Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Workload-Adaptive Multiperspective Indexing for Attribute-Aware Queries over Evolving Sequence Databases

Christos Balaktsis

balaktsis@csd.auth.gr

Aristotle University of Thessaloniki

Neron Michail Panagiotopoulos

Aristotle University of Thessaloniki

Anastasios Gounaris

Aristotle University of Thessaloniki

Ioannis Mavroudopoulos

Aristotle University of Thessaloniki

Antonis Panagiotis Liarokapis

Aristotle University of Thessaloniki

John Paparrizos

The Ohio State University

Research Article

Keywords:

Posted Date: November 25th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11638172/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Workload-Adaptive Multiperspective Indexing for Attribute-Aware Queries over Evolving Sequence Databases

Christos Balaktsis[1] · **Neron M. Panagiotopoulos**[1] · **Anastasios Gounaris**[1] · **Ioannis Mavroudopoulos**[1] · **Antonis P. Liarokapis**[1] · **John Paparrizos**[2,1]

Received: date / Accepted: date

Abstract Sequence databases underpin analytical log processing, including process monitoring, cybersecurity auditing, and web interaction analysis. Expressive pattern queries must combine temporal ordering, regular-expression operators, and attribute-aware predicates across multiple grouping perspectives. Existing approaches either assume fixed sequence groupings or lack reusable indices for expressive multiperspective queries, while eager per-perspective materialisation incurs storage and maintenance costs. To this end, we present SIESTA-MP, a workload-adaptive, multiperspective inverted-indexing framework for attribute-aware pattern querying over evolving sequence databases. The framework generalises case-centric sequence indexing, prominent in fields like business processes, to arbitrary attribute-derived grouping perspectives and embeds full event attribute vectors within pairwise inverted-index entries, enabling in-index evaluation of value predicates and cross-event bindings without raw log access. We formulate persistent pair selection as an online index-selection problem and approximate it through a cost-based lifecycle combining on-demand extraction, transient caching, and incremental maintenance driven by query demand and ingestion costs. Experiments on real-world and synthetic logs show that SIESTA-MP reduces incremental indexing time by 28–88% and index storage by 92–96% relative to eager materialisation. Compared with state-of-the-art approaches, SIESTA-MP achieves up to 8.7× faster indexing on a Spark cluster, 2.3–10.8× faster queries over persisted pairs, and up to 26× faster query execution in general.

Keywords Adaptive indexing · Event log · Multiperspective complex queries · Query workload

[1] Aristotle University of Thessaloniki, Greece
E-mail: {balaktsis,npanagii,gounaria,mavroudo,aliarok}@csd.auth.gr
[2] The Ohio State University, USA
E-mail: paparrizos.1@osu.edu

1 Introduction

From business-process executions and ERP transaction logs to market value observations, network telemetry, and security audit trails, many information systems record their behaviour as sequences of events [63,33,36,64,45,34]. Each event captures an activity at a particular point in time, typically enriched with contextual attributes such as the responsible actor, affected entity, or transaction amount [1, 60]. Such collections, commonly referred to as event logs or *sequence databases*, are analysed to reveal behaviours of interest, supporting applications ranging from process optimisation to security investigation [49,32,3,61]. Their analytical value lies not merely in the individual events, but in the *patterns* formed by their temporal relationships: which activities precede others, which alternatives occur, which steps are omitted, and under which conditions a behaviour arises [25,50,23,22].

Crucially, the same event collection can give rise to different sequences depending on the *perspective* under which events are grouped and ordered [2,56,54]. For instance, a business-process analyst may ask "Which applications involved an offer creation followed by its cancellation?", or instead "Which employees repeatedly create offers that are subsequently cancelled?". While both questions concern the same activities, they target different (grouping) entities; the former examines the process from an application-centric perspective, whereas the latter adopts a resource-centric one, potentially revealing different behaviours. Beyond the choice of perspective, meaningful analysis also requires expressive patterns that go beyond fixed activity sequences [62,11]. E.g., "Which employees created offers exceeding $10,000 that were later cancelled by a different employee, without any intervening modification?" combines structural operators, including optionality,

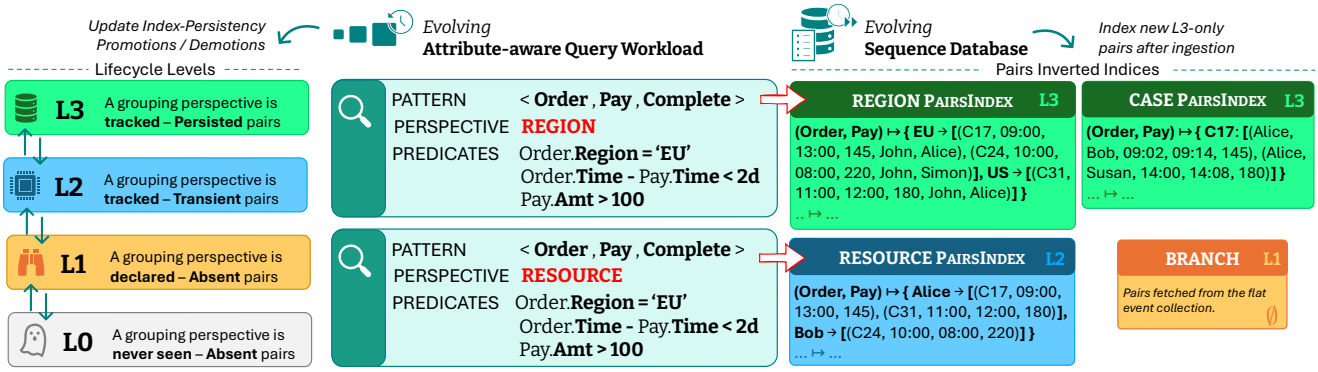

Fig. 1 Motivating example of multiperspective, attribute-aware pattern querying and workload-adaptive indexing. The same activity pattern and predicates yield different matching sequences under different grouping perspectives.

alternatives, repetition, and negation, with value predicates on individual events and cross-event attribute bindings.

Despite decades of research on pattern detection and mining over event sequences [44,11,29], the efficient evaluation of expressive, attribute-aware patterns across different grouping perspectives over the same event collection remains a fundamental challenge. Supporting this form of analysis at scale is a physical data management problem. Queries must combine *structural complexity* (long patterns, alternatives, repetition, optionality, and absence) and *attribute expressiveness* (constant comparisons and constraints across matched events) [62,29], alongside *multiperspectiveness* (different, potentially overlapping event groupings that induce different sequences) [55,20,21]. Figure 1 illustrates how the same attribute-aware pattern yields different matches across grouping perspectives, motivating selective materialisation of perspective-specific activity-pair indices. Cross-event constraints refer to the association of `Time` attribute between ordering and paying, without explicitly matching for a specific timestamp. Meanwhile, queries must also search the accumulated event history, subject to their specified temporal bounds, rather than only events within a transient streaming window [14]. Finally, as the event collection continues to grow, attributes may be sparse or evolve, and previously unused perspectives may become relevant as the analytical workload changes [31,9].

These requirements introduce competing demands on how event sequences are stored and queried. Evaluating patterns directly over the underlying event collection offers flexibility, as different grouping perspectives, structural (regexp) operators, and attribute predicates can be accommodated without predefined access structures. However, querying an increasingly large history may repeatedly incur costly grouping, ordering, and matching operations, particularly for long patterns involving cross-event conditions. Precomputing and retaining reusable structures, such as inverted indices, can alleviate these costs, but supporting multiple grouping perspectives and diverse patterns substantially increases the amount of information to materialise and maintain. Moreover, as new events arrive and analytical interests shift, previously useful structures may become redundant while new ones become desirable. The central, long recognised challenge in workload-driven physical design [40,5] is therefore to reconcile expressive, multiperspective querying over evolving event collections with efficient retrieval and affordable incremental maintenance.

Existing approaches provide important, yet complementary, solutions for this challenge. Inverted-index-based systems, including ELK-Elasticsearch [30] and case-centric sequence indices such as SIESTA [48,46], accelerate retrieval over event repositories; however, they do not provide the workload-adaptive, perspective-specific activity-pair structures studied here. In particular, SIESTA's pair indices are tied to a predetermined sequence grouping and do not support direct evaluation of neither single-value nor cross-event attribute predicates. While Elasticsearch supports indexed event retrieval and query-time perspective grouping, it does not provide reusable, workload-adaptive activity-pair indices across grouping perspectives, leaving repeated historical pattern queries dependent on query-time correlation, and exposed to search-space explosion risks.

Alternatively, search-oriented event-query languages are distinct from selectively materialising and incrementally maintaining reusable pair indices for multiple perspectives: they provide event filtering and sequence correlation. Complex event processing and relational pattern recognition, including Apache Flink's `MATCH_RECOGNIZE`, support such expressive temporal patterns, attribute conditions, and configurable grouping through NFAs [29,62,11,65,67,35]. Even so, their application to repeated queries over accumulated event histories can incur substantial costs: evaluating increasingly long windows, relatively long patterns or reprocessing historical events increases state management and pattern-matching overhead, while eagerly materialising historical sequences across multiple perspectives shifts the burden to storage and incremental maintenance [31,14,46].

Labelled property graph (LPG) representations capture events and their relationships across multiple entities, enabling expressive multiperspective behavioural queries through Cypher [20,21]. However, their application to large, evolving sequence databases faces two major challenges. First, evaluating long and complex patterns, particularly those involving negation, can lead to a combinatorial explosion of intermediate matches, resulting in substantial memory consumption and query latency. Second, although graph databases support incremental updates, maintaining the temporal relationships between events as new data arrives can entail considerable reconstruction and indexing overhead, especially across multiple perspectives. Thus, LPG-based approaches face scalability challenges in both complex pattern evaluation and incremental maintenance.

To address these limitations, we introduce *SIESTA-MP*, a workload-adaptive MultiPerspective indexing framework for attribute-aware complex pattern queries over evolving sequence databases. Building upon SIESTA [48,46], SIESTA-MP extends its case-centric indexing architecture to support multiple coexisting grouping perspectives and expressive attribute constraints. Rather than replicating a complete index for each perspective, SIESTA-MP retains a shared event and activity indexing layer and organises its access structures into perspective-specific activity-pair indices (Figure 2). Each index entry identifies the corresponding group and stores the timestamps and contextual attribute vectors of both participating events. This design enables value predicates and cross-event attribute bindings to be evaluated directly over retrieved index records, avoiding additional joins with the underlying event repository. Perspective–pair combinations without materialised indices remain queryable through extraction from the shared layer.

To control the cost of maintaining these structures, SIESTA-MP introduces a workload-adaptive retention lifecycle that selectively materialises and maintains perspective–pair indices according to observed query demand. Each pair-level structure can occupy one of three states: *absent*, *transient* (LRU-cached), or *persistent* (Delta-backed and incrementally maintained). A cost-based retention policy guides promotions and demotions by balancing estimated query execution savings against index construction and recurring maintenance costs, using decayed workload statistics to adapt to changing query patterns. Consequently, frequently accessed perspective–pair combinations benefit from reusable persistent indices, whereas less frequently requested combinations avoid unnecessary maintenance commitments. Experimentally, this selective materialisation reduces the index footprint by 92–96% and incremental ingestion time by 28–88% relative to eager multiperspective indexing, without restricting access to historical events across perspectives.

Table 1 Capabilities for multiperspective, attribute-aware pattern queries over evolving event histories. *Attr.*: event and cross-event predicates; *Persp.*: alternative event groupings; *PairIdx.*: persistent pair-access structs; *Inc.*: incremental index maintenance; *Adapt.*: workload-driven physical design. Symbols ✓, ×, and / denote supported, absent, and partial or implementation-dependent capabilities, respectively.

Approach	Attr.	Persp.	PairIdx.	Inc.	Adapt.
SPM [4,26,53,44]	/	×	×	×	×
CEP [62,12,6]	✓	/	×	/	×
SQL [65,35]	✓	✓	×	×	×
LPG [20,21]	✓	✓	×	/	×
ELK [30]	/	✓	×	✓	×
SIESTA [48,46]	/	×	✓	✓	×
SIESTA-MP (ours)	✓	✓	✓	✓	✓

Moreover, persisted pair indices accelerate query execution by 2.3–10.8× compared with the case-centric SIESTA.

After visiting the related works targetting the same problem (Sec. 2), this paper makes the following contributions:

1. **Problem formulation (Section 3).** We formalise attribute-enriched sequence databases, attribute-derived grouping perspectives, and queries combining regular-expression-style patterns with value and cross-event constraints. We cast the choice of persistent perspective–pair structures as an online index-selection problem that accounts for query execution, index construction, and recurring maintenance costs.
2. **Workload-adaptive multiperspective indexing (Section 4).** We develop a shared base layer and perspective-specific pair inverted indices that retain both events' contextual attributes. A cost-based lifecycle governs their transient caching, selective persistence, and incremental maintenance, with promotions and demotions driven by decayed workload statistics and estimated costs.
3. **Implementation and evaluation (Sections 5 and 6).** We implement SIESTA-MP on top of SIESTA using Apache Spark and Delta Lake, and evaluate it on several real-world logs (BPI & Loghub) and a synthetic dataset.

2 Related Works

Prior research on event-sequence querying has largely emphasised either physical support for repeated historical access or expressive pattern and grouping semantics, rather than their joint optimisation. Indeed, the ability to specify alternative grouping perspectives or cross-event predicates does not necessarily imply that their evaluation is backed by reusable, incrementally maintained sequence indices. Bridging this gap is the focus of our work. Table 1 summarises where existing approaches stand.

Inverted-indexing approaches. The closest physical-design alternatives are two inverted-index-based systems, which

offer different forms of reusable access to accumulated event histories. SIESTA [48,46] maintains distributed, sequence-oriented inverted indices over activity occurrences and ordered pairs, enabling repeated structural complex[1] pattern queries and incremental updates as the log grows. Its indices, however, are constructed for a predetermined case-based grouping, and do not retain the attribute information that originally accompanies events. Consequently, changing the grouping perspective requires deriving and indexing another sequence organisation, while attribute-aware conditions cannot be evaluated directly from index records. Elasticsearch [30] builds persistent, incrementally updated inverted indices over documents, but the document model forces a structural choice at ingestion time. Indexing each sequence as a document require a pre-defined (typically case-centric) event-grouping criterion, yielding an inverted index equivalent in perspective to SIESTA. Its Event Query Language (EQL) supports temporal sequence correlation and query-time grouping through, allowing different grouping perspectives to be queried without necessarily rebuilding the underlying index, yet bringing higher latencies. Eager materialisation is an alternative that multiplies storage and update costs. Indexing each event as a separate document dissolves the sequence structure entirely, requiring post-retrieval joins over event documents for every query; a regime well known from the literature to degrade sharply as pattern length grows [51,33].

Graph representations. Labelled property graph (LPG) representations offer the richest multiperspective expressiveness among the approaches considered here. Esser and Fahland [20,21] demonstrate that multi-dimensional event data can be encoded in a single graph database, with events, entities, and their temporal directly-follows relationships represented as nodes and labelled edges. Cypher queries can then express complex multi-entity behavioural patterns without a predetermined case notion, with different grouping perspectives naturally realised by varying the entity type traversed, and attribute-aware predicates expressed through standard property access. Despite benefiting from graph-specific access structures and query optimisation, long patterns involving negation and cross-event predicates may still require extensive graph traversal and intermediate-result processing, potentially increasing query latency and memory consumption. LPG-based approaches do not directly provide reusable pair-indices for repeated historical pattern queries. On the maintenance side, evolving event collections require updating temporal relationships between events, potentially across multiple grouping perspectives. These costs motivate an alternative physical design based on selectively materialised activity-pair indices.

Complex event processing. CEP systems, including SASE [62], FlinkCEP [6], and the broader CEP literature [11,29], support expressive temporal patterns, repetition, negation, and attribute predicates across matched events. Their execution model primarily targets continuously registered patterns over event streams, rather than repeated ad hoc queries over accumulated histories. Temporal windows commonly bound the lifetime of partial matches, while historical evaluation may require replaying events or employing dedicated archival mechanisms. Although grouping can be specified per query, changing perspectives generally requires separate pattern evaluation rather than reuse of persistent, perspective-specific pair indices. Moreover, NFA-based execution may accumulate substantial partial-match state for long patterns involving optionality or nondeterministic matching, increasing memory and processing costs depending on the workload. Systems such as DejaVu [14] extend CEP to archived streams, but retain the same window-bounded, partition-fixed execution model without addressing workload-adaptive materialisation of historical partitions across perspectives. Cayuga [12] is the most index-oriented CEP system: it maintains static and dynamic predicate indices over active automaton instances to accelerate transition evaluation and avoid redundant predicate checks across concurrent queries. These indices are query-specific and held in memory for the duration of stream monitoring. Cayuga therefore does not resolve the storage or scalability limitations inherent to automaton-based execution over large, evolving collections.

Relational approaches. SQL `MATCH_RECOGNIZE` [65,41] provides declarative row-pattern matching with `PARTITION BY`, ordering, regular-expression-style operators, and predicates referencing matched rows. A grouping perspective can therefore be selected for a query rather than fixed at ingestion; the challenge is repeatedly organising and matching the corresponding historical partitions. T-ReX [35] demonstrates that historical pattern search need not rely on naive scans: its segment-oriented language and search-space-aware optimizer prune variable-length candidates and share aggregate computation, sometimes building temporary aggregate-specific indices during execution. These optimisations substantially improve complex historical pattern evaluation, but they explicitly concern execution *within a query*: common sub-patterns for varying queries are not benefited. Our work addresses this complementary physical-design problem rather than claiming greater pattern expressiveness than such SQL approaches.

Pattern and Process discovery. Classical sequential pattern mining (SPM) [4,26,53,44] discovers recurring activity

[1] Structural/Temporal complex patterns refer to ordered sequences of activities that include regexp operators, such as Kleene-Star, negation etc. E.g., $\langle A?\ B^{+} \sim C \rangle$: sequences that optionally include activity A directly followed by at least one B and C never follows.

sequences using occurrence (how many *instances* of an activity), ordering, and gap constraints (how many *irrelevant activities* intervene between two activities). Although it can reveal valuable behavioural regularities, its main objective is discovering patterns rather than repeatedly evaluating analyst-specified, attribute-constrained queries under changing groupings. Attribute conditions and cross-event bindings are generally not represented in reusable, perspective-specific sequence postings in these frameworks. Related process-mining research addresses richer semantics through data-aware declarative constraints [59], declarative discovery [13,47], and instance-level analysis [27]. These contributions inform the types of questions analysts wish to answer, but do not by themselves answer repeatedly queried patterns over an evolving event collection.

Workload-adaptiveness. Database cracking [37,58] and automatic physical design [10,5,8,31,40] establish that query workload should govern which access structures are built and retained. Our framework is inspired by this principle, but operates in a different design space: candidate structures are not column partitions or relational indices but activity-pair records whose identity emerges dynamically from the observed query stream, retention must account for recurring maintenance cost under continuous ingestion and not only read acceleration, and the shared base layer means selective materialisation supplements a common structure rather than choosing among independent full replication per perspective. The aim is therefore neither eager materialisation for every perspective nor repeated reconstruction from the base history, but a selectively retained family of attribute-aware pair indices governed by observed workload demand.

Overall, existing approaches either provide efficient indexed access to historical event sequences under fixed grouping perspectives, or support expressive multiperspective querying without corresponding evolving sequences or reusable sequence indices. What remains unaddressed is a non-eager multiperspective inverted-indexing strategy that supports complex, attribute-aware pattern queries over evolving event histories, without requiring either full materialisation of every perspective or repeated reconstruction at query time. Our approach, SIESTA-MP, fills this gap through selective, workload-adaptive materialisation and incremental maintenance of attribute-enriched activity-pair indices across grouping perspectives.

3 Formal Model and Problem Definition

The workloads discussed in Sections 1 and 2 require three capabilities beyond the classical sequence-database model: contextual event attributes for expressive conditions, flexible grouping of the same event collection into different sequence perspectives, and a workload-dependent choice of which data would be beneficial to maintain in some quick-access structures. In this section, we first introduce the event-sequence database model, and then formulate the query workloads under perspectives and attribute-aware predicates. Table 2 summarises the presented notation.

3.1 Plain Sequence Databases

Let $\mathscr{A}$ be a finite set of *activities* (i.e., activity labels or event types), and EV the universe of events. An **event** $ev \in EV$ is the atomic unit of observation, characterised at minimum by an activity label and a timestamp: $ev = (ev.act,\ ev.ts,\ \ldots)$, where $ev.act \in \mathscr{A}$ and $ev.ts \in \mathbb{N}$. The activity projection $\zeta : EV \to \mathscr{A}$ is defined as $\zeta(ev) = ev.act$.

A *flat event collection* $\mathscr{E}$ is a finite multiset of events without an a priori grouping into sequences. A **sequence** $S = \langle ev_1, \ldots, ev_n \rangle$ is a finite tuple of events $\{ev_i\} \subseteq \mathscr{E}$ ordered by strictly increasing timestamps, i.e., $ev_i.ts < ev_{i+1}.ts$ for $1 \leq i < n$. Its length is $|S| = n$, and the derived, zero-based position of ev_i is $pos(ev_i, S) = i - 1$. When convenient, we identify S with its activity projection $\langle \zeta(ev_1), \ldots, \zeta(ev_n) \rangle$. Accordingly, a **sequence database** (or *log*) is a triple $\mathscr{L} = (\mathscr{S}, \mathscr{I}, \gamma)$, where $\mathscr{S}$ is a finite set of sequences, $\mathscr{I}$ is a finite set of *sequence identifiers*, and $\gamma : \mathscr{S} \to \mathscr{I}$ is a bijection assigning a unique identifier to each sequence. The size of the database is $|\mathscr{L}| = |\mathscr{S}|$. [2]

In the classical setting, sequences are typically identified by a predefined case identifier, while contextual event attributes are not part of the basic sequence abstraction. We next extend this model to support different grouping perspectives and attribute-aware queries.

3.2 Attribute-aware Sequence Databases

The classical sequence-database model assumes a fixed assignment γ that determines sequence membership, typically instantiated through a designated case-identifier attribute. This restricts multiperspective analysis, where different queries over the same event collection require different groupings [42]. Moreover, real-world events rarely consist solely of an activity label and a timestamp [2,28]. In practice, they are enriched with *contextual attributes*—such as the responsible resource, organisational unit, geographic region, customer segment, or execution outcome—over which analytical queries are naturally dependent to operate.

Let $\mathscr{K} = \{k_1, \ldots, k_d\}$ be a finite set of *attribute keys*, each associated with a value domain $Dom(k_j)$. An **attribute-enriched event** is a tuple $ev = (ev.k_1, \ldots, ev.k_d)$,

[2] In process mining, $\mathscr{L}$ corresponds to an *event log*, sequences to *traces* or *cases*, and identifiers to *case identifiers*. We use both terminologies interchangeably.

where $ev.k_j \in Dom(k_j) \cup \{\perp\}$ and $\perp$ denotes an absent value. The activity label and timestamp are distinguished, non-missing keys $k_{act}, k_{ts} \in \mathscr{K}$, with $ev.act = ev.k_{act}$, $ev.ts = ev.k_{ts}$, $Dom(k_{act}) = \mathscr{A}$, and $Dom(k_{ts}) = \mathbb{N}$. The remaining keys define the *contextual attribute space* $\hat{\mathscr{K}} = \mathscr{K} \setminus \{k_{act}, k_{ts}\}$. For notational convenience, we write $ev.\mathbf{a} \triangleq [ev.k_j \mid k_j \in \hat{\mathscr{K}}]$ for the contextual attribute vector of ev, $\mathbf{a}[k]$ for the value of key k, and $\mathbf{a}[\mathscr{K}']$ for its projection onto some other $\mathscr{K}' \subseteq \hat{\mathscr{K}}$.

We generalise sequence grouping by treating the log as a flat collection of attribute-enriched events and inducing sequences through an attribute-based grouping function.

Definition 1 (Attribute-Based Grouping Function) Given a flat event collection $\mathscr{E}$ and a non-empty subset of contextual keys $G \subseteq \hat{\mathscr{K}}$ (the *grouping keys*), the **attribute-based grouping function** $\phi_G : \mathscr{E} \to \prod_{k \in G}(Dom(k) \cup \{\perp\})$ maps each event to its tuple of values over G, i.e., $\phi_G(ev) = (ev.k)_{k \in G}$. Each distinct image value $v \in img(\phi_G)$ defines a *group* $\mathscr{E}_v = \{ev \in \mathscr{E} \mid \phi_G(ev) = v\}$.

We call the pair (G, ϕ_G) a **(grouping) perspective**; when ϕ_G is clear from context, we identify the perspective with its key set G alone. Missing grouping values, if present, are treated as the distinguished value $\perp$.

Example 1 (Attribute-Based Grouping) Consider a collection $\mathscr{E}$ with contextual keys $\hat{\mathscr{K}} = \{k_{case}, k_{resource}\}$, comprising ev_1=(Submit, 09:00, C1, Alice), ev_2=(Review, 10:30, C1, Bob), ev_3=(Submit, 11:00, C2, Alice), ev_4=(Approve, 13:00, C2, Alice), and ev_5=(Review, 14:00, C1, Bob). Applying ϕ_G with $G = \{k_{case}\}$ yields two groups: $\mathscr{E}_{\texttt{C1}} = \{ev_1, ev_2, ev_5\}$ and $\mathscr{E}_{\texttt{C2}} = \{ev_3, ev_4\}$, recovering the classical case-centric partition. Applying ϕ_G with $G = \{k_{resource}\}$ over $\mathscr{E}$ yields a resource-centric partition instead: $\mathscr{E}_{\texttt{Alice}} = \{ev_1, ev_3, ev_4\}$ and $\mathscr{E}_{\texttt{Bob}} = \{ev_2, ev_5\}$. Thus, the two perspectives coexist over the same collection.

Definition 2 (Attribute-Aware Sequence Database) For any grouping function ϕ_G, an **attribute-aware sequence database** $\mathscr{L}_G = (\mathscr{S}_G,\ img(\phi_G),\ \gamma_G)$, where $\mathscr{S}_G = \{S_v \mid v \in img(\phi_G)\}$, is defined as follows:

i. Partition $\mathscr{E}$ into $\{\mathscr{E}_v\}_{v \in img(\phi_G)}$ using ϕ_G;
ii. Order each $\mathscr{E}_v$ by timestamp to obtain a sequence $S_v = \langle ev_1, \ldots, ev_{|\mathscr{E}_v|} \rangle$, and assign each event its derived, zero-based intra-group position $pos_G(ev_i) = i - 1$;
iii. Assign the group value as its unique sequence identifier, i.e., $\gamma_G(S_v) = v$.

Two remarks follow. First, the grouping function is most meaningful over discrete-valued keys; out-of-scope continuously valued attributes would require discretisation before they define useful groups. Second, the classical case-centric database is recovered by $G = \{k_{case}\}$. Different choices of G thus induce different *logical* sequence databases over the same $\mathscr{E}$, without requiring that each be eagerly materialised.

3.3 Attribute-Annotated Queries

We consider two analytical tasks: *pattern detection*, which identifies sequences satisfying an expression, and *pattern exploration*, which seeks supported continuations of a given expression. Both rely on a structural pattern language and attribute constraints over matched events.

A **pattern expression** P over $\mathscr{A}$ is defined inductively by the grammar:

$$P ::= A \mid P_1 P_2 \mid P_1 \,\|\, P_2 \mid P^* \mid P^+ \mid P^? \mid \sim A$$

where $A \in \mathscr{A}$ is an atomic activity label; $P_1 P_2$ denotes sequential composition; $P_1 \,\|\, P_2$ denotes disjunction; and P^*, P^+, and $P^?$ are the standard Kleene operators (zero-or-more, one-or-more, and zero-or-one). The operator $\sim A$ requires the absence of activity A between the surrounding positive matched positions. We write $\text{acts}(P) \subseteq \mathscr{A}$ for the set of activity labels appearing in P. A **grounded instance** selects one disjunctive branch and fixes a repetition count for each Kleene sub-expression, yielding a sequence of positive atomic labels $\langle p_1, \ldots, p_m \rangle$ together with absence conditions induced by negated activities. We write $|P| = m$ for the length of a given grounded instance; for patterns without disjunction or Kleene operators, this length is inherent.

A **match** of P in $S = \langle ev_1, \ldots, ev_n \rangle$, for a grounded instance $\langle p_1, \ldots, p_m \rangle$, is defined by positions $1 \le i_1 < \cdots < i_m \le n$ such that $\zeta(ev_{i_j}) = p_j$ for every $1 \le j \le m$, and any absence conditions introduced by negation are satisfied. The corresponding matched subsequence is $\hat{S} = \langle ev_{i_1}, \ldots, ev_{i_m} \rangle$, denoted $\hat{S} \sqsubseteq S$. An occurrence of $\sim A$ prohibits activity A in the corresponding interval between positive matches.

Two containment policies govern which events between matched positions are permitted. Under **strict contiguity** (SC), matched events must be consecutive in S, i.e., $i_{j+1} = i_j + 1$. Under **skip-till-next-match** (STNM), intervening events may be skipped while matching advances to the next eligible occurrence of the required activity. Unless stated otherwise, we use STNM.

A **look-back** window $\lambda \in \mathbb{R}_{\ge 0} \cup \{\infty\}$ bounds the temporal span of a match, i.e., the elapsed time between its first and last matched events. For matched positions $i_1 < \cdots < i_m$, this requires $ev_{i_m}.ts - ev_{i_1}.ts \le \lambda$.

An **attribute expression** e over matched positions $\{1, \ldots, m\}$ is either:

- a constant $c \in \bigcup_{k \in \mathscr{K}} Dom(k)$,
- an attribute reference $\mathbf{a}_j[k]$ to key k at matched position j, or
- a simple arithmetic combination thereof.

An **attribute constraint** ψ has the form $e_1 \odot e_2$ with $\odot \in \{=, \neq, <, \le, >, \ge\}$. A *value constraint* compares an event attribute against a constant (e.g., $\mathbf{a}_j[amount] > 1000$), whereas a *cross-activity constraint*

relates attributes of two distinct matched positions (e.g., $\mathbf{a}_i[resource] = \mathbf{a}_j[resource]$). A *gap constraint* is a particular cross-activity constraint on the derived positions of matched events: for $1 \le u < v \le m$, define $\mathrm{gap}(u,v) \triangleq \mathbf{a}_v[pos_G] - \mathbf{a}_u[pos_G] - 1$, the number of events intervening between positions u and v in the sequence under G. Thus, $\mathrm{gap}(j, j+1) = 0$ requires the corresponding activities to be directly consecutive, whereas the absence of a gap constraint leaves their distance unrestricted inside λ.

Example 2 (Pattern and constraints) The pattern $\texttt{Submit}^+ \sim \texttt{Reject Pay}$ matches one or more submissions followed by a payment, without an intervening rejection. For a grounded instance with m positive matched positions, the cross-activity constraint $\mathbf{a}_1[resource] = \mathbf{a}_m[resource]$ requires the first matched submission and the payment to have the same resource. A value constraint $\mathbf{a}_m[amount] > 1000$ further restricts the payment amount. Without a gap constraint, arbitrary other activities may intervene between the matched submissions and payment, except for the explicitly prohibited rejection. For comparison, $\mathrm{gap}(1,2) = 0$ on $\langle \texttt{Submit}, \texttt{Review}, \texttt{Pay} \rangle$ would require the matched submission and review to be consecutive, without constraining the distance to the payment activity.

Definition 3 (Attribute-Annotated Query) Given a pattern expression P, a finite conjunction of attribute constraints $\Psi = \{\psi_1, \ldots, \psi_r\}$, a perspective G, and a look-back window λ, an **attribute-annotated query** is a tuple $Q = (P, \Psi, G, \lambda)$. A sequence $S \in \mathscr{S}_G$ **satisfies** Q, written $S \models Q$, if there exists a match of P in S satisfying every $\psi \in \Psi$ inside the look-back window.

Matching uses the specified containment policy, and $\Psi = \emptyset$ yields a purely structural query. An infinite window imposes no look-back restriction, i.e., full historical horizon.

Example 3 (Example query) Under the `case` perspective, consider the expression (with abbreviated attribute names *res* and *amt*):

```
Submit_Declaration[res=$r]
  Handle_Payment[res=$r,amt>1000]+
  ~Reject_Declaration
  Handle_Payment[res=$r,amt>1000].
```

It selects *cases* where a declaration submission is followed by one or more high-value payments and a further high-value payment with no intervening rejection; the variable $\$r$ requires the matched activities to have the same resource, without fixing its value in advance.

Definition 4 (Query Workload) A **query workload** $\mathscr{Q}$ is a finite multiset of attribute-annotated queries $Q_i = (P, \Psi, G, \lambda)$ over potentially different perspectives.

The above models define queries independently of their physical execution. Evaluating such queries requires accessing activity occurrences, their ordering, and associated attributes under potentially different grouping perspectives. Repeatedly deriving this information from the underlying event collection can be costly, whereas maintaining dedicated access structures for every possible perspective and query pattern would introduce substantial storage and maintenance overhead. We formalise this trade-off as an online index-selection problem below.

3.4 Adaptive Materialisation-Selection Problem

Supporting repeated attribute-annotated queries over evolving event collections presents a physical-design trade-off. We consider inverted indices over ordered activity pairs (A,B), mapping each pair to its occurrences within sequences induced by a grouping perspective G. Such perspective-specific indices avoid repeated extraction of temporal relationships at query time, but incur construction and update costs as new events arrive. The central decision is therefore not whether to index entire perspectives eagerly, but *which activity pairs, under which perspectives, are worth maintaining* as workload demand changes.

We consider a sequence of workload steps $t = 1, \ldots, T$, each corresponding to a query or an ingestion batch. Let $\mathscr{G}$ denote the set of grouping perspectives that may be requested, and let $\mathscr{M}_t \subseteq \bigcup_{G \in \mathscr{G}} \{G\} \times \mathscr{A}^2$ be the set of *persistently materialised* perspective–pair indices in effect at step t. A triple $(G,A,B) \in \mathscr{M}_t$ denotes a maintained index for the ordered activity pair (A,B) under perspective G. The perspective-specific set $\mathscr{M}_G(t) = \{(A,B) \mid (G,A,B) \in \mathscr{M}_t\}$ identifies the materialised pairs under G. For example, if $G_c = \{k_{case}\}$ and $G_r = \{k_{resource}\}$, then $\mathscr{M}_t = \{(G_c,A,B), (G_r,A,B)\}$ maintains distinct indices for the same activity pair (A,B) under case- and resource-based grouping, respectively.

We model the cumulative **cost of materialisation decisions** over T workload steps, accounting for query execution (*exec*), incremental index maintenance (*maint*), and index construction (*build*):

$$\begin{aligned}\mathrm{Cost}(\mathscr{M}_{1:T}) = \sum_{t=1}^{T} \big[&\mathrm{exec}(Q_t, \mathscr{M}_t) \\ &+ \mathrm{maint}(\Delta\mathscr{E}_t, \mathscr{M}_t) \\ &+ \mathrm{build}(\mathscr{M}_t \setminus \mathscr{M}_{t-1})\big].\end{aligned} \tag{1}$$

Here Q_t denotes the query, if any, at step t, and $\Delta\mathscr{E}_t$ the newly ingested events, if any. The three terms capture query execution cost (including on-demand pair extraction and index read), incremental maintenance of materialised indices, and construction of indices newly included in $\mathscr{M}_t$, respectively. Construction costs are incurred whenever an index is

Table 2 Notation for the data, query, and indexing models.

Symbol	Meaning
$\mathscr{A}$	Universe of activity labels
$\mathscr{K}, \hat{\mathscr{K}}$	Attribute keys and contextual subset
$Dom(k)$	Value domain of attribute key k
$ev, \mathbf{a}$	Attribute-enriched event and attribute vector
$\mathscr{E}, \mathscr{E}_v$	Flat event collection and group of value v
$\zeta(ev)$	Activity label of event ev
$S, \hat{S} \sqsubseteq S$	Sequence and matched subsequence
G, ϕ_G	Grouping keys and grouping function
$\mathscr{G}$	Set of grouping perspectives
$\mathscr{L}_G$	Logical sequence database induced by G
pos_G	Derived, zero-based intra-group position
P, acts(P)	Pattern expression and its activity labels
ψ, Ψ	Attribute predicate and predicate conjunction
acts(Ψ)	Activities referenced by attribute predicates
P_*, P_+	Activities under optional and repetition operators
$P_\sim$	Negated activities in P
gap(u,v)	Events between matched positions u and v
λ	Temporal look-back window
$Q = (P, \Psi, G, \lambda)$	Attribute-annotated query
$\mathscr{M}_G$	Materialised pair keys under perspective G
$\mathscr{M}_t$	Persistent index configuration at workload step t
$f_t(G,A,B)$	Decayed query frequency for pair (A,B) under G
b_t	Decayed ingestion-batch count
$c_{G,A,B}$	Estimated pair-index construction cost
$\bar{s}_{G,A,B}$	Estimated saving per referencing query
$\tilde{m}_{G,A,B}$	Estimated maintenance cost per batch

materialised, including if it was previously removed. Costs independent of the materialisation decisions, such as storing the underlying event collection, are omitted.

Problem 1 (Adaptive Index Selection) Given an evolving flat event collection and a workload of attribute-annotated queries arriving online, select the materialisation sets $\mathscr{M}_1, \ldots, \mathscr{M}_T$ so as to minimise Eq. (1). At each step t, the configuration $\mathscr{M}_t$ may depend only on queries and ingestion batches observed before that step, without knowledge of future workload demand.

The challenge is that pair candidates emerge as different perspectives and queries are observed, while their potential query-time benefits must be weighed against construction and recurring maintenance costs. In the worst, *eager* case, $|\mathscr{G}|\,|\mathscr{A}|^2$ perspective–pair candidates exist, although only a workload-dependent subset is typically requested. Since future workload demand is unknown, an online strategy must approximate the optimal configuration using observed workload statistics. Section 4 presents our indexing architecture and adaptive retention policy for this purpose.

4 Adaptive Multiperspective Indexing

To address the aforementioned adaptive index-selection problem, we introduce SIESTA-MP, a workload-adaptive multiperspective inverted-indexing framework. We first introduce the shared and perspective-specific index structures that materialisation relies on, followed by an adaptive retention policy addressing the adaptiveness trade-off. Two pattern-query algorithms finally complete the pipeline.

4.1 Index Structure Family

We adopt ordered activity pairs as the basic unit of inverted indexing—originally applied and long evaluated in SIESTA [46]—since they capture reusable ordering relationships between events that underpin the evaluation of expressive pattern queries. Activity-pair relationships depend on the sequence boundaries induced by ϕ_G: the same pair of events (e.g. (`Order`,`Pay`)) may belong to one sequence under one perspective (e.g. same `case`) but to different sequences under another (e.g. different `resource` performed each action). Rather than constructing a complete event-log index for each possible G, we maintain a shared event layer and materialise selected per-perspective pair indices on top of it. This design permits attribute-aware pair retrieval under different groupings while avoiding eager replication of the underlying events.

4.1.1 Shared Event Structures

Two perspective-independent structures form the **shared base layer** of the index family, an example of which is depicted in the first row in Figure 2. Both are maintained unconditionally on every ingest batch and are not replicated per perspective, always acting as fallback structures.

The simplest indexing strategy retains every event in a single table, the SEQUENCETABLE, constructed directly from the flat event collection $\mathscr{E}$. The table exists as *one physical instance*. Its schema contains all original event attributes, and two optionally derived columns are populated for a designated *base* perspective: a grouping column $v = \phi_{G_{base}}(\cdot)$ that records the group value of each event, and a monotonically increasing intra-group position pos_v. Position assignment is incremental via an auxiliary SEQUENCEMETADATA table that stores the last assigned position per group, allowing new batches to extend existing groups without reprocessing. Generally, the row schema is:

$$\text{SEQUENCETABLE} : v \longmapsto \big[(act,\, ts,\, \mathbf{a})\big], \tag{2}$$

physically realises $\mathscr{L}_{G_{base}}$ with $v \in \text{img}(\phi_{G_{base}})$.

A familiar base grouping is $G_{base} = \{k_{case}\}$ (case-centric); hence in practice, SEQUENCETABLE partitions the event collection by case id, and assigns intra-case position

numbers to events. For example, a materialised row is:

$$\begin{aligned}\texttt{C1} \mapsto [&(\texttt{Submit}, 202610180801,\\ &[\text{pos}_{\text{case}} = 0, \mathit{res} = \texttt{Alice}, \ldots]),\\ &(\texttt{Approve}, 202610180914,\\ &[\text{pos}_{\text{case}} = 1, \mathit{res} = \texttt{Alice}, \ldots]), \ldots]\end{aligned}$$

Any additional perspective $G' \neq G_{base}$ (or when $G_{base} = \emptyset$) does not induce a second SEQUENCETABLE; instead, the corresponding $\mathscr{L}_{G'}$ is not stored separately but derived at query time by grouping on $\phi_{G'}$ over the shared table. If no base perspective is declared, SEQUENCETABLE $\equiv \mathscr{E}$. Construction costs $O(|\mathscr{E}|\log|\mathscr{E}|)$, dominated by per-group sorting on ts, with $O(|\mathscr{E}'|\log|\mathscr{E}'|)$ per batch.

Next, the ACTIVITYINDEX provides a per-activity inverted view over the SEQUENCETABLE:

$$\text{ACTIVITYINDEX} : \mathit{act} \longmapsto \big[(ts, \mathbf{a})\big], \tag{3}$$

mapping each $\mathit{act} \in \mathscr{A}$ to its timestamp-ordered occurrences together with the full contextual attribute vector. Because every attribute key is retained in $\mathbf{a}$, any perspective G can be projected from the shared ACTIVITYINDEX at any time by filtering on $\mathbf{a}[G]$. Construction requires a single pass over the SEQUENCETABLE, followed by ordering the occurrence lists, yielding $O(|\mathscr{E}|\log|\mathscr{E}|)$ time and $O(|\mathscr{E}|)$ space. For an incoming batch $\mathscr{E}'$, updates cost $O(|\mathscr{E}'|)$ under timestamp-ordered ingestion. For queries anchored on a single activity type, the ACTIVITYINDEX avoids a full SEQUENCETABLE scan by accessing only the relevant occurrence list. Moreover, it provides the shared access structure for both on-demand extraction and persistent construction of perspective-specific pair indices.

4.1.2 Per-Perspective Pair Indices

Building on the shared ACTIVITYINDEX, ordered activity relationships can be materialised as an index for a particular grouping perspective G. Such an index admits at most $|\mathscr{A}|^2$ ordered activity-pair keys, although materialising all of them may incur unnecessary storage and maintenance overhead. For now, let $\mathscr{M}_G \subseteq \mathscr{A}^2$ denote the set of pairs under G grouping, chosen to be materialised, leaving their selection to the adaptive formulation of Section 3.4.

The PAIRSINDEX is the primary query acceleration structure derived from the shared layer. For each $(A,B) \in \mathscr{M}_G$, it materialises intra-sequence occurrences in which A is eventually followed by B within the same group and, optionally, within a temporal look-back λ (if $\lambda \neq \infty$). Since ϕ_G determines the sequence boundaries, qualifying pairs are extracted from the corresponding activity lists in the ACTIVITYINDEX. For distinct activities ($A \neq B$), extraction follows the STNM policy: the earliest eligible following B is paired with an A, consuming the matched endpoints for that pair key. Self-pairs are different: when $A = B$, every two consecutive occurrences within λ are retained, including overlapping pairs, which are essential for reconstructing occurrence chains during query validation.

Algorithm 1 Build PAIRSINDEX and LASTCHECKED

Require: Persp. G, updated groups $\mathscr{V}$, pair keys $\mathscr{M}_G$, look-back λ
Ensure: PAIRSINDEX, LASTCHECKED

for all $V \in \mathscr{V}$ **do**
  $\mathscr{G}_V \leftarrow \{(\mathit{act}, ts, \mathbf{a}) \in \text{ACTIVITYINDEX} : \mathbf{a}[G] = V\}$
  for all $A \in \mathscr{A}$ **do**
    $E_A \leftarrow \big[(ts, \mathbf{a}) \mid (\mathit{act}, ts, \mathbf{a}) \in \mathscr{G}_V,\ \mathit{act} = A\big]$.**sortBy**$(ts)$
  for all $(A,B) \in \mathscr{M}_G : E_A \neq \emptyset \wedge E_B \neq \emptyset$ **do**
    $\tau_{AB} \leftarrow \text{LASTCHECKED}_G$.**get**$(A,B,V)$.**orElse**$(-\infty)$
    $P_{AB} \leftarrow$ **EXTRACTPAIRS**$(V,A,B,E_A,E_B,\tau_{AB},\lambda)$
    if $P_{AB} \neq \emptyset$ **then**
      $\text{PAIRSINDEX}_G[A,B]$.**extend**(P_{AB})
      $\text{LASTCHECKED}_G[A,B,V] \leftarrow \max_{p \in P_{AB}} p.ts_B$
LASTCHECKED_G.**prune**$\big(\{(A,B,V,\tau) \mid ts_{\min} - \tau \geq \lambda\}\big)$

Algorithm 2 Pair extraction under STNM

procedure EXTRACTPAIRS$(V,A,B,E_A,E_B,\tau_{AB},\lambda)$
  $\mathit{pairs} \leftarrow [\,]$
  if $A = B$ **then** ▷ overlapping consecutive self-pairs
    for $j \leftarrow 1$ **to** $|E_A| - 1$ **do**
      $(t_1, \mathbf{a}_1) \leftarrow E_A[j-1]$; $(t_2, \mathbf{a}_2) \leftarrow E_A[j]$
      if $t_2 > \tau_{AB}$ **and** $t_2 - t_1 \leq \lambda$ **then**
        pairs.APPEND$(A,A,V,t_1,t_2,\mathbf{a}_1,\mathbf{a}_2)$
    return *pairs*
  $i \leftarrow 0$; $\mathit{prev} \leftarrow \tau_{AB}$
  for all $(ts_A, \mathbf{a}_A) \in E_A$ **do**
    if $ts_A > \mathit{prev}$ **then**
      $\mathit{matched} \leftarrow$ **false**
      while $i < |E_B|$ **and not** *matched* **do**
        $(ts_B, \mathbf{a}_B) \leftarrow E_B[i]$
        if $ts_B \leq ts_A$ **then** $i \leftarrow i + 1$
        else if $ts_B - ts_A \leq \lambda$ **then**
          pairs.APPEND$(A,B,V,ts_A,ts_B,\mathbf{a}_A,\mathbf{a}_B)$
          $\mathit{prev} \leftarrow ts_B$; $i \leftarrow i + 1$; $\mathit{matched} \leftarrow$ **true**
        else break
  return *pairs*

Example 4 (Extracted pairs) Given a sequence $S_V = \langle A_1, B_2, B_3, A_4, B_5 \rangle$, the (A,B) index contains (A_1,B_2) and (A_4,B_5) under the STNM policy, whereas the (B,B) index retains the overlapping self-pairs (B_2,B_3) and (B_3,B_5).

Pair indices assume a consistent temporal extraction horizon; a query with a different bound requires compatible index contents or an on-demand extraction under that bound. Formally, the index is organised as

$$\text{PAIRSINDEX}_G : (A,B) \longmapsto \big[(V, ts_A, ts_B, \mathbf{a}_A, \mathbf{a}_B)\big], \tag{4}$$

where each record carries the grouping value V, both timestamps, and the *full* contextual attribute vectors $\mathbf{a}_A, \mathbf{a}_B$ of both matched events. This is the defining design choice: attribute-conditioned queries can be evaluated directly against index records, without joining back to the SEQUENCETABLE.

The LASTCHECKED$_G$ is an auxiliary structure that maps (A,B) of PAIRSINDEX$_G$ to a set of group-specific watermarks (V,τ), where τ denotes the last-matched timestamp of B. These watermarks support incremental pair extraction upon subsequent ingestion batches. Stale entries satisfying $ts_{\min} - \tau \geq \lambda$ can be pruned (also from PAIRSINDEX) to bound memory[3]. Algorithms 1 and 2 describe the construction and incremental maintenance of these structures.

Overall, two properties characterise the index family regarding construction costs. First, the index contains at most $|\mathscr{A}|^2$ possible pair keys, of which only those selected for persistence need to be maintained. Each source event in a group contributes at most one record per target activity type, bounding total storage to $O(|\mathscr{A}| \cdot |\mathscr{E}|)$ records. Second, once constructed, the PAIRSINDEX replaces per-query scan-based pair extraction with direct key lookups on (A,B), amortising scan cost across all future queries referencing the same pair. Per ingestion batch, let $n_{\mathscr{V}} = \sum_{V \in \mathscr{V}} |E_V|$ denote the number of events in the updated groups $\mathscr{V}$. Constructing per-activity sorted lists from the ACTIVITYINDEX costs $O(n_{\mathscr{V}} \log n_{\mathscr{V}})$; for each updated group V and each (A,B), EXTRACTPAIRS performs a two-pointer scan in $O(|E_A| + |E_B|)$, and summing across pairs within a group yields $O(|\mathscr{A}| \cdot n_V)$ per group. The total batch cost is therefore $O(n_{\mathscr{V}} \cdot (|\mathscr{A}| + \log n_{\mathscr{V}}))$. Finally, the LASTCHECKED pruning scan, over at most $|\mathscr{A}|^2 \cdot |\mathrm{img}(\phi_G)|$ entries, does not dominate.

4.2 Adaptive Index Lifecycle

To approximate Problem 1 online, we introduce an *adaptive lifecycle* that governs the construction, caching, and persistent materialisation of pair indices. It assigns an *access level* to each perspective G and each activity pair (A,B) requested under G, while a *retention criterion* determines which pair indices enter or leave the persistent materialisation set $\mathscr{M}_t$.

4.2.1 Lifecycle Levels

The lifecycle defines four levels of progressively richer index availability, ranging from an uninitialised perspective to persistently materialised pair indices. At the perspective level, L0 and L1 distinguish between the absence and establishment of an index structure, respectively. At the pair level, L2 and L3 introduce temporary caching and persistent materialisation of individual activity pairs. These levels progressively refine the physical index layout as follows:

L0 — Perspective declared. The grouping function ϕ_G is declared as an available perspective (e.g., G_{base} perspective) but has not yet been queried. No perspective-specific index structure exists beyond the shared base layer, and G is excluded from incremental maintenance. L0 is optional.

L1 — Perspective established. Upon the first query under G, its logical index structure is initialised, but no pair records are materialised. The grouping value $v = \phi_G(\cdot)$ is computed on demand from the shared structures. Establishing the perspective incurs no footprint, and individual pair indices can subsequently be materialised under it.

L2 — Pair cached (transient). An activity-pair index record (A,B) under an established perspective is extracted on demand and retained in a bounded LRU cache for reuse across queries. Cached entries are evicted when maximum capacity is reached or invalidated upon relevant ingestion (new instances of A or B under G), avoiding repeated extraction without incurring persistent maintenance costs.

L3 — Pair persisted. An activity-pair index (A,B) under an established perspective is physically materialised in PAIRSINDEX, together with its corresponding LASTCHECKED metadata. Unlike L2, the index is incrementally maintained as new events arrive, enabling direct retrieval without repeated extraction.

Example 5 (Lifecycle levels) Consider the query `App_Submitted[res=$r] Payment_Handled[res=$r]` under a resource perspective $G = \{\texttt{res}\}$, asking which resource groups contain an application submission eventually followed by a payment handled, both by that same resource. At **L1**, reached on the query's first occurrence, the shared SEQUENCETABLE is scanned, events are on-the-fly grouped by resource, and within each group the two-pointer pair extraction runs. At **L2**, pair records for (`Submitted`, `Handled`) are held in an LRU cache; the query reads from cache at zero extraction cost, but the entry is invalidated on the next ingest batch that contains new `Submitted` or `Handled` events under G, or evicted by competing pairs. At **L3**, an index record is fully materialised and incrementally maintained; the query issues a partition-pruned key lookup, matching conventional inverted-index latency. Across all levels, the attribute vectors $\mathbf{a}_A$, $\mathbf{a}_B$ stored per record make `res=$r` evaluable in-index without joining back to raw events from the shared structures layer.

In Figure 1, we observe the same activity pairs contained in different co-existing indices retained in different lifecycle levels. `Region` and `Case` indices are physically materialised, while the `Resource` logical PAIRSINDEX lives in the LRU.

4.2.2 Retention Cost Function

For retention decisions, the lifecycle distinguishes three pair-level states: *Absent* (no cached or persisted records), *Transient* (L2), and *Persistent* (L3). An absent pair may belong to a perspective at either L0 or L1. Transient pairs

[3] Pruning assumes approximately monotone batch timestamps [48].

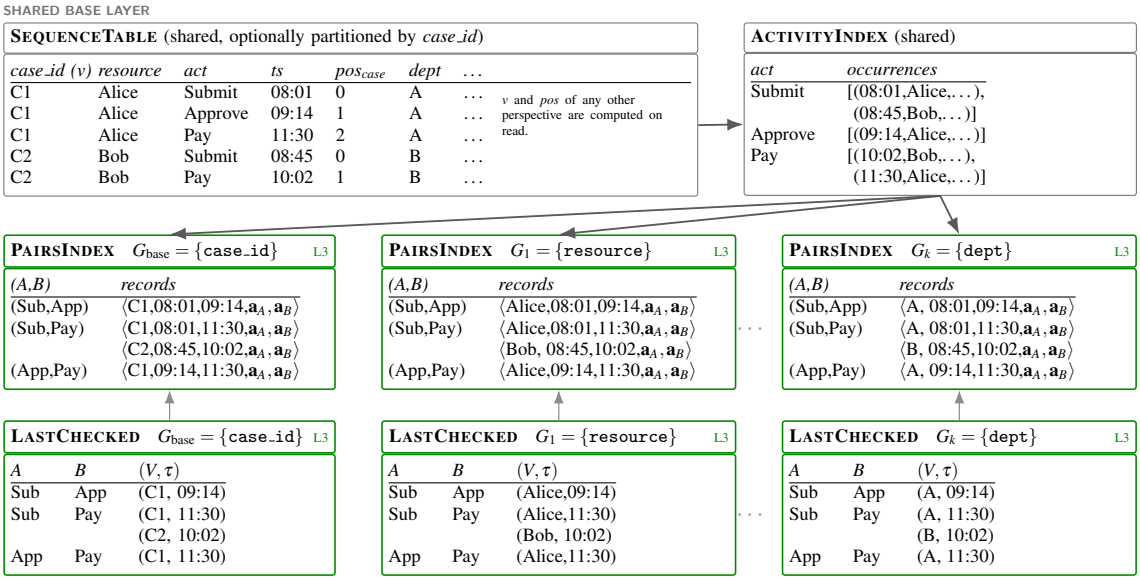

Fig. 2 Overall index family. Per-perspective PAIRSINDEX and LASTCHECKED instances are built upon shared base layer derived by $\mathbf{a}[G]$. Under $G_k = \{\texttt{dept}\}$, `App`→`Pay` yields only one record, because both events share only department A, whereas `Sub`→`Pay` yields two records, since the corresponding source and target events occur in both departments A and B.

are managed by a bounded LRU cache, whose entries are invalidated upon ingestion of their activities under G, rather than incrementally maintained. Persistent pairs, in contrast, incur recurring maintenance costs as new events arrive. The retention criterion therefore determines when persistent materialisation is justified by the expected query-time savings, approximating the trade-off formulated in Problem 1.

We estimate query demand and ingestion frequency using two exponentially decayed counts: $f_t(G,A,B)$ counts queries referencing pair (A,B) under perspective G, while b_t counts ingestion batches. Each corresponding observation increments its count by one. Between observations, either count x decays as $x(t) = x(t_0) \cdot 2^{-(t-t_0)/h}$, where $x(t_0)$ is its value at the last update, t is the current workload step, and h is a configurable half-life. Both statistics are tracked regardless of pair materialisation, allowing the potential maintenance cost to be estimated before an index is persisted, rather than assuming zero cost.

For an established perspective G, we approximate the cost trade-off of Eq. (1) at the individual pair level: the **persistence criterion** compares the estimated reduction in query execution cost against the construction and maintenance costs of $\text{PAIRSINDEX}_G[A,B]$:

$$f_t(G,A,B) \cdot \bar{s}_{G,A,B} \;>\; c_{G,A,B} + b_t \cdot \bar{m}_{G,A,B}, \tag{5}$$

where $c_{G,A,B}$ is the estimated one-time build cost, $\bar{m}_{G,A,B}$ the estimated maintenance cost per ingestion batch, and $\bar{s}_{G,A,B}$ the estimated *potential saving per referencing query execution* from persistent lookup compared with on-demand extraction or transient cache access. Crucially, a cold extraction supplies evidence of the possible saving: its observed extraction latency can be compared with an estimated persistent lookup latency, even though that query itself experienced no saving. Cache hits similarly provide a lookup-cost estimate. Maintenance estimates are calibrated from maintained pairs and subsequently refined by measured ingestion costs; before any maintenance measurement they remain estimates. This allows repeated cold accesses to justify persistent materialisation, even when ingestion invalidates the transient cache between queries.

The criterion is a break-even *heuristic*, not an exact decomposition of Eq. (1): interactions among queries, shared scans, cache hits, and batch costs mean that index benefits need not be fully additive. It compares the potential savings implied by recent query demand against the estimated construction cost and maintenance burden, using exponentially decayed workload statistics.

4.2.3 Moving across Levels

The persistence criterion governs transitions between pair-level states. We refer to the transition of a pair from Absent or Transient to Persistent (L3) as *promotion*, and its removal from persistent storage as *demotion*. To prevent frequent oscillations near the break-even threshold, we introduce a **hysteresis band** $\varepsilon \in [0,1)$. An optional minimum query count $n_{\min}$ further guards against premature materialisation based on insufficient observations.

A pair is **promoted** when its estimated query-time saving exceeds the combined construction and maintenance costs by a margin determined by ε:

$$f_t(G,A,B)\,\bar{s}_{G,A,B} > (1+\varepsilon)\big(c_{G,A,B} + b_t\,\bar{m}_{G,A,B}\big). \tag{6}$$

Promotion additionally requires satisfying the optional minimum query count. Conversely, a persistent pair is **demoted**

when its estimated saving falls below the discounted maintenance cost:

$$f_t(G,A,B)\,\bar{s}_{G,A,B} < (1-\varepsilon)\,b_t\,\bar{m}_{G,A,B}. \quad (7)$$

Note that demotion excludes the construction cost $c_{G,A,B}$, which has already been incurred. The separation between the two thresholds provides hysteresis, allowing existing materialisation decisions to remain unchanged under small workload fluctuations.

Promotion is evaluated after query execution, using the updated demand statistics and cost estimates. Demotion is evaluated during ingestion, before applying incremental maintenance to each persisted pair index. This allows indices whose expected benefit no longer justifies their maintenance burden to be released before incurring the cost of the incoming batch. Both decisions rely only on workload observations available at evaluation time.

Finally, upon promotion, the corresponding pair index is constructed from the shared structures and subsequently maintained incrementally. Upon demotion, its persisted records and LASTCHECKED watermarks are removed, while decayed workload statistics and historical cost estimates are retained for future decisions. If the pair is materialised again, its construction cost is incurred again.

4.3 Query Pipelines

The index structures of Section 4.1 provide access to ordered activity pairs and the contextual attributes of their events. Building on the query-processing paradigm of SIESTA [46], our approach evaluates patterns over compact pseudo-sequences reconstructed from pair-index records, rather than scanning complete event sequences. The query pipeline uses these records for two tasks: *detection* identifies groups (i.e., perspective-induced sequences from the log) satisfying $Q = (P,\Psi,G,\lambda)$, while *exploration* identifies statistically supported continuations of a pattern. Detection first leverages activity pairs implied by pattern P in order to prune candidate groups, then validates the full query over the reconstructed pseudo-sequences using a CEP engine. Exploration invokes detection for candidate extensions.

Pair retrieval. Since both tasks rely on perspective-aware activity pairs, we address this need with the general function **fetch**(G,A,B) that utilises both the aforementioned pair extraction method and the indexing structures. It returns records of schema (4) for pair (A,B) under perspective G. For a Persistent pair (L3), it reads the materialised index; for a Transient pair (L2), it uses the LRU cache; and for an Absent pair, it extracts records from the shared structures using Algorithm 2 and caches the result. For example, detecting $\langle a\,b\,c\rangle$ under G may retrieve (a,b) from L3, (b,c) from L2, and (a,c) through on-demand extraction in a single query. All three paths return the same record format.

4.3.1 Detection

Given $Q = (P,\Psi,G,\lambda)$, Algorithm 3 identifies satisfying groups $\mathscr{R} \subseteq \mathrm{img}(\phi_G)$ in two phases. The *lookup* phase uses on-line extracted and materialised pair records to prune candidates. The *validation* phase evaluates the query over pseudo-sequences reconstructed from the fetched records.

The *lookup phase* first identifies the activity pairs needed to evaluate the query. For each grounded branch of P, it constructs the set OP of ordered pairs between non-negated activities. Among these, only pairs connecting mandatory (non-optional) activities are used to prune candidate groups. Specifically, let $P_\star \subseteq \mathrm{acts}(P)$ denote activities appearing under optional operators ($*$ or ?). The required set RP consists of pairs in OP whose source and target are not in $P_\star$. A group lacking any required pair can therefore be excluded.

Additional pairs are retrieved to support validation, without participating in pruning. Self-pairs provide the occurrences needed to evaluate attribute constraints, repetition, and negation. Let P_+ additionally include activities under Kleene+, $P_\sim$ the negated activities, and $\mathrm{acts}(\Psi)$ the activity labels referenced by attribute predicates. Cross-position constraints, including positional gaps, are denoted by $\Psi_\times \subseteq \Psi$ and require retrieving all relevant occurrences.

For negated activities, the lookup also constructs *anchor pairs* $\mathrm{anch}(A,P)$. These connect a negated activity A to its nearest preceding positive activities, or to its nearest following positive activities if no predecessor exists. Optional neighbours are traversed until a non-optional activity is reached. These pairs are fetched like any other member of SP but never participate in pruning: they supply the occurrence of A that a self-pair chain cannot, e.g., when A occurs once in a group, which validation embeds into the pseudo-sequences discussed later.

Finally, the complete fetch set $FP = OP \cup SP$ combines ordered positive pairs with the additional pairs required for validation. The lookup retrieves their records through **fetch**, intersects candidate groups only for pairs in RP, and passes the resulting records and groups to validation.

The distinction between required and fetched pairs is particularly important for attribute-aware validation. For example, consider a sequence $\langle A_1,B_2,D_3,F_4,B_5,C_6,B_7\rangle$ under perspective G, with the (G,A,B) and (G,B,B) indices materialised. Due to STNM, $\textsc{PairsIndex}_G[A,B]$ retains only (A_1,B_2), discarding (A_1,B_5) and (A_1,B_7) because A_1 has already been matched. This suffices for structural pruning, since (A_1,B_2) witnesses the pattern $\langle A,B\rangle$. However, attribute-aware validation may require later occurrences. For instance, the predicate $\mathbf{a}_B[\mathit{amount}] > \mathbf{a}_A[\mathit{amount}]$ may fail for B_2 but hold for B_5 or B_7, potentially leading to a false rejection if these occurrences are unavailable.

To preserve such alternatives, the lookup additionally retrieves self-pairs. Specifically, $\textsc{PairsIndex}_G[B,B]$ contains

Algorithm 3 Pattern Detection (DETECT)

Require: $Q = (P, \Psi, G, \lambda)$, ϕ_G
Ensure: $\mathscr{R} \subseteq \text{img}(\phi_G)$
▷ ordered pairs of positive activities
$OP \leftarrow \{(p_i, p_j) \mid i < j,\ p_i \notin P_\sim,\ p_j \notin P_\sim\}$
▷ required pairs: optional-free endpoints; participate in pruning
$RP \leftarrow \{(A,B) \in OP \mid A \notin P_\star,\ B \notin P_\star\}$
$needs_all \leftarrow P_\sim \neq \emptyset \vee \Psi_\times \neq \emptyset$
$SP \leftarrow \emptyset$
for all $p_i \in \text{acts}(P)$ **do**
▷ self-pair: reconstruct all occurrences of p_i
  if $p_i \in \text{acts}(\Psi) \vee p_i \in P_\sim \vee p_i \in P_+ \vee needs_all$ **then**
    $SP \leftarrow SP \cup \{(p_i, p_i)\}$
▷ deliver p_i-occurrences via nearest positive neighbour
  if $p_i \in P_\sim$ **then**
    $SP \leftarrow SP \cup \text{anch}(p_i, P)$
▷ fetch pairs: full fetch set, no pruning on $FP \setminus RP$
$FP \leftarrow OP \cup SP$
$\mathscr{R} \leftarrow \text{img}(\phi_G)$; $\mathscr{I} \leftarrow \emptyset$
for all $(A,B) \in FP$ **do**
  $\mathscr{I}_{AB} \leftarrow$ **fetch**(G, A, B)
▷ intersect candidate groups only on required pairs
  if $(A,B) \in RP$ **then**
    $\mathscr{R} \leftarrow \mathscr{R} \cap \{r.V \mid r \in \mathscr{I}_{AB}\}$
  $\mathscr{I} \leftarrow \mathscr{I} \cup \mathscr{I}_{AB}$
return VALIDATE$(Q, \mathscr{I}, \mathscr{R}, \phi_G)$

Algorithm 4 Validation (VALIDATE)

Require: $Q = (P, \Psi, G, \lambda)$; $\mathscr{I}$: index records; $\mathscr{R}$: candidate groups; ϕ_G
Ensure: $\mathscr{R}' \subseteq \mathscr{R}$
$\mathscr{R}' \leftarrow \emptyset$
for all $V \in \mathscr{R}$ **do**
  $\Pi_V \leftarrow \{(r.act_x, r.ts_x, r.\mathbf{a}_x) \mid r \in \mathscr{I},\ r.V = V,\ x \in \{A,B\}\}$
  $\hat{S}_V \leftarrow \Pi_V$.**sortBy**$(ts, tieBreak)$ ▷ pseudo-sequence
  $\mathscr{R}' \leftarrow \mathscr{R}' \cup \{V \mid \hat{S}_V \models Q\}$ ▷ evaluate via CEP engine
return $\mathscr{R}'$

the overlapping pairs (B_2, B_5) and (B_5, B_7), whose records collectively recover all three occurrences and their attribute vectors. Accordingly, self-pairs are retrieved for activities in $\text{acts}(\Psi) \cup P_\sim \cup P_+$, or for all activities when *needs_all* holds.

The *validation phase* finally completes the detection mechanism. Algorithm 4 deduplicates the event occurrences in the fetched records and restores their temporal order to form $\hat{S}_V$. Each occurrence retains its original timestamp and contextual attributes. The original position pos_G is carried in the attribute vector of materialised records. For on-demand extraction, it is computed before validation and retained with any L2 cache entry. Thus, a constraint such as $\text{gap}(u,v) = 0$ is checked against the original intra-group positions, not against positions in the compact pseudo-sequence. The CEP engine evaluates structural matching, attribute constraints, and negation over these records. By Lemma 2, $\hat{S}_V$ is sufficient for evaluating Q over $S_V \in \mathscr{L}_G$: it contains every event occurrence the CEP engine requires, in the correct relative order, without requiring any further join-back to the SEQUENCETABLE.

Regarding total complexity, retrieving pair records costs $O(\sum_{(A,B)\in FP} n_{AB})$, where n_{AB} is the number of fetched records for a pair. Validation is bounded by $O(|\mathscr{R}| \cdot |\hat{S}| \cdot CC(P))$, where $|\hat{S}|$ is the average pseudo-sequence length and $CC(P)$ denotes the pattern-dependent CEP processing cost. The latter dominates for larger $|\mathscr{R}|$. Together these give $O(\sum_{(A,B)\in FP} n_{AB} + |\mathscr{R}| \cdot |\hat{S}| \cdot CC(P))$.

Finally, we complete our detection approach description with two lemmas ensuring soundness and correctness.

Lemma 1 (Lookup soundness) *Every group satisfying Q survives candidate pruning over the required pairs RP.*

Proof Let $S_V \models Q$. By construction, for every $(A,B) \in RP$, a witnessing match contains an occurrence of A followed by an occurrence of B within λ. Under STNM, such a co-occurrence guarantees at least one extracted (A,B) pair for V, even if different occurrences are matched. Thus, V belongs to the candidate groups of every required pair and survives their intersection. □

Lemma 2 (Pseudo-sequence fidelity) *Let $V \in \mathscr{R}$ be a candidate group and let $S_V \in \mathscr{L}_G$ be its corresponding sequence. The pseudo-sequence $\hat{S}_V$ constructed by* VALIDATE *is a sufficient projection of S_V for evaluating Q: $\hat{S}_V$ contains relatively ordered every event occurrence required by the CEP engine to correctly determine whether $S_V \models Q$.*

Proof Sorting deduplicated event occurrences by timestamp (with deterministic tie-breaking) recovers their original relative order, since pair records preserve the original event annotations including pos_G. Positional gap constraints are checked against pos_G, not against the position of an event in the compact pseudo-sequence. It remains to show that $\mathscr{I}$ provides every occurrence the CEP engine may need to witness a match or to rule one out.

Unconstrained activities ($A \in \text{acts}(P)$, $A \notin \text{acts}(\Psi) \cup P_\sim \cup P_+$, $needs_all =$ **false**). For such activities, correctness follows from the STNM semantics: a single co-occurrence record per source event is sufficient to certify that the structural pattern is satisfied in the group. Every pair $(A,B) \in FP$ is fetched, and their flattened source and target events cover at least one witnessing occurrence of each activity at the required pattern position, which is all the CEP engine requires.

Constrained activities ($A \in \text{acts}(\Psi) \cup P_\sim \cup P_+$, or $needs_all =$ **true**). Under STNM, PAIRSINDEX$_G[A,\cdot]$ stores only the first occurrence of each target per source event; alternative occurrences are not directly available from non-self pair records. For this pair-based argument, we assume no activity labels two positions of a branch. Let M be a match of Q in S_V within λ and let a be an occurrence of A that M binds (if $A \notin P_\sim$) or that lies in a region governed

by the negation (if $A \in P_\sim$); we show that $a \in \mathscr{I}$. All events of the span of M are within λ of one another.

(i) a has a neighbouring occurrence of A within λ. PAIRSINDEX$_G[A,A]$ is fetched by construction of SP, and its *overlapping* self-pairs link every two successive A-occurrences at temporal distance at most λ. Hence a is an endpoint of a chain record, and flattening the chain delivers a together with its attribute vector.

(ii) a is isolated, i.e. no other occurrence of A lies within λ of it. Then no chain record contains a, and we use the pairs with the other events of M. If $A \notin P_\sim$, M contains an event e of a non-optional positive element of type $B \neq A$ inside a λ-window. If e follows a, then $(A,B) \in OP$ and a is an eligible source whose first following B is within λ; hence the record $(a,\cdot)$ exists. If e precedes a, then $(B,A) \in OP$ and, by isolation, a is the first A after e; hence a is the target of the record whose source is e, or of an earlier record if e was already consumed. If $A \in P_\sim$, the nearest non-optional positive neighbour of A in P is bound in M to an event within λ of a, and its type belongs to $\mathrm{nbr}^-(A,P)$ (or $\mathrm{nbr}^+(A,P)$ if no predecessor exists) by definition; the same argument applies to $\mathrm{anch}(A,P) \subseteq SP$.

In both cases $a \in \mathscr{I}$, so the occurrences needed to evaluate attribute constraints, absence conditions, repetition semantics, and cross-event bindings are present in $\mathscr{I}$. □

The lemma extends to the full grammar (Kleene, disjunction, negation) via the rewriting and anchor-pair retrieval of Section 4.3.3. Together, Lemmas 1 and 2 establish that candidate pruning preserves satisfying groups and that the reconstructed pseudo-sequences retain the information required for pattern evaluation.

4.3.2 Exploration

Pattern exploration identifies supported continuations of a sequential pattern $P = \langle p_1, \ldots, p_m \rangle$. Algorithm 5 obtains candidate activities x from witnessed ordered pairs with source p_m under perspective G, without scanning the entire activity alphabet. Equivalently, $\mathscr{C} = \{x \in \mathscr{A} \mid \mathbf{fetch}(G, p_m, x) \neq \emptyset\}$. For each candidate, DETECT evaluates $Q_x = (\langle P,x \rangle, \Psi, G, \lambda)$ and reports the fraction π_x of satisfying groups. The lookup records for the common prefix P can be reused across extensions.

Exploration is dominated by the detection calls for candidate extensions. Candidate generation enumerates witnessed outgoing pair keys of p_m; its cost depends on the keys examined, rather than on $|\mathscr{A}|$. Without accounting for prefix reuse, a conservative bound for the detection calls is $O\big(\sum_{x \in \mathscr{C}} [\sum_{(A,B) \in FP_x} n_{AB} + |\mathscr{R}_x| \cdot |\hat{S}_x| \cdot CC(P_x)]\big)$, where FP_x is the fetch set for the extended pattern and $\hat{S}_x$ its average pseudo-sequence. Reusing prefix records reduces repeated lookup work across candidates.

Algorithm 5 Exploration (EXPLORE)

Require: $Q = (P, \Psi, G, \lambda)$, ϕ_G
Ensure: $\mathscr{X} = \{(x, \pi_x)\}$
1: $\mathscr{C} \leftarrow \{x \in \mathscr{A} \mid \mathbf{fetch}(G, p_m, x) \neq \emptyset\}$
2: **for all** $x \in \mathscr{C}$ **do**
3: $\quad \mathscr{R}_x \leftarrow \textsc{Detect}((\langle P,x \rangle, \Psi, G, \lambda),\ \phi_G)$
4: $\quad \pi_x \leftarrow |\mathscr{R}_x| / |\mathrm{img}(\phi_G)|$
5: **return** $\{(x, \pi_x) \mid x \in \mathscr{C}\}$

Alternative approaches. Original SIESTA presents two alternatives to *accurate* exploration method, which EXPLORE adopts: a *fast* heuristic that ranks candidates by upper bounds derived from pre-computed pair statistics, and a *hybrid* method that applies the accurate evaluation only to the top-k candidates of the fast ranking [48]. Neither carries over directly to the present setting. Such statistics aggregate completions irrespective of event attributes and therefore cannot reflect the attribute predicates Ψ of a query; the hybrid method inherits this limitation through its shortlisting stage. For expressions without attribute predicates, adapting the SIESTA algorithms is straightforward. We nevertheless leave their adaptation, and the maintenance of the statistics they require under arbitrary perspectives, out of the scope of the present work.

4.3.3 Handling Complex Pattern Operators

The preceding algorithms focus on sequential patterns. The remaining operators are supported through pattern rewriting and branch-wise evaluation. Rewriting determines which pairs are retrieved, while validation preserves the original pattern semantics.

Disjunction. For $P = P_1 \parallel \cdots \parallel P_n$, lookup and candidate pruning are performed separately for each branch P_i, and the resulting candidate groups are combined by union: $\mathscr{R} = \bigcup_{i=1}^{n} \mathscr{R}_i$. Branches admitting a single-event match alternatively use the ACTIVITYINDEX, since such matches cannot be guaranteed by pair records. A branch whose positive elements are all optional is satisfied by the empty match, imposes no pruning, and $\mathscr{R}_i = \mathrm{img}(\phi_G)$. The fetched occurrences are supplied to the CEP engine, which evaluates the original disjunction ($\mathscr{R}$). Keeping branches separate during pruning avoids imposing pair requirements from mutually exclusive alternatives. Detection is presented for a single branch; the full version wraps it in a loop over $\{P_i\}_{i=1}^{n}$, unions the results, and evaluates *needs_all* once over the whole pattern.

Kleene and optionality. At parse time, optional constructs are rewritten as $P^* \equiv P^+ \parallel \varepsilon$ and $P^? \equiv P \parallel \varepsilon$, recursively where needed. Under our query semantics, a complete ε branch is satisfied regardless of Ψ and therefore contributes all groups $\mathrm{img}(\phi_G)$ without requiring pair lookup. Non-empty branches are processed normally.

Negation. Negated activities $\sim A$ express absence rather than co-occurrence and therefore do not participate in candidate pruning. For validation, self-pairs retrieve consecutive occurrences of A, while anchor pairs recover isolated occurrences that self-pairs cannot capture. Moreover, negation requires retrieving all positive activity occurrences, since an occurrence selected by STNM may violate an absence condition while a later one satisfies it. VALIDATE incorporates these occurrences into the reconstructed pseudo-sequences, where the CEP engine evaluates the prohibited intervals.

5 System Design and Pipeline Implementation

The algorithms and lifecycle levels presented in the previous sections are implemented under a multiperspective indexing framework. We introduce SIESTA-MP: a distributed system based on Apache Spark (PySpark) and Delta Lake [7] over MinIO object storage, extending the original SIESTA architecture [46,48] towards workload adaptiveness and multiperspectiveness. Delta Lake supports transactional updates, while its Parquet-backed storage enables column pruning and predicate pushdown during index retrieval.

5.1 Storage Structures

The four index structures of Section 4.1 are realised as Delta tables. The shared SEQUENCETABLE stores one row per event, optionally partitioned by a designated base grouping value v (e.g., `case_id`), while the ACTIVITYINDEX is partitioned by activity act. For each perspective G with materialised pair indices, PAIRSINDEX$_G$ and LASTCHECKED$_G$ reside in separate perspective-specific Delta tables. Both tables are partitioned by activity pair (A,B), enabling partition pruning when retrieving or maintaining records under G. This Parquet-based layout follows the Spark ecosystem; alternative LSM-based storage designs would support the same application, introducing space–latency trade-offs, as discussed in [46,43].

The deliberate design choice for PAIRSINDEX and the primary point of departure from prior pair-indexed formulation of SIESTA [48] is that each pair-record is proposed to carry the full contextual attribute vectors $\mathbf{a}_A$ and $\mathbf{a}_B$ of both matched events. The motivation behind this is to allow fast constraint evaluation during pattern validation step: since it is performed over arranged fetched events, attribute-conditioned queries can be inherently evaluated via in-place predicate evaluation if events carry their contextual information along, avoiding joins with the shared index layer.

These vectors are introduced and implemented in two alternative representations: column-wise and map-based. In the former, each contextual key $k_j \in \hat{\mathscr{K}}$ is represented by dedicated columns for the two events, supporting column pruning and predicate pushdown. The latter stores each event's attributes in a key–value map, accommodating heterogeneous or evolving schemas at the cost of less efficient attribute-specific access. Both representations preserve the same pair-record semantics and are evaluated in Section 6.

Finally, a *perspective catalog*, implemented as a Delta table with one row per perspective G, records its grouping keys, look-back parameters, lifecycle state, and the statistics required by the retention criterion. These include the decayed query frequency $f_t(G,A,B)$, ingestion-batch count b_t, estimated build cost $c_{G,A,B}$, potential query-time saving $\bar{s}_{G,A,B}$, and maintenance cost $\bar{m}_{G,A,B}$. Pair-level states and statistics are stored in a JSON field within each perspective row, avoiding an $O(|\mathscr{A}|^2)$ expansion of catalog rows. An in-memory singleton serves catalog accesses, with write-through persistence for promotions and write-back updates for batched statistics. The catalog schema can evolve as new statistics are introduced. The transient L2 cache is a bounded LRU keyed by (G,A,B) and is invalidated when ingestion affects the corresponding perspective.

5.2 Incremental Ingestion and Query Execution

Incremental ingestion follows SIESTA's batch-wise processing model [48], extended with selective pair-index maintenance. Each incoming batch first updates the shared SEQUENCETABLE and ACTIVITYINDEX, independently of the current materialisation configuration. For established perspectives with persistent pair indices, applying ϕ_G to the incoming events identifies the affected groups $\mathscr{V}$. Original intra-group positions are derived from the corresponding timestamp ordering when requested.

Before maintaining L3 pair indices, the ingestion statistics are updated and the demotion criterion (Eq. (7)) is evaluated. Indices selected for demotion, together with their LASTCHECKED watermarks, are removed before incurring maintenance costs for the incoming batch. For the remaining materialised pairs, Algorithm 1 builds per-activity occurrence lists for affected groups, extracts new pairs using the LASTCHECKED watermarks, and updates both perspective-specific structures. The measured batch costs then refine the maintenance estimates in the catalog. The updated statistics are also used to evaluate any further materialisation decisions before ingestion completes.

For query execution, a detection query is compiled into a Spark DAG in two stages: driver-side *pattern preprocessing* and distributed *lookup and validation*. The preprocessor rewrites optional and Kleene operators, resolves disjunctive branches (Section 4.3.3), and derives the per-branch lookup sets (Algorithm 3). During lookup, the catalog determines the source of each requested pair: an L3 Delta index, an L2 cache entry, or on-demand extraction from the shared structures. Where supported by the physical representation,

attribute predicates are pushed down during retrieval. The retrieved records are combined by group value V, including the required self-pair records, and redistributed so that each logical sequence is processed within one Spark task.

We deliberately switch SIESTA's CEP engine SASE [62] to OpenCEP[4] [39] for the validation step: it allows distributed CEP execution via `mapPartitions` UDF operations. The pattern and constraints are compiled on the driver once, and broadcast to executors. Each executor processes the assigned groups independently. Each task reconstructs a pseudo-sequence $\hat{S}_V$ from the retrieved occurrences, ordered by pos_G values (or timestamps, equivalently), as required by Lemma 2. OpenCEP then evaluates the pattern over that pseudo-sequence as an in-memory event stream. Since groups are validated independently, no cross-task coordination is needed during CEP evaluation. Thus, validation scales with executor parallelism.

Similarly, exploration reuses the same pipeline. Lookup records for the common pattern prefix are computed once and cached, while candidate extensions require only their additional pair retrieval and group-level validation.

6 Evaluation

We evaluate SIESTA-MP through six research questions that examine its adaptive behaviour, indexing overhead, query performance, scalability, and design choices:

RQ1: How do query latency and index materialisation evolve under workload-driven adaptation (Sec. 6.2)?
RQ2: How much indexing time and storage does adaptive materialisation save compared with eager indexing (Sec. 6.3)?
RQ3: How does SIESTA-MP compare with existing systems for attribute-aware complex queries across grouping perspectives and various lengths (Sec. 6.4)?
RQ4: How do indexing and query execution scale with the number of executors and dataset size (Sec. 6.5)?
RQ5: How do retention parameters and physical representations affect system performance (Sec. 6.6)?

Throughout the section, we refer to our proposal as SIESTA-MP, and to the original, case-centric SIESTA [48, 46], which eagerly indexes every activity pair of a single grouping, as SIESTA. We further denote by *Brute-force* the eager multiperspective SIESTA alternative that materialises every activity pair under any available perspective.

Brute-force is not an existing system; we realise it by running SIESTA once per perspective, each time designating the perspective's grouping attribute as the sequence identifier and storing the resulting indices in a separate storage namespace, which serves the queries posed under that perspective. The original SIESTA indices do not retain event attributes, so SIESTA cannot in general answer attribute-aware queries [48]. To compare the indexing strategies rather than the query fragments they support, we extend the pair records of SIESTA, and hence of Brute-force, with the attribute vectors of both events, exactly as in SIESTA-MP (Eq. (4)); all three systems thus evaluate attribute predicates in-index with the same validation step.

6.1 Experimental Setup

The complete implementation details of the evaluating variants and the experimental suite of competitors and workloads setup are included in our public GitHub repository[5].

Datasets. We use five real-world process logs from the BPI Challenge [15, 16, 19, 17, 18], a synthetic log with controlled sequence lengths and balanced attribute distributions, and two system logs from Loghub-2.0 [66, 38], BGL and HDFS (Table 3). In the system logs, a sequence is the event stream of a node (BGL) or of a block (HDFS), and activities are the log templates (`EventId`). We retain only categorical perspectives with cardinality in $[3, 500]$, more than 1.2 distinct values per trace on average, and a median of at least two distinct activities per group, excluding sparse attributes, non-discrete fields, and perspectives with highly skewed value distributions. For each dataset, we use the `case` perspective and up to three such attributes.

Systems. Besides SIESTA-MP, SIESTA and Brute-force, we compare against three competitors, each queried in its native language. *ELK* is Elasticsearch [30] (v8.13.4), queried through its Event Query Language (EQL) using the `sequence by` operator. *MR* is Apache Flink's SQL `MATCH_RECOGNIZE` [65], which partitions and orders the event relation with `PARTITION BY` and `ORDER BY` per query. *LPG* is the event knowledge graph model of Esser and Fahland [21], stored in the embedded graph DBMS Kuzu [24] (v0.11.3) and queried in Cypher. We use Kuzu rather than Neo4j due to resource limitations. We additionally report *Upper-bound*, an hand-tuned Cypher LPG variant that is not a competitor but an optimistic reference, encoding knowledge of the matching semantics.

Workloads. Query patterns are sampled as contiguous subsequences of real groups (witnesses), with attribute predicates imputed from the witness events, so that every query has a non-empty result and reported latencies reflect genuine matching work. *Structural* queries carry no attribute predicate, *attribute-aware* queries carry literal predicates and cross-event bindings, and *complex* queries additionally apply complex operators. *Skewed* workloads direct 80% of the queries to a small hot set of pairs, whereas *uniform* workloads draw pairs uniformly from the pair space.

[4] `https://github.com/ilya-kolchinsky/OpenCEP`

[5] `http://github.com/siesta-tool/siesta/releases/v4.1.1`

Table 3 Datasets. *Groups* is the number of sequences under the `case` perspective.

Dataset	Domain	Events	Groups	Perspectives (besides `case`)
BPIC 2011 [15]	hospital	150.3K	1,143	`Section`, `org:group`, `Producer code`
BPIC 2012 [16]	loan applications	262.2K	13,087	`lifecycle:transition`, `org:resource`
BPIC 2015 [19]	building permits	52.2K	1,199	`org:resource`
BPIC 2017 [17]	loan applications	1.20M	31,509	`EventOrigin`, `Action`, `lifecycle:transition`, `org:resource`
BPIC 2018 [18]	agricultural grants	2.51M	43,809	`doctype`, `subprocess`, `org:resource`
BGL [66, 38]	supercomputer system log	4.63M	69,227	`Component`, `Level`, `Type`
HDFS [66, 38]	distributed file system log	11.17M	575,061	`Level`, `Component`
Synthetic	n/a	11.5K	2,000	`resource`, `department`, `region`

Protocol. Each query has a timeout of 200 s for every system. For every system, the matched groups are checked against a ground-truth matcher that runs over the same canonical events (identical traces, positions, timestamps and attribute values). Each system is measured in its best case: competitors query pre-built indices or graphs, and SIESTA-MP is measured *warm*, i.e., after the pairs requested by each query have been persisted by an unmeasured run, unless stated otherwise (see Sec. 6.2 for cold measurements.).

Hardware. Single-node experiments run on a machine with an AMD Ryzen 5 3600X (6 cores, 12 threads) and 22 GB of RAM, using Apache Spark 4.0 and Delta Lake 4.0 [7] over MinIO, containerised with Docker. In the competitor comparisons of Section 6.4, each system receives the same budget of 16 GB of RAM and 12 cores: Kuzu uses a 12 GB buffer pool and 12 threads; Elasticsearch an 8 GB heap in a 16 GB container; and SIESTA-MP a 5 GB driver container and a 10.5 GB Spark worker. The distributed experiments (Section 6.5) run on a cluster of four such machines, one hosting the Spark driver and three hosting the executors.

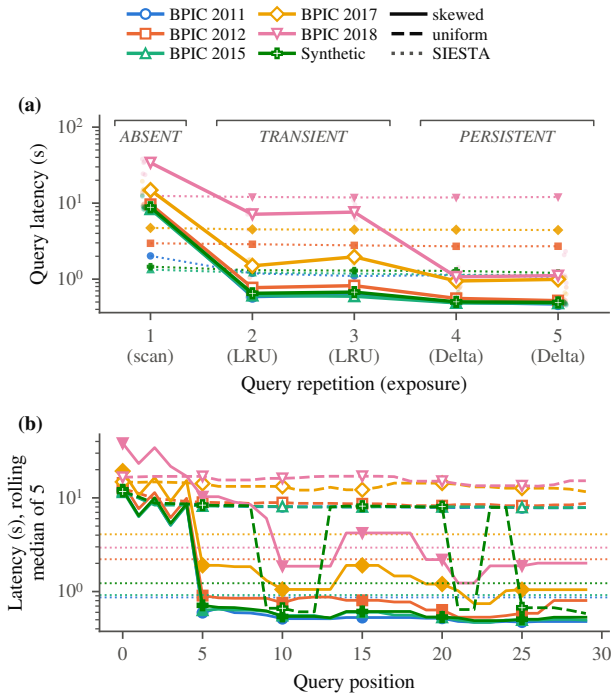

Fig. 3 Workload-driven adaptation in pattern detection under the classical `case` perspective. SIESTA-MP outperforms SIESTA when referencing materialised pairs, due to faster reads from the smaller index.

6.2 Adaptive Indexing

Warm-up under a static log. We first examine how query latency evolves as the index warms. Each dataset is ingested once, no additional batches arrive, and a fixed workload over the five highest-coverage pairs of the case perspective is executed five times against an initially empty pair index. SIESTA answers the same queries from its case index.

The first execution incurs the full cold-start cost of extracting the required pairs on demand from the shared structures. The retention policy then serves the pairs from the transient LRU cache on the second and third exposure, and from persisted Delta tables once their decayed query count reaches the promotion threshold ($n_{\min} = 3$). As shown in Figure 3a, all six datasets follow this tier sequence; SIESTA-MP and SIESTA return identical results on every query. Latency drops from 8.3–34.0 s at the first exposure to 0.47–1.11 s at the fifth, a 15–31× reduction, with the largest one on BPIC 2018 (34.0 s to 1.11 s). From the second exposure on, SIESTA-MP is faster than SIESTA on every dataset, and once its pairs are persisted it is 2.3–10.8× faster (0.47–1.11 s against 1.07–12.04 s), since SIESTA-MP reads a few small per-pair tables, whereas SIESTA filters a larger index holding every pair. The persisted tier is the fastest one: the LRU tier is slower on every dataset, markedly so on BPIC 2018 (7.1 s against 1.1 s). Persistence therefore not only preserves the latency advantage across ingestion cycles, but also delivers most of it.

Skewed vs. uniform workloads. Next, Figure 3b depicts how workload skew affects convergence. Two workloads of 30 case-perspective queries each start from a fresh ingest and an empty catalog: a *skewed* workload directs each query to the three highest-coverage pairs with probability 0.8, and a *uniform* workload draws pairs uniformly from the pair space. Under skew, the hot pairs are served from persisted tables from about the 11th query on. Over queries

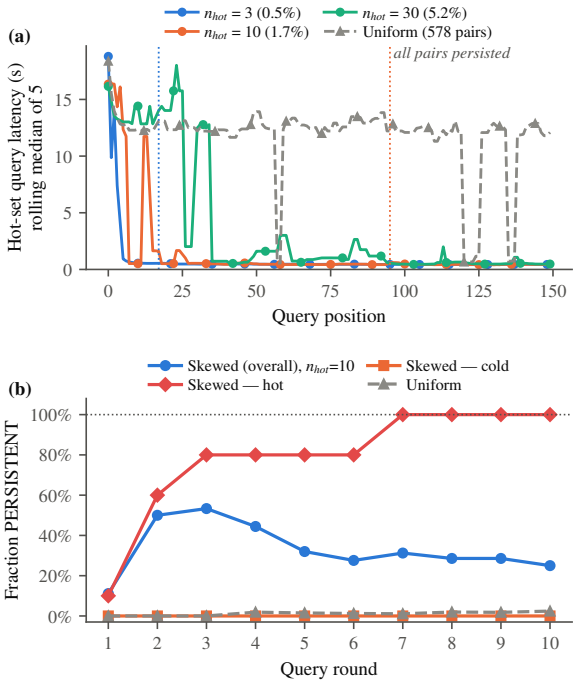

Fig. 4 Effect of the hot-set size on retention convergence (BPIC 2017, `lifecycle:transition`, 578 pairs). (a) Latency of hot-set queries (rolling median of 5) against query position; vertical lines mark the query at which every hot pair is persistent. (b) Fraction of pairs persisted at the end of each round of 15 queries.

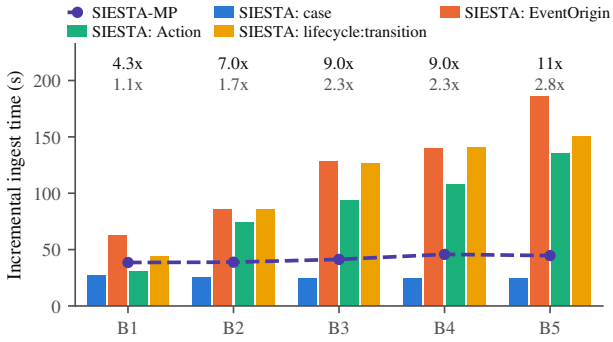

Fig. 5 Incremental ingest time per batch on BPIC 2017: Brute-force's per-perspective ingests (bars) against SIESTA-MP (line), with 3 hot pairs per perspective (12 in total). Top: speed-up of SIESTA-MP over Brute-force. Bottom: speed-up divided by perspectives count.

10–30, hot queries take a median of 0.48–1.16 s, against 1.00–9.61 s for SIESTA (2.1–8.3× faster), and the rolling median falls to about 2 s or less within the first ten queries on every dataset, with transient increases when cold queries cluster (BPIC 2018). Under the uniform workload, almost no pair is queried often enough to be promoted: 0.1–6.8% of the queries are served warm, except on the synthetic log (40%), whose pair space is significantly smaller.

Queries on pairs that are not persisted pay the cost of not indexing up front. A cold query requires an on-demand extraction from the shared structures, which takes 7.9–15.1 s against 0.69–3.43 s for SIESTA, which has every pair indexed. This is inherent to adaptive materialisation: SIESTA-MP trades the latency of the first, infrequent accesses to a pair for the construction, storage and maintenance cost that SIESTA pays for every pair (Sec. 4.2.2).

Figure 4 isolates the effect of the hot-set size on BPIC 2017 under the `lifecycle:transition` perspective (7 groups, 578 co-occurring pairs). Skewed workloads with $n_{\text{hot}} \in \{3, 10, 30\}$ (0.5%, 1.7% and 5.2% of the pair space) and a uniform workload each run 150 queries, in 10 rounds of 15, from a fresh ingest. Smaller hot sets converge faster: every hot pair is persistent at query 17 for n_{hot}=3 and at query 95 for n_{hot}=10, whereas for n_{hot}=30 only 22 of the 30 hot pairs (73%) are persistent after 150 queries, since each hot pair receives proportionally fewer queries. No cold pair is ever persisted under any skewed workload. Under the uniform workload, 3 of the 124 touched pairs (2.4%) are persisted. Across datasets, this fraction ranges from 2.4% to 28.7% and depends entirely on how each log's pair distribution concentrates the uniformly drawn queries: BPIC 2011: 5.3%, BPIC 2012: 17.9%, BPIC 2015: 4.1%, BPIC 2017: 2.4%, BPIC 2018: 28.7%, Synthetic: 11.7%.

The uniform workload is an adversarial case for any demand-driven strategy rather than a representative one. It assigns the same demand to every co-occurring pair, including pairs of rarely co-occurring activities, whereas analytical query workloads are typically highly repetitive; in the Amazon Redshift fleet, for instance, 80% of the queries are exact repetitions of earlier ones in half of the clusters [57]. Overall, SIESTA-MP rapidly promotes frequently queried pairs to persistent indices, substantially reducing query latency, while avoiding unnecessary materialisation under less repetitive workloads. Under uniform demand, the policy instead favours on-demand extraction, trading cold-query latency for lower maintenance costs.

6.3 Indexing Cost: Time and Space

Ingest cost avoided. In many domains a curated list of relevant perspectives is unavailable, so an eager deployment maintains the indices of every declared perspective on every batch, performing one independent ingest per perspective that rebuilds the respective structures [46]. We compare Brute-force against SIESTA-MP on the incremental batches B1–B5 of BPIC 2011, 2012, 2015, 2017 and the synthetic log (BPIC 2018 is excluded due to the duration of Brute-force's ingests). SIESTA-MP performs one ingest into the shared structures, followed by the incremental maintenance of its persistent pairs; its hot set results from a multi-perspective query phase after B0, which persists three hot pairs per perspective. To attribute the saving, we also measure an *all-pairs* configuration of SIESTA-MP that pins every

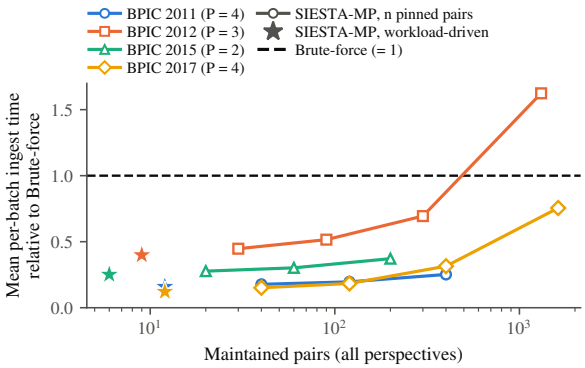

Fig. 6 Mean per-batch ingest time of SIESTA-MP relative to Brute-force against the number of maintained pairs over all perspectives: 10, 30 and 100 pinned pairs per perspective and all pairs (lines), and the workload-driven hot set (stars).

co-occurring pair of every perspective as persistent. It materialises the same pairs as Brute-force, but with a single shared ingest and one table per pair, instead of one full ingest per perspective with one pair table each. Comparing Brute-force with this configuration therefore yields an *architecture gain*, which combines the saving of the shared ingest with the overhead of the per-pair layout, while comparing the all-pairs configuration with SIESTA-MP isolates the *pair-selection gain* of persisting only the demanded pairs; the overall speed-up is the product of the two.

SIESTA-MP reduces the total ingest time by 28–88% (a 1.39–8.21× speed-up; Table 4). The two sources of the saving weigh differently across logs. On BPIC 2012 and 2017, maintaining only the demanded pairs dominates (pair-selection gains of 4.07 and 6.20). On BPIC 2011 and 2015, where all-pairs maintenance was not measured, we can only bound the architecture gain (at most 3.97 and 2.69). On the synthetic log, whose pair space is small (224 pairs), both gains are modest. Moreover, Brute-force's cost grows as the log grows, whereas SIESTA-MP's does not (Fig. 5): on BPIC 2017, Brute-force's per-batch ingest rises from 164.3 s (B1) to 496.1 s (B5), driven by the coarse perspectives, whose few large groups are touched by every batch, while SIESTA-MP stays between 38.5 s and 45.8 s. The per-batch speed-up thus grows from 4.3× to 11.1× (1.1–2.8× per perspective). Of SIESTA-MP's 38.5 s on B1, 6.5 s update the shared structures, 26.5 s maintain the 12 persistent pairs, and 5.5 s evaluate retention. The per-batch speed-up also grows on BPIC 2011 (4.5 to 8.0×) and BPIC 2015 (3.1 to 4.8×), fluctuates within 2.0–2.9× on BPIC 2012, and stays at about 1.4× on the synthetic log.

The saving relies on selectivity (Fig. 6). On every dataset, pinning 10, 30 and 100 persistent pairs per perspective raises SIESTA-MP's mean per-batch ingest monotonically, and persisting all pairs raises it further. On BPIC 2017, it rises from 42 s (workload-driven) to 52, 63 and 108 s, and to 259 s for all 1,621 pairs, against 344 s for Brute-force. The relative cost rises highest on logs where eager ingestion is cheap. Brute-force ingests a BPIC 2012 batch in 56 s on average, against 344 s on BPIC 2017, although both logs maintain about 400 co-occurring pairs per perspective, whereas SIESTA-MP's cost decreases less (22 against 42 s per batch for the workload-driven hot set). On such logs, persisting most pairs exceeds the eager cost: the all-pairs configuration takes 455.4 s against 280.5 s on BPIC 2012 (1.62×, an architecture gain of 0.62, where the per-pair layout's overhead outweighs the saving of the shared ingest), and 177.5 s against 172.6 s on the synthetic log; on BPIC 2017, only its first batch exceeds Brute-force (335.9 against 164.3 s). The cause is the one-table-per-pair layout, whose per-table metadata and clean-up overhead accumulates over thousands of pair tables and, unlike Brute-force's cost, does not shrink with the size of the ingested data.

Space–time trade-off. We next relate storage to query latency. On the case perspective, SIESTA indexes every pair up front, whereas SIESTA-MP stores the shared structures and persists the pairs that a skewed workload keeps querying (50 length-2 queries in 5 rounds of 10, with 80% of them on 10 hot pairs); SIESTA replays the same workload. After the fifth round, SIESTA-MP requires 2.8–9.2× less storage than SIESTA (Table 5), and from the second round on it answers the workload 1.9–6.0× faster at the round median. Most of SIESTA's storage is spent on pairs that are never queried: 90.3% (BPIC 2012), 92.9% (BPIC 2017) and 99.2% (BPIC 2015) of its pair rows belong to pairs that the workload never touches.

The gap widens with the number of perspectives. Under a multiperspective workload (50 length-2 queries, 5 hot pairs per perspective), Brute-force keeps one full index per perspective, whereas SIESTA-MP keeps the shared structures, the per-perspective overlays and the hot pairs. SIESTA-MP then requires 12–22× less storage than Brute-force, a 92–96% reduction (Table 5), and its median latency is lower than Brute-force's under every perspective except one, discussed below. Eager materialisation is moreover impractical for querying under coarse perspectives, where an eager index turns the log into a few very large sequences, each validated by a single task. Under `lifecycle:transition`, `EventOrigin` and `Action`, Brute-force's queries take 37–184 s or time out (BPIC 2017, `lifecycle:transition`), whereas SIESTA-MP answers them in 0.6–5.4 s; Brute-force's queries under attribute perspectives were sampled (two per perspective) because of their duration. The exception is BPIC 2017's case perspective, where SIESTA-MP (6.05 s) is slightly slower than Brute-force (4.82 s): with demand spread over four perspectives, fewer case pairs reach the promotion threshold and more case queries run cold.

Table 4 Total ingest time over 5 batches. *Persp.*: number of perspectives; *All pairs*: SIESTA-MP with every co-occurring pair of every perspective pinned as persistent, i.e., the same pairs as Brute-force, but with a single shared ingest and one table per pair; *Arch.*: architecture gain (Brute-force vs. all pairs), combining the saving of the shared ingest with the overhead of the per-pair layout; *Pair sel.*: pair-selection gain (all pairs vs. SIESTA-MP), isolating the effect of persisting only the demanded pairs. The speed-up is the product of the two gains. †All-pairs maintenance was not measured (more than 3,000 pairs per perspective); the pinned run with 100 pairs per perspective gives a lower bound.

Dataset	Persp.	Brute-force	SIESTA-MP	All pairs	Speed-up	Arch.	Pair sel.
BPIC 2011	4	999.8 s	159.6 s	≥251.8 s†	6.26×	≤3.97	≥1.58
BPIC 2012	3	280.5 s	111.9 s	455.4 s	2.51×	0.62	4.07
BPIC 2015	2	414.5 s	104.2 s	≥154.0 s†	3.98×	≤2.69	≥1.48
BPIC 2017	4	1717.9 s	209.1 s	1297.0 s	8.21×	1.32	6.20
Synthetic	4	172.6 s	123.8 s	177.5 s	1.39×	0.97	1.43

Overall, selective materialisation substantially reduces ingestion time and storage relative to eager multiperspective indexing, while generally improving query latency; these benefits depend on the fraction of pairs selected for persistence and the distribution of query demand.

6.4 Query Performance against Competitors

No single baseline captures all the requirements studied in this work (Section 1); Section 2 discusses where each falls short. Since a per-sequence ELK index fixes one grouping at ingestion, we index each event as a document, so that EQL assembles the sequences of any perspective at query time; the LPG holds case entities only, so other perspectives are formed by joining event nodes on equal attribute values.

We compare the systems on attribute-aware complex queries. Under the case perspective, each query carries 1–3 literal predicates as well as cross-event bindings and combines disjunction and negation (30 queries per length); the attribute-perspective workload is described below. For the LPG, we evaluate two Cypher formulations, both executed entirely by Kuzu. The *plain* formulation is the generic one: a single `MATCH` binds every pattern event to the same group, with `NOT EXISTS` for negation and `IN` for disjunction. The *step-by-step* formulation, denoted *Upper-bound (LPG)*, fetches events progressively along the pattern: a chain of `WITH` stages keeps, for each group, only the earliest position at which it matched the previous element, and for a negated gap keeps only the latest preceding event before checking the gap. Upper-bound is therefore a reference rather than a competitor. It encodes the existence-semantics facts on which SIESTA-MP's own validation relies, is tailored to the workload, and returns group identifiers only; an analyst cannot be expected to hand-write such a query for every pattern, and no other system is given an equivalent advantage.We treat the plain formulation as the LPG's native performance, and report Upper-bound only as an optimistic reference for what a graph database reaches when an expert hand-tunes every query. Figures 7 and 8 report the median latency per pattern length; a length whose median query fails (timeout, memory error, crash or wrong result) is drawn above the timeout.

Figure 7 covers all seven real logs under the case perspective, for lengths 11–20. MR and the plain LPG answer only the shortest patterns, of lengths 11–13 at most, in 12–193 s, and fail at every longer length on every log; MR evaluates each query by rescanning the partitioned

	Case perspective		All perspectives	
BPIC	SIESTA	SIESTA-MP	Brute-force	SIESTA-MP
2012 (3)	15.1	5.3 (2.8×)	57.6	4.7 (12.3×)
2015 (2)	47.1	5.1 (9.2×)	112.7	5.0 (22.5×)
2017 (4)	293.2	89.5 (3.3×)	1,138	74.9 (15.2×)

Table 5 MB-storage of eager and adaptive materialisation after the skewed workloads, on the case perspective and over all perspectives on BPIC datasets (number of perspectives in parentheses).

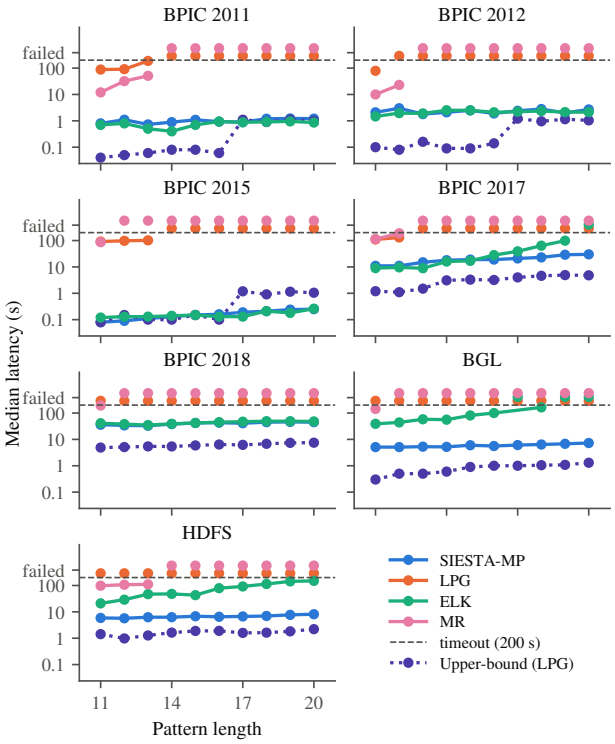

Fig. 7 Attribute-aware complex queries under the case perspective: median latency per pattern length. Upper-bound is hand-tuned Cypher.

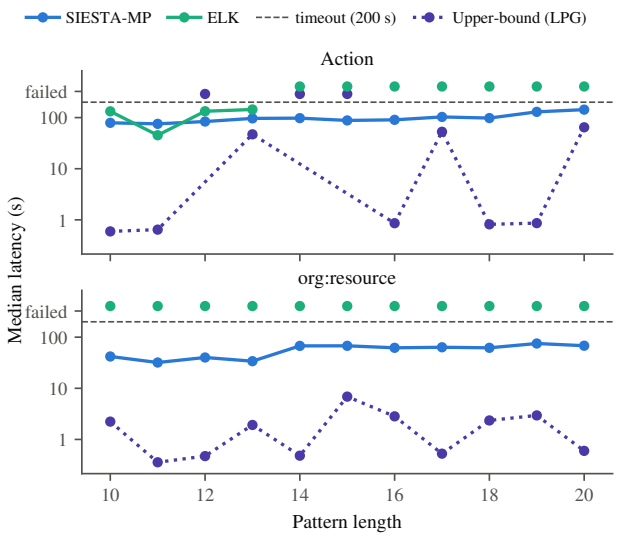

Fig. 8 Attribute-aware complex queries on BPIC 2017 under the `Action` and `org:resource`: median latency per pattern length.

relation, and plain Cypher materialises the joins among all candidate events of a group. SIESTA-MP and ELK answer far more, and are close to each other on BPIC 2011, 2012, 2015 and 2018, within about a factor of two at every length, with medians of at most 3 s on the first three logs and 33–49 s on BPIC 2018. They diverge on the logs with many and long sequences. On BPIC 2017, ELK is slightly faster up to length 13 (8.9–9.6 against 11–15 s), but its latency then grows to 99 s at length 19 and fails at length 20, whereas SIESTA-MP grows from 11 to 30 s. On BGL and HDFS, ELK is 8–26× and 4–18× slower than SIESTA-MP, respectively, and fails at most BGL lengths from 17 on, whereas SIESTA-MP stays at 5–8 s on both logs. SIESTA-MP's latency grows moderately with length on every log, by 1.3–3.1× from length 11 to 20 (the largest growth on BPIC 2015, from 0.08 to 0.25 s). As a reference, Upper-bound (LPG) is the fastest on most logs: 3.7–17× faster than SIESTA-MP on BPIC 2017, BPIC 2018, BGL and HDFS, and sub-second on BPIC 2011, 2012 and 2015. From length 17 on, its latency rises to about 1 s, comparable to SIESTA-MP and ELK on BPIC 2011, still about 2× faster on BPIC 2012, and slower on BPIC 2015.

Figure 8 repeats the comparison on BPIC 2017, whose attribute perspectives distribute events most evenly across groups among our logs, under `Action` and `org:resource`, for lengths 10–20. Here, each length has 5 attribute-aware queries with 1–3 literal predicates and no cross-event bindings, combining 1–3 operators among optional, one-or-more, disjunction and negation; since EQL cannot express the optional and one-or-more operators, ELK's medians are taken over the queries it can express. No system knows these perspectives a priori; SIESTA-MP builds their pairs on demand from its shared structures. The plain LPG formulation is not shown: it already fails beyond length 13 on every log under the case perspective (Fig. 7), and timed out on a sample of the `Action` queries. Here, the gap between SIESTA-MP and ELK becomes decisive. ELK fails at every length under `org:resource` and from length 14 on under `Action`, answering lengths 10–13 in 45–144 s, as assembling the sequences of these large groups at query time becomes prohibitive. SIESTA-MP's median query succeeds at every length of both perspectives, in 76–143 s under `Action` and 32–75 s under `org:resource`.

Under `Action`, some of its queries time out, but fewer than 30% at every length, so the medians reflect answered queries; these timeouts coincide with memory terminations of its Spark worker, which holds each group whole during validation. Its latency is higher than under the case perspective because these groups are much larger than cases, and each is validated within a single task. Upper-bound (LPG) answers every `org:resource` length in 0.4–6.9 s, but fails at `Action` lengths 12–15. Its failures are negation queries that exhaust memory: a group formed by joining on an attribute value holds hundreds of thousands of events, and Cypher can only pair candidate events against each other, whereas SIESTA-MP validates each group in a single pass. The Upper-bound reference shows that hand-tuned graph queries can be faster under the case perspective, but they require query-specific reformulations and still fail on the negation queries of the `Action` perspective.

Overall, SIESTA-MP maintains comparatively stable latency as pattern length increases and remains competitive with the strongest native-query alternative under the case perspective. Its advantage becomes more pronounced for long sequences and attribute-derived perspectives, where competing native queries increasingly fail, whereas SIESTA-MP's median query succeeds at every tested length. Hand-tuned graph queries can be faster, but require query-specific reformulations and also exhibit failures under coarse perspectives.

6.5 Scalability and Distributed Execution

We replicate BPIC 2017 by sampling traces and re-inserting them under new trace identifiers (×1–×3 its size), and vary the executor cores on the four-machine cluster. We compare SIESTA against SIESTA-MP on structural and attribute-aware case-perspective queries, and evaluate SIESTA-MP under the `Action` and `org:resource` perspectives.

Indexing. On ×2 and ×3, SIESTA-MP indexes 4.7–8.7× faster than SIESTA at every core count (Fig. 9a). Since it builds only the shared structures, SIESTA-MP is already fast at three cores, whereas SIESTA needs additional cores to reduce its pair-index construction (up to 2.0× faster from 3 to 18 cores). SIESTA-MP also grows more slowly with the data: at 18 cores, tripling the log from ×1 to ×3 increases SIESTA's indexing time by 1.75× (145.8 to 255.4 s) and

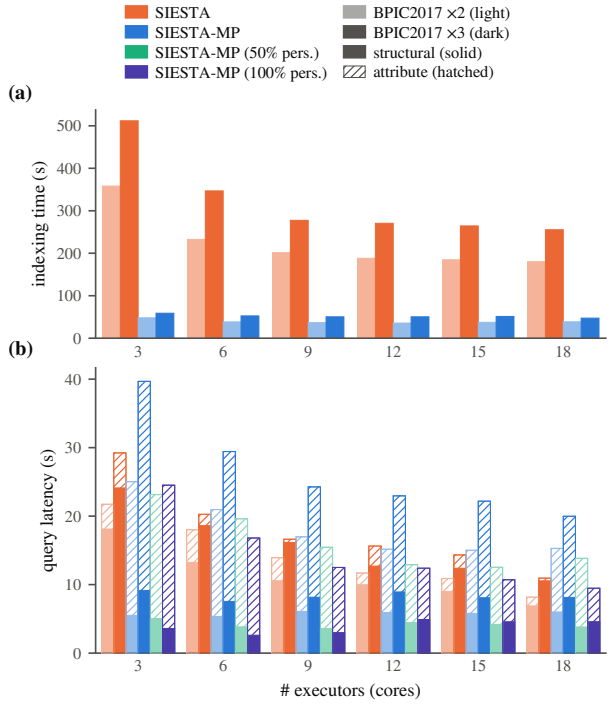

Fig. 9 Scale-out ((a) indexing; (b) querying) on replicated BPIC 2017.

SIESTA-MP's by 1.48× (31.9 to 47.3 s). Part of SIESTA's cost is the shuffle that co-locates the events of each group for pair extraction, which is measured growing with the data from 0.8–1.3 s per executor on ×1 to 3.1–4.7 s on ×3, whereas SIESTA-MP's ingest shuffles 0.05–0.55 s per executor in every configuration (9–20× less at 18 cores).

Query latency. SIESTA-MP answers structural queries 1.1–3.3× faster than SIESTA in every configuration, with latency that barely varies with the number of cores, whereas SIESTA approaches it only with more cores (Fig. 9b). In the default configuration, attribute-aware queries run cold: their pairs are not yet persisted and are extracted on demand from the shared structures. This extraction benefits from more cores (up to 2.0× from 3 to 18), but remains slower than reading SIESTA's index (up to 1.9× at 18 cores). Persisting half of the queried pairs (×2) narrows this gap only slightly. With all of them persisted (×3), SIESTA-MP is faster than SIESTA at every core count, by 2.3–7.1× for structural and 1.2–1.3× for attribute-aware queries. Under the `Action` and `org:resource` perspectives, structural and `Action` attribute-aware queries take 0.9–2.4 s at every data size, whereas `org:resource` queries grow almost linearly with the data (14.3, 27.9 and 40.5 s at 18 cores).

Summarising, SIESTA-MP achieves 4.7–8.7× faster distributed indexing than SIESTA and exhibits slower indexing-time growth with increasing data size. However, query scalability is constrained by the distribution of events across perspective-specific groups, as large, skewed groups limit parallelism during validation.

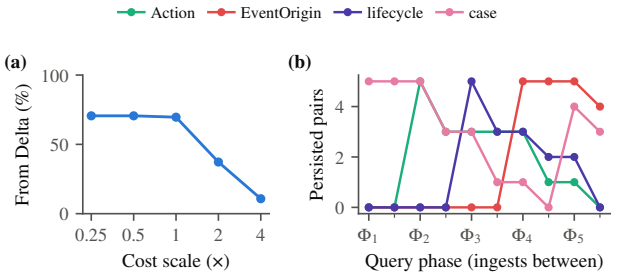

Fig. 10 Sensitivity of the retention policy on BPIC 2017. (a) Share of pairs served from persisted tables when the estimated costs are scaled by 0.25–4×. (b) Persisted pairs per perspective after each query phase (Φ) and ingestion in-between, as the hot perspective moves from case to `Action`, `lifecycle:transition`, `EventOrigin` and back to case.

6.6 Sensitivity Analysis

We study the retention policy on BPIC 2017 with incremental batches: each run starts from B0 and alternates query phases of length-2 pair queries with the ingestion of B1–B5, at which demotion is evaluated (Section 4.2.3).

Hysteresis band. We sweep $\varepsilon \in \{0, 0.05, 0.15, 0.3, 0.5\}$ under graded Zipf [52] demand over 12 pairs, with rankings rotated between phases and a half-life of 60 s. Across all settings, 120 queries trigger the same 9 promotions, 12 demotions and 2 re-promotions, with marginal differences in median latency (1.19–1.33 s) and maintenance time (41.6–44.3 s). The demand guard delays promotion until estimated savings exceed build costs by 3–6×, while demotions occur after demand has largely decayed. Thus, the hysteresis band has negligible impact under the tested workload.

Cost misestimation. We scale the estimated build and maintenance costs ($c_{G,A,B}$ and $\bar{m}_{G,A,B}$) by 0.25–4× under a bursty workload (bursts of 30 queries, 90% hot, alternating with quiet phases of 6 queries, 10% hot). Underestimated costs are harmless: decisions match those under the measured costs, with about 70% of pair accesses served from persisted tables (Fig. 10a) and a median latency of 0.61–0.64 s. Overestimated costs degrade gracefully: fewer pairs are persisted (37% and 11% of accesses at 2× and 4×), and median latency rises at most 2.2×.

Drift. Each query phase directs a burst to a different perspective. Every burst persists all five hot pairs of its perspective, whereas the pairs of earlier perspectives are demoted gradually over the following ingestions, as their decayed demand fades, so that each perspective's footprint shrinks to zero once it is no longer queried (Fig. 10b). When the workload returns to the case perspective, four of its five pairs are re-promoted within the burst. Over the 150 queries, the policy performs 19 promotions and 17 demotions, with a median latency of 0.72 s; re-adapting to four raises the total maintenance to 117 s, against 43 s for a single perspective.

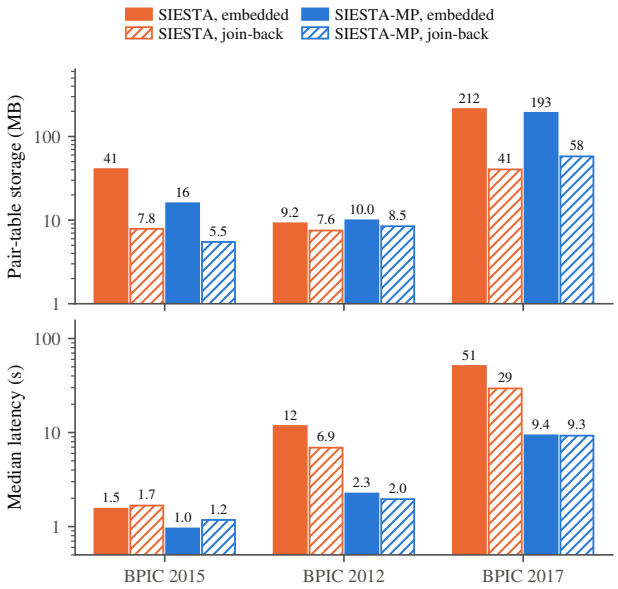

Fig. 11 Attribute vectors embedded in PAIRSINDEX records vs. joined back from shared structures at query time.

Attribute representation. Section 5.1 introduces two physical representations of the attribute vectors in PAIRSINDEX records: a column-wise one , with dedicated columns per attribute key, and a map-based one, with a key–value map. We compare them on HDFS and BGL, with 30 structural and 30 attribute-aware queries of lengths 3–15 over persisted pairs. The two designs behave almost identically: pair-fetching latency differs by less than 5 s in every comparison, and end-to-end query latency differs by roughly 0.3%.

Embedded attributes vs. join-back. Embedding the attribute vectors of both events in every pair record trades storage for in-index predicate evaluation. To quantify this trade-off, we index BPIC 2012, 2015 and 2017 twice with SIESTA and SIESTA-MP: once with embedded vectors (the default), and once without them, in which case the attributes of a pattern with predicates are joined back at query time from ACTIVITYINDEX, after pruning and only for the pattern's events (15 structural and 15 attribute-aware queries of lengths 2–15, case perspective, both systems warm). Join-back reduces the pair-table storage by 1.2–5.3× for SIESTA and 1.2–3.3× for SIESTA-MP, most on BPIC 2015 and 2017 and least on BPIC 2012 (Fig. 11). Its effect on median latency is small for SIESTA-MP, between 1.2× slower (BPIC 2015) and 1.15× faster (BPIC 2012), whereas SIESTA becomes faster with join-back on BPIC 2012 and 2017 (12 to 6.9 s and 51 to 29 s).The latency effect of join-back is thus dataset-dependent, and may relate to log characteristics, requiring further investigation.

Concluding, the retention policy adapts to shifting query demand and is relatively robust to hysteresis and cost estimation errors. Physical attribute layouts exhibit negligible performance differences, while embedding attributes introduces a dataset-dependent storage–latency trade-off.

6.7 Discussion and Limitations

Two limitations follow from the experiments. First, adaptivity defers cost rather than eliminating it: a query on a pair that is not persisted pays an on-demand extraction (7.9–15.1 s against 0.69–3.43 s for SIESTA in Section 6.2), so SIESTA-MP favours workloads with repeated demand, which analytical workloads typically exhibit [57], and its savings shrink, or reverse, if most pairs must be persisted (Fig. 6). Second, validation dominates the latency of long patterns over large groups. Since each executor validates one group per task and pair records are skewly distributed across the groups of coarse perspectives, the largest groups bound both parallelism (Section 6.5) and memory, since each group is held whole by a single validation worker; and the hand-tuned Upper-bound LPG formulation answers case-perspective patterns 3.7–17× faster on four logs (Section 6.4). These observations motivate the future directions of Section 7, namely cheaper validation through selective pruning and a finer-grained distribution of large groups.

7 Conclusions and Future Work

We presented SIESTA-MP, a workload-adaptive inverted-indexing framework for expressive, attribute-aware pattern queries over evolving sequence databases. By combining a shared event and activity layer with selectively materialised perspective-specific pair indices, SIESTA-MP supports multiperspective querying without eagerly indexing every perspective. A cost-based retention lifecycle dynamically caches, persists, and demotes pair indices according to query demand and maintenance costs, while embedded event attributes enable in-index evaluation of value and cross-event predicates.

Experiments on seven real-world logs and one synthetic dataset demonstrate 28–88% lower incremental ingestion time and 92–96% lower multiperspective index storage than eager materialisation. Persisted indices achieve 2.3–10.8× faster queries than the case-centric indexing baseline, while comparisons with state-of-the-art approaches demonstrate the benefits of SIESTA-MP for long, attribute-aware patterns across different grouping perspectives.

Future work will focus on reducing the overhead of on-demand pair extraction and CEP-based validation through selective pruning, and improving distributed execution through finer-grained processing of large, skewed groups. We also plan to investigate more efficient persistence layouts to reduce per-pair maintenance overhead and automatic tuning of retention parameters under evolving workloads.

References

1. Van der Aalst, W.M.: Object-centric process mining: dealing with divergence and convergence in event data. In: International Conference on Software Engineering and Formal Methods, pp. 3–25. Springer (2019)
2. van der Aalst, W.M.P.: Object-centric process mining: Unraveling the fabric of real processes. Mathematics **11**(12), 2691 (2023). DOI 10.3390/math11122691. URL http://dx.doi.org/10.3390/math11122691
3. Agarwal, B., Mittal, N.: Hybrid approach for detection of anomaly network traffic using data mining techniques. Procedia Technology **6**, 996–1003 (2012)
4. Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proceedings of the Eleventh International Conference on Data Engineering, pp. 3–14 (1995). DOI 10.1109/ICDE.1995.380415
5. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in sql databases. In: VLDB, vol. 2000, pp. 496–505 (2000)
6. Apache Flink: FlinkCEP — complex event processing for Flink. https://nightlies.apache.org/flink/flink-docs-stable/docs/libs/cep/ (2024)
7. Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., Van Hovell, H., Ionescu, A., Łuszczak, A., et al.: Delta lake: high-performance acid table storage over cloud object stores. Proceedings of the VLDB Endowment **13**(12), 3411–3424 (2020)
8. Bruno, N., Chaudhuri, S.: Automatic physical database tuning: A relaxation-based approach. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pp. 227–238 (2005)
9. Bruno, N., Chaudhuri, S.: An online approach to physical design tuning. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE) (2007). DOI 10.1109/ICDE.2007.367928
10. Chaudhuri, S., Narasayya, V.: Autoadmin "what-if" index analysis utility. ACM SIGMOD Record **27**(2), 367–378 (1998)
11. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Computing Surveys **44**(3), 1–62 (2012). DOI 10.1145/2187671.2187677. URL http://dx.doi.org/10.1145/2187671.2187677
12. Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.M., et al.: Cayuga: A general purpose event monitoring system. In: Cidr, vol. 7, pp. 412–422 (2007)
13. Di Ciccio, C., Mecella, M.: On the discovery of declarative control flows for artful processes. ACM Transactions on Management Information Systems **5**(4), 24:1–24:37 (2015). DOI 10.1145/2629447
14. Dindar, N., Güç, B., Lau, P., Ozal, A., Soner, M., Tatbul, N.: Dejavu: declarative pattern matching over live and archived streams of events. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 1023–1026 (2009)
15. van Dongen, B.: Real-life event logs - hospital log (2011). DOI 10.4121/UUID:D9769F3D-0AB0-4FB8-803B-0D1120FFCF54. URL https://data.4tu.nl/articles/_/12716513/1
16. van Dongen, B.: Bpi challenge 2012 (2012). DOI 10.4121/UUID:3926DB30-F712-4394-AEBC-75976070E91F. URL https://data.4tu.nl/articles/_/12689204/1
17. van Dongen, B.: Bpi challenge 2017 (2017). DOI 10.4121/UUID:5F3067DF-F10B-45DA-B98B-86AE4C7A310B. URL https://data.4tu.nl/articles/_/12696884/1
18. van Dongen, B., Borchert, F.F.: Bpi challenge 2018 (2018). DOI 10.4121/UUID:3301445F-95E8-4FF0-98A4-901F1F204972. URL https://data.4tu.nl/articles/_/12688355/1
19. van Dongen, B.B.: Bpi challenge 2015 (2015). DOI 10.4121/UUID:31A308EF-C844-48DA-948C-305D167A0EC1. URL https://data.4tu.nl/collections/_/5065424/1
20. Esser, S., Fahland, D.: Storing and querying multi-dimensional process event logs using graph databases. In: C. Di Francescomarino, R. Dijkman, U. Zdun (eds.) Business Process Management Workshops, pp. 632–644. Springer International Publishing, Cham (2019)
21. Esser, S., Fahland, D.: Multi-dimensional event data in graph databases. Journal on Data Semantics **10**(1), 109–141 (2021)
22. Faloutsos, C., Ranganathan, M., Manolopoulos, Y.: Fast subsequence matching in time-series databases. ACM Sigmod Record **23**(2), 419–429 (1994)
23. Fan, Y., Ye, Y., Chen, L.: Malicious sequential pattern mining for automatic malware detection. Expert Systems with Applications **52**, 16–25 (2016)
24. Feng, X., Jin, G., Chen, Z., Liu, C., Salihoğlu, S.: Kùzu graph database management system. In: Conference on Innovative Data Systems Research (CIDR) (2023)
25. Fournier-Viger, P., Lin, J.C.W., Kiran, R.U., Koh, Y.S., Thomas, R.: A survey of sequential pattern mining. Data Science and Pattern Recognition **1**(1), 54–77 (2017)
26. Garofalakis, M.N., Rastogi, R., Shim, K.: SPIRIT: Sequential pattern mining with regular expression constraints. In: Proceedings of the 25th International Conference on Very Large Data Bases (VLDB), pp. 223–234 (1999)
27. George, L., Cadonna, B., Weidlich, M.: Il-miner: Instance-level discovery of complex event patterns. Proceedings of the VLDB Endowment **10**(1), 25–36 (2016)
28. Ghahfarokhi, A.F., Park, G., Berti, A., van der Aalst, W.M.: Ocel: a standard for object-centric event logs. In: European Conference on Advances in Databases and Information Systems, pp. 169–175. Springer (2021)
29. Giatrakos, N., Alevizos, E., Artikis, A., Deligiannakis, A., Garofalakis, M.: Complex event recognition in the big data era: a survey. The VLDB Journal **29**(1), 313–352 (2020)
30. Gormley, C., Tong, Z.: Elasticsearch: the definitive guide: a distributed real-time search and analytics engine. " O'Reilly Media, Inc." (2015)
31. Graefe, G., Kuno, H.: Self-selecting, self-tuning, incrementally optimized indexes. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 371–381 (2010)
32. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. ACM SIGMOD Record **29**(2), 1–12 (2000). DOI 10.1145/335191.335372. URL http://dx.doi.org/10.1145/335191.335372
33. Hayamizu, Y., Kawamichi, R., Goda, K., Kitsuregawa, M.: Benchmarking and performance analysis of event sequence queries on relational database. In: Technology Conference on Performance Evaluation and Benchmarking, pp. 110–125. Springer (2018)
34. He, S., He, P., Chen, Z., Yang, T., Su, Y., Lyu, M.R.: A survey on automated log analysis for reliability engineering. ACM computing surveys (CSUR) **54**(6), 1–37 (2021)
35. Huang, S., Zhu, E., Chaudhuri, S., Spiegelberg, L.: T-rex: Optimizing pattern search on time series. Proceedings of the ACM on Management of Data **1**(2), 1–26 (2023)
36. Huang, Y., Zhang, L., Zhang, P.: A framework for mining sequential patterns from spatio-temporal event data sets. IEEE Transactions on Knowledge and data engineering **20**(4), 433–448 (2008)
37. Idreos, S., Kersten, M.L., Manegold, S., et al.: Database cracking. In: CIDR, vol. 7, pp. 68–78 (2007)
38. Jiang, Z., Liu, J., Huang, J., Li, Y., Huo, Y., Gu, J., Chen, Z., Zhu, J., Lyu, M.R.: A large-scale evaluation for log parsing techniques: How far are we? In: Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis, pp. 223–234 (2024)
39. Kolchinsky, I., Schuster, A.: Efficient adaptive detection of complex event patterns. Proc. VLDB Endow. **11**(11), 1346–1359 (2018). DOI 10.14778/3236187.3236190. URL https://doi.org/10.14778/3236187.3236190

40. Kotidis, Y., Roussopoulos, N.: DynaMat: A dynamic view management system for data warehouses. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 371–382 (1999). DOI 10.1145/304182.304215
41. Lambrecht, L., Findling, T., Heid, S., Knüdeler, M., Grust, T.: Democratize match_recognize! Proceedings of the VLDB Endowment **18**(12), 5251–5254 (2025)
42. Li, G., de Murillas, E.G.L., de Carvalho, R.M., Van Der Aalst, W.M.: Extracting object-centric event logs to support process mining on databases. In: International Conference on Advanced Information Systems Engineering, pp. 182–199. Springer (2018)
43. Luo, C., Carey, M.J.: Lsm-based storage techniques: a survey: C. luo, mj carey. The VLDB Journal **29**(1), 393–418 (2020)
44. Mannila, H., Toivonen, H., Inkeri Verkamo, A.: Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery **1**(3), 259–289 (1997). DOI 10.1023/a:1009748302351. URL `http://dx.doi.org/10.1023/A:1009748302351`
45. Marin-Castro, H.M., Tello-Leal, E.: Event log preprocessing for process mining: a review. Applied Sciences **11**(22), 10,556 (2021)
46. Mavroudopoulos, I., Balaktsis, C., Gounaris, A.: A comprehensive scalable framework for cloud-native pattern detection with enhanced expressiveness. Information Systems **140**, 102,725 (2026). DOI https://doi.org/10.1016/j.is.2026.102725. URL `https://www.sciencedirect.com/science/article/pii/S0306437926000396`
47. Mavroudopoulos, I., Balaktsis, C., Varvoutas, K., Kougka, G., Gounaris, A., Comuzzi, M.: Declarative process mining in big data scenarios using an application-agnostic framework. Process Science **2**(1), 6 (2025)
48. Mavroudopoulos, I., Gounaris, A.: Siesta: A scalable infrastructure of sequential pattern analysis. IEEE Transactions on Big Data **9**(3), 975–990 (2022)
49. Mavroudopoulos, I., Tsichlas, K., Gounaris, A.: Sequential pattern detection: similarities and differences across various fields: I. mavroudopoulos et al. Data Mining and Knowledge Discovery **39**(4), 38 (2025)
50. Mooney, C.H., Roddick, J.F.: Sequential pattern mining–approaches and algorithms. ACM Computing Surveys (CSUR) **45**(2), 1–39 (2013)
51. González López de Murillas, E., Reijers, H.A., Van Der Aalst, W.M.: Connecting databases with process mining: a meta model and toolset. Software & Systems Modeling **18**(2), 1209–1247 (2019)
52. Newman, M.E.: Power laws, pareto distributions and zipf's law. Contemporary physics **46**(5), 323–351 (2005)
53. Pei, J., Han, J., Wang, W.: Mining sequential patterns with constraints in large databases. In: Proceedings of the eleventh international conference on Information and knowledge management, pp. 18–25 (2002)
54. Pei, J., Han, J., Wang, W.: Constraint-based sequential pattern mining: the pattern-growth methods. Journal of Intelligent Information Systems **28**(2), 133–160 (2007)
55. Pinto, H., Han, J., Pei, J., Wang, K., Chen, Q., Dayal, U.: Multi-dimensional sequential pattern mining. In: Proceedings of the tenth international conference on Information and knowledge management, pp. 81–88 (2001)
56. Raedt, L.D., Zimmermann, A.: Constraint-based pattern set mining. In: proceedings of the 2007 SIAM International conference on Data Mining, pp. 237–248. SIAM (2007)
57. van Renen, A., Horn, D., Pfeil, P., Vaidya, K., Dong, W., Narayanaswamy, M., Liu, Z., Saxena, G., Kipf, A., Kraska, T.: Why TPC is not enough: An analysis of the Amazon Redshift fleet. Proceedings of the VLDB Endowment **17**(11), 3694–3706 (2024). DOI 10.14778/3681954.3682031
58. Schuhknecht, F.M., Jindal, A., Dittrich, J.: An experimental evaluation and analysis of database cracking. The VLDB Journal **25**(1), 27–52 (2016)
59. Schönig, S., Di Ciccio, C., Maggi, F.M., Mendling, J.: Discovery of Multi-perspective Declarative Process Models, p. 87–103. Springer International Publishing (2016). DOI 10.1007/978-3-319-46295-0_6. URL `http://dx.doi.org/10.1007/978-3-319-46295-0_6`
60. Van Der Aalst, W., Adriansyah, A., De Medeiros, A.K.A., Arcieri, F., Baier, T., Blickle, T., Bose, J.C., Van Den Brand, P., Brandtjen, R., Buijs, J., et al.: Process mining manifesto. In: International conference on business process management, pp. 169–194. Springer (2011)
61. Vautier, A., Cordier, M.O., Quiniou, R.: An inductive database for mining temporal patterns in event sequences. In: International Joint Conference On Artificial Intelligence, vol. 19, p. 1640. LAWRENCE ERLBAUM ASSOCIATES LTD (2005)
62. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 407–418 (2006). DOI 10.1145/1142473.1142520
63. Wu, H., Salzberg, B., Zhang, D.: Online event-driven subsequence matching over financial data streams. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pp. 23–34 (2004)
64. Yeshchenko, A., Mendling, J.: A survey of approaches for event sequence analysis and visualization. Information Systems **120**, 102,283 (2024). DOI https://doi.org/10.1016/j.is.2023.102283. URL `https://www.sciencedirect.com/science/article/pii/S0306437923001199`
65. Zemke, F., Witkowski, A., Cherniack, M., Colby, L.: Pattern matching in sequences of rows. Tech. rep., ANSI Standard Proposal (2007). ISO/IEC JTC1/SC32 WG3
66. Zhu, J., He, S., He, P., Liu, J., Lyu, M.R.: Loghub: A large collection of system log datasets for ai-driven log analytics. In: 2023 IEEE 34th international symposium on software reliability engineering (ISSRE), pp. 355–366. IEEE (2023)
67. Ziehn, A.: Complex event processing for modern data processing environments (2026)