

Data models and query languages

Тимофей (@timofey256)

26.03.2026

What the subchapter was about?

- Data Models for Graph Data
- Storing Matrices, DataFrames, Arrays
- **Event-sourcing and CQRS.**

What the subchapter was about?

- Data Models for Graph Data
- Storing Matrices, DataFrames, Arrays
- **Event-sourcing and CQRS.**

Let's talk about:

- When to use ES.
- Common approaches to build ES-driven systems
- CQRS
- Challenges in ES systems.

Problem: mutable state

- Traditional approach: overwrite current state in the DB
- We lose: audit history, past state reconstruction, temporal queries

Problem: mutable state

- Traditional approach: overwrite current state in the DB
- We lose: audit history, past state reconstruction, temporal queries

Example: "Your bank account shows €500. But how did it get there?"

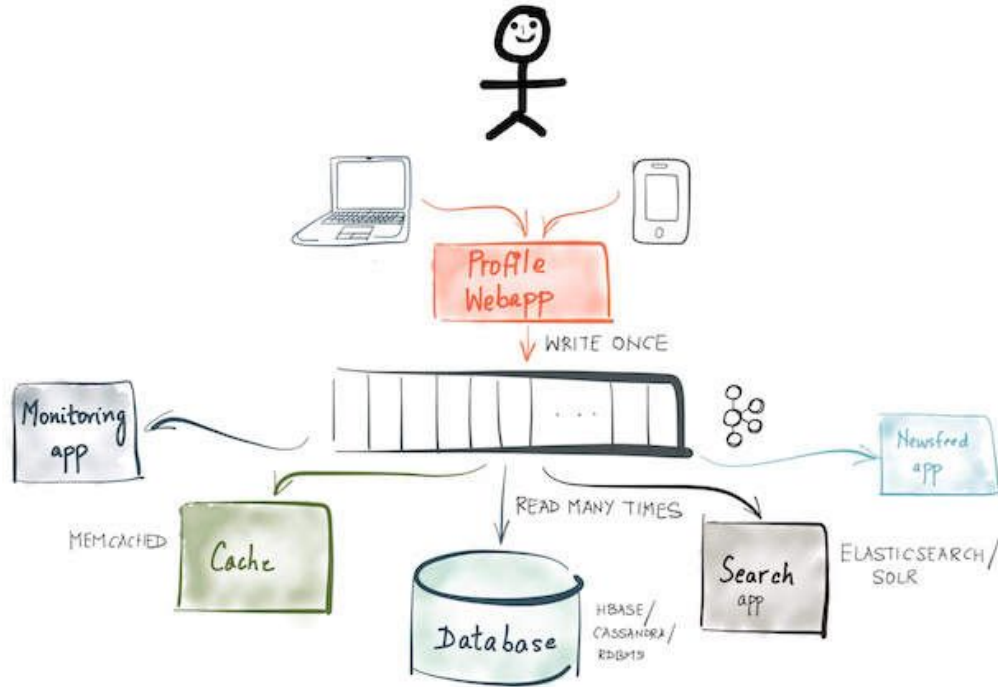
Solution: Event sourcing

- Instead of storing current state, append *immutable* events.
- State is derived, not stored.

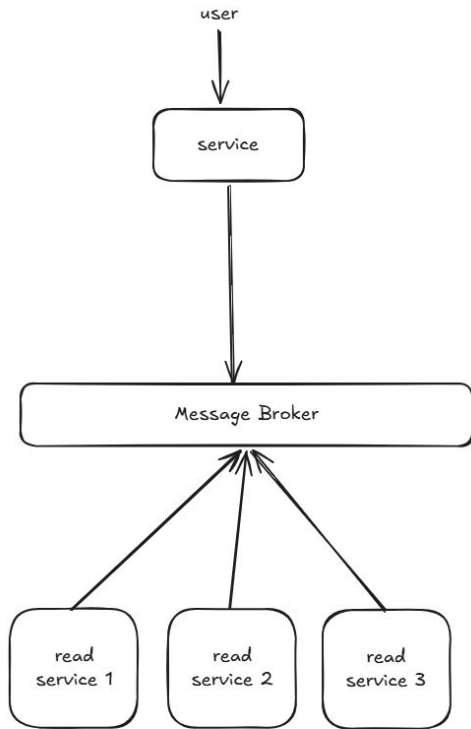
Example: no separate "balance" column:

$$\text{balance} = \text{sum}(\text{'Deposited'}) - \text{sum}(\text{'Withdrawn'})$$

What does it look like (conceptually)?



What does it look like (concretely)?



Approach 1: Only Kafka

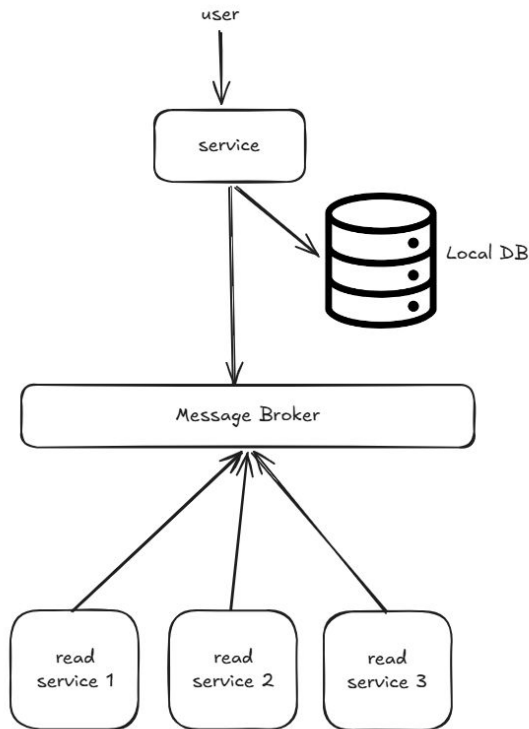
Good:

- Simple
- Familiar
- Event-driven (with all benefits of this)

Bad:

- No atomicity.
- No guarantee of publish succeeding.
- Duplicates.
- Ordering guarantee only on partition level.
- Concurrency problems
- Event-driven but NOT event-sourced

What does it look like (concretely)?



Approach 2: Save to database and publish to Kafka

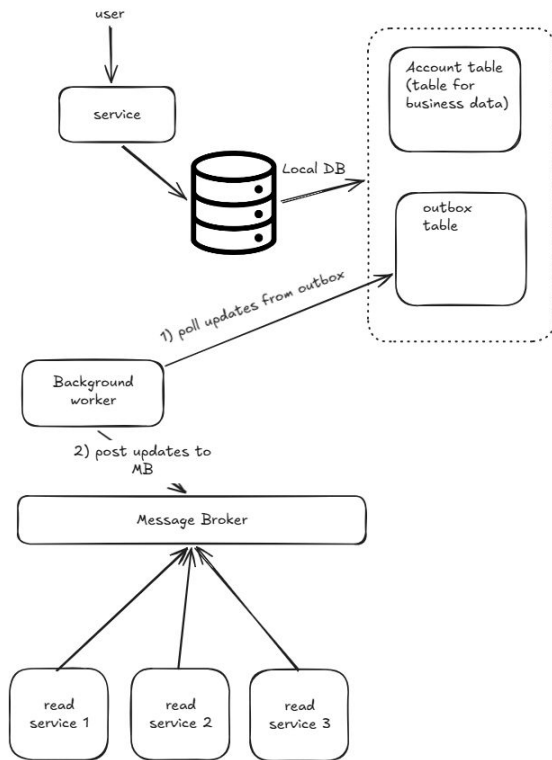
Good:

- Local durability => source of truth
- Support for constraints within the aggregate.
- Simple reads

Bad:

- Dual-writes => inconsistency + hard recovery
- Still not clear how to handle concurrency problems.
- Event-driven but NOT event-sourced

What does it look like (concretely)?



Approach 3.1: Use outbox pattern

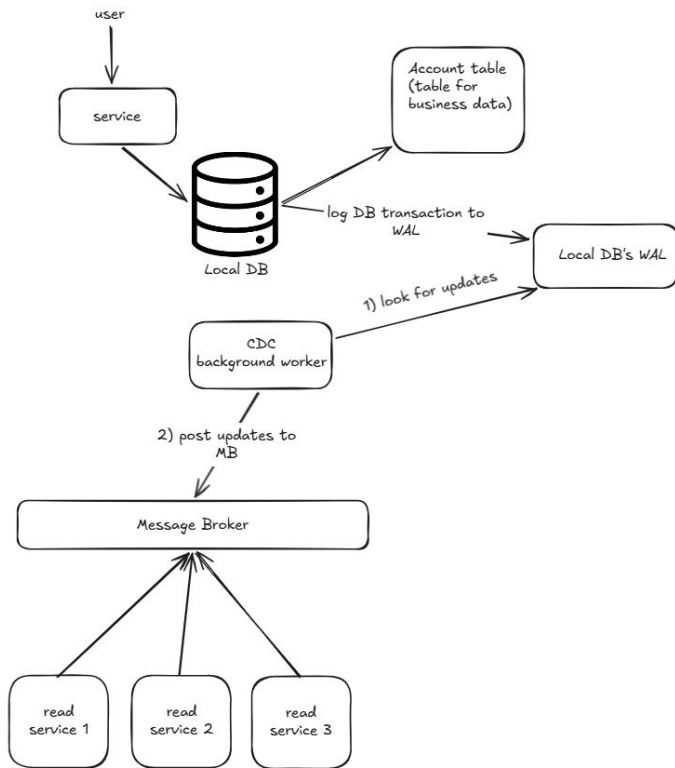
Good:

- Atomicity
- No dual-writes
- No lost events (reliability)
- Durable

Bad:

- Complexity
- No concurrency handling
- Duplicates (but less)
- Event-driven but NOT event-sourced

What does it look like (concretely)?



Approach 3.2: Use Change Data Capture (CDC)

Good & bad same as outbox but

CDC:

- Easier to adopt
- Has less control over the event shape

What does it look like (concretely)?

Approach 3.1 + 3.2: How to fix concurrency and idempotency?

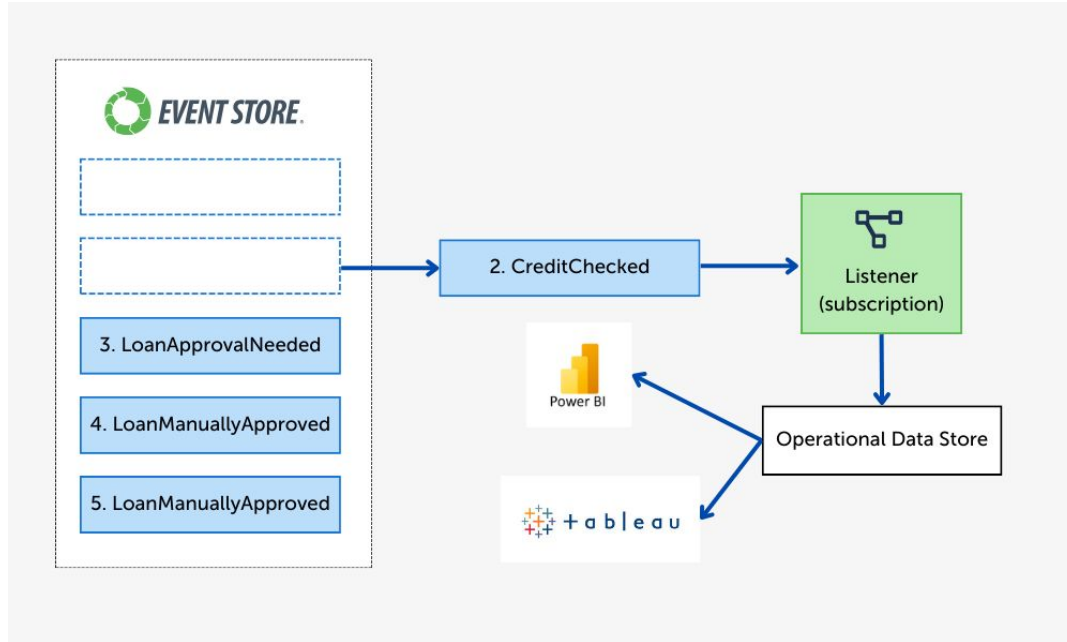
Concurrency

- Versioning
- Single writer per aggregate + Partition by aggregate ID.
- ?

Idempotency (how to fight duplicates)

- Store Event ID with each event: consumer stores event id => sees duplicate => ignore
- Design events idempotently (“set balance to X” rather than “increment by N”)
- ?

What does it look like (concretely)?



Approach 4: Use “event-native” DBs as KurrentDB (EventStoreDB)

Good:

- Truly event-**sourced** = events are source of truth
- Built-in optimistic concurrency
- Built-in subscription model
- No dual-writes

Bad:

- Higher modeling complexity (event design, aggregates, projections)

How to make queries efficient in ES system?

Problem:

- Different query and store patterns
- Need to scale read-side independently.
- Unclear boundaries between responsibilities.

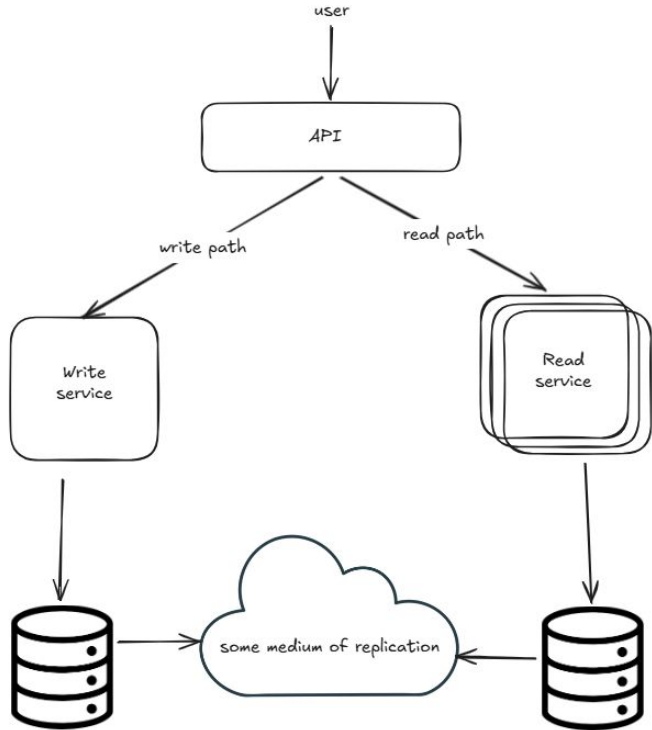
How to make queries efficient in ES system?

Problem:

- Different query and store patterns
- Need to scale read-side independently.
- Unclear boundaries between responsibilities.

=> separate read and write side

Command-Query Responsibility Separation (CQRS)



- Separate write and read path
- Updates need time to propagate from write DB to read DB => eventual consistency
- The medium of replication from write DB to read DB could be:
 - 1) posting messages to message broker
 - 2) direct write DB polling
 - 3) etc.

Other problems in event-sourced systems

Problem 1: consumer lag

Problem:

- Projections don't keep up with the write side

Solution:

- Keep projection build speed as a significant SLI

Problem 2: eventual consistency

Problem:

- User performs a write and (often) expects to see it on read-side immediately.
- “Read-your-writes”

Solution:

- Versioning tokens
- Session routing
- UX focuses (optimistic UI)

Problem 3: projection rebuilding

Problem:

- Sometimes we need to rebuild projections.

Solution:

- Snapshotting
- Blue/Green projection rebuild

Problem 4: GDPR

Problem:

- Some countries require giving a user a right to request a complete erasure of user's data from your system ("right to be forgotten").
- If we use event sourcing => we can't delete records from the event bus => we can't delete user's data.

Solution:

- Cryptoshredding: before inserting an event record to the event log, generate encryption key, encrypt the event record with this key, store the event record in event log and the encryption key in some DB.
- User requests data erasure => delete the en-/decryption key => we can't decrypt the data

Problem 5: migrating event log schema

Problem:

- Similarly to DBs, we sometimes need to migrate event log schema.

Solution:

- Same mechanisms as migrating DB schema:
 - Version schemas. Then, consumers just need to parse the event log according to the schema.
 - Upcasting.
 - Rebuild the full store (expensive, long).

References + further reading

- [1] [Оптимизация хранения transactional outbox в Postgres](#)
- [2] [Надежная отправка событий в Apache Kafka: от CDC до Transactional Outbox](#)
- [3] DDIA Ch. 12
- [4] [Greg Young - CQRS and Event Sourcing](#)
- [5] [KurrentDB](#)