https://doi.org/10.31891/csit-2026-2-19

Ihor YANKIN
PhD student, Department of Computer Systems and Technologies,
Odesa I. I. Mechnikov National University, Odesa, Ukraine,
https://orcid.org/0000-0003-4423-8267
e-mail: yankiniigor@gmail.com

Yurii GUNCHENKO
Doctor of Technical Sciences, Professor, Department of Computer Systems and Technologies, Odesa I. I. Mechnikov National University, Odesa, Ukraine,
https://orcid.org/0000-0003-4423-8267
e-mail: gunchenko@onu.edu.ua

Received: 02/04/2026
Accepted: 04/05/2026
Published: 31/05/2026

UDC 004.22

FOUNDATIONAL ABSTRACTIONS FOR CORE ENTITIES AND QUERY MECHANISMS IN EVENT-SOURCED SYSTEMS

Event-sourced systems represent application state as a deterministic function of an immutable event history, so queries operate over histories and derived views rather than over a single current-state snapshot. The absence of shared, implementation-agnostic definitions and semantic contracts makes nominally similar queries non-comparable and hinders portable cost reasoning. The goal of the study is to develop a compact, implementation-agnostic formal foundation for core entities and query mechanisms in event-sourced systems, which enables mechanism-level cost analysis independent of specific technologies. A theoretical methodology based on formalization and mechanism-level analysis is proposed. The study defines a minimal set of core entity abstractions that determine representation and interpretation in event-sourced systems (events, streams, aggregates, projections, snapshots, versions). On this basis, querying is formalized as contract-defined deterministic evaluation over immutable histories and organized into four mechanism groups: reconstruction, temporal, cross-stream, and retroactive replay, each specified through explicit scope and cut rules with declared ordering/merge, correlation, and version-normalization policies. Portable cost envelopes are derived by expressing selection and evaluation costs through selected evidence size and amortized per-event processing, including explicit contributions from normalization and replay shortening by snapshots. The study formalized implementation-agnostic core abstractions and contract-defined query mechanisms for event-sourced systems and derived cost envelopes. A theoretical experiment on a synthesized banking event dataset confirmed internal consistency, replay equivalence, and reproducibility under semantics-preserving transformations. The proposed formalization fixes the semantic degrees of freedom required for reproducible and comparable querying over immutable event histories and provides a reusable basis for mechanism-level cost reasoning across implementations. Further research should extend the framework toward practice-complete semantics by formalizing admissibility rules under distributed time uncertainty or read-model staleness under eventual consistency.

Keywords: event-sourced systems, event-driven architectures, query semantics, formalization, cost modeling

Introduction

Event-sourced systems treat application state as a function of an immutable history of domain events. Rather than persisting only the latest values of entities as is usually the case in most systems, semantically meaningful records are appended to a log, and state is reproduced by replaying those records in a strict order. In such architectures, queries operate not over a single entity but over histories of events, which are usually reflected in the actual business events [1]. As a result, the choice and discipline of the underlying abstractions determine which questions can be answered, under which guarantees, and at what computational cost.

The importance of this topic is underscored by current software practice. With the growth of distributed, microservice-oriented systems, there is a natural fit towards architectures that preserve explicit histories and support independent evolution of services. Moreover, regulatory and business demands for explainability and audit trails further require that answers be justifiable in terms of underlying events rather than only the present state.

A variety of engineering approaches have been adopted to make queries in such systems practical and satisfy the requirements of such systems, such as snapshots, correlation indexes, dynamic indexes, etc [2]. These techniques often succeed

operationally, yet they are introduced and evaluated in isolation. A significant amount of research in the field is case studies, presenting specific use-cases, which can hardly be generalized without further formalization. In the absence of theory-grounded definitions, empirical observations lack the invariants needed for replication and synthesis. Moreover, it becomes difficult to accumulate results across those studies, to derive reusable cost models, or to investigate optimization strategies that are portable across implementations.

The scientific problem addressed in this work is the absence of explicit, shared definitions and semantic contracts that fix what a query over an immutable event history means. In particular, key elements of query interpretation - what constitutes admissible evidence, how scope is delimited, how a cut is defined and justified, which ordering and composition rules apply when evidence spans multiple histories, and how mixed-version events are normalized - are often left implicit or scattered across code and operational conventions. As a result, queries that appear identical at the surface may, in fact, operate under different semantics, which makes their behavior non-comparable across various systems.

The object of study is the process of state reconstruction and query answering over immutable event histories in event-sourced architectures.

The subject of study is the set of methods and models of query mechanisms over event histories, including the minimal abstractions and invariants that determine query semantics and enable implementation-agnostic cost reasoning.

Therefore, this article aims to develop a compact, implementation-agnostic formalization of the core abstractions and query mechanisms, which is necessary to provide a stable foundation for subsequent research and to enable clean separations between layers. By pinning down these bases, comparable reasoning about costs becomes possible at the mechanism level, and subsequent work can attach analyzable envelopes without dependence on a particular case.

To achieve this aim, the article defines the minimal set of core entity abstractions that determine representation and interpretation of information in event-sourced systems - events, streams, aggregates, projections, snapshots, and versions; formalizes the primary query mechanism groups over immutable histories by organizing them into reconstruction, temporal, cross-stream, and retroactive replay mechanisms and specifying them in terms of scope and cut; and derives cost envelopes expressed through the size of the selected events and amortized per-event processing costs, including the effect of normalization and the replay-shortening role of snapshots.

Problem statement

Let an event-sourced system be given in which evidence about domain changes is represented by an immutable set of events ε. The events are organized into a family of streams, where each stream is a finite, totally ordered history associated with a stream key. Each event carries, at minimum, a stream key identifying the stream it belongs to, an event-time value from a time domain T, and a version label from a version set V. Query answering is performed by a deterministic interpretation procedure that consumes an admissible subset of events in a specified order and accumulates them into a state in a state space Σ, after which a readout produces an answer in an answer domain A. Under evolution, a deterministic version interpretation policy specifies how events of different versions are brought to a common interpretation before they are processed.

As input data, we consider: the stored streams; a query scope specification defining which stream or set of streams is admissible; a cut specification defining which events within the scope are admissible; an ordering or merge specification defining how a deterministic evaluation order is obtained whenever evidence spans multiple streams; an optional correlation specification defining how related events are identified across streams; a version interpretation specification defining how mixed-version histories are handled; and a deterministic interpretation procedure together with its initial state and readout definition. For retroactive queries, the input additionally includes a deterministic rule for generating an alternative history from a declared cut and a rule for comparing outcomes.

The desired results are: a minimal, implementation-agnostic set of formal core entity abstractions sufficient to determine representation and interpretation of information in event-sourced systems independently of a particular storage engine or framework; a formalization of the primary query mechanism groups, stated in terms of scope and cut over the previously defined entities; and mechanism-level cost envelopes that follow from the same definitions and are expressed through the number of events selected under the declared scope and cut, the amortized per-event processing cost of deterministic interpretation, the amortized per-event overhead induced by the stated version normalization policy, and the replay-shortening effect of snapshot-assisted evaluation tied to an explicit cut.

A result is considered correct if the produced answers and derived states are justified by deterministic evaluation over exactly the admissible evidence fixed by the declared entity definitions and the declared query contract. A result is considered reproducible if repeating the evaluation under the same inputs and conditions yields the same outcomes. Results are considered comparable across implementations only when the full definition and query contract are identical, meaning that entity meanings, admissible evidence selection, and interpretation policies coincide. The constraints are: immutability of events; total order within each stream; determinism of all components that influence entity interpretation and query admissibility; and semantic preservation under auxiliary acceleration, meaning that any use of precomputed intermediate state must not change the outcomes defined by the same scope, cut, and interpretation policy.

Review of the literature

Early works like Kabbedijk et al. (2012) [3] classified event sourcing as a sub-pattern of the CQRS pattern aimed at improving scalability and variability in systems, underlining the conceptual connection to established architectural paradigms. However, as event sourcing has gained significant traction for its ability to produce reliable, auditable, and scalable systems, more research has begun to appear on such systems and their features. Overeem et al. (2021) [1] note that event sourcing is widely adopted in industry but has not received as much attention from the scientific community, leading to ad hoc solutions and inconsistent terminology. Their empirical study of 19 event-sourced systems is one of the most fundamental studies on the topic, despite being focused fully on practical adaptation and existing practice. They identify common challenges, but without providing a unifying theory.

Recent research has begun to address this gap by systematically comparing approaches in event-sourced architectures. Hlybovets and Yankin (2025) [4] provide a high-level introduction to event sourcing and compare it to traditional state-storage approaches. Alongi et al. (2022) [5] in their study focus on observability in event-sourced systems and contribute a rigorous definition of software architecture observability, along with design principles to augment common patterns for improved auditability. Lytvynov and Hruzin (2024) [6] tackle the problem of causal event ordering in distributed CQRS with an event-sourcing architecture approach. They formalized existing solutions of delivery guarantee and evaluated these approaches under performance and complexity criteria. By comparing such strategies in a unified framework, they developed a new method and proved it to be the most effective solution to mitigate out-of-order event anomalies according to their evolution. Another their study [7] takes a step toward comparing different approaches propose a decision-support framework for selecting among CQRS with event-sourcing architectural variations. They introduce formal metrics (based on a set of representative test projects [8]) to evaluate variations on complexity and performance dimensions.

All those studies provide different definitions for highly used query mechanisms, though mostly informally described. For instance, Fowler's original description (2005) [9] notes that with a complete event history, one can perform temporal queries (reconstructing application state at any point in time) or even execute retroactive event replay to correct past mistakes by replaying an amended event sequence, not providing the readers with any formalization or implementation approaches.

In essence, recent works are moving the field from isolated case studies with informal definitions towards a unified theoretical foundation for event-sourced systems. This enables not only reproducible comparison of different mechanisms, but also the derivation of analyzable cost envelopes for queries across any event-sourced implementation.

Materials and methods

By core entity abstractions are meant the minimal, implementation-agnostic constructs that determine how information is represented and manipulated in an event-sourced system, independently of any particular storage engine or framework. Event-sourced systems are fundamentally bound to events and, as a consequence, operate a small number of specific structures - events, streams, aggregates, projections, and indexes - as a primary organizing principle rather than as incidental implementation details. These entities shape the semantics and cost of queries and serve as the basis on which all further work and more complex concepts are constructed. A formal definition of these abstractions is therefore an important step, as it unifies their meaning within the event-sourced setting and prevents query behavior from being implicitly tied to any particular implementation or infrastructure. Once these core entity abstractions are defined with the necessary formality, subsequent sections and future research can rely on them as a stable foundation for specifying query mechanisms, analyzing costs, and comparing different implementations. The relationships between these abstractions are summarized in Fig. 1.

An event is the smallest operational unit in an event-sourced system, representing a single, domain-relevant change that has occurred in the application. Formally, an event can be understood as an immutable record that captures the type of change, its associated data, and the necessary identifiers and timestamps to locate it within a history. Once appended to the event log, an event is never modified or deleted; instead, it becomes part of the history from which all derived state is reconstructed. This makes events the primary source of truth [2] application state is obtained by applying a sequence of events, and higher-level artefacts such as projections, snapshots, and indexes are computed by processing events according to specific rules.

A stream is the primary structural abstraction used to organize those events in an event-sourced system. It represents a totally ordered sequence of events associated with a particular key, most commonly an aggregate identifier or a logical category. Formally, for a given key k, a stream can be written as a finite sequence of the related events (1).

$$s_k = \langle e_1, e_2, \ldots, e_n \rangle \quad (1)$$

Streams are essential because they provide both the ordering and the scope required for deterministic reconstruction of information from histories and for temporal reasoning.

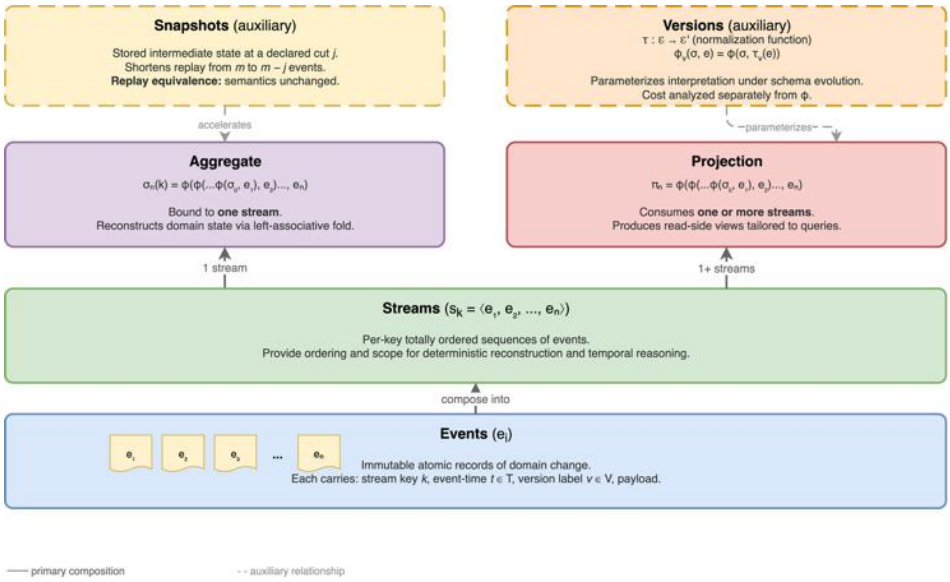

Fig. 1. Entity hierarchy

An aggregate is the primary logical unit around which events and streams are organized in an event-sourced system. Informally, it represents a coherent "thing" in the domain whose changes are recorded as events and whose history is stored in a dedicated stream. Each aggregate instance is identified by a key, and all events that describe changes to that instance are appended to the corresponding stream for that key. In this way, events provide the atomic records of change, streams collect those records into totally ordered histories, and aggregates provide the conceptual boundary that determines which events belong together and which do not. As aggregates are not stored physically, they are obtained by replaying the events of their stream in order and accumulating their effect on an abstract state. Formally, we can define an initial aggregate state as σ_0. Then, for a given aggregate identifier k with stream s_k, the reconstructed state after n events can be defined as a left-associative application of application functions over that stream in a strict order (2). This definition makes explicit that the aggregate state is a pure function of its immutable event history and that any query about the "current state" of an aggregate is, in principle, reducible to this reconstruction process over its stream.

$$\sigma_n(k) = \varphi(\varphi(\dots\varphi(\sigma_0, e_1), e_2)\dots, e_n)\ e_1, \dots, e_n \in s_k \tag{2}$$

Aggregates, by design, provide only a local and operational view: to obtain the current state, the events of the corresponding stream must be replayed by applying the transition function to each event in order. This is acceptable for occasional reconstruction of a single aggregate, but quickly becomes impractical for frequent or wide queries, where the same computation would have to be repeated many times. A projection is introduced precisely to remove these restrictions. Whereas an aggregate state is the result of folding a single stream into a domain state space, a projection is a derived, read-side abstraction that folds one or more streams into a separate state space chosen for querying. In contrast to aggregates, which are bound to one stream and to a domain-oriented state space, a projection may combine events from many streams and maintain a state space that is explicitly tailored to the intended queries [7]. Let's define a space of streams as *S*. Given an ordered sequence of input events belonging to that space, projection can be defined in a way close to the aggregate's definition (3).

$$\pi_n = \varphi(\varphi(\dots\varphi(\sigma_0, e_1), e_2)\dots, e_n)\ e_1, \dots, e_n \in S \tag{3}$$

This distinction determines the respective roles of aggregates and projections in the overall architecture. Compared to the aggregates, projections provide derived views that combine and reorganize information from one or more histories according to a specified perspective.

Aggregates and projections together define how state and derived views are obtained from event histories, but both rely conceptually on replaying sequences of events: aggregates by folding a single stream into a domain state, projections by folding one or more streams into a read-optimized state space. As histories grow, the cost of reconstruction from the beginning of a stream or from the start of all relevant streams becomes a limiting factor, even if the underlying fold functions remain simple [11]. Snapshots are introduced precisely to mitigate this cost without changing the semantics defined for aggregates and projections. A snapshot is an auxiliary abstraction that records a precomputed state together with an explicit reference to a cut in the underlying histories. For an aggregate, a snapshot captures the aggregate state after processing events up to a given position in its stream; for a projection, an analogous snapshot captures the projection state after consuming all events up to a specified cut of the input histories. The essential property of snapshots is replaying equivalence: resuming reconstruction from a snapshot and applying the

remaining events must yield the same result as applying the corresponding fold from the initial state over the entire history. In this sense, snapshots do not introduce new primary semantics; they preserve the definitions of aggregate and projection state while bounding the length of history that must be processed on demand. As a result, they provide a controlled mechanism for trading storage and update overhead for reduced read-time computation, enabling aggregates and projections to remain usable in the presence of long-running streams and expensive rebuilds. Fig. 2 illustrates the difference between full replay and snapshot-assisted reconstruction, showing that both paths produce an identical terminal state while the snapshot variant processes only the suffix beyond the stored cut.

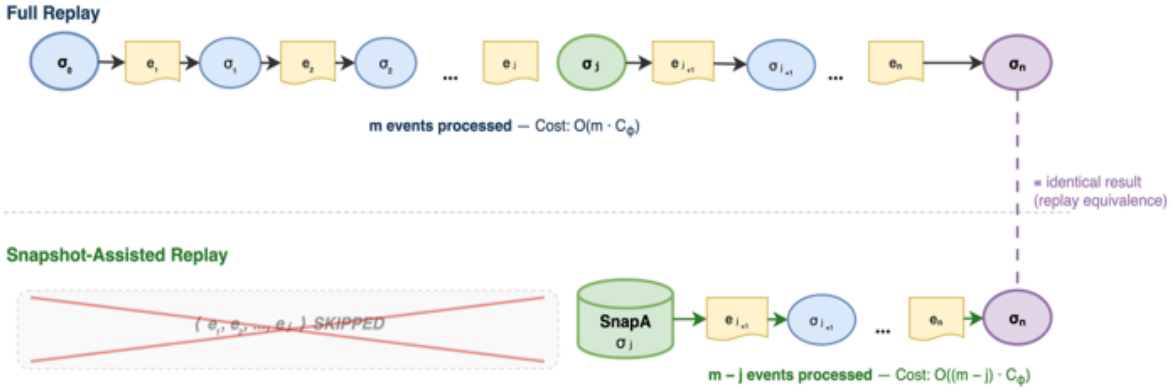

Fig. 2. Snapshot-assisted reconstruction

Alongside snapshots, which bound reconstruction cost without changing the underlying semantics, version information forms another auxiliary abstraction that refines how the previously defined entities are interpreted over time. While events, streams, aggregates, and projections specify what is stored and how state and views are derived from histories, versions control how individual events are to be understood when schemas and domain assumptions evolve [12]. At the level of a single event, a version can be treated as a specific property of the event with a special functional meaning: it determines which interpretation rules, schemas, or invariants apply to that event. Unlike ordinary payload fields, the version does not describe the domain change itself; instead, it governs the processing of the event by aggregates, projections, and other consumers. The purpose of introducing versions as an explicit abstraction is to separate the immutable history of what happened from the evolving interpretation of how that history is to be processed. In practice, event structures and meanings change as fields are added or removed, constraints are tightened or relaxed, and business rules are refined. If version information is handled only implicitly inside application or projector code, the resulting mixture of old and new assumptions is difficult to analyze, compare, or generalize. By contrast, presenting version as a dedicated, semantically significant property makes it possible to define compatibility relations and interpretation policies. It can be formalized as a normalization function (4) which maps raw events from the original event set $\mathcal{E}$ into a canonical representation $\mathcal{E}'$.

$$\tau: \mathcal{E} \to \mathcal{E}' \tag{4}$$

Aggregate transition functions and projection update functions then operate on these normalized events (5). In this formulation, the cost of applying τ_v can be analyzed separately from the cost of φ, and different policies v correspond to different ways of handling mixed-version histories.

$$\varphi_v(\sigma, e) = \varphi(\sigma, \tau_v(e)) \tag{5}$$

Given that definition, versions do not alter the fundamental model in which application state is a function of event histories, but complement it with variation of interpretations.

The abstractions introduced in this section - events, streams, aggregates, projections, snapshots, and versions – fully define how information is represented and interpreted in an event-sourced system, independently of any implementation. Events provide the smallest units of domain change; streams organize these units into per-key histories; aggregates specify how state is reconstructed from a single history; projections generalize this reconstruction to read-oriented views over one or more histories; snapshots bound the cost of reconstruction without altering semantics; and versions parameterize the interpretation of events under schema and domain evolution. Together, they establish the layer on which all subsequent reasoning about querying must be based.

On this foundation, query behavior in the system can be described without referring to specific technologies, but as operations over these abstractions.

By core query mechanisms are meant the minimal, implementation-agnostic operations by which answers are derived from event histories and their deterministic derivations, independently of any particular query engine, database, or messaging platform. In the event-sourced setting, querying is constrained and enabled by the previously defined entities: events provide immutable evidence, streams provide per-key ordering, aggregates and projections define reconstruction rules, snapshots bound replay cost, and version policies parameterize interpretation under evolution. A mechanism is included in this category only if it can be expressed in terms of these abstractions, if its inputs and outputs admit a clear semantic contract, and if its behavior can be reasoned about without reference to storage-specific optimizations.

Under this selection principle, the scope is restricted to four mechanism groups that capture the dominant semantics of querying in event-sourced systems. Reconstruction mechanisms describe how state or derived views are obtained from event histories through deterministic replay. Temporal mechanisms formalize how histories are queried through time-based cuts, windows, and change evaluation. Cross-stream mechanisms specify how answers are derived when the relevant information spans multiple streams, requiring explicit correlation evidence rather than an assumed global order. Retroactive replay mechanisms formalize what-if and branch analysis by evaluating alternative histories while preserving immutability of the authoritative log. In this way, the four groups can be assumed not as disconnected techniques but as a coherent, compositional model of querying over event histories (Fig. 3).

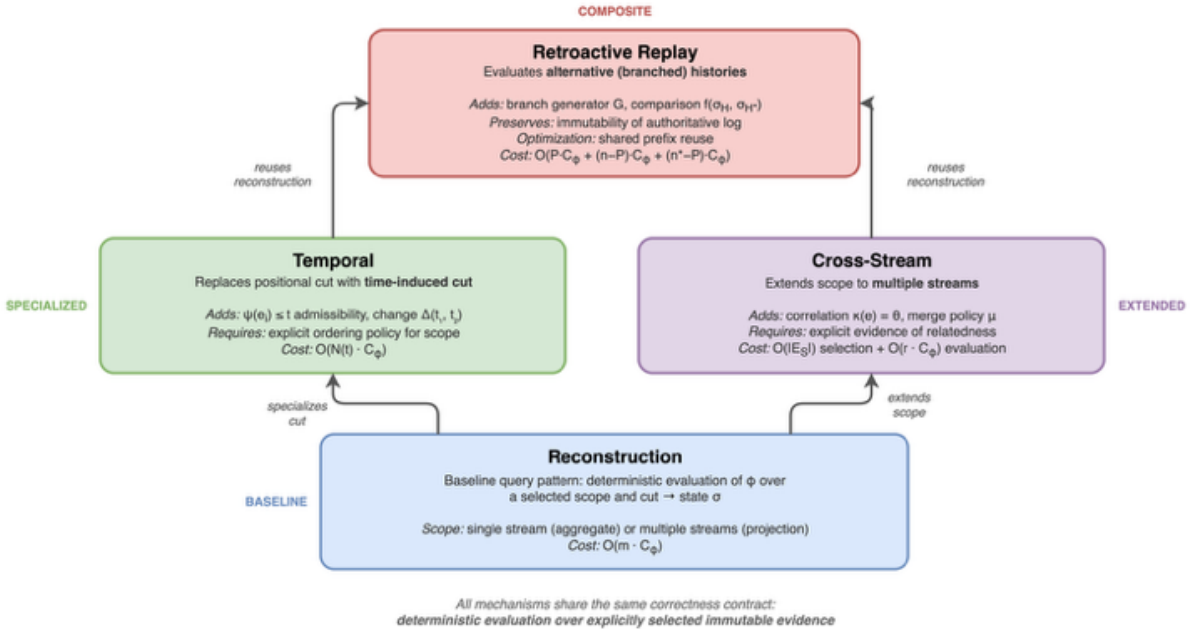

Fig. 3. Four query mechanism groups

Reconstruction mechanisms constitute the fundamental query pattern in event-sourced systems: a result of a query is defined as the result of evaluating a deterministic interpreter over a selected portion of immutable history. The formal expressions used for reconstruction are the same as earlier definitions of aggregate (2) and projection (3) state. In other words, replay is simultaneously the semantic definition of state and the baseline way of obtaining that state for a query. The main difference and the reason why it is placed in both sections is that while entity abstractions establish the objects of interpretation, reconstruction mechanisms formalize the act of searching: the rules are instantiated on demand for a chosen scope and cut to produce an answer. With this definition, reconstruction is treated as the fundamental query pattern: it defines what it means for a read operation to be correct in an event-sourced system, namely, to be equivalent to evaluating the deterministic interpreter on the corresponding history prefix and then applying the query's readout function.

Within this group, variations are expressed by changing the bound components without changing the semantics mentioned above. A single-history reconstruction (which represents aggregate) binds the scope to one stream; a multi-history reconstruction (which represents projection) binds the scope to a set of streams together with a selection or merge policy that yields the ordered input history used for evaluation. Snapshot-assisted reconstruction (which represents an aggregate made from a snapshot) binds an earlier cut together with a stored intermediate state so that evaluation starts from that cut rather than from the beginning, while remaining equivalent to full replay for the same final cut.

Cost estimations can be obtained directly from the reconstruction semantics because evaluation is executed one by one, following the strict order of events in the stream. Let n_{sel} denote the number of events included by the selection, and let C_φ denote the amortized cost of processing one event by the interpreter (considering the amortized cost is important as the cost of applying functions to the events can be very variable). Then, the reconstruction cost can be defined as $O(n_{sel} * C_\varphi)$. For the cases, when a normalization function τ with amortized cost of C_r is applied, the reconstruction cost will increase to $O(n_{sel} * (C_\varphi + C_r)$, which makes explicit that version handling increases reconstruction cost linearly in the number of processed events. With a snapshot at cut j, only the suffix of length $m - j$ is processed, which reduces the cost to $O((n_{sel} - j) * C_\varphi)$ which making it explicit that snapshots reduce cost by shortening the replay prefix and not by changing the semantics.

The expressions above underscore a linear relationship between runtime and the length of the selected history; therefore, all optimizations in this group operate by reducing the effective replay length or by controlling the per-event processing cost.

Temporal query mechanisms build directly on reconstruction by replacing an index-based cut with a time-induced cut. Instead of selecting a prefix by position, a temporal query selects the subset of history that is valid up to a time instant t [13] and then applies the same deterministic interpreter as in the reconstruction mechanism. Let the function $\psi: \mathcal{E} \to T$ map each event to its event time. For a stream $s_k = \langle e_1, \dots, e_n \rangle$ and point in time t, the temporal cut can be defined as the maximal prefix whose events satisfy $\psi(e_i) \leq t$, captured by the cut index (6).

$$i(t) = max\{i \mid 1 \leq i \leq n \wedge \psi(e_i) \leq t\} \quad (6)$$

Temporal querying also supports change semantics by relating two-time cuts t_1 and t_2, where $t_1 < t_2$. Conceptually, change over $[t_1, t_2]$ is defined by evaluating how the reconstructed state differs across that segment. This can be expressed as a reconstruction from the beginning at both cuts, followed by a domain-specific difference operator ω (7).

$$d(t_1, t_2) = \omega(\sigma_n(i(t_2)), \sigma_n(i(t_1))) \quad (7)$$

Considering that the $\sigma_n(i(t_2)$ is a state of applying suffix $i(t_2)$-$i(t_1)$ to a $\sigma_n(i(t_2)$, computation can be simplified, reducing the duplication of computations (8).

$$d(t_1, t_2) = \omega(\sigma_n(\sigma_n(i(t_1)), i(t_2) - i(t_1)), \sigma_n(i(t_1))) \quad (8)$$

The other option requires a definition of a specific function to compute the difference between two states, not by reconstructing the states but rather by operating on the sequence of the events themselves (9).

$$d(t_1, t_2) = \chi(e_{t_1}, e_{t_1+1}, \dots, e_{t_2}) \quad (9)$$

Despite the efficiency of that method, the trade-off lies in the necessity of defining the function on the events subset, which usually binds to a specific case and can be applied only on a subset that satisfied certain requirements defined by the function.

As temporal queries follow the same principle as reconstruction, the cost is also linear in the number of events selected by the temporal predicate. Let $i(t)$ be the number of events included up to time t under the query's scope and ordering policy, and let C_φ be the amortized cost of one interpreter step. Then the cost can be defined as $O\big(i(t) * C_\varphi\big)$.

For an interval $[t_1, t_2]$, the replay length is governed by the number of events whose timestamps fall after the first cut and up to the second cut and the cost of the interpreter function C_ϕ, so the evaluation cost is $O\big(C_\phi + (i(t_2) - i(t_1)) * C_\varphi\big)$.

Finally, because time alone does not define a total order across multiple histories, the temporal semantics remains well defined only under an explicit ordering policy for the chosen scope; this policy becomes part of the temporal query contract, ensuring that the same time constraint yields a reproducible selected history and, therefore, a reproducible answer. Fig. 4 illustrates how a late event, whose event time precedes a previously appended event, creates ambiguity between append order and event-time order and demonstrates why an explicit admissibility rule is required for reproducible temporal cuts.

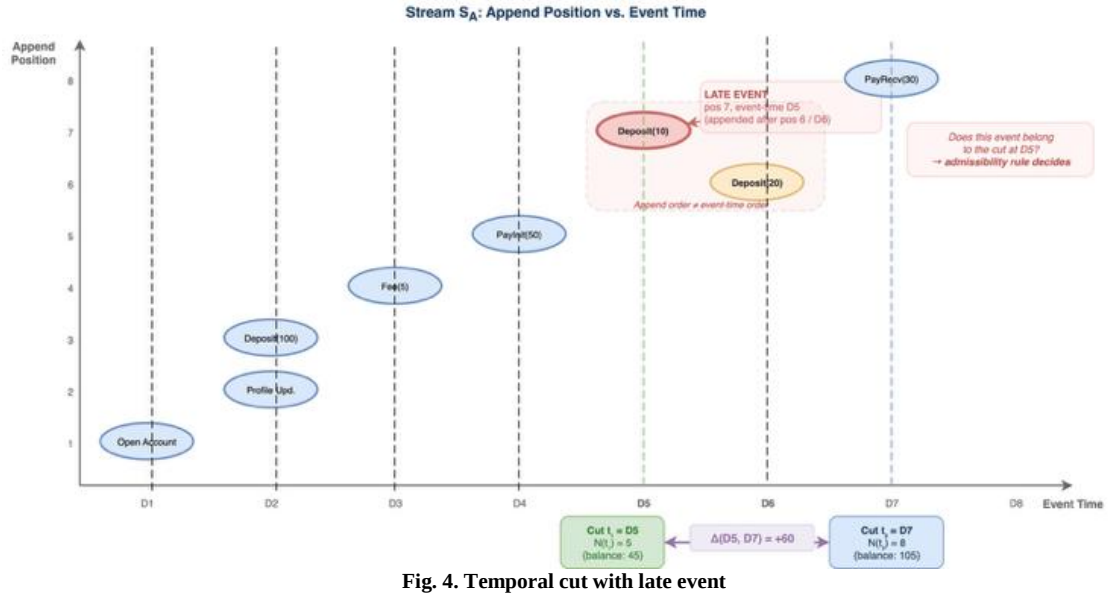

Fig. 4. Temporal cut with late event

Both reconstruction and temporal queries operate over a single stream of events. However, many analytically and operationally relevant questions in event-sourced systems cannot be confined to a single stream prefix even when a time constraint is provided, because the required evidence is distributed across multiple streams. Cross-stream query mechanisms address this setting by defining how multiple histories are composed under explicit evidence of relatedness, rather than under an assumed global order [14]. Therefore, cross-stream queries are characterized by a multi-stream scope and an explicit correlation semantics. Let S be a set of streams and E_S be the set of all events contained in that scope. As the order of events is maintained within a stream, the events belong to, cross-stream

querying needs to define a correlation predicate to maintain the order in multiple streams. Let's introduce a correlation extractor $\kappa: \mathcal{E} \rightarrow K$ and interpret a query parameter $\theta \in K$ as the correlation criterion. Then, the correlated event set can be defined as a set where all events share the same correlation key (10).

$$C(S, \theta) = \{e \in \mathcal{E}_S \mid \kappa(e) = \theta\} \quad (10)$$

With that, the result is defined by a deterministic function of the events selected by explicit correlation evidence. When an ordered evaluation is required, the semantics is extended by introducing an explicit policy μ that maps the correlated set into an ordered sequence up to a cut c, after which the same deterministic evaluation principle as in reconstruction is applied.

The cost of cross-stream queries can be split into two parts. First is the cost of the event selection, and the second is the cost of the evaluation. Let $N_S = \mathcal{E}_S$ and let $r = \mathcal{C}(S, \theta)$. If correlation evidence is computed by scanning the scoped history, the selection cost is linear in the scope size and can be defined as $O(N_S)$. If an auxiliary mapping from correlation keys to event references exists, selection can be reduced to the cost of retrieving the r correlated elements $O(r)$. For the evaluation, the same cost as for the reconstruction is applied, and the overall cost will be the sum of those two parts. Fig. 5 illustrates the complete cross-stream query process: correlation-based selection across multiple streams, version normalization, merge into a single evaluation sequence, and deterministic interpretation.

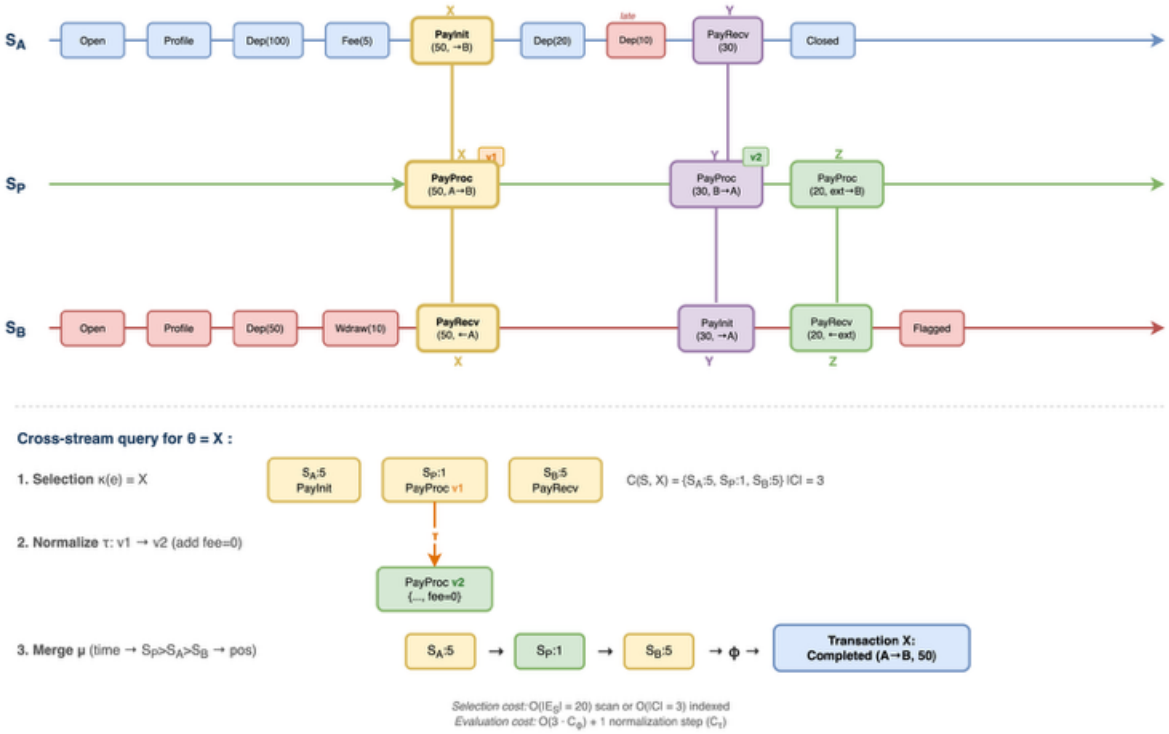

Fig. 5. Cross-stream correlation

A different class of questions arises when the history itself is treated as an object of analysis: it becomes necessary to evaluate how answers would change if some part of the event history were different, while keeping the authoritative log immutable [14]. Retroactive replay mechanisms formalize this setting by defining queries over alternative histories (branches) and by specifying the semantics of comparing outcomes between the authoritative history and a counterfactual variant.

A branch is defined as an alternative event history constructed from the authoritative history by applying a controlled transformation from a chosen cut. Let H denote the authoritative ordered history under the query scope, and let $H[1 \ldots \vartheta]$ denote its prefix up to cut ϑ. Let G denote a branch generator that, given a cut ϑ and the authoritative history, produces a modified suffix Δ_ϑ, while preserving the prefix. The branched history, then, can be defined as a concatenation of the prefix H_ϑ with the suffix (11).

$$H^*_\vartheta = H[1 \ldots \vartheta] \parallel \Delta_\vartheta \quad (11)$$

With that, the authoritative log remains unchanged, and the branch is treated as a separate history that is evaluated under the same interpretation rules as the authoritative one.

Retroactive replay answers are defined by evaluating the same reconstruction semantics on both histories, and the purpose of branch analysis is typically expressed as a comparison function applied to the two outcomes (12).

$$f(\sigma_{H_\vartheta}, \sigma_{H^*_\vartheta}) \quad (12)$$

The difference is computed under identical interpreters, with only the history suffix changed. Because the same reconstruction semantics is reused, retroactive replay inherits the same correctness contract: each answer is justified by the events in the evaluated history, and reproducibility holds as long as the branch generator G is deterministic.

The cost structure of retroactive replay follows directly from its definition as repeated reconstruction. Let n be the length of the authoritative history prefix used for evaluation, and let $n^\star$ be the length of the branched history segment processed. If $C_n$denotes the amortized cost per event step of the interpreter, then evaluating both histories from the beginning will cost $O(n \cdot C_n + n^\star \cdot C_n)$. However, retroactive replay is naturally optimized by reusing the reconstructed state at the branch point. Since $H$and $H^\star_\vartheta$ share the same prefix $H[1..\vartheta]$, a single reconstruction of that prefix can be shared, and only the suffixes beyond m need to be evaluated independently, reducing the complexity to $O\big(\vartheta \cdot C_\varphi + (n-\vartheta) \cdot C_\varphi + (n^\star - \vartheta) \cdot C_\varphi\big)$. Fig. 6 shows the shared prefix processed once and the two suffixes diverging from the branch point, with the resulting cost reduction compared to naive full evaluation of both histories.

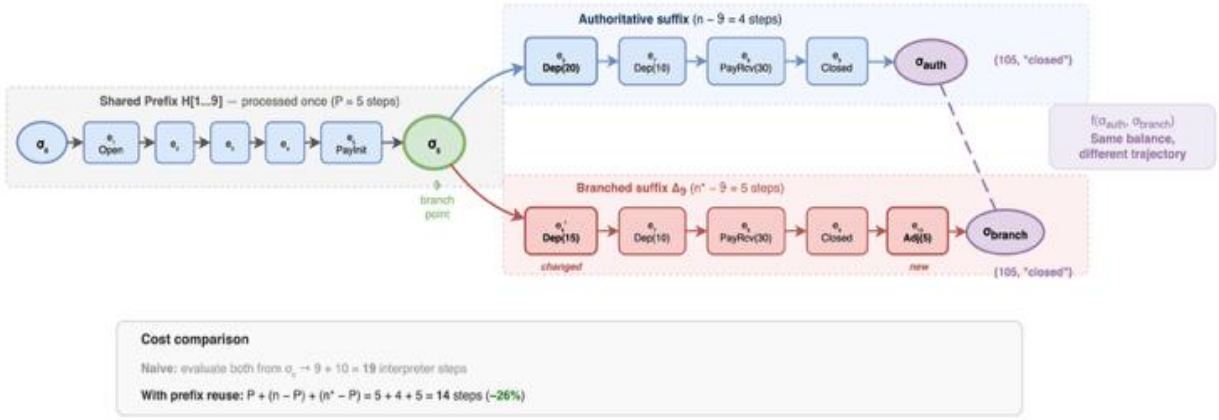

Fig. 6. Retroactive replay

The core query mechanisms formalized in this section define how answers are derived from event histories. Reconstruction provides the baseline search semantics by binding a scope and a cut to the relevant interpreters and evaluating them over the selected history prefix, which also yields direct, linear cost envelopes in the number of processed events. Temporal querying is obtained by inducing cuts and windows from event time and then applying the same reconstruction semantics to time-bounded prefixes or segments, thereby formalizing state-at-time and change-over-time answers under explicit ordering assumptions. Cross-stream querying extends the framework to multi-history scopes by selecting evidence through explicit correlation semantics and, when ordered evaluation is required, by introducing an explicit merge policy that makes reproducibility and correctness well defined. Retroactive replay completes the framework by treating histories as evaluable objects, defining branch histories that preserve an authoritative prefix while modifying a suffix, and computing branch outcomes and differences under identical interpreters. Taken together, these four groups form a single model in which query meaning is expressed as deterministic evaluation over explicitly selected histories, and query cost is reasoned about from the size of the selected evidence and the per-event processing cost.

Experiments

To evaluate the internal consistency of the proposed formalizations and cost envelopes, a synthetic banking-domain dataset was constructed in which every core entity and query mechanism group is instantiated under explicit contracts. The evaluation criterion is as follows: if the query meaning is fixed by the declared contract, the resulting answer is uniquely determined from the immutable event history under the corresponding interpreter, and semantics-preserving transformations (such as snapshot-assisted replay) produce equivalent readouts at the same cut, then the proposed abstractions are sufficient and internally consistent.

The dataset (Table 1, Fig. 7) consists of 20 events distributed across three streams: S_A and S_B, each defining an aggregate (accounts *A* and *B*), and S_P, which provides payment-processing events for a cross-stream projection. The dataset was designed to exercise all mechanism groups: correlation keys *(X, Y, Z)* span multiple streams for cross-stream querying; a deliberately late event (id 13, appended at position 7 with event time D5 < D6) stresses temporal admissibility; and "Payment Processed" events appear in two schema versions (*v1* and *v2*) to exercise normalization. Auxiliary specifications are given in Table 2: snapshot records for replay-shortening, a normalization policy (*v1* → *v2* by adding *fee=0*), and a deterministic branch rule for retroactive replay.

Table 1

Synthetic banking-domain dataset

Event id	Stream id	Append pos	Event time	Event type	Payload	Corr key	Version
1	S_A	1	D1	Open Account	{initialBalance:0}	–	–
2	S_B	1	D1	Open Account	{initialBalance:0}	–	–
3	S_A	2	D2	Profile Updated	{newContact:…}	–	–
4	S_B	2	D2	Profile Updated	{newContact:…}	–	–
5	S_A	3	D2	Deposit	{amount:100}	–	–
6	S_B	3	D2	Deposit	{amount:50}	–	–
7	S_A	4	D3	Fee Charged	{amount:5}	–	–
8	S_B	4	D3	Withdrawal	{amount:10}	–	–
9	S_A	5	D4	Payment Initiated	{amount:50,target:B}	X	–
10	S_P	1	D4	Payment Processed	{amount:50,from:A,to:B}	X	v1
11	S_B	5	D4	Payment Received	{amount:50,from:A}	X	–
12	S_A	6	D6	Deposit	{amount:20}	–	–
13	S_A	7	D5	Deposit	{amount:10}	–	–
14	S_B	6	D6	Payment Initiated	{amount:30,target:A}	Y	–
15	S_P	2	D7	Payment Processed	{amount:30,from:B,to:A,fee:0}	Y	v2
16	S_A	8	D7	Payment Received	{amount:30,from:B}	Y	–
17	S_A	9	D8	Account Closed	{reason:…}	–	–
18	S_P	3	D8	Payment Processed	{amount:20,from:ext,to:B,fee:0}	Z	v2
19	S_B	7	D8	Payment Received	{amount:20,from:ext}	Z	–
20	S_B	8	D9	Account Flagged	{flag:"Review"}	–	–

Table 2

Auxiliary specifications

Item	Specification
$Snap_A$ (Aggregate A)	Cut: S_A pos 5. Stored state: {balance: 45, status: "active"}
$Snap_P$ (Payment's projection)	Cut: time ≤ D4. Stored state: {completed: {X}}
Normalization policy	"Payment Processed" v1 is normalized to v2 by adding field fee=0 before interpretation.
Branch rule	Branch from S_A pos 5: replace event at S_A pos 6 Deposit(amount=20) with Deposit(amount=15); append Adjustment(amount=5) at new pos 10.

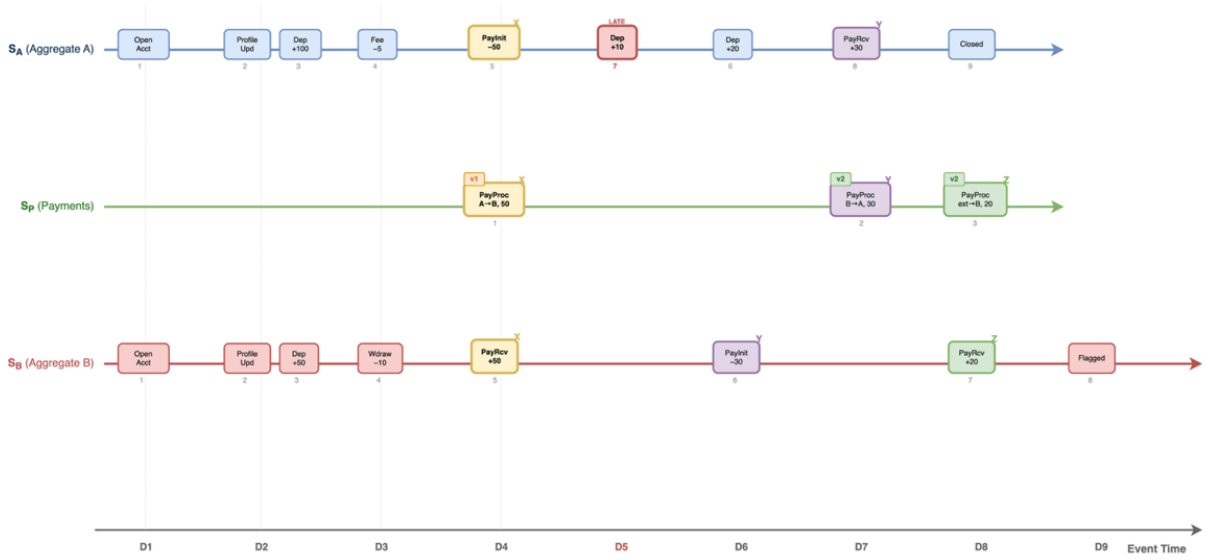

Fig. 7. Experimental dataset

Each mechanism group was evaluated by instantiating the corresponding contract on this dataset. The contracts, semantic outputs, and cost envelopes are summarized in the results section.

Results

The results of the evaluation are presented in Table 3. For reconstruction, the full replay and snapshot-assisted replay return identical readouts at the same cut, which demonstrates replay equivalence and shows that snapshots are a semantic-preserving optimization: they reduce work solely by shortening the replayed prefix ($m-j$ instead of m), not by changing interpretation. For temporal querying, the obtained states at *t1* and *t2* differ exactly as implied by the admissibility rule; the late event affects the later cut but not the earlier one, which illustrates why a temporal cut must explicitly define how out-of-order evidence is treated to keep "state at time t" well-defined and comparable. For cross-stream querying, the transaction-level answer depends on explicit correlation selection and a deterministic merge policy; the cost envelope separates selection work from evaluation work and makes clear how correlation indexes conceptually reduce selection from $|E_{scope}|$ to $|C|$. For retroactive replay, the branch scenario demonstrates that alternative-history evaluation is reproducible when the branch generator is deterministic, and its cost is bounded by reprocessing only the suffix beyond the shared prefix, yielding the expected benefit from prefix reuse.

The obtained results confirm that the same semantic objects defined in the article are sufficient to specify and reproduce query answers without appeal to implementation-specific details.

Table 3

The results of the experiment

mechanism group	contract	semantic output	cost envelope
Reconstruction	scope S_A; cut pos 9; interpreter S_A; readout {balance,status}	{balance:105, status:"closed"}	selected events m=9; interpreter steps=9; normalization steps=0
Reconstruction with snapshots	scope S_A; cut pos 9; start from $Snap_A$ at pos 5	same as full replay	replayed events m−j=4 (j=5); interpreter steps=4; normalization steps=0
Temporal	scope S_A; cuts t1=D5 and t2=D7; admissibility excludes late-appended evidence at t1	at D5: {45,"active"}; at D7: {105,"active"}; Δ(D5,D7]=+60	admissible prefix sizes N(t1)=5, N(t2)=8; interval size=3; interpreter steps proportional to these counts
Cross-stream (by X correlation)	scope {S_A, S_B, S_P}; θ=X; κ=correlation_key; μ=time then S_P>S_A>S_B then pos; normalize v1→v2	Transaction X: Completed (A→B, 50)	selection cost is $\|E_{scope}\| = 20$ under scanning versus $\|C\| = 3$ under keyed retrieval; evaluation processes 3 correlated events plus 1 normalization step (for the v1 PaymentProcessed).
Retroactive replay	branch from S_A pos 5 by rule; evaluate authoritative to pos 9 and branch to pos 10; reuse shared prefix	authoritative: {105,"closed"}; branch: {105,"closed"} (trajectory differs)	prefix length P=5 processed once; suffix lengths: auth=4, branch=5; total steps=14 vs 19 naive

Discussion

The alignment analysis indicates that the proposed formalization functions as a contract-level semantic layer for event-sourced querying. This strengthens the reliability of empirical conclusions and enables comparison between different systems, as correctness and cost claims can be evaluated against the same explicitly declared invariants rather than against implementation-specific conventions.

Existing empirical characterizations and experience reports typically do not provide a compact semantic contract. Observability-oriented studies improve traceability and diagnosis, yet generally treat query meaning as embedded in code and runtime behavior. The present work contributes by specializing the general principles into an implementation-agnostic contract that matches event histories, derived views, evolution handling, and replay-based semantics.

To position the proposed formalization more precisely, Table 4 provides a structured comparison with three established paradigms that address related aspects of querying over data that changes or accumulates over time: temporal relational databases as standardized in SQL:2011 [15], formal frameworks for Complex Event Processing (CEP) [16], and traditional state-storage (CRUD) querying as practiced in conventional relational systems. The comparison is organized along nine dimensions that reflect the core concerns identified in the present work: data model, query target, state reconstruction, temporal reasoning, cross-entity correlation, retroactive analysis, schema evolution handling, cost model portability, and immutability guarantees.

The comparison reveals that each paradigm occupies a distinct position in the design space. Temporal databases provide built-in time-based querying through standardized SQL extensions but embed query semantics in the engine's execution model rather than in a declarative, implementation-agnostic contract; consequently, cost reasoning remains engine-specific and is not portable across implementations [15]. CEP formal frameworks share the principle of operating over immutable streams and offer well-studied automata-based complexity analysis [16], yet they are oriented toward stateless pattern detection and do not formalize the stateful abstractions central to event sourcing: aggregates, projections, snapshots, and version normalization. Traditional CRUD systems store only the current state, which precludes temporal queries, retroactive replay, and any form of history-based reasoning without

external supplementation [1]. The proposed formalization occupies the gap between these paradigms: it is simultaneously stateful (reconstruction and projection), temporal (time-induced cuts with explicit admissibility), correlation-aware (cross-stream querying with declared evidence and merge policy), and history-preserving (retroactive replay with branch semantics), while its cost envelopes remain portable because they are expressed through selected evidence size and amortized per-event processing rather than through engine-specific execution plans.

Table 4

Comparison of the proposed formalization with related query paradigms

Dimension	Proposed ES formalization	Temporal databases (SQL:2011)	CEP formal frameworks	Traditional CRUD
Data model	Immutable append-only event log partitioned into per-key streams	Bitemporal tables (valid-time and transaction-time columns)	Unbounded event streams with schema	Mutable current-state rows
Query target	Histories and derived views (aggregates; projections)	Temporal relations with period predicates	Event patterns and composite events	Current state of entities
State reconstruction	Explicit deterministic replay: φ over declared scope and cut	Implicit via temporal JOIN and AS OF predicates	Not applicable (stateless pattern matching)	Not applicable (state is directly stored)
Temporal reasoning	Time-induced cuts with explicit admissibility rules	Native: FOR SYSTEM_TIME AS OF, BETWEEN...AND	Temporal operators within pattern language	Not natively supported
Cross-entity correlation	Explicit extractor κ with declared merge policy μ	JOIN with temporal alignment (period overlaps)	Sequence and conjunction operators with selection policies	JOIN on current foreign keys
Retroactive / what-if analysis	Branch generator G with outcome comparison under identical φ	Requires external versioning or manual bitemporal manipulation	Not natively formalized	Not supported
Schema evolution	Explicit normalization function τ with separable cost Cτ	ALTER TABLE with migration; no per-query normalization contract	Schema-on-read or external adaptation	ALTER TABLE with migration
Cost model	Portable: O(m · Cφ), explicit contributions from τ and snapshot shortening	Engine-specific (index structure, query plan dependent)	Automata-based complexity bounds (per-event amortized)	Engine-specific (B-tree, hash, query plan)
Immutability guarantee	By definition: events are never modified or deleted	Transaction-time rows are immutable; valid-time rows are mutable	Stream elements are immutable in transit	Rows are freely mutable

In practice, the model can be used as an audit template. Different implementations can be mapped by isolating the scope, cut rule, interpreter, normalization, and cost envelope. Once stated in this form, technologies are comparable by whether they implement the same contract, and performance claims become interpretable at the mechanism level.

The applicability of the model is bounded by its assumptions. It presumes deterministic interpreters and requires explicit ordering/merge policies in multi-stream settings. It resolves distributed uncertainties or read-model staleness under eventual consistency, which are the typical issues for the event-sources systems.

Conclusions

The work addresses the scientific problem of the absence of explicit, shared definitions and semantic contracts that fix what it means to query immutable event histories in event-sourced systems, which makes nominally similar queries non-comparable and obstructs reusable cost reasoning.

The research has developed a compact, implementation-agnostic foundation in which representation and interpretation are fixed by a minimal set of core entity abstractions, and query meaning is fixed at the mechanism level by organizing querying into four groups defined through explicit scope and cut rules. The proposed formalization enables reproducible semantic comparison and portable mechanism-level cost envelopes expressed through selected evidence size and amortized per-event processing, including explicit contributions of normalization and replay shortening.

Prospects for further research include extending the framework toward practice-complete semantics by formalizing admissibility rules under distributed time uncertainty or read-model staleness under eventual consistency.

ADDITIONAL INFORMATION

AUTHOR CONTRIBUTIONS

Conceptualization, Y.G.; methodology, I.Y.; validation, Y.G.; formal analysis, I.Y.; investigation, I.Y.; data curation, Y.G..; writing-original draft preparation, I.Y.; writing-review and editing, Y.G.; visualization, I.Y.; supervision, Y.G.; project administration, Y.G. All authors have read and agreed to the published version of the manuscript.

DECLARATION ON THE USE OF GENERATIVE ARTIFICIAL INTELLIGENCE TOOLS

In preparing this work, the author used DeepL Translate and Grammarly for: grammar and spelling checks, paraphrasing, and rephrasing. After using these tools/services, the author reviewed and edited the content and takes full responsibility for the content of this publication.

1. Overeem M., Spoor M., Jansen S., Brinkkemper S. An empirical characterization of event sourced systems and their schema evolution—Lessons from industry. Journal of Systems and Software. 2021. Vol. 178. Art. 110970. DOI: 10.1016/j.jss.2021.110970

2. Lima S., Correia J., Araujo F., Cardoso J. Improving observability in event sourcing systems. Journal of Systems and Software. 2021. Vol. 181. Art. 111015. DOI: 10.1016/j.jss.2021.111015

3. Kabbedijk J., Jansen S., Brinkkemper S. A case study of the variability consequences of the CQRS pattern in online business software. Proceedings of the 17th European Conference on Pattern Languages of Programs (PLoP'12), Irsee, Germany, July 2012. New York : ACM, 2012. P. 1–10. DOI: 10.1145/2602928.2603078

4. Hlybovets A., Yankin I. Event sourcing pattern and its application. International Scientific Technical Journal "Problems of Control and Informatics". 2025. Vol. 70, № 3. P. 74–84. DOI: 10.34229/1028-0979-2025-3-7

5. Alongi F., Bersani M., Ghielmetti N., Mirandola R., Tamburri D. Event-sourced, observable software architectures: An experience report. Software: Practice and Experience. 2022. Vol. 52. P. 2127–2151. DOI: 10.1002/spe.3116

6. Lytvynov O. A., Hruzin D. L. Critical causal events in systems based on CQRS with event sourcing architecture. Radio Electronics, Computer Science, Control. 2024. № 3. P. 119–143. DOI: 10.15588/1607-3274-2024-3-11

7. Lytvynov O., Hruzin D. Decision-making on Command Query Responsibility Segregation with Event Sourcing architectural variations. Technology Audit and Production Reserves. 2025. № 4. P. 37–59. DOI: 10.15587/2706-5448.2025.337168

8. Mohapatra S. K., Prasad S. Finding representative test case for test case reduction in regression testing. International Journal of Intelligent Systems and Applications. 2015. Vol. 7, № 11. P. 60–65. DOI: 10.5815/ijisa.2015.11.08

9. Fowler M. Event Sourcing. Martin Fowler website. 2005. URL: https://martinfowler.com/eaaDev/EventSourcing.html (дата звернення: 12.12.2005).

10. Sadegi M. Design and evaluation of persistent storage techniques for event-based data : master's thesis. Universität Innsbruck, 2025. URL: https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-85544

11. Overeem M., Spoor M., Jansen S. The dark side of event sourcing: Managing data conversion. 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017 : proceedings. IEEE, 2017. P. 193–204. DOI: 10.1109/SANER.2017.7884621

12. Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM. 1978. Vol. 21, № 7. P. 558–565. DOI: 10.1145/359545.359563

13. Rozsnyai S., Vecera R., Schiefer J., Schatten A. Event Cloud—Searching for correlated business events. 2007 IEEE Conference on E-Commerce Technology : proceedings. IEEE, 2007. P. 409–420. DOI: 10.1109/CEC-EEE.2007.47

14. Chaudhuri S., Narasayya V. AutoAdmin "what-if" index analysis utility. ACM SIGMOD Record. 1998. Vol. 27, № 2. P. 367–378. DOI: 10.1145/276305.276337

15. Kulkarni K., Michels J.-E. Temporal features in SQL:2011. ACM SIGMOD Record. 2012. Vol. 41, № 3. P. 34–43. DOI: 10.1145/2380776.2380786

16. Grez A., Riveros C., Ugarte M. A formal framework for complex event processing. 22nd International Conference on Database Theory (ICDT 2019) : proceedings. Dagstuhl : Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2019. P. 5:1–5:18. (LIPIcs ; vol. 127). DOI: 10.4230/LIPIcs.ICDT.2019.5

Ігор ЯНКІН, Юрій ГУНЧЕНКО
Одеський національний університет імені І. І. Мечникова

ФУНДАМЕНТАЛЬНІ АБСТРАКЦІЇ БАЗОВИХ СУТНОСТЕЙ ТА МЕХАНІЗМІВ ЗАПИТІВ У ПОДІЄ-ОРІЄНТОВАНИХ СИСТЕМАХ

Подієво-орієнтовані системи подають стан застосунку як детерміновану функцію незмінної історії подій, тому запити виконуються над подіями та похідними представленнями, а не над єдиним агрегатом поточного стану. Відсутність спільних, незалежних від реалізації визначень і семантичних контрактів робить номінально подібні запити непорівнюваними та перешкоджає обґрунтуванню вартості. Метою дослідження є розробка незалежного від реалізації формального підґрунтя для базових сутностей і механізмів запитів у подієво-орієнтованих системах, що дає змогу здійснювати аналіз вартості на рівні механізмів незалежно від конкретних технологій. Запропоновано теоретичну методологію, засновану на формалізації та аналізі на рівні механізмів. Дослідження визначає мінімальний набір абстракцій базових сутностей, які задають представлення та інтерпретацію в подієво-орієнтованих системах. На основі цих сутностей, запити формалізовано як детерміноване обчислення з контрактами, що визначено над незмінними історіями та впорядковано у чотири групи механізмів: реконструкції, часові запити, запити, що поєднують потоки та ретроактивне відтворення; кожну групу задано через явні правила області

видимості та зрізу, із задекларованими політиками впорядкування/злиття, кореляції та нормалізації версій. Оцінки вартості отримано шляхом вираження вартості вибору подій та обчислення через розмір вибраних доказів і амортизовану обробку подій, включно з застосуванням політик нормалізації та скороченням реконструкції завдяки збереженим станам. У роботі формалізовано базові абстракції, що не залежать від реалізації, та визначені контрактами механізми запитів для подієво-орієнтованих систем і виведено оцінки вартості. Теоретичний експеримент на штучному банківському наборі подій підтвердив внутрішню узгодженість, еквівалентність відтворення та повторюваність за умови збереження семантичних перетворень. Запропонована формалізація фіксує семантичні ступені свободи, необхідні для відтворюваних та порівнюваних запитів над незмінними історіями подій, і надає основу для оцінки вартості на рівні механізмів у різних реалізаціях. Подальші дослідження мають розширити формалізацію у бік семантики, що буде орієнтована на практику, шляхом формалізації сутностей та запитів за умов невизначеності або застарілості даних в умовах тимчасової неузгодженості даних.

Ключові слова: подіє-орієнтовані системи, подіє-орієнтована архітектура, семантика запитів, формалізація, моделювання витрат.