TECHNICAL WHITE PAPER

RathORM and Hibernate

Developer effort, query code, reusable projections, and cache behavior

Karthikeyan
CEO Tranz Technologies

7 October 2026

Measured query-body examples. Transparent setup costs. A clear distinction between code size, developer effort and runtime behavior.

RathORM 2.1.8-beta | Hibernate ORM 7.4.12.Final
Evidence-scoped comparison edition

Contents

1. Executive assessment

2. Scope and the meaning of "time taken"

3. Architecture and what is being reused

4. Cache audit: three meanings of entity reuse

5. Query workload and semantic contract

6. Measured query-body code counts

7. Time to write simple and complex queries

8. Existing local performance evidence

9. Runtime benchmark protocol

10. Complex-query loading and N+1

11. API development and projection reuse

12. Decision guide

13. Publication-ready claims and unresolved evidence

14. Reproduction and provenance

Appendices A-D: exact query bodies, shared declarations, generated SQL and collection templates.

Tranz Technologies | Technical white paper | 7 October 2026

Author: Karthikeyan, CEO Tranz Technologies

RathORM version: `2.1.8-beta`. Hibernate documentation baseline: ORM `7.4.12.Final`. Hibernate is the comparison target; Spring Data is discussed separately as an optional productivity layer.

Evidence status: query-body line counts are measured from supplied examples. Both frameworks' example files compile on Java 21; RathORM projection SQL was generated locally. Fifty existing targeted tests passed. Human implementation time, Hibernate execution latency, database throughput, and complete API line counts have not been measured. This paper does not claim a measured overall performance winner.

1. Executive assessment

RathORM has a credible productivity hypothesis for PostgreSQL read APIs: declare an API-facing projection once, reuse its columns and relationships, and express each endpoint's changing filters and pagination with a small query body. Its web integration can also reuse metadata for parsing filters, validating field names, converting identifiers, and building list responses. These mechanisms can reduce repeated application work across a family of endpoints.

That hypothesis contains three separate effects. Reusing a projection can reduce authoring effort. Caching an entity definition can reduce repeated runtime reflection. Caching query results can avoid eligible database reads. A result-cache hit does not itself reduce the time a developer spends designing an API, and metadata caching does not itself eliminate database queries. A useful white paper must keep all three effects separate.

In the exact examples accompanying this paper, RathORM query bodies use 4 versus 5 nonblank lines for a simple list, 5 versus 6 for a filtered list, and 6 versus 10 for a joined projection compared with concise Hibernate HQL. A nested projection uses 5 versus 13 lines compared with a Hibernate native-SQL body. These are limited call-site measurements. The nested comparison excludes Hibernate JSON decoding, and the starter declarations occupy 47 RathORM lines versus 19 Hibernate lines. The evidence therefore supports shorter displayed query bodies, not a blanket claim of fewer total application lines.

The strongest product message supported today is: **RathORM centralizes PostgreSQL read projections and relationship declarations so applications can reuse them across select queries and API endpoints.** A numerical claim such as "APIs are created five times faster" still requires a controlled developer study. A numerical runtime claim requires equivalent outputs, cache states, transactions, SQL plans, and database workloads.

2. Scope and the meaning of "time taken"

2.1 Development time

Development time is elapsed time from a defined task start to a correct, accepted implementation. For a query task, completion requires compiling, executing against the supplied fixture, validating outputs, and passing the acceptance checks. For an API task, it also requires request parsing, authorization scope, error handling, response serialization, pagination behavior, and tests. Merely typing a query expression is not task completion.

Measure setup time, first-query time, incremental-query time, validation time, and rework time separately. A reusable projection may impose a larger first-use cost while making later endpoints cheaper. The relevant business outcome is cumulative effort over the real endpoint portfolio, including change requests.

2.2 Runtime time

Runtime measurements answer different questions: startup and metadata initialization; construction and compilation of a query; connection acquisition; database execution; row decoding; nested JSON handling; cache lookup and copying; and complete HTTP response latency. The sum of these components determines the cost observed by API users.

For example, a metadata lookup improvement may be visible in a tight loop and negligible in an API dominated by a database scan. A result-cache hit may eliminate a database select but still require connection acquisition in an enclosing runtime scope, entity reconstruction, serialization, and a separate count query. Report each boundary explicitly.

2.3 What this edition measures

Metric	Status	Interpretation
Query-body nonblank LOC	Measured	Exact examples and fixed formatting
Shared declaration LOC	Measured	Cost visible alongside query bodies
RathORM source behavior	Inspected and partly test-verified	Describes this checkout
RathORM metadata timing	Single coarse test observation	Framework-internal diagnostic only
Developer minutes per task	Not measured	No defensible productivity multiplier yet
PostgreSQL query latency	Not measured	No comparative p50 or p95 claims
Complete API LOC	Not measured	Controller, policy, serializers and tests excluded
Hibernate snippets	Java 21 compilation passed	Not executed against a live database

3. Architecture and what is being reused

3.1 RathORM's model/projection split

The repository separates writable `Model` types from read-oriented `Entity` projections. A model declares physical table columns. A projection declares selected columns and aliases, a base table, joins, base criteria, and optional relationships. A query then adds the parameters and ordering relevant to an individual use case. The example classes use the instance query API, so their syntax requires no instrumentation plugin.

`AccountView` selects only account id and name. `AccountEntryView` selects those fields plus an entry id through an inner join. `AccountWithEntries` selects account fields plus a nested collection of visible entry ids. These declarations are reusable application assets. They remain useful even when result caching is disabled.

The generated SQL appendix confirms that the flat joined view becomes one SQL select with an inner join. The nested view becomes one select containing a correlated `json_agg` subquery with a bound visibility criterion. This verification checks metadata and SQL generation, not the execution plan or the returned payload of a live database.

3.2 Hibernate baseline

Hibernate maps persistent Java objects and offers HQL, Criteria, native SQL, and managed sessions. A fair query comparison uses its concise projection APIs instead of forcing every read through a verbose repository/service/DTO stack. Read-only use can also avoid some managed-state costs, and stateless access is an alternative. Hibernate Session API

For the supplied flat comparisons, Hibernate uses HQL projections into records. A Criteria alternative is shown for the filtered query because dynamic filter construction is a different authoring problem from a fixed HQL string. Native SQL is used for the nested JSON case to make a PostgreSQL aggregation strategy available on both sides. SelectionQuery API

3.3 Comparable layers

Keep a core comparison of RathORM query bodies versus Hibernate query bodies. Add a separate API comparison using the same HTTP framework, authentication policy, response schema, and transaction policy. Finally, evaluate existing team productivity layers: RathORM's `EntityService` on one side, and a reusable Hibernate query/endpoint layer on the other.

Spring Data JPA is an additional framework, not Hibernate itself. If a team already uses it, excluding its existing repositories and reusable Specifications could overstate RathORM's incremental advantage. Conversely, comparing a bare RathORM core query with an entire Hibernate controller and test suite is not a valid line-count comparison. Classify code by responsibility before counting it.

4. Cache audit: three meanings of entity reuse

4.1 Model metadata cache

`MetadataRegistry.tableOf()` uses a process-wide `ConcurrentHashMap` and `computeIfAbsent()` to parse and store model table metadata. Repeated resolution can avoid repeated reflection and validation. This is a runtime optimization; the model declaration still has to be authored. Local source: MetadataRegistry.java (framework source reference)

4.2 Entity-definition cache

`QueryGenerator.getEntityDefinition()` looks up an entity class in `ENTITY_CACHE`, parses a definition on a miss, and stores it. Parent-cycle detection uses a thread-local resolving set. Its miss path does not use `computeIfAbsent()`, so concurrent first resolutions can parse the same definition more than once. Do not describe this cache as an unconditional "exactly once" parser under concurrency.

The cached object is a definition, not a fully parameterized result and not proof of complete SQL compilation reuse. `getAll()` still builds a select from the definition. Cold-start metadata cost and warm select-generation cost should be measured independently. Local source: QueryGenerator.java (framework source reference)

4.3 Result cache

`DefaultResultCache` maintains a synchronized bounded LRU map with TTL expiration and reconstructs entity objects on reads. The default constructor configures 500 entries and a 60-second TTL. The cache is disabled in runtime configuration unless explicitly supplied. An entry is a result list, so 500 entries does not imply a limit of 500 rows or a specific memory budget.

More importantly, availability is not integration. `EntityQuery.list()` directly compiles and executes SQL without consulting `ResultCache`. The integration inspected here is in `EntityList.hydrate()`, reached

through eligible legacy/delegate query paths. That path first compiles SQL, then decides whether the result is cacheable. A warm result hit therefore does not eliminate all query-construction work.

4.4 Eligibility in this checkout

Read path	Result-cache behavior observed in source
`EntityQuery.of(...).list()`	No result-cache lookup
`EntityDelegate.findAll()` on a plain `@Cached` projection	Can be eligible if runtime cache enabled
`EntityList` with an explicit filter	Ineligible
`EntityList` with required parent criteria	Ineligible
`EntityList` with recursive anchor criteria	Ineligible
`EntityList` with ordering or grouping	Ineligible
`EntityList` with a nonempty limit/offset	Ineligible
`EntityList` with dynamic columns	Ineligible
Active managed write transaction	Cache read bypassed on this integration path
Already hydrated `EntityList` instance	Reuses that instance's delegate list

The annotation is detected by `EntityDelegate.findAll()`. Other `find()` construction paths pass `false` for the cache flag. Base ordering or grouping can also make an annotated `findAll()` ineligible. Static `@BaseCriteria` is part of the definition and compiled SQL; eligibility only checks the runtime criteria fields. Do not generalize the runtime filter exclusion into a claim that every projection with base criteria is ineligible. Local sources: EntityDelegate.java (framework source reference), EntityList.java (framework source reference)

4.5 Correctness boundaries to disclose

The cache key type supports runtime, database, schema, entity, SQL, and parameters. However, the inspected `EntityList` integration supplies literal database `default` and schema `public`, alongside a runtime identity string. This is not evidence of automatically deriving actual catalog/schema or tenant context. A runtime reused across changing schema, role, or row-level-security context needs an isolation test before publication.

The documentation describes deep snapshots, but the implementation copies top-level maps with `new LinkedHashMap<>(entity.toMap())`. Reads copy the top-level snapshot again. Nested maps, lists, dates, arrays, or other mutable values can remain shared. The existing test verifies top-level id isolation; it does not establish arbitrary deep isolation. A nested-result cache claim needs mutation tests and, if necessary, a repair.

`RathOrmRuntime` clears its configured cache on the successful owning transactional commit path. External transaction owners, writes by other services, and other runtime instances must be examined separately. A TTL bounds eventual expiration; it does not make every cache hit consistent with every concurrent write. Whole-cache invalidation is simple but can reduce hit rates in write-heavy workloads.

All `DefaultResultCache` methods that touch entries are synchronized. Copying large results while holding a cache lock may become a concurrency cost. This is a reason to benchmark contention and payload size, not a claim that the implementation is slow. Cache stampedes and maximum result size also deserve explicit tests. Local source: DefaultResultCache.java (framework source reference)

4.6 Hibernate cache comparison

Hibernate separates persistence-context caching, optional second-level caching, and query-result caching. Query caching needs configuration and per-query opt-in; cache layout may store full fetched data or identifiers requiring further cache/database retrieval. Its query-cache validation uses table-update timestamps. A comparison must report provider, layout, invalidation policy, and observed hit rate rather than saying Hibernate has no entity cache. Hibernate caching documentation

These mechanisms are not identical to RathORM projection reuse. One is a reusable application representation, another is retained mapping metadata, and another is retained data. The developer study tests authoring reuse; the runtime study tests metadata and retained-data behavior separately.

5. Query workload and semantic contract

Use a small accounts/entries domain to make both implementations inspectable. An account has `id`, `name`, and `active`; an entry has `id`, `account_id`, and `visible`. Do not let one implementation read all columns while the other selects only two. Table contents, schema, indexes, bindings, page size, output fields, and sorting must agree.

Case	Task	Output contract	Pagination unit
S1	First 25 accounts	id and name, ordered by id	Accounts
S2	First 25 accounts matching active flag	Same fields, bound flag, ordered by id	Accounts
C1	Join active accounts to visible entries	account id, name, entry id; order by both ids	Joined rows
C2	Active accounts with visible entry ids nested	Parent id/name and child collection	Accounts

C1 and C2 are both complex reads but solve different problems. C1 returns up to 25 joined rows and may repeat a parent. C2 returns up to 25 parents, each with its child collection. Comparing their runtimes directly as interchangeable output shapes would be invalid. Every result must pass the case-specific contract before timing is accepted.

For C2, normalize an absent collection consistently: the generated `json_agg` can yield SQL null when no child matches. Decide whether the API returns null or an empty list. The sample does not impose child ordering; treat child ids as a set during semantic validation or add explicit ordering on both sides and recount the code. Duplicates must still be detected. Highly skewed child counts should be represented in the dataset.

5.1 Additional deep-comparison tasks

The initial examples cover simple reads and two representative relationship shapes. The full evaluation should additionally include: dynamic combinations of filters; a three-table projection; grouped revenue totals with HAVING; a correlated existence filter; a recursive hierarchy; keyset pagination; exporting a large stream; and changing an existing API projection after release. For each, define results and acceptance tests first. Do not publish assumed LOC or minutes for tasks not implemented.

Complexity is multidimensional. Number of tables, dynamic predicates, output nesting, policy rules, aggregation semantics, pagination rules, and data volume all matter. A ten-line query can be harder to validate than a forty-line one. Report complexity attributes rather than relying only on the word "complex."

6. Measured query-body code counts

The counting rule is physical nonblank lines, including braces where present and each SQL text-block line. Query-body snippets exclude imports, class and method declarations, shared mappings/projections, connection/transaction wrappers, request handling, serialization, tests, and dependency configuration. Formatting is fixed by the supplied generator; compressing fluent chains or SQL onto one line would change this metric without necessarily improving maintainability.

Case	RathORM	Hibernate baseline	Hibernate LOC	Body reduction
S1 simple list	4	HQL projection	5	20.0%
S2 filtered list	5	HQL projection	6	16.7%
S2 filtered list alternative	5	Criteria projection	9	44.4%
C1 joined projection	6	HQL projection	10	40.0%
C2 nested children	5	Native SQL with JSON aggregate	13	61.5%

Body reduction is `(Hibernate body LOC - RathORM body LOC) / Hibernate body LOC`. The Criteria alternative is not added to the HQL baseline to inflate it. The C2 figure compares query bodies only: RathORM's projection path decodes relationships through its normal handler, while the Hibernate body returns `Object[]` with a JSON column and does not include JSON-to-response conversion. It is not an equivalent complete-API LOC result.

6.1 Shared setup and total effort

Across the supplied suite, RathORM declarations use **47** nonblank lines, compared with **19** Hibernate lines. RathORM explicitly declares both models and multiple read projections. The Hibernate example uses table-mapped field-access classes and two projection records; its C2 query uses native SQL and a raw row representation. No getters/setters are added solely to make Hibernate look longer.

The four primary RathORM bodies total 20 lines. The matching Hibernate bodies total 34. Adding all shared declarations gives 67 versus 53 lines before method signatures/imports and omitted API concerns. This starter-suite subtotal is larger for RathORM despite the shorter query bodies. It is not a full implementation count, and the C2 representation difference remains. It demonstrates why setup cost must be included and why repeated use matters.

Do not attribute every line of shared setup to each query. Report one-time shared cost and incremental endpoint cost. Publish separate totals for query definitions, mappings, DTOs/projections, endpoint code, framework configuration, generated code, validation, and tests. Generated code should be recorded separately from human-authored source; generated SQL still matters to correctness and runtime.

6.2 Break-even model

Let `S_R` and `S_H` be setup effort, and `q_R` and `q_H` be incremental effort for semantically equivalent uses. Then total effort for n uses is `S_R + n*q_R` versus `S_H + n*q_H`. If `q_H > q_R`, the break-even point is `n > (S_R - S_H)/(q_H - q_R)`. Use minutes to analyze productivity and LOC to analyze code size; never exchange one for the other.

For an illustrative code-size calculation only, if setup differs by 28 lines and each additional equivalent query saves 4 lines, the setup difference is offset after seven such uses and total RathORM code becomes smaller on the eighth. This is arithmetic, not a forecast: projections may be shared unevenly, query bodies may be reused on both sides, and equivalent response adapters alter setup. The real study should use observed setup and marginal effort for its actual endpoint portfolio.

7. Time to write simple and complex queries

7.1 Current results table

Task	RathORM developer minutes	Hibernate developer minutes	Comparative runtime
S1 simple list	Not measured	Not measured	Not measured
S2 filtered list	Not measured	Not measured	Not measured
C1 joined projection	Not measured	Not measured	Not measured
C2 nested API read	Not measured	Not measured	Not measured
Reuse existing projection in another API	Not measured	Not measured	Not measured

These fields are deliberately unfilled. Typing a generated sample with an assistant, estimating minutes from LOC, or timing a single expert familiar only with RathORM would not support a general human-productivity claim. The paper includes data collection templates so measured values can replace these fields without changing their meaning.

7.2 Controlled developer study

Recruit developers with recorded Java, PostgreSQL, RathORM, and Hibernate familiarity. Use a pilot to identify unclear tasks and estimate variability, then choose a sample size based on the effect size the study intends to detect. A small convenience sample can provide exploratory evidence but should not be presented as population-level proof.

Use a paired crossover design where practical. Developers perform equivalent but nonidentical tasks with both frameworks. Randomize framework order and counterbalance task variants so memory of the first solution does not determine the second. Provide equivalent documentation, editor support, and warm-up exercises. Freeze whether assistants, autocomplete, copied templates, and repository scaffolding are permitted.

Run two task conditions. In the first-use condition, developers begin with the common application shell and implement mappings, projections, and endpoint logic. In the reuse condition, equivalent domain mappings and reusable query components already exist; developers implement an additional read or API variation. Do not give RathORM an existing projection while withholding an equivalent reusable Hibernate query component.

Record task start, first compiling implementation, first accepted output, final acceptance, active time, interruptions, debugging, and rework. Mark incomplete tasks as incomplete with their observed duration; do not silently exclude them or replace them with an arbitrary successful time. Keep the task timer running through framework-caused debugging and verification.

For the API condition, acceptance should include valid and invalid filters, maximum page size, stable ordering, empty results, malformed id handling, not-found behavior, authorization scope, a parent with zero children, and

response JSON equality. The task ends when this contract passes. Session recording can support auditability with participant consent.

7.3 Analysis and claim wording

Report medians, distributions, paired differences, confidence intervals, completion rate, defects, and experience strata. Separate framework setup from marginal query/API effort. Predefine exclusion rules. Report both task-level effects and portfolio-weighted effects using a declared workload mix.

Productivity reduction is `(T_H - T_R)/T_H`; speed ratio is `T_H/T_R`. A 50% time reduction corresponds to twice as fast for that measured task, not to twice as fast for every application. Publish sample size, baseline versions, task scope, and whether existing projections were provided alongside any percentage.

The strongest future claim will likely be conditional: "For this repeated PostgreSQL read-API workload, developers using the existing RathORM projections completed accepted endpoints in X median minutes versus Y under the specified Hibernate baseline." Until those values exist, use mechanism-based wording and the measured snippet counts.

8. Existing local performance evidence

The existing targeted test command was run on macOS `27.0.1`, aarch64, Temurin Java `21.0.7`, with a reported maximum JVM heap of 6144 MB:

```
./mvnw -o
-Dtest=ResultCacheTest,QueryApiTest,QueryContractTest,PerformanceAssessmentTest,WebAdapterTest test
50 tests; 0 failures; 0 errors; 0 skipped
10,000 uncached model parses: 35 ms
10,000 cached metadata resolutions: 1 ms
10,000 entity-definition resolutions: reported 0 ms
10,000 select-generation iterations: 2 ms
```

The 0 ms observation means below the displayed whole-millisecond resolution, not zero computation. The test uses one ordered timing run without a controlled Hibernate baseline. JIT warm-up, cache state, iteration ordering, and timer truncation affect interpretation. Do not convert these numbers into production request latency, a reliable 35-times speed multiplier, or developer implementation minutes.

Other assessment tests use a recording JDBC double to verify statement counts and resource behavior. Such tests provide useful evidence that a generated relationship query is sent as one SQL statement in that path. They do not measure PostgreSQL network latency, real server CPU, disk I/O, transaction isolation, index selection, or throughput. Existing test source (framework source reference)

9. Runtime benchmark protocol

9.1 Environment and isolation

Use a disposable PostgreSQL benchmark database, never the application database referenced by local credentials. Pin JDK, JVM flags, heap, GC, PostgreSQL version, JDBC driver, pool size, hardware, operating system, and framework artifacts. Record dataset seed, index definitions, and exact cache configuration. Use the same serialization library and response payload contract in HTTP comparisons.

Match transaction semantics before matching code syntax. Specify whether reads run in autocommit or an explicit read transaction and the isolation level. A convenient scope wrapper on one side must not silently add commits or change snapshot semantics relative to the other. Measure connection acquisition both separately and within end-to-end request latency.

Warm up separately from measurement, and record the warm-up procedure. Execute multiple independent process runs, alternate framework order, and retain distributions. A JMH-style harness is suitable for JVM-only construction and decoding; database benchmarks need a sustained harness with explicit database and pool state. Do not time an unconsumed lazy query object.

9.2 Cache-state matrix

Experiment	Metadata	Result cache	Purpose
Cold application	Cold	Empty/disabled	Startup and first accepted query
Warm, uncached S1-S2	Warm	Disabled on both sides	Baseline query/read cost
Warm, uncached C1-C2	Warm	Disabled on both sides	Relationship SQL and decoding
Eligible cached lookup	Warm	Warm on both configured sides	Retained-result benefit
Cache miss after invalidation	Warm	Cleared	Recovery cost
Write-heavy read mix	Warm	Enabled where supported	Realistic hit rate and consistency
Large nested payload	Warm	Enabled/disabled separately	Copying, memory, contention

The supplied S1-S2-C1-C2 RathORM bodies use `EntityQuery`, so they do not exercise its result cache. Add a separately identified unfiltered, unordered `@Cached` delegate case for a cache-hit experiment. Do not relabel a warm metadata query as a warm result-cache query. Hibernate's provider and query-cache settings must also be explicit.

9.3 Dataset design

Use at least small, medium, and large fixtures, such as 1,000, 100,000, and 1,000,000 accounts, with declared entry counts and deterministic seeds. These are proposed dataset sizes, not datasets already run. Include zero-child parents, median fan-out, high fan-out, and a skewed tail. Vary active/visible selectivity and repeated-parameter popularity rather than repeatedly requesting one trivial row.

Publish DDL and indexes and run ANALYZE after seeding. For relationship reads, start with an index supporting `entries.account_id` and evaluate whether the visibility predicate changes the best index. Keep database settings and statistics equal. Record plans for every query with `EXPLAIN (ANALYZE, BUFFERS)` outside the timed application loop.

9.4 Metrics

Collect p50, p95, p99, throughput, failures, rows returned, bytes serialized, SQL statements, database CPU, buffer hits/reads, connection wait, allocation, GC, memory retained, and cache hit/miss/invalidation counts. Collect query construction, database work, materialization, and serialization independently when instrumentation permits.

A cache hit can reduce select statements to zero for an eligible read while still requiring nonzero CPU and response work. A nested single statement can be slower than a carefully batched alternative at very high fan-out. A lower average can hide worse tail latency under contention. Report tradeoffs alongside wins.

10. Complex-query loading and N+1

RathORM's inspected `@SubEntity` implementation emits a correlated JSON aggregation expression inside the parent select. That avoids an application round trip per parent for this relationship shape. It can also avoid repeating wide parent columns on every child row, although JSON encoding/decoding introduces its own costs.

The inspected generator intentionally builds child core selects without recursively including further relationship subqueries. Therefore, the one-statement example does not establish arbitrary-depth nested graph loading. Include a depth-two workload before claiming multi-level graph coverage. Parent pagination, child criteria, child ordering, and null normalization need individual acceptance checks.

Hibernate supports several fetch strategies and native SQL; N+1 is a workload/configuration outcome, not a mandatory Hibernate behavior. The HQL 7.4 documentation says collection fetch-join pagination is fixed on databases supporting limits and offsets in subqueries. PostgreSQL comparisons must inspect actual generated SQL rather than repeating older universal in-memory-pagination claims. Hibernate HQL guide

For C2, benchmark a native JSON projection, an appropriately fetched graph, and a batched alternative where they satisfy the same API contract. Keep managed graphs distinct from flat or nested DTO responses. An intentionally naive lazy loop may be useful as an educational failure case, but it must not be the only Hibernate control.

11. API development and projection reuse

11.1 Work that RathORM can centralize

`EntityService` combines field allowlisting, request criteria parsing, ordering, bounded pagination, list execution, total counts, id conversion, and response envelopes. `FieldAllowlist` derives names from metadata or accepts explicit fields. These components can remove repeated plumbing from a portfolio of similar read endpoints. Local source: EntityService.java (framework source reference)

That productivity comes from reusable API machinery and entity definitions. A result cache may accelerate an eligible lookup, but it does not implement access policy, decide the response contract, or validate a new endpoint. Attribute the benefit to the component doing the work.

11.2 Work still required

A production endpoint must define routing, authentication, tenant scope, field exposure, business validation, response semantics, error reporting, transaction ownership, and tests. Metadata-derived allowlisting is not proof that every exposed field is appropriate for every role or that every allowed filter is indexed. A custom allowlist may be required for a public API even when the entity contains more columns.

The inspected list service fetches rows and also executes `EntityDelegate.count()`. A "single-query endpoint" claim would therefore be inaccurate for this list response: the nested data select may be one statement while the response adds another count statement. Count cost can dominate complex/grouped queries and should be included in the API benchmark.

The service accepts subentity-related arguments in its public signature, but the inspected `get()` method dispatches only to list or fetch-by-id. Do not infer implemented subentity routing from the parameter names alone. Composite-key `findById` is explicitly rejected in the query generator; include that limitation in capability comparisons rather than demonstrating only single-key models.

11.3 Reuse experiment

Build five equivalent endpoint variations over an established projection: a base list, a bound filter, a new sort option, a count-bearing response, and an export. Then introduce a change to a selected field and a relationship criterion. Record first-use and marginal effort, affected files, regression defects, and tests changed.

Provide equivalent reusable components to both teams, or explicitly label a second experiment as evaluating each framework's supplied integration layer. Centralization can improve consistency, but a defect in a shared projection can also affect several endpoints. Include change impact and review effort in the study.

12. Decision guide

Project condition	Implication for evaluation
Many similar PostgreSQL read projections	RathORM reuse is a strong hypothesis to test
Small number of simple reads	Setup cost may dominate any shorter query body
Existing mature Hibernate query/API layer	Measure incremental migration value against that layer
Repeated full-list lookup with stable data	Inspect result-cache eligibility and stale-data tolerance
Highly dynamic filters and pagination	Do not assume current RathORM result-cache coverage
Very large child collections	Compare JSON aggregation, fetched rows, batching and payload limits
Rich domain updates	Evaluate update semantics and business workflow separately from read LOC
Changing schema/role in a shared runtime	Verify cache-context isolation before enabling result reuse
Nested mutable cached values	Verify deep isolation before making safety claims

The framework choice should follow the application's actual workload and team. A read-API paper cannot establish superiority for every write workflow, migration strategy, or deployment topology. Conversely, a broad feature checklist does not substitute for measuring repeated projection work in the use case RathORM targets.

13. Publication-ready claims and unresolved evidence

Supported now

The local source defines reusable entity projections, cached metadata, and a generic read-service layer. Generated example SQL demonstrates an inner join and one-level nested JSON aggregation. The fixed-format query bodies are smaller than the displayed Hibernate alternatives, and the shared setup is larger. Fifty targeted tests passed in this run.

Needs measurement or repair before a stronger claim

Developer-time improvements need controlled task results. Runtime improvements need a live database comparison. Complete API code savings need semantically complete endpoint implementations on both sides. Result-cache safety needs nested mutation and context-isolation coverage. Broad cache availability needs integration/eligibility evidence for the exact query path being advertised. Multi-level nested relationships need their own implementation and tests.

Suggested publication language

"RathORM makes PostgreSQL read projections reusable across queries and API endpoints. In the supplied examples, the query bodies require fewer nonblank lines than the compared Hibernate HQL or native-SQL bodies. Shared setup and complete endpoint behavior are separate costs. Developer-time and end-to-end runtime improvements remain to be established through controlled measurements."

Avoid "Hibernate always needs N+1," "all RathORM selects are cached," "zero runtime overhead," "35 times faster than Hibernate," and "61.5% less API code." Each would extend beyond the evidence in this edition.

14. Reproduction and provenance

Run the existing targeted test command from the repository root. Run `python3 docs/comparison/tools/create_examples.py` to regenerate examples and `loc-results.json`. Run `python3 docs/comparison/tools/verify_examples.py` after the tests to compile the RathORM examples using the existing test classpath and regenerate SQL/provenance records. With the reference Hibernate and Jakarta jars in the scratch directory, that command also compiles the Hibernate examples. That probe does not connect to PostgreSQL.

This Tranz Technologies publication preserves the supplied comparison findings and examples. Reproduction commands refer to the original framework study workspace; raw example files, test logs and study templates are not included in this website edition. Artifacts describe the recorded source baseline; changing the production code requires rechecking the claims and regenerated examples. The repository README references an older dependency version, so this edition takes `pom.xml` and inspected source as its local baseline.

Primary external sources were accessed on 7 October 2026: Hibernate releases, Hibernate introduction, HQL guide, ORM user guide, Session API, and SelectionQuery API. Supplied Hibernate snippets compile against Hibernate ORM 7.4.12.Final and Jakarta Persistence 3.2.0. Compilation and documentation review are not replacements for live database acceptance tests.

Appendix A. Exact query bodies

The blocks below are generated from `loc-results.json`. The accompanying Java files contain imports, shared declarations, and method signatures. RathORM methods must run inside an appropriate runtime/connection scope; Hibernate methods receive an active session. Neither standalone example file initializes a database or constitutes a complete API.

S1_RathORM (4 nonblank lines)

```
return EntityQuery.of(AccountView.class)
    .orderBy(OrderBy.asc("a.id"))
    .limit(25)
    .list();
```

S1_Hibernate_HQL (5 nonblank lines)

```
return session.createSelectionQuery(
    "select a.id, a.name from Account a order by a.id",
    AccountRow.class)
    .setMaxResults(25)
    .getResultList();
```

S2_RathORM (5 nonblank lines)

```
return EntityQuery.of(AccountView.class)
    .where(WhereCriteria.eq("a.active", active))
    .orderBy(OrderBy.asc("a.id"))
    .limit(25)
    .list();
```

S2_Hibernate_HQL (6 nonblank lines)

```
return session.createSelectionQuery(
    "select a.id, a.name from Account a where a.active = :active order by a.id",
    AccountRow.class)
    .setParameter("active", active)
    .setMaxResults(25)
    .getResultList();
```

S2_Hibernate_Criteria (9 nonblank lines)

```
var cb = session.getCriteriaBuilder();
var cq = cb.createQuery(AccountRow.class);
var a = cq.from(Account.class);
cq.select(cb.construct(AccountRow.class, a.get("id"), a.get("name")));
cq.where(cb.equal(a.get("active"), active));
cq.orderBy(cb.asc(a.get("id")));
return session.createSelectionQuery(cq)
    .setMaxResults(25)
    .getResultList();
```

C1_RathORM (6 nonblank lines)

```
return EntityQuery.of(AccountEntryView.class)
    .where(WhereCriteria.eq("a.active", true)
        .and(WhereCriteria.eq("e.visible", true)))
    .orderBy(OrderBy.asc("a.id").thenAsc("e.id"))
    .limit(25)
    .list();
```

C1_Hibernate_HQL (10 nonblank lines)

```
return session.createSelectionQuery("""
    select a.id, a.name, e.id
    from Account a join Entry e on e.accountId = a.id
    where a.active = :active and e.visible = :visible
    order by a.id, e.id
    """, AccountEntryRow.class)
    .setParameter("active", true)
    .setParameter("visible", true)
    .setMaxResults(25)
    .getResultList();
```

C2_RathORM (5 nonblank lines)

```
return EntityQuery.of(AccountWithEntries.class)
    .where(WhereCriteria.eq("a.active", true))
    .orderBy(OrderBy.asc("a.id"))
    .limit(25)
    .list();
```

C2_Hibernate_Native (13 nonblank lines)

```
return session.createNativeQuery("""
    select a.id, a.name,
        (select json_agg(child)
         from (select e.id from entries e
               where e.account_id = a.id and e.visible = :visible) child) as entries
    from accounts a
    where a.active = :active
    order by a.id
    """, Object[].class)
    .setParameter("visible", true)
    .setParameter("active", true)
    .setMaxResults(25)
    .getResultList();
```

Appendix B. Shared declarations

RathORM declarations (47 nonblank lines)

```
@Table(name = "accounts")
public static class Account extends Model {
    @DbColumn(type = SqlType.BIGINT, primaryKey = true)
    public static final String id = "id";
    @DbColumn(type = SqlType.VARCHAR, length = 100)
    public static final String name = "name";
    @DbColumn(type = SqlType.BOOLEAN)
    public static final String active = "active";
}

@Table(name = "entries")
public static class Entry extends Model {
    @DbColumn(type = SqlType.BIGINT, primaryKey = true)
    public static final String id = "id";
    @DbColumn(type = SqlType.BIGINT)
    public static final String account_id = "account_id";
    @DbColumn(type = SqlType.BOOLEAN)
    public static final String visible = "visible";
}

public static class AccountView extends Entity {
    @BaseTable(alias = "a")
    public static EntityTable<Account> account = () -> columns(
        column("a", Account.id, "id"),
        column("a", Account.name, "name"));
}

public static class AccountEntryView extends Entity {
    @BaseTable(alias = "a")
    public static EntityTable<Account> account = () -> columns(
        column("a", Account.id, "id"),
        column("a", Account.name, "name"));
    @ChildTable(alias = "e", conditions = "e.account_id = a.id", join = Join.INNER)
    public static EntityTable<Entry> entry = () -> columns(
        column("e", Entry.id, "entry_id"));
}

public static class VisibleEntryView extends Entity {
    @BaseTable(alias = "e")
    public static EntityTable<Entry> entry = () -> columns(column("e", Entry.id, "id"));
    @BaseCriteria
    public static WhereCriteria visible = WhereCriteria.eq("e.visible", true);
}

public static class AccountWithEntries extends Entity {
    @BaseTable(alias = "a")
    public static EntityTable<Account> account = () -> columns(
        column("a", Account.id, "id"),
        column("a", Account.name, "name"));
    @SubEntity(conditions = "e.account_id = a.id")
    public Class<VisibleEntryView> entries = VisibleEntryView.class;
}
```

Hibernate declarations (19 nonblank lines)

```
@Entity(name = "Account")
@Table(name = "accounts")
public static class Account {
    @Id
    public Long id;
    public String name;
    public Boolean active;
}

@Entity(name = "Entry")
@Table(name = "entries")
public static class Entry {
    @Id
    public Long id;
    @Column(name = "account_id")
    public Long accountId;
    public Boolean visible;
}

public record AccountRow(Long id, String name) {}
public record AccountEntryRow(Long id, String name, Long entryId) {}
```

Appendix C. Actual generated RathORM SQL

```
AccountView: Select a.id As "id", a.name As "name" From "public"."accounts" as "a"
Parameters: []
AccountEntryView: Select a.id As "id", a.name As "name", e.id As "entry_id" From "public"."accounts"
 as "a" Inner Join "public"."entries" as "e" On e.account_id = a.id
Parameters: []
AccountWithEntries: Select a.id As "id", a.name As "name", (Select Json_Agg(Sub) From (Select e.id
As "id" From "public"."entries" as "e" Where e.account_id = a.id And ("e"."visible" = ?)) Sub) As
"entries" From "public"."accounts" as "a"
Parameters: [true]
```

Appendix D. Collection templates

`developer-study.csv` records participant, framework, task, condition, durations, completion and validation. `runtime-study.csv` records per-run environment, workload, cache state, latency, throughput, errors and resource metrics. Empty cells mean no observation, not zero. The templates are not benchmark results.

Maintain an evidence ledger with four statuses: source-confirmed mechanism, locally verified behavior, comparative observation, and untested hypothesis. Every figure in the final public edition should link to a raw record or executable example. Keep the honest limitations visible in the video as well as the paper.