrades to write lock on first write — can fail with `SQLITE_BUSY` at the upgrade point
- `IMMEDIATE`: acquires a reserved lock at `BEGIN` — fails fast if another writer is active
- `EXCLUSIVE`: acquires an exclusive lock — blocks all readers during the transaction

Use `IMMEDIATE` for write transactions in InvenioRDM to fail early rather than mid-transaction:

```
engine = create_engine(
    "sqlite:///invenio.db",
    connect_args={"check_same_thread": False},
    isolation_level="IMMEDIATE",   # all write transactions use
        BEGIN IMMEDIATE
)
```

This prevents the worst case: a transaction that has done 200 ms of work, then fails to upgrade its lock and must be retried from scratch.

Connection pool strategy

SQLite does not support true connection-per-thread multiplexing. The right pool depends on deployment:

Deployment	Pool	Rationale
Single-process (dev, demo)	`StaticPool`	one connection, no contention
Gunicorn sync workers (2–4)	`NullPool` + `check_same_thread=False`	each request gets a fresh connection; WAL handles concurrent reads
Gunicorn async (gevent/asyncio)	`NullPool` + thread-local	SQLite is not thread-safe across event loops
Celery worker	`StaticPool` per worker process	one connection per process, serialized in-process

```
# dev / demo
engine = create_engine("sqlite:///invenio.db",
        poolclass=StaticPool,
                        connect_args={"check_same_thread": False})

# production (2–4 Gunicorn sync workers)
engine = create_engine("sqlite:///invenio.db", poolclass=NullPool,
                        connect_args={"check_same_thread": False},
                        isolation_level="IMMEDIATE")
```

Celery long-write mitigation

The two high-risk Celery patterns — bulk reindex and embargo lifting — should be broken into small committed batches rather than single large transactions:

```
# Instead of: one transaction over 10 000 records
# Do: commit every 500 records
BATCH = 500
for i in range(0, len(record_ids), BATCH):
    with db.session.begin():
        for rid in record_ids[i:i+BATCH]:
            update_record(rid)
    # write lock released between batches
    # other writers (Flask deposits) get a turn
```

FTS5 index maintenance via triggers rather than explicit `INSERT INTO works_fts(works_fts) VALUES('rebuild')` also avoids holding the write lock for the entire corpus.

Concurrency ceiling

Empirical benchmarks on commodity hardware (commonmeta-rs experience + SQLite literature):

Concurrent writers	Behaviour
1	No contention, full throughput
2–4	< 5 % contention loss; `busy_timeout=2000` sufficient
8–16	Noticeable queuing; 5–10 % of requests hit busy_timeout at peak
> 16	Write throughput degrades; consider serializing writes through a queue

For a small institution (< 50 concurrent users, < 5 simultaneous deposits), 2–4 Gunicorn workers with `busy_timeout=5000` and `BEGIN IMMEDIATE` is stable. Beyond ~20 simultaneous writers the single-writer model becomes a bottleneck and PostgreSQL becomes the right choice.

Key Technical Challenges

1. invenio-db: SQLAlchemy + SQLite compatibility (Medium–Hard)

Issue: invenio-db and its dependents use several PostgreSQL-specific features:

- **JSONB columns**: `invenio-records` stores records as `JSONB`. SQLite has the JSON1 extension but no dedicated JSONB type. SQLAlchemy's `postgresql.JSONB` must be replaced with `sa.Text` (with JSON serialization in application code) or `sa.JSON` (mapped to TEXT in SQLite).
- **`postgresql.UUID`**: Used for primary keys in many Invenio tables. Must be mapped to `sa.String(36)` or `sa.Text` on SQLite. SQLAlchemy 2.x has `sa.Uuid` that works on both.
- **Alembic migrations**: Many existing migration scripts contain `op.execute("ALTER TABLE ... USING expr::jsonb")` or PostgreSQL-specific DDL. These need SQLite-safe equivalents (SQLite has limited `ALTER TABLE` — no `DROP COLUMN` before 3.35, no type changes).

- **`postgresql.ARRAY`**: Used in a few places for array-type columns. Replace with JSON text.
- **`RETURNING` clause**: Used in some insert operations; supported in SQLite ≥ 3.35.
- **`ON CONFLICT DO UPDATE` (upsert)**: Supported in SQLite ≥ 3.24 — this one is fine.

Mitigation: Use SQLAlchemy's dialect-conditional column type mapping (pattern exists in projects like Flask-Security-Too). Fork `invenio-db` into `invenio-db-sqlite` that re-exports all models with SQLite-compatible types.

Effort: ~3–4 weeks for the ORM/migration layer. The migration scripts are the hardest part.

2. invenio-search: SQLite FTS5 replacement (Hard)

This is the most significant engineering challenge. `invenio-search` is tightly coupled to the OpenSearch query DSL and aggregations API.

What needs replacing:

- `RecordsSearch`, `CommunitySearch` (class-based query builders using `opensearch-dsl`)
- Aggregation/facet computation: OpenSearch `terms`, `date_histogram`, `range` aggregations have no direct SQLite equivalent — must be reimplemented as SQL GROUP BY queries
- Permission filtering: currently done as OpenSearch query filters; must become SQL WHERE clauses
- Sorting: `_score`, `updated`, `publication_date` — easy in SQL
- Pagination: `from/size` → SQL `LIMIT/OFFSET`
- Nested queries on JSONB (e.g. `creators.person_or_org.name`) — must be pre-extracted into indexed columns (similar to how commonmeta-rs extracts title/subjects into the works table)

Suggested approach: Implement `invenio-search-sqlite` as an alternative backend, similar to how `invenio-search` itself abstracts ES vs OpenSearch. The interface contract is the `Search` class (`.query()`, `.filter()`, `.aggs`, `.execute()`). A SQLite backend would translate these into SQL at query time.

For facets: pre-extract key fields (resource type, access status, publisher, language, year) into indexed columns in the record table, then use GROUP BY for aggregations. This is exactly the pattern in commonmeta-rs (`type`, `language`, `date_published`, `provider` as first-class columns alongside the compressed blob).

Effort: 6–8 weeks for a minimal viable implementation (basic search + 4–5 facets). Full parity with OpenSearch facets would take longer.

3. Celery broker: SQLite via kombu (Easy)

`kombu` (Celery's messaging library) has a SQLite transport via its SQLAlchemy backend. Replace Redis/RabbitMQ with:

```
CELERY_BROKER_URL = "sqla+sqlite:////path/to/celery.db"
CELERY_RESULT_BACKEND = "db+sqlite:////path/to/results.db"
```

Caveats: SQLite broker has known performance issues under high concurrency but is acceptable for Starter/small-institution workloads (few concurrent background tasks). The SQLAlchemy transport polls; not suitable for real-time task fan-out at scale.

Effort: ~1 day configuration, ~1 week testing/stabilization.

4. Concurrent writes (Low–Medium risk)

SQLite in WAL mode supports one writer and multiple concurrent readers. InvenioRDM's Celery workers + Flask web workers all writing simultaneously could hit the single-writer limit.

Mitigation: - WAL mode already in use in commonmeta-rs (proven at scale for reads) - Writes in InvenioRDM are bursty, not sustained — SQLite handles typical deposit workflows - `PRAGMA busy_timeout = 5000` prevents most `SQLITE_BUSY` errors - For demo/development: single-process deployment avoids the issue entirely

Effort: Low — mostly configuration and error handling.

5. File storage

InvenioRDM's `invenio-files-rest` stores file metadata in PostgreSQL (buckets, object locations, checksums) and binary content in S3 or local filesystem. The file metadata tables are small and simple — straightforward to port to SQLite. Binary storage already works with local filesystem.

Effort: ~1–2 weeks.

Rust / commonmeta-rs Work Required

The dragoman/commonmeta-rs Rust layer would need:

Write path from Python

Currently commonmeta-rs is the **write** engine (imports Crossref/DataCite/ORCID/ROR into SQLite) and dragoman is the **read** server. InvenioRDM's deposit workflow writes new records via Python/SQLAlchemy.

Two options: 1. **Python writes directly to SQLite** (via SQLAlchemy + same schema) — simplest 2. **Python calls Rust via FFI or subprocess** — complex, avoid

Option 1 is preferred: InvenioRDM Python code writes records in the same commonmeta schema. The `metadata` BLOB uses zstd-compressed commonmeta JSON, which Python can produce via `commonmeta-py` (the Python sibling library) + `zstandard` package.

FTS5 index maintenance

When Python inserts/updates records, FTS5 triggers (or explicit rebuild calls) must keep `works_fts` in sync. SQLite FTS5 content tables with `content=` need explicit population via `INSERT INTO works_fts(works_fts) VALUES('rebuild')` or row-level triggers.

Recommendation: Add INSERT/UPDATE/DELETE triggers to the schema to maintain the FTS5 index automatically. This is straightforward SQLite DDL.

Effort: ~1 week to add triggers + test from Python.

`commonmeta-py` alignment

The Python library `commonmeta-py` must produce JSON that `commonmeta-rs` can round-trip. Some schema drift between the Python and Rust implementations may need reconciliation.

Effort: 1–2 weeks.

Effort Breakdown

Component	Effort	Difficulty
SQLAlchemy/invenio-db SQLite compat	3–4 weeks	Medium
Alembic migration scripts for SQLite	2–3 weeks	Hard
invenio-search SQLite FTS5 backend	6–8 weeks	Hard
Facet aggregations via SQL	3–4 weeks	Medium

Component	Effort	Difficulty
Celery broker via kombu SQLite	1 week	Easy
File metadata tables	1–2 weeks	Easy
FTS5 write triggers from Python	1 week	Easy
commonmeta-py/rs schema alignment	1–2 weeks	Medium
Integration testing + stabilization	3–4 weeks	Medium
Documentation + packaging	1–2 weeks	Easy
Total	**~22–31 weeks**	

A focused 2-person team (1 Python, 1 Rust/Python) could deliver a working Starter prototype in 4–5 months and production-ready small-institution support in 6 months.

What Scope to Cut for an MVP

A minimal “works for demo servers and local dev” version only needs:

1. SQLite record storage (replaces PostgreSQL for the `rdm_records`, `rdm_drafts` tables)
2. SQLite FTS5 search for records (title + description full-text, 3 facets: type, year, access)
3. SQLite for communities (minimal: community slug, title, owner)
4. **Drop** users/auth — use SQLite with existing `flask-security-too` SQLite support (it works)
5. **Drop** requests/review workflow — defer to v2
6. SQLite broker for Celery (vs Redis)

This narrows the estimate to ~10–14 weeks and produces something useful for the Starter and local development use cases.

FTS5 vs Tantivy for Faceted Search

The key distinction

Faceted search has two independent components:

Component	What it does	SQLite answer
Full-text search		**FTS5**

Component	What it does	SQLite answer
	Rank documents by keyword relevance (BM25)	
Facet counts	COUNT(*) per bucket (type, year, access, …)	**SQL GROUP BY**

FTS5 is not a facet engine. OpenSearch happens to bundle both in one query. In SQLite they are two separate queries against the same file, but both are in-process and sub-millisecond at Starter scale (< 500K records).

Concrete query pattern for InvenioRDM record search:

```
-- 1. Full-text search: get ranked rowids
SELECT rowid FROM works_fts
WHERE works_fts MATCH 'climate change'
ORDER BY rank
LIMIT 10 OFFSET 0;

-- 2. Apply per-row permission + facet filters on pre-extracted
      columns
SELECT w.id, w.metadata
FROM works w
WHERE w.rowid IN (…fts rowids…)
  AND w.access_status = 'open'
  AND w.type = 'Dataset';

-- 3. Facet counts (separate query, shares the same filtered rowid
      set)
SELECT type, COUNT(*) FROM works
WHERE rowid IN (…fts rowids without type filter…)
GROUP BY type ORDER BY 2 DESC;
```

This replaces one OpenSearch request with 2–3 SQLite queries. At < 500K records all three complete in under 10 ms total. No network round-trip (embedded).

What this requires from the schema

Pre-extract the fields users facet on as indexed first-class columns — exactly the pattern in commonmeta-rs (`type`, `date_published`, `language`, `provider`). InvenioRDM would add:

- `resource_type` TEXT (e.g. `"software"`, `"dataset"`)
- `access_status` TEXT (`"open"`, `"restricted"`, `"embargoed"`)
- `publisher` TEXT
- `year` INTEGER (extracted from `date_published`)

- `subject_ids` TEXT (JSON array of subject IDs, queryable via `json_each`)

Date histogram facets use SQLite's `strftime('%Y', date_published)` in GROUP BY — identical to what OpenSearch's `date_histogram` aggregation does internally.

When FTS5 alone is not enough (and Tantivy would help)

FTS5 weaknesses that matter for InvenioRDM:

1. **Field-specific boosting**: FTS5 weights columns equally or by a fixed column weight. OpenSearch boosts `title^3, description^1` per-query. FTS5 column weights are set at `CREATE VIRTUAL TABLE` time, not per-query. Work-around: duplicate title into a separate FTS5 column and query it with `title:term`.

2. **Simultaneous search + facets in one pass**: OpenSearch returns `hits + aggregations` in one response. With FTS5+SQL you need 2–3 queries. For < 500K records this is not a problem; above ~2M it becomes noticeable (~30–50 ms total).

3. **Phrase proximity and field nesting**: FTS5 supports `"exact phrase"` and `NEAR` queries but not OpenSearch's `nested` query on embedded JSON objects. Mitigation: denormalize into pre-extracted columns.

4. **Highlighting + snippets**: FTS5 has `snippet()` and `highlight()` auxiliary functions. Works for basic use; less configurable than OpenSearch highlighting.

When Tantivy would be needed

Tantivy is the Rust equivalent of Lucene and is the engine behind Meilisearch, Quickwit, and Lnx. It has native DocValues (fast fields) that return facet counts in the same index pass as the full-text search, avoiding the N-query pattern above.

Tantivy would be worth the integration complexity only if:

- Record count exceeds ~2M and simultaneous search+facets latency matters
- Field-level relevance boosting per-query is required (journal article title vs abstract)
- Real-time indexing (< 1 s latency from deposit to searchable) is required
- Advanced tokenizers (language-specific stemming, synonyms) are needed

For the **Starter / local dev / demo / small institution** target:

> **FTS5 + SQL GROUP BY is sufficient. Tantivy is not needed.**

The decisive factor is that InvenioRDM at small scale has thousands to low hundreds of thousands of records. SQLite FTS5 scans the entire matching rowid set in memory in this range. Tantivy's segment architecture pays off at millions of documents.

If Tantivy were adopted later, `tantivy-py` (PyO3 bindings, available on PyPI) provides a Python API, and the index can live alongside the SQLite file. The FTS5 virtual table could be dropped and replaced with Tantivy with no change to the rest of the schema.

Recommendation

Phase 1–2: FTS5 + SQL GROUP BY. Sufficient for the stated use cases, zero additional dependencies, proven at 150M+ rows in production (commonmeta.org).

Phase 3+ (optional): Evaluate Tantivy if an institution outgrows SQLite FTS5 latency. The schema pre-extraction pattern is identical between the two; switching is a drop-in replacement of the search layer, not a data model change.

Limitations

Record Count

Metric	Safe	Caution	Not recommended
Total records (FTS5 search)	< 500K	500K – 2M	> 2M
Total records (SQL GROUP BY facets)	< 500K	500K – 2M	> 2M
SQLite file size	< 10 GB	10–50 GB	> 50 GB

At **500K records**, FTS5 keyword search and GROUP BY facets each return in under 5 ms with proper indexes. At **2M records**, latency rises to 50–150 ms for complex faceted queries — still acceptable for most users but noticeably slower. Beyond **2M records**, facet queries without covering indexes can exceed 500

ms. The FTS5 index itself has no hard size limit (commonmeta.org uses FTS5 at 150M+ rows), but the facet-side SQL becomes the bottleneck.

SQLite does not support parallel query execution. A single slow query blocks the thread it runs on. Deploy with multiple worker processes (`WORKERS=4`) to keep other requests responsive while one worker handles a heavy facet query.

Write Concurrency

Metric	Safe	Caution	Not recommended
Gunicorn workers	1–4	4–8	> 8
Simultaneous deposit submissions	1–5	5–20	> 20
Background Celery workers (DB-writing tasks)	1–2	2–4	> 4

SQLite WAL mode allows **one writer at a time**. Readers never block writers and writers never block readers, but two concurrent writes queue against each other. With `busy_timeout = 5000 ms` and `BEGIN IMMEDIATE`, up to ~4–8 writers can contend without user-visible errors; beyond that, the 5-second timeout starts expiring and deposits return HTTP 500.

Concretely: a web deposit transaction holds the write lock for ~20–50 ms. At 4 workers, the effective write throughput ceiling is roughly **80–200 deposits/minute** before queuing becomes visible. For typical small-institution usage (a few simultaneous submitters) this is never reached. The ceiling is hit when background Celery tasks (bulk harvest, embargo lifting, re-indexing) run while users are depositing — mitigate by scheduling bulk writes outside business hours or rate-limiting them explicitly.

Missing Capabilities

These features are either absent or substantially degraded relative to PostgreSQL + OpenSearch:

No geographic / bounding-box search. OpenSearch geo_point and geo_shape fields have no equivalent in SQLite. Institutions relying on spatial search (e.g. geoscience data portals) cannot migrate Stage 2.

No real-time indexing. OpenSearch near-real-time indexing makes a deposit searchable within ~1 second. SQLite FTS5 is updated inside the same commit as the record insert, so it is consistent — but there is no push-based notification. Users who

refresh immediately after deposit will see the result; background harvests that batch-commit every 500 records will have a ~5–30 second gap.

No advanced text analysis. FTS5 `unicode61` tokenizer provides basic Unicode word splitting. OpenSearch supports per-field language analyzers, synonym expansion, stop-word lists, and custom tokenizers. Institutions with multilingual corpora or domain-specific vocabulary will notice lower recall.

No distributed deployment. SQLite is a single-file database on one server. You cannot run two InvenioRDM instances against the same SQLite file, so blue/green deploys and horizontal scaling are not possible. Vertical scaling (bigger VM) is the only option.

No streaming replication or point-in-time recovery. PostgreSQL streaming replication and WAL archiving enable PITR and read replicas. SQLite backup is snapshot-only (`sqlite3 .backup` or `VACUUM INTO`). For SQLite deployments, schedule automated snapshots (e.g. Litestream for continuous WAL streaming to S3) and test restore procedures.

Graduation Threshold

Migrate back to PostgreSQL + OpenSearch when **any** of the following is true:

- Record count exceeds **2M** and faceted search latency is user-visible (> 200 ms p95).
- Simultaneous deposit volume exceeds **20 concurrent writers** sustained over business hours.
- Geographic search, real-time indexing (< 1 s), or multilingual text analysis is required.
- The institution requires horizontal scaling (multiple application nodes).
- The database file exceeds **50 GB** and routine backups cannot complete in the maintenance window.

The schema design (pre-extracted columns, FTS5 virtual table, `works_relations` table) is intentionally close to what a PostgreSQL migration requires. The main delta is replacing the FTS5 index with an OpenSearch index and switching `NullPool` → connection pooling.

The Datasette Precedent

Simon Willison's Core Insight

Simon Willison created Datasette (2017) after a moment of recognition in the shower: a SQLite database file plus a tiny HTTP server is already a complete, queryable, shareable data publication platform. No Postgres cluster. No search daemon. No queue broker. Just a file.

The insight was not about SQLite's performance ceiling but about the **ratio of capability to operational complexity**. A journalist, a researcher, or a small team can publish a dataset by copying a file to a server — the same mental model as attaching a spreadsheet to an email, but with a SQL query interface, an automatic JSON API, and full-text search via FTS5. The infrastructure required to run it is exactly as complex as the infrastructure required to serve a static website.

Datasette's production deployments (data.world, Fly.io's public SQLite work, the UK Parliament API) proved that SQLite-backed web services handle millions of read requests per day without architectural complexity. Willison documented this explicitly:

> "SQLite is the most widely deployed database in the world — it's in every smartphone, every browser, every operating system. The idea that it can't be used for web applications is a myth perpetuated by people who haven't actually tried it."

The architectural corollary is that most web applications are overwhelmingly read-heavy. A single SQLite file served with WAL mode and `mmap_size` set appropriately will outperform a poorly-tuned PostgreSQL instance on the same hardware for read workloads, because reads never contend and the entire hot dataset fits in the OS page cache.

How This Applies to InvenioRDM

InvenioRDM's default stack — PostgreSQL + OpenSearch + Redis + Celery — is designed to handle CERN-scale operations: tens of millions of records, thousands of concurrent users, complex aggregation pipelines, distributed background processing. For that workload, every component earns its keep.

The Starter / small-institution use case is different in kind, not just degree:

- A university department repository holds **10K–100K records**, not 10M.
- Peak concurrent users are **5–20**, not 5,000.

- A deposit workflow that takes 200 ms is fast; one that takes 2 s is acceptable.
- The operator is a research IT generalist, not a database administrator.
- The deployment budget may be a single €40/month VPS.

For this use case, the four-service stack imposes overhead that dwarfs the actual workload: PostgreSQL requires a connection pool, routine VACUUM, index maintenance, and backup infrastructure; OpenSearch requires 2–4 GB of JVM heap, a separate indexing pipeline, and its own backup strategy; Redis requires persistence configuration; Celery requires worker supervision and dead-letter queue monitoring.

The Datasette insight reframed: **a research repository serving 50K records to 10 concurrent users should be deployable and maintainable with the same operational burden as a static website.** A single SQLite file, a single Python process, and a cron-scheduled backup script.

This is exactly the operational model that `sqlite3 .backup` (or Litestream) enables. The database is a file. The file is a backup. Restore is a file copy.

What Datasette Validated for This Design

Datasette's six-year production history provides direct evidence for several design decisions in the proposed InvenioRDM SQLite backend:

- **FTS5 is sufficient for data discovery at < 2M records.** Datasette's default full-text search is FTS5; no Tantivy, no OpenSearch. Willison has written extensively about FTS5's performance characteristics and its fit for data publishing workloads.
- **Generated columns for faceting.** Datasette pre-extracts facet fields from JSON blobs into indexed columns — the same pattern proposed here for InvenioRDM's metadata JSON.
- **WAL mode + single-process writes.** Datasette's architecture is single-writer by design. The same `busy_timeout` + `BEGIN IMMEDIATE` pattern is used and documented.
- **HTTP read replicas via file copy.** Because the database is a file, Datasette instances can share a read-only replica by mounting the same file read-only. InvenioRDM's web tier could do the same for read endpoints if vertical scaling is ever needed.

The key difference is that InvenioRDM has a write path (deposit, versioning, embargo management) that Datasette's typical publishing workflow does not. That is where the WAL concurrency analysis in this document applies and where the departure from pure Datasette patterns begins.

Single Stack vs Scale-Out: Matching Infrastructure to Workload

The Default Assumption Problem

InvenioRDM's default deployment requires six independent services before you can accept a single deposit: PostgreSQL, OpenSearch, Redis, RabbitMQ, Celery workers, and the web application itself. This is appropriate for CERN, which operates at a scale where each service earns its keep. It is not appropriate for a university department running 10K records and five simultaneous users.

The problem is not that the services are wrong — it is that they are mandatory from day one. A new InvenioRDM operator must understand, configure, monitor, backup, and upgrade all six services before the repository is even populated. The operational complexity budget is fully spent before the scientific work begins.

> **The key question is not "can we use PostgreSQL?" but "should we spend operator attention on PostgreSQL right now, given our actual workload?"**

Component-by-Component Graduation

Each service in the InvenioRDM stack solves a real problem at scale. The graduation model replaces the assumption that all problems exist from day one with the principle that each component is added only when the workload creates the problem it solves.

SQLite → PostgreSQL

SQLite handles one writer at a time. PostgreSQL handles hundreds. The graduation trigger is not a record count but a write concurrency pattern: when more than ~10 simultaneous users are regularly submitting deposits, or when background harvest jobs contend with web deposits to the point of user-visible timeouts, PostgreSQL becomes justified. For a small-institution repository where deposits happen a few times per day, SQLite's ceiling is never reached.

SQLAlchemy's dialect abstraction means this graduation is a configuration change (`DATABASE_URL` swap), not a code change — the same model that Django demonstrated a decade ago for test-to-production transitions.

FTS5 → OpenSearch

OpenSearch solves full-text search at scale with advanced text analysis, near-real-time indexing, and distributed query execution. FTS5 solves full-text search for files that fit in RAM. The graduation trigger is when search latency becomes user-visible (typically at > 2M records), when multilingual text analysis or synonym expansion becomes a requirement, or when a deposit must appear in search results within seconds of submission. For a repository of 50K records with a single language collection, OpenSearch is solving a problem that does not exist.

SQLite kombu transport → RabbitMQ

Celery's `kombu` transport layer already supports SQLite as a message broker. Background tasks (DOI registration, format conversion, file integrity checks, email notifications) work correctly against a SQLite queue for any workload a small institution generates. RabbitMQ becomes justified when job throughput is high enough that the SQLite single-writer lock becomes a queue bottleneck — in practice, hundreds of concurrent job submissions per minute. Small institutions produce background jobs in the tens-per-hour range.

Redis → in-process cache or SQLite

Redis in InvenioRDM primarily serves session storage, rate limiting, and Celery result backend. For a single-process deployment, an in-process cache (e.g. `cachelib.SimpleCache`) handles sessions, and the Celery result backend can be the same SQLite database. Redis becomes necessary when the web tier scales to multiple processes on multiple hosts that need shared session state.

The Operational Complexity Budget

Dan McKinley's "choose boring technology" framing is applicable here: every service you add to a stack consumes from a finite budget of operator attention. That attention pays for initial setup, routine upgrades, incident response, backup verification, and capacity planning. For a research IT generalist maintaining a departmental repository alongside other responsibilities, the budget is small.

The four-service stack (PostgreSQL + OpenSearch + RabbitMQ + Redis) exhausts that budget before a single record is deposited. The single-stack alternative (SQLite for everything) preserves the budget for the actual mission: helping researchers manage and publish data.

The cost is asymmetric: over-engineering is paid every day by every operator who doesn't need it; under-engineering is paid only if the workload actually outgrows the simple stack. For the majority of InvenioRDM Starter deployments, the workload never will.

Same Codebase, Configuration-Driven

The goal of this work is that the same InvenioRDM codebase supports both the single-stack and the scale-out configuration, switchable by environment variables with no code changes:

```
# Starter / single-stack
DATABASE_URL=sqlite:///data/invenio.db
SEARCH_BACKEND=sqlite_fts5
BROKER_URL=sqla+sqlite:///data/celery.db
CACHE_TYPE=simple

# Production / scale-out
DATABASE_URL=postgresql+psycopg2://user:pass@db/invenio
SEARCH_BACKEND=opensearch
BROKER_URL=amqp://guest:guest@rabbitmq:5672//
CACHE_TYPE=redis
```

A developer running locally uses the Starter configuration with no Docker. The same application, deployed to a departmental server, uses the same configuration. When the institution grows, the operator changes environment variables and migrates data — they do not fork the codebase or redesign the application.

The Django Precedent

Django established this pattern in 2008 and it has been the default recommendation ever since. Django's `DATABASES` setting accepts an `ENGINE` key that selects the backend (`django.db.backends.sqlite3` or `django.db.backends.postgresql`) while the model definitions, migrations, and business logic are unchanged. The ecosystem further normalized the pattern through `dj-database-url` (originally from Heroku's twelve-factor buildpack), which reads a `DATABASE_URL` environment variable and configures the backend automatically — exactly the convention shown above.

The critical enabler is that Django's ORM emits standard SQL and handles the dialect differences between SQLite and PostgreSQL (type coercions, `RETURNING` clause support, `ON CONFLICT` syntax, transaction isolation) in the backend layer, not in application code. Tens of millions of Django applications have validated this over fifteen years: the dev–prod parity is real and the upgrade path from SQLite to PostgreSQL is a `DATABASE_URL` change plus `pg_dump` / restore.

InvenioRDM uses SQLAlchemy + Flask-SQLAlchemy, which has the same dialect abstraction capability. The `create_engine()` call already accepts a URL; only the InvenioRDM-specific SQL that bypasses SQLAlchemy (raw queries in `invenio-db`, JSON containment operators in `invenio-rdm-records`) needs to be ported. That porting work is the substance of Stage 1 in this proposal — not a reimplementation of the ORM, but an audit and replacement of the small set of PostgreSQL-specific SQL that SQLAlchemy does not abstract.

Rogue Scholar: A Concrete Use Case

Rogue Scholar is a science blog archiving service that mints DOIs for science blog posts and makes them citable, searchable scholarly records. It is built on InvenioRDM, written and hosted by the author of this document, and operated as a single-person project with no dedicated operations staff.

Workload characteristics:

- ~50–100K records (archived science blog posts), growing by a few hundred per week
- Read-heavy: the dominant traffic pattern is researchers looking up, citing, and downloading metadata for archived posts
- Write pattern: periodic RSS feed harvests ingest new posts in background Celery tasks; simultaneous human deposits are rare (curator-only workflow)
- Concurrent users: small; academic traffic, not consumer-scale
- Operator: one person managing the full stack

Current pain points with the scale-out stack:

Running PostgreSQL, OpenSearch, RabbitMQ, and Redis for a ~100K-record, single-operator service means:

- Four services to monitor, upgrade, and back up independently
- OpenSearch JVM heap (2–4 GB) is the largest single cost line on the hosting bill, exceeding the RAM used by the application itself
- Backup requires coordinating PostgreSQL `pg_dump`, OpenSearch snapshot, and file storage — three separate procedures to keep in sync

- A local development environment requires Docker Compose with five containers before the first record can be loaded
- Any infrastructure incident (disk full, OOM, certificate expiry) requires diagnosing across all four services

Why SQLite fits this workload precisely:

The Rogue Scholar workload maps onto SQLite's strengths:

- 50–100K records is deep in the "safe" range; FTS5 search over the full corpus returns in < 5 ms
- The write pattern is sequential batch (RSS harvest commits 50–500 records in one transaction, not 500 simultaneous users submitting forms)
- A SQLite WAL file plus a Litestream sidecar provides continuous backup to S3 — one mechanism instead of three
- The commonmeta-rs integration is already in production for DOI resolution; the same library handles format conversion for import and export, reducing the Python serialization surface area
- Local development becomes `git clone` + one SQLite file

Relationship to commonmeta:

Rogue Scholar's ingestion pipeline already uses commonmeta for converting blog post metadata into scholarly records (title, authors, DOI, references). A `commonmeta-py` PyO3 package would let the InvenioRDM Python layer call the same Rust conversion code that dragoman uses in production, removing the duplicate Python implementation of format normalization that currently exists in the Rogue Scholar harvester.

Expected outcome:

Migrating Rogue Scholar from the full InvenioRDM stack to the SQLite-backed single-stack configuration would:

- Reduce the hosting bill by eliminating the OpenSearch node (largest cost)
- Reduce backup complexity to a single Litestream-managed file
- Make the development environment reproducible from a repo clone in under a minute
- Serve as the reference implementation and integration test bed for the `invenio-db-sqlite` and `invenio-search-sqlite` packages proposed in this document

---

JSON Schema and commonmeta: Schema and Tooling Consolidation

The Four-Layer Schema Problem

InvenioRDM currently maintains four independent schema representations of the same metadata:

Layer	Technology	Purpose
Database columns	SQLAlchemy models	ORM mapping to PostgreSQL
Record validation	JSON Schema (jsonschema)	Content correctness
API serialization	Marshmallow	JSON in/out at HTTP boundary
Search mapping	OpenSearch mapping	Field types, analyzers, indexing

Every time a metadata field is added or changed, all four layers must be updated in sync. The OpenSearch mapping and the SQLAlchemy model use entirely different vocabularies; a `keyword` analyzer in OpenSearch has no natural counterpart in SQLAlchemy. The result is that schema evolution is expensive, error-prone, and requires four-way expertise to review.

JSON Schema as Single Source of Truth

The SQLite migration is the opportunity to collapse these layers. JSON Schema already sits at the centre of InvenioRDM's validation pipeline. The remaining layers can be derived from it:

SQLite DDL generation. A JSON Schema with custom `x-` extension keywords can drive automatic DDL generation for the `works` table. For example:

```
{
  "properties": {
    "doi": { "type": "string", "x-column": true, "x-index":
        true },
    "title": { "type": "string", "x-fts": true },
    "publication_year": { "type": "integer", "x-column": true }
  }
}
```

A small code generator reads the schema and emits `CREATE TABLE` DDL with the correct column types and `CREATE INDEX` statements. The FTS5 content-table columns are derived from the same `x-fts: true` flags. Adding a new indexed field is a one-line JSON change, not a four-file change.

Marshmallow generation. `marshmallow-jsonschema` and `apispec` can generate Marshmallow schemas from JSON Schema definitions. The HTTP serialization layer becomes a thin adapter over the canonical schema rather than a parallel specification.

Validation consolidation. With PostgreSQL gone, the SQLAlchemy model and the JSON Schema were duplicating type information. With a generated DDL layer, the JSON Schema is authoritative and the DDL is a build artifact.

This pattern is not theoretical: OpenAPI 3.x uses the same `x-extension` mechanism for code generation, and projects like `datamodel-code-generator` already generate Python dataclasses from JSON Schema. The delta for InvenioRDM is writing one schema-to-DDL generator (~200 lines of Python) that understands the `x-column`, `x-index`, and `x-fts` extension keywords.

commonmeta as the Interoperability Layer

InvenioRDM's metadata schema is not commonmeta and should not become it. InvenioRDM's schema is DataCite-aligned but carries fields that describe repository management state, not intellectual content:

- Access rights: embargo dates, access conditions, restricted record flags
- Community membership and collection assignment
- Review requests, workflow state, curation events
- File metadata: checksums, sizes, MIME types, storage backends
- Custom fields defined per community
- InvenioRDM-specific PID management (multiple identifier types, PID states)

These have no equivalent in commonmeta, which models *what a work is*, not *how it is managed inside a repository*. Forcing InvenioRDM records into the commonmeta schema would either discard repository management data or require extending commonmeta with InvenioRDM-specific concepts — defeating the purpose of a format-neutral canonical schema.

The right role for commonmeta is at the **import/export boundary**, not in core storage.

Import. When harvesting from Crossref, DataCite, OpenAIRE, or similar sources, commonmeta handles format conversion to a normalized intermediate representation. InvenioRDM maps from that representation to its own schema. One commonmeta converter replaces N format-specific InvenioRDM harvesters; the mapping step (neutral → InvenioRDM) is a thin, stable adapter rather than a full parser for each source format.

Export. When returning BibTeX, CSL-JSON, JATS, or DataCite XML to a depositor or harvester, commonmeta handles the output serialization. InvenioRDM maps its record to the commonmeta intermediate; commonmeta-rs serializes to the target format. One exporter replaces N format-specific Marshmallow serializers for standard scholarly formats.

DOI resolution. dragoman uses commonmeta-rs today for DOI resolution and content negotiation. An InvenioRDM instance could expose the same endpoint without a separate dragoman deployment, because the export path described above already produces the formats dragoman serves.

PyO3 bindings: `commonmeta-py`. The mechanism is a PyO3-based Python package exposing commonmeta-rs conversion functions. InvenioRDM calls into Rust only at conversion boundaries, not for core record storage or retrieval:

```
import commonmeta

# Import: convert external source to neutral form, then map to
        InvenioRDM
external = commonmeta.from_doi("10.5281/zenodo.8340374")
        # fetch + normalize
invenio_record = map_to_invenio_schema(external)            #
        InvenioRDM mapping step

# Export: map InvenioRDM record outward, then serialize
bibtex =
        commonmeta.to_bibtex(commonmeta.from_invenio(invenio_record))
```

This removes the N×M problem (N input formats × M output formats) from the Python codebase without touching InvenioRDM's core schema, data model, or storage format.

What commonmeta does not replace. All Python code concerned with access rights, embargo management, file metadata, community workflows, custom fields, and PID state stays in Python. The commonmeta surface area is strictly the conversion of scholarly bibliographic metadata between formats — a well-defined, bounded responsibility.

Scope for This Migration

The JSON Schema DDL-generation and commonmeta-py integration are independent of the Stage 1/2 database migration:

1. **Stage 1** (PostgreSQL → SQLite): adopt JSON Schema DDL-generation for the SQLite schema; generate Marshmallow schemas from the same JSON Schema rather than writing them by hand.
2. **Stage 2** (drop OpenSearch): derive FTS5 configuration from `x-fts` keywords in the JSON Schema; retire the OpenSearch mapping.
3. **Stage 3 (optional)**: publish `commonmeta-py` (PyO3 bindings); wire it into InvenioRDM's import harvesters and export serializers; retire format-specific Marshmallow schemas for standard scholarly formats. InvenioRDM-specific schema code (access, community, files, PIDs) remains Python and is unaffected.

Stage 3 removes a large surface area of Python serialization code and replaces it with a Rust library tested at 150M+ records in production — without touching the core data model or requiring database schema changes. It can proceed independently after Stages 1 and 2 are stable.

Prior Art and Related Work

- **Datasette (Simon Willison, 2017–present)**: the canonical proof that SQLite is production-viable for data publishing at scale; validated FTS5, generated-column faceting, and WAL single-writer patterns that this proposal adopts directly. See dedicated section above.
- **kombu SQLAlchemy transport**: used in production by smaller Celery deployments.
- **invenio-search abstraction**: the existing ES→OpenSearch migration showed the search layer can be swapped with ~6 weeks of effort; SQLite is a larger gap but the path exists.
- **commonmeta-rs at scale**: the production commonmeta.org instance runs SQLite with >150M works, using WAL + mmap_size for read performance — no signs of ceiling for small-institution record counts (10K–1M records).

Two-Stage Refactoring: PostgreSQL First, Then OpenSearch

Yes — the migration can and should be split into two independent stages. The two stores are decoupled in InvenioRDM's architecture: `invenio-db` owns the primary record store (PostgreSQL), `invenio-search` owns the search index (OpenSearch). Replacing one does not require touching the other.

Why the split is clean

InvenioRDM's indexing pipeline is one-directional:

```
Flask deposit → PostgreSQL (invenio-db)
                      │
                      ▼ (Celery invenio-indexer)
                OpenSearch (invenio-search)
```

The search index is a *derived* view of the primary database. OpenSearch can keep running against a SQLite-backed primary store in Stage 1 with zero changes to the search layer.

Stage 1 — Replace PostgreSQL with SQLite

Goal: eliminate PostgreSQL; keep OpenSearch running unchanged.

Scope:

- `invenio-db-sqlite`: fork/patch of `invenio-db` with SQLite-compatible column types (`sa.JSON` instead of `postgresql.JSONB`, `sa.String(36)` instead of `postgresql.UUID`, generated columns for indexed JSON paths, WAL + busy_timeout pragmas on connect)
- Alembic migration scripts: audit and fix PostgreSQL-specific DDL (type casts, array columns, unsupported `ALTER TABLE` operations)
- Connection pool: `NullPool` + `check_same_thread=False` + `BEGIN IMMEDIATE` transactions
- Celery broker: can stay on Redis at this stage — changing two things at once increases risk
- FTS5 triggers: not needed yet — OpenSearch indexer reads directly from the SQLite DB via SQLAlchemy and sends records to OpenSearch as before

What still runs:

- OpenSearch (unchanged — invenio-indexer reads from SQLite and writes to OpenSearch)
- Redis (unchanged — Celery broker)

- All search, facets, and aggregations (served by OpenSearch as before)

Validation criteria:

- All invenio-rdm-records unit tests pass against SQLite
- Deposit → publish → search round-trip works
- Alembic `upgrade head` runs cleanly on a fresh SQLite file

Effort: ~8–10 weeks.

Task	Weeks
`invenio-db-sqlite` column type mapping	2–3
Alembic migration script audit + fixes	2–3
WAL / connection pool config	1
File metadata tables	1
Integration testing (deposit, search, access)	2

Stage 2 — Drop OpenSearch

Goal: eliminate OpenSearch and Redis; SQLite as the only data store.

Precondition: Stage 1 is complete and stable.

Scope:

- `invenio-search-sqlite`: new backend implementing the `invenio-search` interface (`RecordsSearch`, `CommunitySearch`) using SQLite FTS5 + SQL GROUP BY
- Pre-extract facet fields into indexed columns (or generated columns) during Stage 1 schema work — doing this upfront in Stage 1 avoids a second migration
- Drop `invenio-indexer` Celery task or replace it with FTS5 trigger maintenance
- Replace OpenSearch aggregation results with SQL GROUP BY results
- Replace OpenSearch permission filters with SQL WHERE clauses
- Celery broker: switch from Redis to `sqla+sqlite://` (kombu) at this stage if Redis is to be eliminated entirely

What is no longer needed:

- OpenSearch / Elasticsearch container
- Redis (if broker is also migrated)
- `invenio-indexer` and its Celery tasks

Effort: ~12–14 weeks.

Task	Weeks
`invenio-search-sqlite` FTS5 query backend	4–5
Facet aggregations (GROUP BY + pre-extracted columns)	3–4
Permission filtering via SQL WHERE	2
FTS5 trigger maintenance (replace indexer)	1
Celery broker SQLite migration	1
Integration testing + parity validation	2–3

What to pre-extract in Stage 1 to enable Stage 2

Even though Stage 2 work starts later, the schema design for Stage 1 should include indexed columns for fields that will become SQL facets in Stage 2. Adding these as generated columns in Stage 1 costs almost nothing and avoids a second migration:

```
-- Add to rdm_records_metadata during Stage 1 migration
ALTER TABLE rdm_records_metadata ADD COLUMN resource_type TEXT
    GENERATED ALWAYS AS (json_extract(json,
        '$.metadata.resource_type.id')) STORED;
ALTER TABLE rdm_records_metadata ADD COLUMN access_status TEXT
    GENERATED ALWAYS AS (json_extract(json, '$.access.status'))
        STORED;
ALTER TABLE rdm_records_metadata ADD COLUMN pub_year INTEGER
    GENERATED ALWAYS AS (CAST(strftime('%Y', json_extract(json,
        '$.metadata.publication_date')) AS INTEGER)) STORED;

CREATE INDEX idx_rdm_resource_type ON
        rdm_records_metadata(resource_type);
CREATE INDEX idx_rdm_access_status  ON
        rdm_records_metadata(access_status);
CREATE INDEX idx_rdm_pub_year       ON
        rdm_records_metadata(pub_year);
```

OpenSearch continues to serve search in Stage 1 and these columns are unused. In Stage 2 they become the facet source without any further migration.

Risk comparison

Risk	Stage 1 only	Stage 1 + 2
Search correctness regression	None (OpenSearch unchanged)	Medium (FTS5 parity)
Write concurrency issues	Medium	Medium (same)
Deployment complexity	Lower (still needs OpenSearch)	Lowest (single SQLite file)

Risk	Stage 1 only	Stage 1 + 2
Rollback path	Restore PostgreSQL dump	Restore PostgreSQL + reindex OS
Total calendar time	~3 months	~6 months

Stage 1 alone already delivers the primary value for local development and demo servers: PostgreSQL (the harder service to provision) is gone. OpenSearch running in a container alongside a SQLite-backed Invenio is a manageable setup and a meaningful intermediate state.

Phased Delivery Plan

Phase 1 (months 1–3): Stage 1 — `invenio-db-sqlite`. PostgreSQL replaced. OpenSearch and Redis remain. Validate deposit, publish, search round-trip. Ship as a dev/demo profile.

Phase 2 (months 4–6): Stage 2 — `invenio-search-sqlite`. OpenSearch dropped. FTS5 keyword search + 4–5 facets. Redis/Celery broker migrated to SQLite. Single-file deployment.

Phase 3 (months 6+): Packaging, `invenio-db-sqlite` + `invenio-search-sqlite` on PyPI, commonmeta-py integration for rich metadata, small-institution production hardening.

The Rust work is minimal (write triggers, schema cleanup). The bulk is Python. The commonmeta-rs SQLite schema is a strong template for the record data model; the biggest open question is whether InvenioRDM's community/request/user tables can be ported to SQLite without forking too deeply into each invenio-* package.