Engineer

Event sourcing

keeping state as the ordered log of events that changed it

What it is

Event sourcing is a way of storing data in which the record of what happened is the data. Most applications keep the current state of an order or an account in a row and overwrite it on every change. An event-sourced application keeps the changes instead: each one is appended to a log as an event, and the current state is worked out from that log whenever it is needed. Martin Fowler, who described the pattern in 2005, put it in one line: “Capture all changes to an application state as a sequence of events.”

An event records something that has already happened. Greg Young’s CQRS Documents ask for every event to be named as a verb in the past tense, such as `CustomerRelocated` or `CargoShipped`, because the name is part of the language the business uses about its own work. A fact about the past cannot be undone, so events are never edited or removed. Young compares the log to an accountant’s ledger, where a wrong entry is not rubbed out but corrected by a new one, and he applies the same rule to deletion. When a customer takes two pairs of socks back out of a cart, the log gains a “removed” event. Young calls such an entry a reversal: the cart ends up as if the socks had never been added, and the log still shows that they were.

The events are kept in streams, one per thing whose state they describe. Young, who builds on domain-driven design, calls that thing an aggregate. Each event carries the stream’s id and a version number that is unique and sequential within that stream, so the order of events is never in doubt. To load an order, the application reads the order’s events by version and applies each one in turn. Young points out how little the store itself has to do: at its simplest it offers two operations, reading every event of one stream in order and appending new events to it.

That makes the log the system of record. Fowler notes that either the log or the current state can be the official record. In an event-sourced system it is the log, and everything else, a table of current balances, a search index, a report, is a copy derived from it that can be thrown away and built again.

Use the concept in a decision

Put “What is event sourcing?” to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

Fowler names three things a complete log makes possible. A complete rebuild: the application state can be discarded and recreated by replaying every event onto an empty application. A temporal query: the state at any past moment is the result of replaying the events up to that moment. And event replay: when an event turns out to have been wrong, the system can reverse it and the events after it, then replay the corrected sequence. The same log is also an audit trail that nobody had to remember to write, and Fowler mentions support and debugging as everyday uses: a support team can see exactly what a user did, and a developer can replay production events in a test environment.

Young shows the payoff with two teams at an online retailer, both asked for a report on items that customers remove from their carts. The team that stores only current state starts recording removals now, and its report has data from that day on. The team that stores events writes the same report handler and runs it over the log from the beginning, so its first report reaches back years. A question nobody thought to ask when the system was built can still be answered, because the system kept what happened rather than what it ended up as.

Events are also how other systems learn about a change. Young observes that many systems save their state and then publish an event to a queue, which strictly needs a two-phase commit between the two, since a crash in between leaves them disagreeing. His alternative is to treat the event store itself as the queue: a sequence number on every event lets a separate process publish the events in order, so there is only one write. And because the store is append-only,

it is easier to scale. Young notes that append-only storage needs far fewer locks than storage that updates in place, and that the aggregate id is the only key an event store ever has to be partitioned by.

The costs are real, and Fowler states them. Packaging every change as an event is a style of programming that not everyone is comfortable with, and the pattern is very hard to add later to a system that was not built around it. The store also answers only one kind of question, “give me this stream”. Young’s example is that an event store cannot tell you which users are named Greg; questions like that need a separate read model, built from the events, which is why he treats event sourcing and CQRS, separating the models for writing and reading, as a pair.

What it looks like in practice

The smallest working event store is one PostgreSQL table, close to the layout Young gives (an aggregate id, the event’s data and a version). Here it holds the stock of one item on sale. Each event carries a change to the number of units on sale rather than the new total, the form Fowler calls the most straightforward to reverse: “add ten” is undone by subtracting ten, while “set to 110” has lost the value it replaced.

```
CREATE TABLE stock_events (
    item_id  text    NOT NULL,
    version  integer NOT NULL,
    event    text    NOT NULL,
    order_id text,
    units    integer NOT NULL,  -- change to the units on sale
    PRIMARY KEY (item_id, version)
);

INSERT INTO stock_events VALUES
    ('item-881', 1, 'ItemStocked',   NULL,   100),
    ('item-881', 2, 'UnitsHeld',     'A-17',  -2),
    ('item-881', 3, 'HoldExpired',   'A-17',   2),
    ('item-881', 4, 'UnitsHeld',     'B-02',  -1),
    ('item-881', 5, 'HoldConfirmed', 'B-02',   0);

-- current state: fold the whole stream
SELECT sum(units) AS on_sale FROM stock_events
WHERE item_id = 'item-881';                    -- 99

-- temporal query: the item as it stood after version 2
SELECT sum(units) AS on_sale FROM stock_events
WHERE item_id = 'item-881' AND version <= 2;   -- 98
```

The primary key does the work Young assigns to an optimistic concurrency check. Two checkouts that both read the stream at version 5 and both decide to hold a unit will each try to append version 6:

```
INSERT INTO stock_events VALUES ('item-881', 6, 'UnitsHeld', 'C-40', -1);
INSERT INTO stock_events VALUES ('item-881', 6, 'UnitsHeld', 'D-11', -1);
-- the second fails: (item-881, 6) already exists
```

PostgreSQL's documentation describes a primary key over a group of columns as making that combination unique, and an insert that would duplicate it raises an error. The checkout that lost reloads the stream, checks that a unit is still on sale, and tries again as version 7. Neither of them ever sold a unit that was not there, and the log shows both decisions in the order they were made.

The note Selling the last ticket exactly once works through the same stock problem and shows where this shape gets tight. Its first answer keeps the count in one row and changes it with a single conditional `UPDATE`, and it advises against optimistic locking on a hot counter, because a version column turns contention into retries. For the hottest item it puts every request on one Kafka partition, read by a single consumer that keeps the count in memory and commits its decisions in batches, and calls this the LMAX shape. Fowler's article on the LMAX architecture, one of the note's sources, describes event sourcing at its purest: a trading engine with no database at all, whose whole state can be derived from its input events, which a journaler stores durably so they can be replayed. The note's checkout uses events of a different kind. Each step writes a message to an outbox table, change data capture publishes it, and the tables remain the record. Those events announce changes; in event sourcing, the events are the record.

Where it goes wrong

Replaying the log repeats its side effects. A handler that sends a confirmation email or charges a card does it again when the log is replayed, and Fowler points out that external systems cannot tell a replay from real work. Rebuilding a read model can then send every customer last year's emails. Put every external system behind a gateway that knows when the application is replaying and drops outgoing calls during a replay, so the domain logic never has to know. Queries need the same care: Fowler's example is a currency conversion, where a replayed event needs the exchange rate of its own day, not today's, so the gateway should record the answers it received and return them on replay.

A busy stream turns into retries. Optimistic concurrency assumes conflicts are rare. When many writers append to one stream at once, every conflict means reloading the stream and trying again, and under a burst of traffic the retries become the load, which is the warning the note above gives about version columns. Draw streams so that one rarely has two writers at the same moment: one stream per order rather than one for a whole product line. Where a single stream really is hot, give it a single writer, as in the LMAX design, so that there is nothing left to conflict.

Long streams load slowly. Replaying thousands of events to load one object costs time on every request, and Fowler lists slow rebuilds among the reasons to keep a current-state copy. Young's answer is the rolling snapshot: the serialised state of a stream at a known version, kept in a table of its own and written in the background by a separate process, so that loading means the newest snapshot plus the events after its version. Keep snapshots out of the event log itself: Young describes a snapshotter on a busy stream failing its concurrency check again and again. He also recommends building without snapshots, since they can always be added later to make loading faster.

Readers fall behind the writer. Every read model is built from the events after they are written, so it trails the log. Fowler warns that systems reading the log are liable to be out of step with it, and a user who submits a form and then sees a list without the change will submit it again. Have each read model record the last event it applied, so its lag can be measured and it can be rebuilt from the start, and show users the outcome of their own command from the command's response rather than from a read model that may not have caught up.

Further reading

- Event Sourcing — Martin Fowler's description of the pattern: complete rebuild, temporal query, event replay, and gateways that keep replays from reaching other systems.
- CQRS Documents by Greg Young — events as facts in the past tense, reversal instead of delete, a two-table event store, rolling snapshots and the event store as a queue.
- The LMAX Architecture — an in-memory trading engine whose entire state is derived from journaled input events.
- PostgreSQL: Constraints — unique and primary-key constraints over a group of columns, the database's side of an optimistic concurrency check.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is event sourcing?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/event-sourcing/