Event Sourcing vs. CRUD Systems with Audit Log

1 Research Question

How does an Event Sourcing architecture compare to CRUD systems with an independent audit log, when it comes to scalability, performance and traceability?

2 Problem Statement

Traceability of data changes and user actions is often a core, non-functional requirement for business and compliance purposes. In traditional CRUD (Create, Read, Update, Delete) systems, this is typically implemented via an **Audit Log**, which is a secondary mechanism. While functional (and often sufficient), this approach has several downsides:

- **Secondary Source of Truth and Data Loss:** The Audit Log is a secondary source of truth, where the "real" state resides in the primary entity tables. This introduces complexity, as the audit log relies on application logic (or database triggers) to correctly capture changes, creating a constant risk of **divergence and data loss** if logging fails or is incomplete.
- **Loss of Business Intent:** Most standard audit logs only log technical differences (e.g., `oldValue: 10`, `newValue: 20`). They do not store the **business intent** of the action (the *reason* a change was made). For instance, an `UPDATE` on a `balance` field could represent a "Refund," a "Fee Charged," or a "Manual Correction." This lack of context makes historical analysis and debugging significantly harder.
- **Inflexible Reconstruction:** It is computationally difficult to reconstruct a complete, reliable historical state from an Audit Log, as it requires complex `JOIN`s between the audit records and the latest entity snapshot. This difficulty is increased when business logic evolves, because re-interpreting old log entries according to new rules is error-prone.

Event Sourcing as a Solution

Event Sourcing offers a fundamentally different approach: Immutable events are appended to a **write-only Event Log**, which is the **single source of truth**. The correct state of the system at any point in time can reliably be reconstructed by replaying every event, inherently retaining all history and business intent. Event Sourcing is typically implemented together with **Domain Driven Design**, which is an approach to software development that attempts to design the software as closely to the business domain as possible.

The Actual Problem

While ES systems promise a reliable historical state and the preservation of business intent, they come with increased complexity in their architecture and implementation.

CRUD systems, on the other hand, are inherently simpler and the Audit Log might be enough for many use cases. This thesis aims to quantify the differences and tradeoffs of the two approaches in terms of performance, traceability and scalability (database scaling).

3 Project

To demonstrate the topics and answer the research question, I want to evaluate a course enrollment and grading system which I will build twice. One implementation will be built using event sourcing and DDD, the other one will be built as a simple layered CRUD architecture with a standard audit log (storing `entityId`, `userId`, `operation`, `oldValue`, `newValue`).

To make realistic assumptions about performance and flexibility of both approaches in production, I want to make the project requirements complex enough to a point where queries and relationships involve several tables (see proposed use cases below).

I will focus on the use cases that are relevant to my research question and leave out other parts that would otherwise be critical in a production system (e.g. authentication, middlewares, security).

I want to follow the recommended ES / CQRS approach of materializing projections into a database to speed up state reads.

Use Cases

1. `EnrollmentCommand`
   During course registration week, hundreds of students enroll in and drop courses while instructors update grades.
   Goal: Measure performance of concurrent writes on the same entity and read-heavy loads like listing all students in a course
2. `SumAllCreditsQuery`
   Sum up all credits a student has accumulated from their finished courses
   Goal: Compare time to query a `StudentCredits` projection vs. the time it takes to `JOIN` the tables and `SUM` the credits in the CRUD implementation.
3. `ShowHistoricalGradesQuery`
   Show all grade changes for student #123 in Mathematics between April 1 and June 1, including who made the change and what the grade was at each point in time.
   Goal: Measure the time required to reconstruct the historical state and the business intent (who, why) from the Audit Log versus the time required to replay the event stream.

Tech Stack

- **Database:** PostgreSQL for both implementations to ensure a fair comparison that is not skewed by differences in the underlying technology

- **Backend:** SpringBoot
  - Axon for the EventSourcing implementation
- **Testcontainers**
  - Axon Server Container
  - PostgreSQL Container

4 Proposed Outline

1. **Basics**
   1. Motivation
   2. Goal of the thesis
   3. Structure of the paper
2. **Related Work / Literature Research**
   - Layered Architecture Foundations
   - DDD Architectural Foundations
   - Event Sourcing and Event-driven Architecture
     - Formal definition (immutability of events & event stream as single source of truth)
   - Differences between replaying an audit log and an event stream
   - concurrency and consistency
     - ACID vs BASE
     - concurrency control: optimistic locking vs. pessimistic locking
     - how the aggregate version number is relevant for optimistic concurrency control
     - ways to scale reads and writes
3. **Proposed Method**
   1. Project outline
   2. Project requirements
   3. Tech stack
   4. Measurement methods:
      Benchmarking via load testing. Time taken per request (Median, P95); CPU utilization, database IO; Database size for a fixed number of operations; …
   5. Demonstrate concurrency control and ways to scale the database

4. **Results**
    1. Performance Benchmarking Results
    2. Traceability and Replay Analysis
    3. Concurrency and Scalability Analysis
5. **Discussion / Future Work**